Use of Peyton Jones' Contract Descriptive Language to Evaluate Different Value Processes

L.J.M.V.R.Balalla
Index No: 13000128

Supervisors: Dr. Chamath Keppitiyagama,
Dr. Kasun Gunawardana

May 2018

Submitted in partial fulfillment of the requirements of the
B.Sc (Hons) in Computer Science Final Year Project (SCS4124)

Declaration

I certify that this dissertation does not incorporate, without acknowledgment, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name: L.J.M.V.R.Balalla

……

Signature of Candidate Date:

This is to certify that this dissertation is based on the work of

Mr. L.J.M.V.R.Balalla

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Supervisor Name: Dr. Chamath Keppitiyagama

……

Signature of Supervisor Date:

Abstract

Contracts play a major role in financial markets. Financial contracts vary in different aspects from one to another and as such, they need proper formalization and mechanism for valuation. Peyton Jones have introduced a contract descriptive language that can be used for both representation and valuation of financial contracts. This language has become popular among the research community and hence many improvements and applications have been made with this language since its introduction.

This research aims to explore different processing activities (value processes) that can be applied to Peyton Jones' contract descriptive language. By doing so the language becomes more powerful from its functionalities. In this research, we mainly explore two such processing activities. Those are, generating a calendar for contractual obligations and valuating stochastic processes using Monte Carlo simulation.

Creating a model to generate a calendar for a contract written in Peyton Jones' contract descriptive language helps users to track the timeline of the contract. We introduce a model that has the capability of generating a calendar for a given contract. This model consists of calendar definition, a set of combinators for calendars and a set of evaluation semantics for the conversion from a contract to the calendar.

Peyton Jones have proposed a valuation model for their contract descriptive language. They used a lattice valuation model to calculate the present value of future cash flows of the contracts. They have claimed that instead of this lattice valuation model, other valuation techniques can be used. Our aim is to support this claim by creating a model to valuate contracts in Peyton Jones' contract descriptive language using Monte Carlo simulation. A simple Monte Carlo simulation method described by Boyle for contracts is used for valuation of contracts. The Monte Carlo simulation and evaluation semantics of contracts are implemented with stochastic processes.

Preface

This thesis focuses on introducing different processing activities related to Peyton Jones' contract descriptive language. It is mainly based on the work done by Peyton Jones et al. in developing the domain specific language for financial contracts. As such, we used their language components as the basis of our designs. This thesis summarizes their work in Chapter 1.

Chapter 3 contains proposed models from these researches. Calendar model proposed in this chapter is entirely my work and it only relies on Peyton Jones' contract descriptive language for representations of contracts. The concept behind this model has not been proposed in any other study related to this research area. Proposed stochastic process valuation semantics are a modification of valuation semantics introduced by Peyton Jones et al. and it uses the Monte Carlo simulation method to discount the value of a contact. This valuation approach has never been directly combined with evaluation semantics before. It is a major achievement of this research.

Chapter 4 contains a description for implementation of the Peyton Jones' contract descriptive language and implementation of the proposed model. Contract Descriptive Language implementation is a slight modification of Peyton Jones et al. implementation. All the implementations of proposed models are done from the scratch and not been present in any other study.

Acknowledgement

I would like to express my sincere appreciation to my principal Supervisor, Dr. Chamath Keppitiyagama and Co-supervisor Dr. Kasun Gunawardana for their constant guidance and encouragement, without which this work would not have been possible. For their unwavering support, I am truly grateful. I am also grateful to all lecturers in University of Colombo School of Computing, especially Dr. Mindika Premachandra and Dr. K. D. Sandaruwan for their support towards the successful completion of this research.

My sincere thanks go out to our research coordinator Dr. H.E.M.H.B.Ekanayake for his encouragement and support in keeping this research focused and on-track. I extend my gratitude to Dr. Damith Karunarathne and Dr. Ruwan Weerasinghe for the immense guidance they offered me by providing their valuable feedback as examiners. Their advice and suggestions encouraged me to carry out this one year research project more successfully and present a useful outcome at the end.

Foremost my special thanks to my parents for providing me a solid foundation in edu cation and all the courage and love gave me on every moment. They are the guiding stars which strengthen me to become the person who I am today.

Finally, I express my sincere gratitude for all my friends who supported and encouraged me on all cause of challenges I faced during this research. All the help given by everyone to make this research a success owns my great appreciation.

Table of Contents

Table of Figures

Acronyms

FCs	Financial Contracts
FDs	Financial Derivatives
FMs	Financial Markets
FP	Functional Programming
DSL	Domain Specific Language
DSELs	Domain Specific Languages
HCCL	Haskell Contract Combinator Library
LSEG	London Stock Exchange Group
SPL	Stochastic Process Language
PFPL	Probabilistic Functional Programming Library
CS	Computer Science
USD	US Dollar
AUD	Australian Dollar
NZD	New Zealand Dollar
VP	Value Process
SP	Stochastic Process
SVP	Stochastic Value Process
ZCB	Zero Coupon Bond

Chapter 1 - Introduction

1.1 Background to the Research

1.1.1 Financial Market

A financial market is a context where people trade financial securities, commodities, and other fungible[1] items of value at low transaction costs and at prices that reflect supply and demand [1]. Financial markets do thousands of transactions per second. There are many kinds of financial instruments traded in those transactions.

1.1.2 Financial Contracts

In financial markets, one of the main concerns is to legally document and process financial derivatives. Some financial derivatives are so complex and there is no unified format to represent them [1]. A financial derivative(FD) is a contract between two or more parties based on a financial asset [2]. And many different types of FDs are there in the world and it is cumbersome to manage each of them separately. Because there is no universal way to manage them and different derivatives have different formats.

Following are some financial derivatives which are commonly traded in financial markets,

- Zero Coupon Bonds - A zero-coupon bond, is a debt security that doesn't pay interest but is traded at a deep discount [2].
- European Options - A European option is an option that can only be exercised at the end of its life, at its maturity [3].

[1] Interchangeability with other individual goods or assets of the same type.

- American Options[2] - An American option is an option that can be exercised anytime during its life. American options allow option holders to exercise the option at any time prior to and including its maturity date [4].
- Asian Options - An Asian option is an option whose payoff depends on the average price of the underlying asset over a certain period of time as opposed to at maturity [5].

These contacts can get combined together to generate complex contracts. Consider the following story of a contract.

"An investor bought an American call option on 1st Jan 2017 on stock XYZ with a strike price of 100GBP. And need to pay a premium of 5GBP on 1st Feb 2017. This call option will get expired on 1st July 2017"

This contract consists of an American option and a zero coupon bond. Likewise, contracts can combine to generate new contracts.

1.1.3 Informal Contract Management

There are many problems that can arise in connection with informal modeling and representation of contracts and their execution. According to Andersen, J. et al. [1] those are (i) disagreement on what a contract actually requires; (ii) agreement on contract, but disagreement on what events have actually happened (event history); (iii) agreement on contract and event history, but disagreement on remaining contractual obligations; (iv) breach or malexecution of contract; (v) entering bad or undesirable contracts/missed opportunities; (vi) bad coordination of contractual obligations with production planning and supply chain management; (vii) impossibility, slowness or costliness in evaluating state of company affairs.

As an example of consequences of those, a major French investment bank has costs of about 50 million. Euro per year and about half due to legal costs in connection with contract disputes and the other half due to malexecution of financial contracts [1].

Most of the contracts are comprised of smaller subcontracts. Those composite contracts are harder to represent and valuate. When different processing activities need to be applied to different contracts each contract need to have a separate model for each processing activity. So from the look of it, we can say that it is very cumbersome. In the industry, if we can represent

[2] Thorough analysis will be done in section 3.3

all contracts in the world in a single format, then the amount of work that needs for processing will reduce drastically.

1.1.4 Domain Specific Language (DSL)

Today DSLs are very common in the financial industry. There can be many advantages of using a DSL over a general purpose language and one of them is, high level of abstraction. According to P.Hudak [6], there are many different ways that can be used to build a DSL. DSL which proposed by Peyton Jones et al. [7] is a Domain Specific Embedded Language (DSEL). DSEL is a DSL, which implemented using another general purpose language. One of the main advantages of this method is that DSEL inherits characteristics from its mother language. Languages such as MetaOCaml, Template Haskell, and C++ are proven to be good candidates for building DSELs [8].

1.1.5 Peyton Jones' Contract Descriptive Language

Peyton Jones et al. [7] has proposed a language which consists of combinator libraries for observables and contracts. According to this language observable is a time-varying quantity such as interest rate. In the contract language, the observables are defined as data whose type is ***Obs a***, where a can be any type. And a combinator for contracts is a function which always returns a contract.

This contract descriptive language provide greater flexibility in representing contracts. It is a powerful tool so that it can preserve important information about the contract in a single line. Let's consider two simple contracts called ***C1*** and ***C2***.

C1 = Receive $100 on date t1

C2 = Transfer £200 on date t2

From the Peyton Jones' contract descriptive language, we can represent these two contracts as follows,

C1 = scaleK 100 (get (truncate t1 (one **USD**)))

C2 = scaleK 200 (give (truncate t2 (one **GBP**)))

When compared with alternative methods, where they need a large amount of code to do the same. And when it's come to comparing two contracts this language made it very easy

and efficient. Because both these contracts are defined using the same set of combinators, it is easy to compare them together. In this case, ***scaleK, truncate*** and ***one*** are common to both contracts and ***get*** (represents a cash inflow), ***give*** (represents a cash inflow) are unique for each contract. Another important feature of this language is that contracts can be combined together to form new contracts. As an example, a new contract ***C3*** can be defined as follows,

C3 = and **C1 C2**

C3 = and (scaleK 100 (get (truncate t1 (one **USD**))))
(scaleK 200 (give (truncate t2 (one **GBP**))
)

C3 states that its contract holder receives ***$100*** at ***t1*** and transfer of ***£200*** at ***t2***. Likewise, any contract can be defined using this contract descriptive language.

Figure 1 Shows 10 combinators for contracts introduced by Peyton Jones et al. [7]. As for observables, a combinator for observables is a function which always returns an observable. Figure 2 Shows 5 combinators for observables introduced by Peyton Jones et al. [7].

`zero :: Contract`
`zero` is a contract that may be acquired at any time. It has no rights and no obligations, and has an infinite horizon. (Section 3.4.)

`one :: Currency -> Contract`
`(one k)` is a contract that immediately pays the holder one unit of the currency `k`. The contract has an infinite horizon. (Section 3.2.)

`give :: Contract -> Contract`
To acquire `(give c)` is to acquire all `c`'s rights as obligations, and vice versa. Note that for a bilateral contract `q` between parties A and B, A acquiring `q` implies that B acquires `(give q)`. (Section 2.2.)

`and :: Contract -> Contract -> Contract`
If you acquire `(c1 'and' c2)` then you immediately acquire both `c1` (unless it has expired) and `c2` (unless it has expired). The composite contract expires when both `c1` and `c2` expire. (Section 2.2.)

`or :: Contract -> Contract -> Contract`
If you acquire `(c1 'or' c2)` you must immediately acquire either `c1` or `c2` (but not both). If either has expired, that one cannot be chosen. When both have expired, the compound contract expires. (Section 3.4.)

`truncate :: Date -> Contract -> Contract`
`(truncate t c)` is exactly like `c` except that it expires at the earlier of `t` and the horizon of `c`. Notice that `truncate` limits only the possible *acquisition date* of `c`; it does *not* truncate `c`'s rights and obligations, which may extend well beyond `t`. (Section 3.4.)

`then :: Contract -> Contract -> Contract`
If you acquire `(c1 'then' c2)` and `c1` has not expired, then you acquire `c1`. If `c1` has expired, but `c2` has not, you acquire `c2`. The compound contract expires when both `c1` and `c2` expire. (Section 3.5.)

`scale :: Obs Double -> Contract -> Contract`
If you acquire `(scale o c)`, then you acquire `c` at the same moment, except that all the rights and obligations of `c` are multiplied by the value of the observable `o` at the moment of acquisition. (Section 3.3.)

`get :: Contract -> Contract`
If you acquire `(get c)` then you must acquire `c` at `c`'s expiry date. The compound contract expires at the same moment that `c` expires. (Section 3.2.)

`anytime :: Contract -> Contract`
If you acquire `(anytime c)` you must acquire `c`, but you can do so at any time between the acquisition of `(anytime c)` and the expiry of `c`. The compound contract expires when `c` does. (Section 3.5.)

Figure 1: Primitives for Defining Contracts

konst :: $a \rightarrow Obs\ a$
(*konst* x) is an observable that has value x at any time.

lift :: $(a \rightarrow b) \rightarrow Obs\ a \rightarrow Obs\ b$
(*lift* f o) is the observable whose value is the result of applying f to the value of the observable o.

$lift_2$ :: $(a \rightarrow b \rightarrow c) \rightarrow Obs\ a \rightarrow Obs\ b \rightarrow Obs\ c$
($lift_2$ f o_1 o_2) is the observable whose value is the result of applying f to the values of the observables o_1 and o_2.

date :: *Obs Date*
The value of the observable *date* at date s is just s.

instance Num a => Num (Obs a)
All numeric operations lift to the *Obs* type. The implementation is simple, using *lift* and $lift_2$.

Figure 2: Primitives for Defining Observables

1.1.6 Financial Contract Valuation

There are many financial models that can be used to valuate financial contracts. Among those models, only three families of numerical methods are widely used in industry. Those are partial differential equations, Monte Carlo Simulation, and lattice methods. Using the language proposed by Peyton Jones et al. [7], they have created a model to valuate contracts using the lattice valuation method.

1.1.7 Monte Carlo valuation for options

Monte Carlo model is widely used in valuating options [4]. Steps used in Monte Carlo valuation for options contracts,

- Generate a large number of possible, but random, price paths for the underlying (or underlying) via simulation.
- Then calculate the associated exercise value of the option for each path.
- These payoffs are then averaged and discounted to today.
- This result is the value of the option.

One advantage of using Monte Carlo simulation over other simpler analytical models is that it provides more statistical information about the discounted value [9].

1.2 Research Problem and Research Questions

1.2.1 Research Questions

1.2.1.1 Question 1

Is it possible to create a model to generate the calendar[3] for a contract written in Peyton Jones' contract descriptive language?

Calendar of a contract is a time line that represents every transaction that scheduled to happen. Buyer, Seller and the Issuer of a contact need to keep track of the time line of that contract. Generating a calendar will benefit all associated parties.

1.2.1.1 Question 2

Can Peyton Jones' contract descriptive language be used to valuate contracts using Monte Carlo simulation?

Peyton Jones et al. [7] showed that contracts in proposed language can be valuated using the lattice valuation model. And he stated that the language can be used to evaluate other processing activities as well. But has not been proved.

1.3 Aims and Objectives

The intention of this research is to discover possibilities of applying different processing activities to Peyton Jones' contract descriptive Language.

- Create a model to generate the calendar for a contract written in Peyton Jones' contract descriptive language.

[3] Calendar is an action schedule of a contract that consists of rights and obligations of that contract. A Proper definition will be given in section 5.1.1.

- Create a model to represent stochastic processes.
- Create a model to generate Monte Carlo simulation for contract value.
- Valuate Options described in Peyton Jones' contract descriptive language using Monte Carlo simulation.

1.4 Delimitations of Scope

For the first research question, proposed models work only on Peyton Jones et al. [7] proposed language. Because of the time constraint, following financial derivatives are only considered,

- Zero Coupon Bonds
- American Options
- European Options

And when complex contracts are needed, only previously mentioned contracts are combined.

All the implementations are done using language Haskell and test only for that language.

1.5 Motivation for the research

Financial contracts play one of the most important roles in the financial world. Thus, financial contracts have a high frequency of use in the field of finance. Among the vast variety of financial contracts that are being created every day, most of them end up being very complex due to their requirement of being able to represent various business needs and agreements. Further, these complex business contracts contain subcontracts that have their own life cycles.

Peyton Jones' contract descriptive language was developed with the intention of making it easier to represent and valuate those contracts. But one of the main restriction that can be seen in that research is that the valuation of these contracts has been done only using lattice valuation model. In the industry, other valuation models like Monte Carlo valuation are very popular. Other than valuation, there are many processing activities that need to be done for financial contracts.

One of the main motivational factors of this research is the keynote speech done by Jean-Marc Eber at The Domain-Specific Languages for Financial Systems (DSLFIN) 2013 workshop [10]. In this speech, he empathized the need of a calendar which can detect all meaningful events that will or may happen in the future with regards to financial contracts.

The main reason for selecting Peyton Jones' contract descriptive language is that it is the root of most of the DSLs in the financial market. And it was well documented and many researchers have tested it for different types of financial contracts.

1.6 Methodology

Peyton Jones et al. [7] has used a precise and scientific methodology when expressing contract valuation. For this research, the same method is followed when building other processing activities. This methodology can be divided into two layers.

- Abstract evaluation semantics
- Concrete implementation

First, a mathematical model is created for the conversion of any contract written in Peyton Jones' contract descriptive language into a process. Then operations are defined over those processes. Then comes the implementation of those processes to match to a real-world activity.

1.7 Outline of the rest of the desertation

In Chapter two, **Literature Review**, illustrates the current status of the research domain, especially targeting towards the Peyton Jones' contract descriptive language and how contract management is done. Then this chapter reviews different valuation methods used for Peyton Jones' contract descriptive language. Finally, it discusses representation and valuation of stochastic processes.

In Chapter three, **Design**, consist of language models we have proposed and justifications for each design component. The first half of this chapter proposes a design of a model to generate

a calendar for Peyton Jones' contract descriptive language with justifications. And in the second part, we introduce to a language model which can use to simulate the contract value of a contract written in Peyton Jones' contract descriptive language with Monte Carlo Method. All the design decisions are justified.

In Chapter four, **Implementation**, illustrate an implementation of the Peyton Jones' contract descriptive language and models proposed in Chapter 4. All the implementations are done using the language Haskell. This chapter also introduces Haskell and its lazy evaluation principles. Codes are provided to illustrate important implementation decisions.

In Chapter five, **Evaluation and Conclusion**, example contract evaluations are done for models introduced in Chapter 4 as proof of concept. This chapter gives a conclusion for research questions and research objectives based on findings. Next, it describes some limitations in proposed models. Finally, this chapter concludes by introducing toward some future research and experiments possible.

Chapter 2 - Literature Review

This chapter illustrates the current status of the research domain, especially targeting towards the Peyton Jones' contract descriptive language and how contract management is done. Then this chapter reviews different valuation methods used for Peyton Jones' contract descriptive language. Finally, it discusses representation and valuation of stochastic processes.

2.1 Related work

When considering the past decade, there had been several domain specific languages were created for different subdomains related to financial contracts. RISLA is an example of such attempt [11]. This language is used to define interest rate products offered by a bank. As the first attempt to represent financial contracts, Lee [12] has attempted to develop a formal language for electronic contracts via having a common logic model. According to him, a product can be described by representing its cash flows. But after that, almost all those researches were based on Peyton Jones et al. [13] contract description language.

We can divide those researches into two main categories. One category is aiming to improve the language further and apply to other domains. Mediratta [14] and Christiansen et al. [15] applied the Peyton Jones' contract descriptive language for specific domains. Figure 4 summarizes their contributions. Another category is that to use this language for contract valuation. Ahnfelt [16] introduced a language called SPL which can be used to valuate stochastic processes. Figure 3 illustrates how researches have been evolved in this domain.

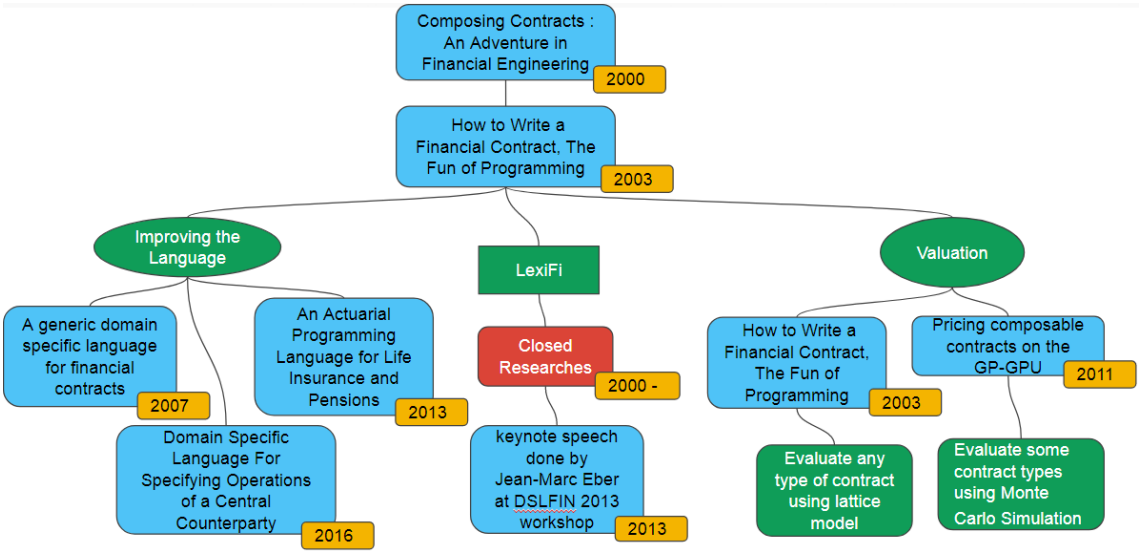

Figure 3: Background Summary

Reference	Key Findings	Drawbacks
Christiansen et al. [9] (MSc Thesis)	• Introduced a DSL for pension and life insurance products. • This DSL can be used for represention, administration, reserve calculations, and audit life insurance and pension products.	• Only applicable for pension and life insurance products.
Med ıtta [7]	• He has tried to test and improve the Peyton Jones language. • He represented and valuated credit default swap and power reverse dual currency swap using this language. • His conclusion was that this language is generic enough to represent and valuate even a new contract.	• Same as in Peyton Jones al. [1] paper, only the lattice evaluation method used.
A felt [8]	• Introduced a language for specifying stochastic processes, called SPL. • This SPL can be used to calculate the price of a range of financial contracts. • Done a Monte Carlo simulation for options using SPL.	• It can't be used to valuate American options.

Figure 4: Related Work

Lexifi [10] had done many researches on this domain and most of their commercial products are based on them. They have done research for pricing and operational management based on the Peyton Jones' contract descriptive language. LexiFi has built a software stack based on this language, implementing all the generic operations listed in Figure 5. They have done those research for many years with the involvement of leading researchers in computer science.

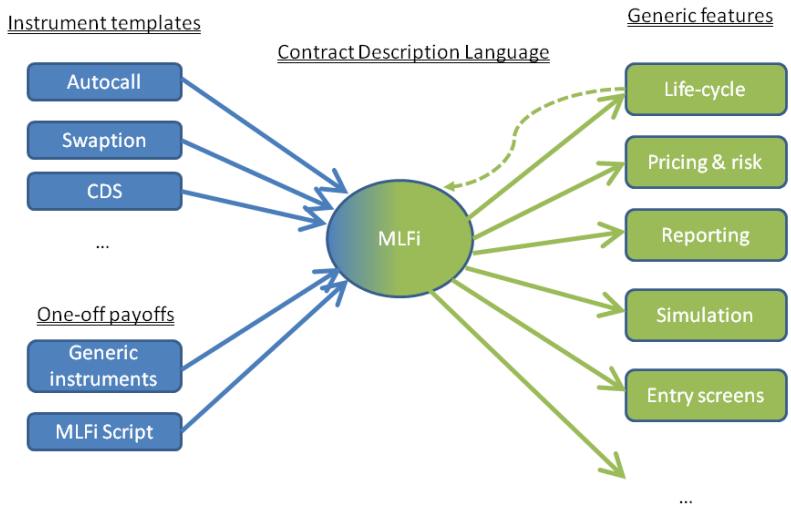

Figure 5: Contract Management

Peyton Jones et al. [13] republished their language with some slight changes. They removed ***truncate*** combinator and added two new combinators called ***cond*** and ***when***. One of the main reasons to do this is to make the language more flexible.

2.2 Contract Management

Contract management is as important as Contract valuation. One important purpose of building Peyton Jones Contract Descriptive Language is to manage contracts [13]. Lexifi has done some research regarding contract management using Peyton Jones' contract description language [17]. They have identified several key features in this language.

- Contract description defines the rights and obligations of the parties both precisely and exhaustively, and independently of any valuation methodology.
- Supports any type or combination of the underlying asset(s).

- Has a well-defined and unambiguous semantics, which makes it possible to reason about contract descriptions and define operations in a generic way.
- Contract description is a "portable" piece data which can be exchanged between computer systems and across stakeholders.

Because of these features, Peyton Jones' contract description language is somewhat suitable for contract management.

But it was not sufficient enough to properly manage all types of contracts. So there are many researches that have been done to Improve Peyton Jones' Contract Descriptive Language for contract management other than valuation.

2.2.1 Compositional Specification of Commercial Contracts

Andersen, J. et al. [1] extends Peyton Jones' contract description language for specifying financial contracts to the exchange of money, goods, and services among multiple parties. In their paper, they have discussed how this extended language can be used for contract management.

2.3 Evaluating Value Processes

Peyton Jones et al. [7] has implemented a value process to generate the value of a contract at a given time. A value process is defined as a partial function of time to a random variable.

$$\mathcal{PR}\ a = \mathcal{DATE} \hookrightarrow \mathcal{RV}\ a$$

They have implemented this value process based on the lattice valuation method. They have introduced a set of compositional evaluation semantics for evaluating composite contract for valuation. Those evaluation semantics first converts the contract into a value process. Those semantics are shown in figure 6 and figure 7.

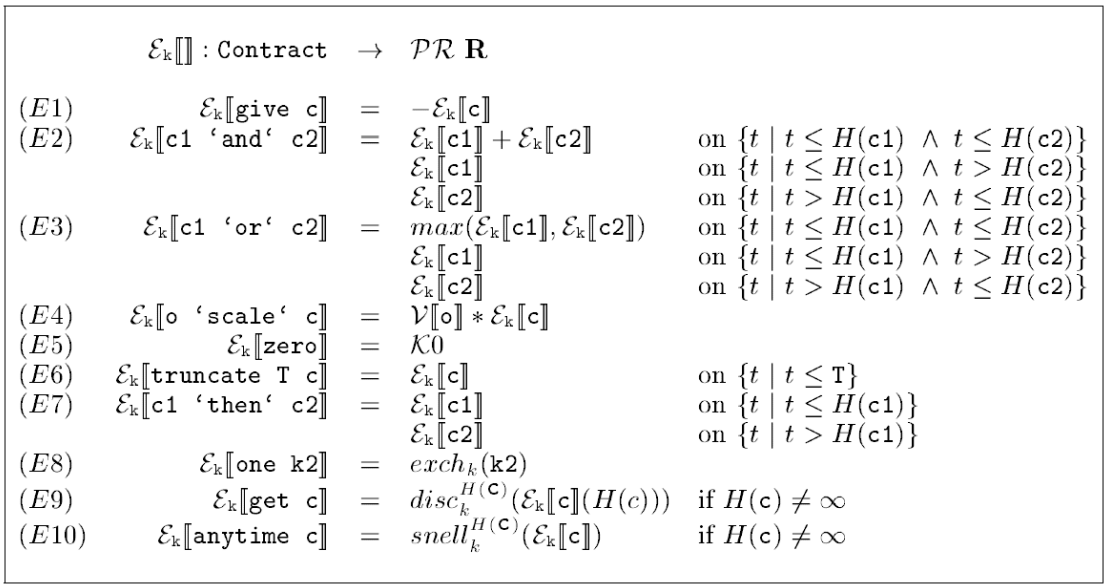

Figure 6: Evaluation Semantics for Contracts

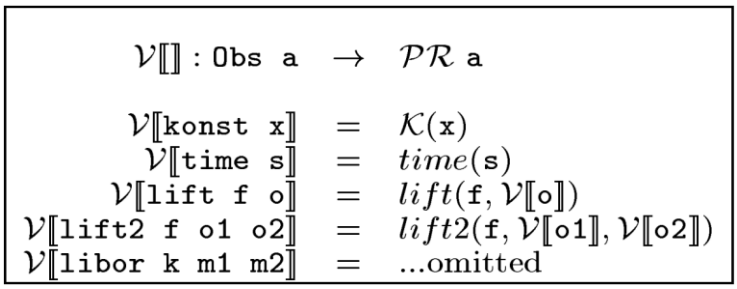

Figure 7: Evaluation Semantics for Observables

After converting into a value process, a financial valuation model is used to convert the value process into concrete implementation. Various mathematical models are used to create such models and lattice approach is the easiest among them. This model represents the value of a contract or an observable using a lattice data structure. Interest rate can also be represented by a lattice data structure. Figure 8 contains such a representation of interest rate over time. Each column of this tree represents a discrete time step.

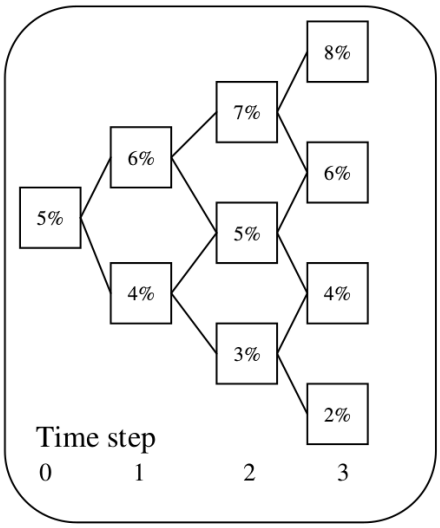

Figure 8: A Short-Term Interest Rate Evaluation

As mentioned, value processes are also modeled as a lattice. Consider the following example contract.

```
(get (scaleK 10 (truncate t (one GBP)))
```

Figure 9 shows the value process of this contract. This value process is generated using the lattice valuation model.

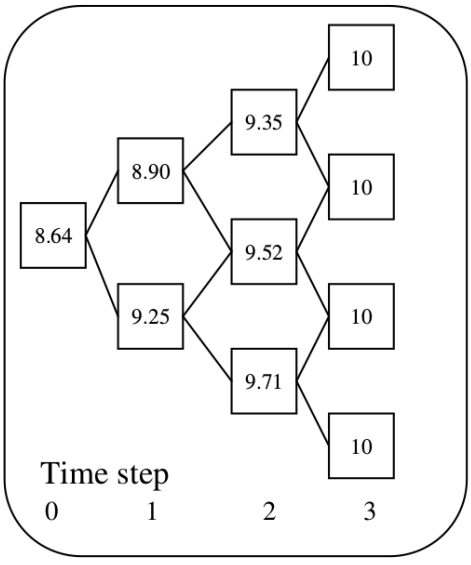

Figure 9: A Valuation Lattice

2.3.1 Valuating Stochastic Processes

Distributions that change over time are called stochastic processes. Ahnfelt [18] has introduced a language for specifying stochastic processes, called SPL. This SPL can be used to calculate the price of a range of financial contracts. They have tested their models for different types of options. And they have done a Monte Carlo simulation as well. But one drawback of this SPL is that it can't be used to valuate American Options. This is due to the semantics of SPL, which assumes that future events can be known in the present. Figure 10 represents a summary of contract valuation for contract descriptive language.

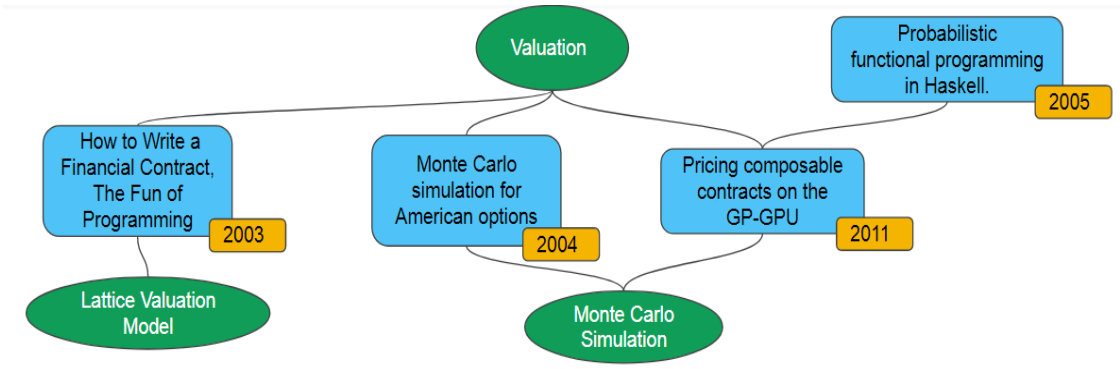

Figure 10: Contract Valuation

2.3.2 Probabilistic Functional Programming Library (PFPL) for Haskell

FPPL is a library for probabilistic functional programming [19]. They used this language for specifying stochastic processes. In this section, I will explain some functionalities of this language.

The basic idea of FPPL is to represent a distribution as a list of all possible outcomes (the sample space) coupled with their probability, which is a real number between 0 (impossible) and 1 (certain).

```
data Dist a = D [(a, Probability)]
```

D data constructor won't create distributions directly. Separate functions were given to create each distribution. To create discrete uniform distribution ***uniform*** construct can be used.

```
data Coin = Heads | Tails
flip :: Dist Coin
flip = uniform [Heads, Tails] Printing
```

Such distributions can be combined together using the ***joinWith*** combinator. And the operator **>>=** can be used when events in a distribution are dependent on each other. So every ***joinWith*** and **>>=** create the Cartesian product of the distributions.

```
flip2 = joinWith both flip flip
```

2.3.3 Representing Stochastic Processes

FPPL model stochastic processes as a list of distributions, ***[Dist a]***. SPL used FPPL with some modifications to represent contracts written Peyton Jones' Contract Descriptive Language as stochastic processes. And Ahnfelt [18] provide an implementation of SPL that performs Monte Carlo simulation using GPGPU.

2.3.4 Monte Carlo Simulation for Valuation

Phelim et al. [4] elaborate, uses of Monte Carlo simulation in security pricing. According to this paper, Contracts and observables are often models as continuous-time stochastic processes. And the price of a contract can be expressed as the expected value of its discounted payouts. So for pricing those contracts, Monte Carlo simulation can be used. According to Phelim et al. [4], Monte Carlo simulation for security pricing can be done using following steps,

- Simulate sample paths of the underlying state variables (e.g., contract prices and interest rates) over the relevant time horizon. Stimulate these according to the risk-neutral measure.
- Evaluate the discounted cash flows of a security on each sample path, as determined by the structure of the security in question.
- Average the discounted cash flows over sample paths.

2.3.5 American options and Monte Carlo simulation

Valuation of American options using Monte Carlo simulation presents some difficulties [20][21]. Monte Carlo methods are required for options that depend on multiple underlying securities or that involve path dependent features. Since the determination of the optimal exercise time depends on an average over future events, Monte Carlo simulation for an American option has a “Monte Carlo on Monte Carlo” feature that makes it computationally complex.

Chapter 3 - Design

This chapter consists of language models we have proposed and justifications for each design component. The first half of this chapter proposes a design of a model to generate a calendar for Peyton Jones' contract descriptive language with justifications. And in the second part, we introduce to a language model which can use to simulate the contract value of a contract written in Peyton Jones' contract descriptive language with Monte Carlo method. All the design decisions are justified.

3.1 Calendar Model

As stated in section 1.2, the first research question is to check the ability to develop a model to generate the calendar for a contract written in Peyton Jones' contract descriptive language.

According to the research methodology, we have developed several calendar models and tested them with examples. Among those models, we have chosen a final model that aligns with our calendar definitions.

3.1.1 Calendar Definition

Calendar for a contract can be defined in many ways. According to Jean-Marc Eber [10] calendar is defined as a mechanism of detecting all meaningful events that will or may happen in the future. And it is an action schedule of a contract that consists of rights and obligations of that contract. For our calendar models, we have the same definition but they have slightly different functionalities. Our alternative calendar model will yield all possible cash flows without considering any past or future decisions. But in the final calendar model, it will yield cash flows only relate to the current decision status.

3.1.2 New Data Types

Related to this model we have introduced two new data types. Those are ***Event*** and ***Calendar***. ***Event*** is a single day on our calendar with a set of possible transactions. ***Event*** is implemented as a list of possible cash inflows and cash outflows.

```
type Event = [Int]
type Calender = Obs Event
```

Calendar is an observable of Events. In other words, Calendar is a function of time to a calculated event. Figure 11 represents the concept behind an event and a calendar.

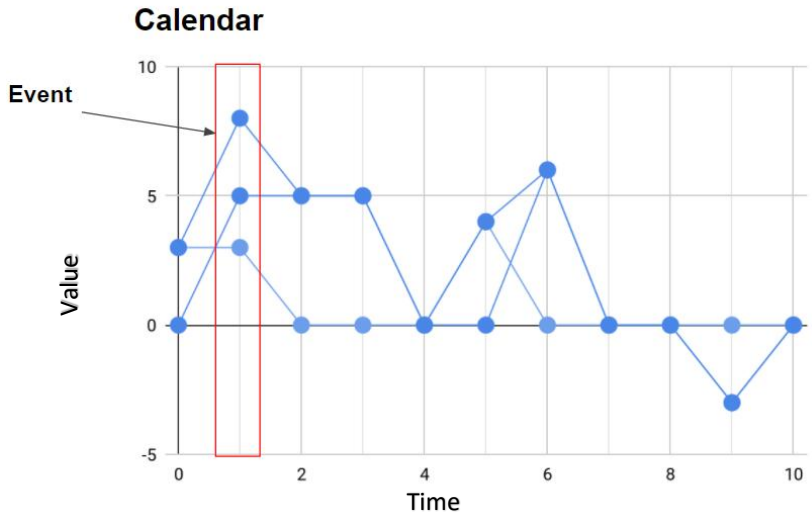

Figure 11: An Event in the Calendar

3.1.3 Evaluation Semantics

To convert Contract into a Calendar set of evaluation semantics were introduced. Figure 13 represents evaluation semantics for the final calendar model. For this model, we have considered time as a relative observable for each calendar. The time starts with 0 and has discrete values which are considered as months.

Let's map each line of evaluation semantics into our calendar definition. The ***β*** function maps contracts into its relevant calendar. First consider the equation which maps ***Zero*** contract into ***zeroCal***. ***Zero*** contract has no rights or obligations. So it should return an empty calendar. According to the definition, ***zeroCal*** represents an empty calendar. ***One k*** represents a contract

with a cash inflow (right) of one at the acquisition date of the contract. It will get maps with a calendar called ***oneCal k*** which represents a cash inflow of one at time 0. ***Give c*** makes all rights of contract c into obligations. To represent this in the calendar we used ***giveCal*** combinator. In our model minus values represent obligations and plus values represent rights. For ***And*** combinator and ***Or*** combinator for contracts, they represent combinations contracts. To combine calendars of two contracts we introduced a new combinator called ***zipCal***. For contracts combined with ***Cond*** combinator, their relative calendars should come to an absolute time scale. To do this we have introduced a combinator called ***shift***.

$$\beta[Contract] \quad \rightarrow \quad Calendar$$

$$\begin{aligned}
\beta[Zero] &= zeroCal && && && 12.1\ (5.1)\\
\beta[One\ \ k] &= oneCal\ \ k && && && 12.2\ (5.2)\\
\beta[Give\ \ c] &= giveCal\ \ \beta[c] && && && 12.3\ (5.3)\\
\beta[o\ \ \text{`}Scale\text{`}\ \ c] &= scaleCal\ \ o\ \ \beta[c] && && && 12.4\ (5.4)\\
\beta[c1\ \ \text{`}And\text{`}\ \ c2] &= zipCal\ \ \beta[c1]\ \ \beta[c2] && && && 12.5\ (5.5)\\
\beta[c1\ \ \text{`}Or\text{`}\ \ c2] &= zipCal\ \ \beta[c1]\ \ \beta[c2] && && && 12.6\ (5.6)\\
\beta[Cond\ \ (Obs\ \ o)\ \ c1\ \ c2] &= shift\ \ \beta[c1]\ \ t && on\ \ \{t|(o\ \ t)\} && 12.7\ (5.7)\\
&= shift\ \ \beta[c2]\ \ t && on\ \ \{t|!(o\ \ t)\} &&\\
\beta[When\ \ (Obs\ \ o)\ \ c] &= shift\ \ \beta[c]\ \ t && on\ \ \{t|(o\ \ t)\} && 12.8\ (5.8)\\
&= zeroCal && on\ \ \{t|!(o\ \ t)\} &&
\end{aligned}$$

Figure 12: Contract evaluation semantics (Final Model)

When gradually building the final contract evaluation model, we came across some alternative calendar models which slightly differ from our calendar definition. One such model is given in Figure 13.

$$\beta[Contract] \quad \rightarrow \quad Calendar$$

$$\begin{aligned}
\beta[Zero] &= zeroCal && && \text{13.1 (5.9)}\\
\beta[One \quad k] &= oneCal \quad k && && \text{13.2 (5.10)}\\
\beta[Give \quad c] &= giveCal \quad \beta[c] && && \text{13.3(5.11)}\\
\beta[o \quad `Scale` \quad c] &= scaleCal \quad o \quad \beta[c] && && \text{13.4(5.12)}\\
\beta[c1 \quad `And` \quad c2] &= zipCal \quad \beta[c1] \quad \beta[c2] && && \text{13.5 (5.13)}\\
\beta[c1 \quad `Or` \quad c2] &= zipCal \quad \beta[c1] \quad \beta[c2] && && \text{13.6(5.14)}\\
\beta[Cond \quad (Obs \quad o) \quad c1 \quad c2] &= geValue \quad \beta[c1] \quad t && on \quad \{t|(o \quad t)\} && \text{13.7 (5.15)}\\
&= getValue \quad \beta[c2] \quad t && on \quad \{t|!(o \quad t)\} &&\\
\beta[When \quad (Obs \quad o) \quad c] &= getValue \quad \beta[c] \quad t && on \quad \{t|(o \quad t)\} && \text{13.8(5.16)}\\
&= getValue \quad zeroCal \quad t && on \quad \{t|!(o \quad t)\} &&
\end{aligned}$$

Figure 13: Contract evaluation semantics (Alternative Model)

3.1.4 Combinators for Calendar

We Introduced 6 new combinators for calendars. Bellow section presents a detailed description of each combinators.

zeroCal	***oneCal***	***giveCal***
scaleCal	***zipCal***	***shift***

```
zeroCal :: Calendar
oneCal :: Calendar
```

These two are the most primitive types of calendars in this model. As described in the contract descriptive language, these combinators are the main building blocks of complex Calendars.

zeroCal is a Calendar, which returns ***zero*** for all the locations in the timeline. This represents an empty calendar whereas ***oneCal*** represents a Calander which returns an event with value 1 at the time 0 and an event with value 0 at every other time. Figure 14 illustrates a representation of ***zeroCal*** and ***oneCal*** on how the value changes with time.

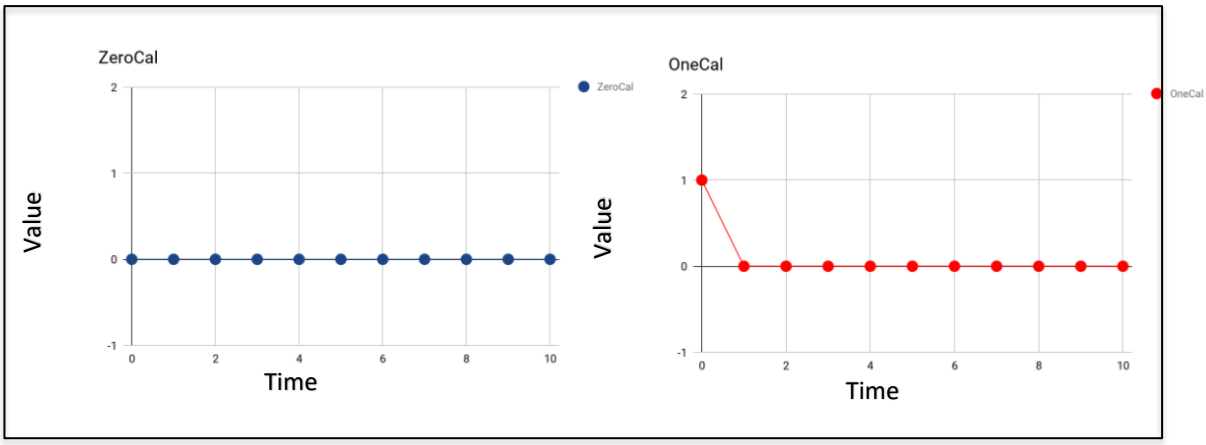

Figure 14: ***zeroCal*** and ***oneCal*** Combinators

```
giveCal :: Calendar -> Calendar
```

giveCal combinator returns the negation of a Calender which represent a cash outflow. Figure 15 represents how calendar events are changed when ***giveCal*** combinator is applied.

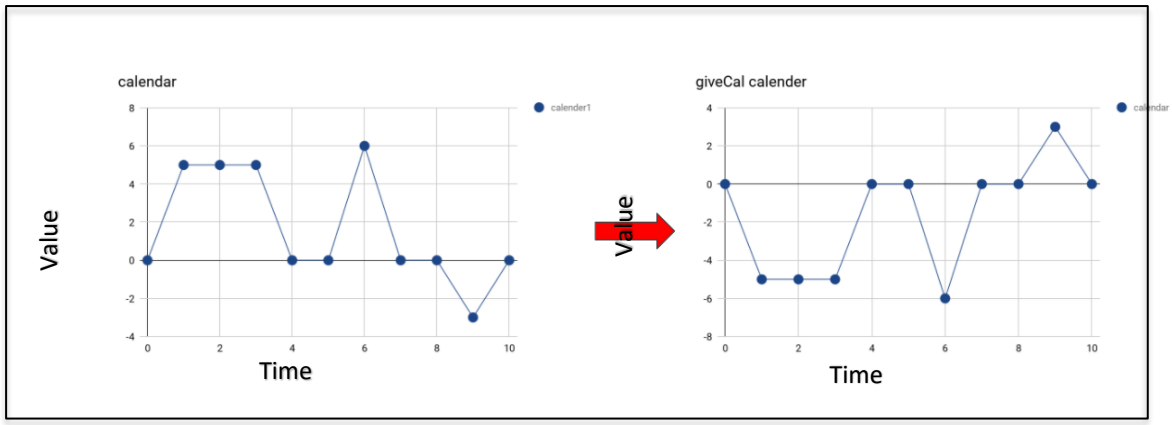

Figure 15: giveCal Combinator

```
scaleCal :: Obs Int -> Calendar -> Calendar
```

scaleCal will scale each value of the Calender with the corresponding value of the observable. Figure 16

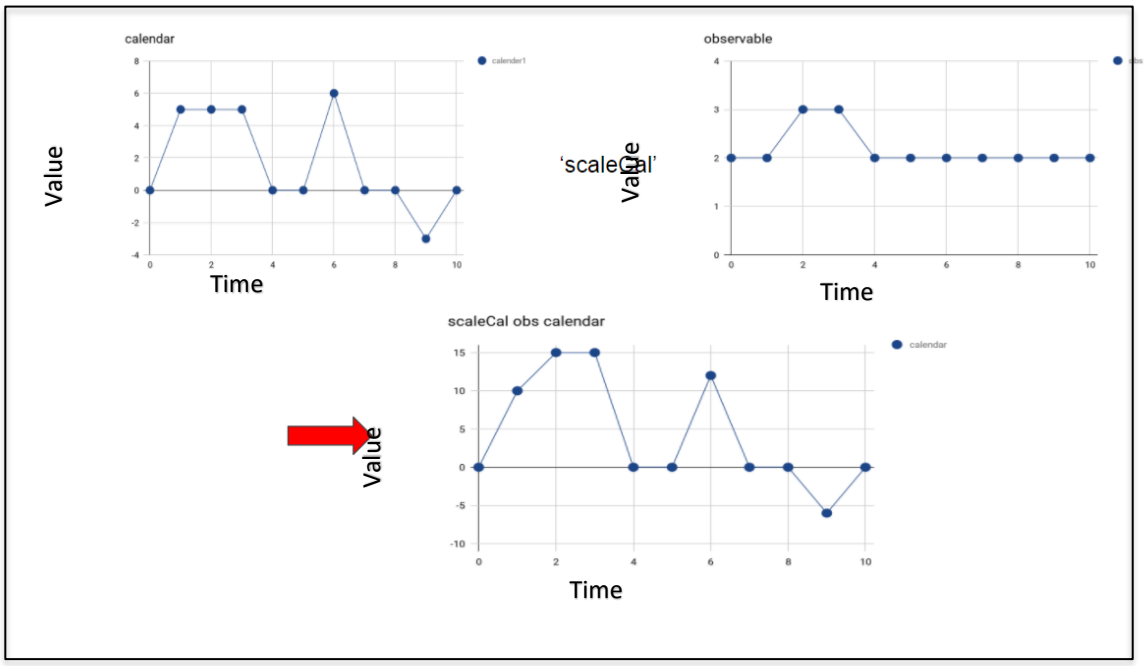

Figure 16: scaleCal Combinator

```
zipCal :: Calendar -> Calendar -> Calendar
```

zipCal combinator is used to merge two calendars. And it combines their events and returns one composite event for a given time.

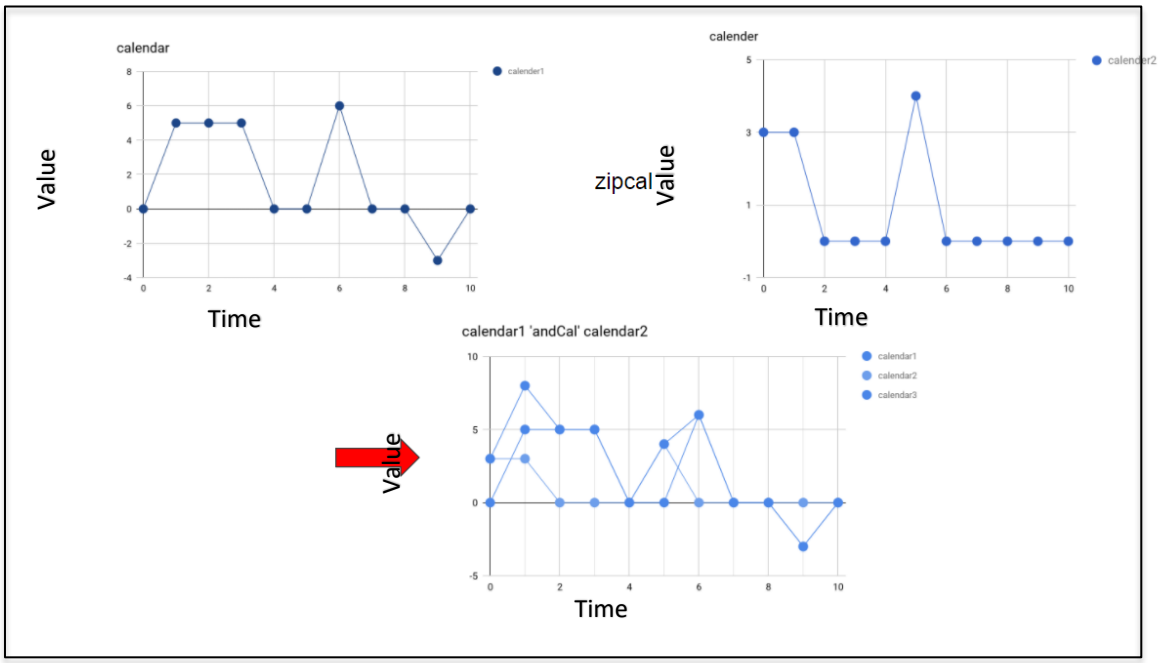

Figure 17: zipCal Combinator

```
shift :: Calender -> Time -> Calender
```

This combinator is more useful when combining two calendars in different time zones. ***shift*** function will shift all its event positions from a given Time value. Figure 18 represents

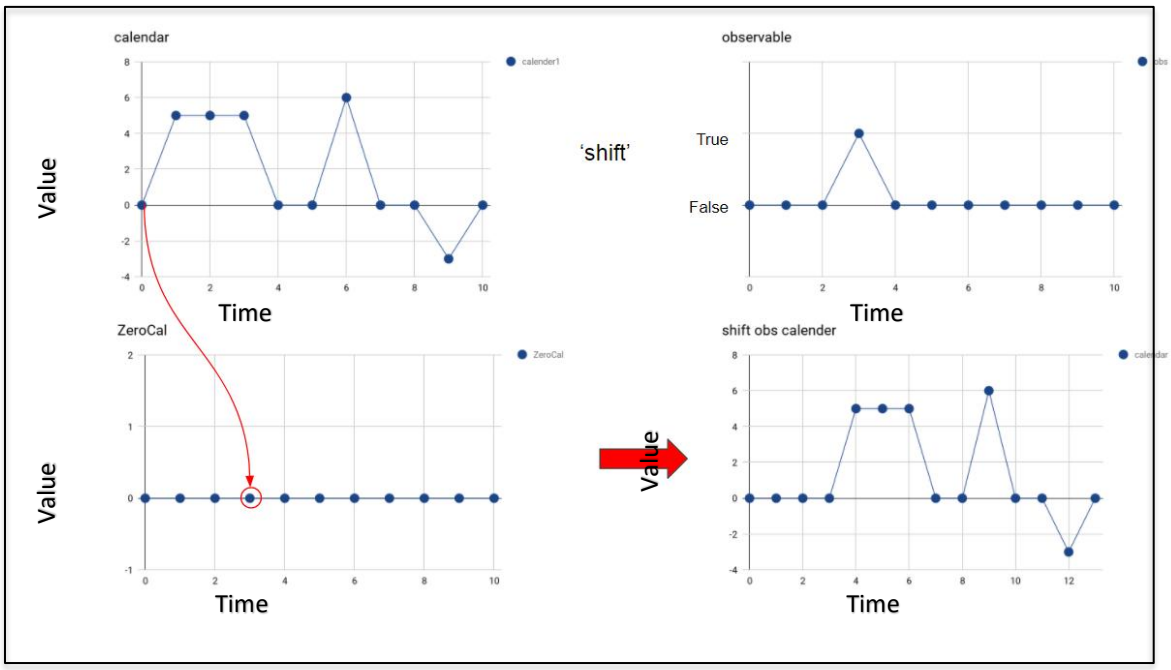

Figure 18: shift Combinator

3.2 Monte Carlo simulation for option pricing

3.2.1 Representing Stochastic Processes

We have created a simple model to represent stochastic processes for Monte Carlo simulation. This model uses a new data type to hold the normal distribution representation. This data type consists of the mean and the standard deviation of a normal distribution.

```
data Dist = Dist mean std
```

And value process is represented as an observable of distributions. Which means that for each discrete time, it returns a distribution of values. This representation of stochastic processes and distributions can get changed in different implementations and different contexts.

3.2.2 Interest Rate Model

We have modeled Interest rate as an observable of normal distributions. For each time step, it returns a distribution that represents how the interest rate can get varied. To build the interest rate model following information needs to be given,

- Interest Rate on the considering date.
- Maximum percentage of the interest rate that can go up or down.
- Number of paths considering for to generate distributions.

3.2.3. Simulating Contract Value

For Monte Carlo Simulation random paths for the contract values, need be generated. Then the simulator returns a value distribution for a given discrete time point. In Peyton Jones' contract descriptive language, they have used a function called ***disc*** to get the discounted value for a particular contract. So this proposed simulator can be directly plugged into this ***disc*** function.

One of the major components in Monte Carlo simulation is the random path generator. In this research, we have designed a simple random path generator which is implemented using language Haskell. Functional Programming languages such as Haskell makes it harder to create random number generators because functions in pure functional languages are immutable in nature. A thorough discussion about the implementation of this random number generator will be done in chapter 4. Random path generator creates random paths for the interest rate.

This random path generator first creates random walks for the interest rate. Figure 19 represents such paths that generate by the simulator. After generating paths for the interest rate, those paths are used to discount the value and generate value paths. This process is called contract value simulation. In our design, paths are infinitely long. And 'number of paths' is a parameter used in the Interest rate model.

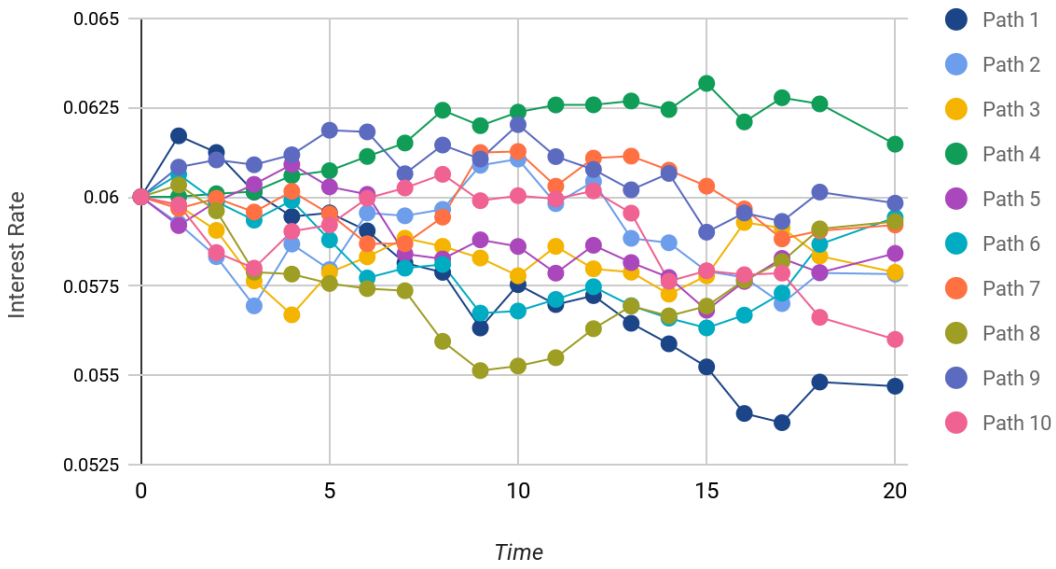

Figure 19: Interest Rate Paths Simulation

Parameters are passed through a model. ***interestWalk*** function create an array of random walks for a given starting interest rate value.

```
interestWalk :: Model -> Float -> Time  -> [[Float]]
```

This ***Model*** parameter can be found in the evaluation semantics proposed by Peyton Jones' language. The same ***Model*** parameter is used for this implementation as well. In the proposed evaluation semantics, function definition has the same set of parameters.

3.2.4 Contract Evaluation Semantics

Importance of this proposed model is that it uses the same evaluation semantics for contracts, proposed by the Peyton Jones' contract descriptive language. Instead of returning a value process this model returns an Observable of value distributions. Figure 6 contains the evaluation semantics for contracts used in contract descriptive language. Following function definition, shows the newly proposed evaluation semantics. The only difference is the return type. This is because proposed semantics supports valuating stochastic processes.

```
evalTermsAtP :: Model -> Time -> Terms -> Obs Dist
```

Proposed evaluation semantics use the same set of combinators as the original model. But instead of returning a simplified contract, it returns a stochastic process. To build up stochastic processes we have introduced a new set of combinators. The only difference between these set of combinators and initial combinators are that instead of combining contracts these combinators combine stochastic processes.

3.2.5 Combinators for Stochastic Processes

To combine stochastic processes we have modified existing set of combinators as follows,

*3.2.5.1 **zeroP***

zeroP is a stochastic process which needs to be equal to ***zero*** combinator. ***zero*** combinator implies no obligation or zero value at any given time. When converting this into a stochastic process, because it is a known variance become zero and the mean is also equal to zero.

```
zeroP :: Obs Dist
```

*3.2.5.2 **oneP***

This combinator is the equivalent stochastic process for the ***one*** combinator. ***one*** combinator is a contract with one value at the acquisition date of the contract. This value has no ambiguity. Because of this, the variance of ***oneP*** is zero and the mean is equal to the contract value.

```
oneP :: Amount -> Obs Dist
```

*3.2.5.3 **giveP***

This is the equivalent stochastic process combinator for ***give***. ***give*** combinator implies a cash outflow. ***giveP*** represents a cash outflow by negating the mean value.

3.2.5.d ***scaleP***

scaleP combinator act as the ***scale*** combinator in Peyton Jones' language. This combinator scales the value process by a given scaling factor.

```
scaleP ::Obs Decimal -> Obs Dist -> Obs Dist
```

3.2.5.e ***andP***

andP combinator applies ***and*** operator for two stochastic processes. When combining two processes together, the new process equals to an observable with a mean of the addition of two distribution means and variance is the addition of two variances.

```
andP :: Obs Dist -> Obs Dist -> Obs Dist
```

3.2.5.f ***orP***

orP combinator acts as the ***or*** combinator in Peyton Jones' contract descriptive Language. When parsing for valuation, ***orP*** combinator selects the stochastic process which as the highest mean value. This can be changed according to the implementation. The model says that, choose the most profitable contract according to their distribution values.

3.2.4 Monte Carlo simulation

Simulation is needed when discounting the value of a contract. In Peyton Jones' contract descriptive language, ***disc*** function acts as a discounting function for contracts. In the proposed model, this disc function is modified so that it does a Monte Carlo simulation for value and return the distribution corresponding to a particular time step. In evaluation

semantics, ***when*** combinator converted into a value process by using this ***disc*** function. In the proposed model, ***when*** is parsed into an observable of distributions instead of a value process.

Chapter 4 - Implementation

This chapter illustrates an implementation of the Peyton Jones' contract descriptive language and models proposed in Chapter 4. All the implementations are done using the language Haskell. This chapter also provides an introduction to Haskell and its lazy evaluation principles. Codes are provided to illustrate important implementation decisions.

4.1 Why Haskell?

Peyton Jones et al. [1] has used Haskell when implementing the contract descriptive language. This research also used Haskell as the primary language for implementations. Following are the main reasons to use Haskell,

- Haskell is a declarative language
- It supports lazy evaluation
- Built-in Characteristics

Haskell has many important features that can be used when building a domain specific language [8]. For an example, in the implementation, lazy evaluations in Haskell made recursive computations efficient.

4.1.1 Lazy evaluation

We used lazy evaluation and its behavior in our implementations. It is used to implement infinite lists and large recursions. Let's understand the behavior of lazy evaluation. Consider the following example,

```
magic :: Int -> Int -> [Int]
magic 0 _ = []
magic m n = m : (magic n (m+n))
```

We can see that `(magic 1 1)` returns the Fibonacci number series as an infinite list. This is possible because of the lazy evaluation. This expression won't evaluate unless otherwise a value is requested.

If we asked for the value at index 2, `((magic 1 1) !! 2)` then it will evaluate the list until 2nd index and returns the value at index 2.

4.2 Peyton Jones' Contract Descriptive Language

There are many implementations of Peyton Jones' Contract Descriptive Language in many different languages. We have implemented the basic functionalities of this language using a Haskell implementation. This implementation consists of following components,

- Contract definition and implementation
- Time, Period, Random value and Observable implementation
- Currencies and amounts implementation
- Contracts combinators
- Observables combinators
- Contract types
- Contract evaluation semantics
- Contract valuation model

4.2.1 Contract

For our research, we have used an implementation of contracts which have a name parameter. Because of this managing, the contract becomes much easier. In this implementation contracts are implemented as follows,

```
data Contract = Contract Name Terms deriving Show

name :: Contract -> Name
name (Contract n t) = n

terms :: Contract -> Terms
terms (Contract n t) = t
```

```
type Name = String
data Terms =
   Zero
   | One  Amount
   | Give Terms
   | And  Terms Terms
   | Or   Terms Terms
   | Cond   (Obs Bool)   Terms Terms
   | Scale   (Obs Int)   Terms
   | When   (Obs Bool)   Terms
   deriving Show
```

Contract is defined as a data type and it has two main components. Which are ***Name*** and ***Terms***. ***Name*** is a string that use to identify a particular contract. And ***Terms*** are defined recursively.

4.2.2 Time

```
data PeriodName = Month | Months
type Time   = Integer
```

For this implementation, we measured time as discrete months. And it can be changed according to the requirement. ***Time*** is represented by an integer and it identifies a particular month.

4.2.3 Currencies and amounts

```
data Currency = AUD | NZD | USD deriving (Eq, Show)
data Amount = Amt Decimal Currency
```

In this implementation, there are three currency types. They are ***Australian Dollars***, ***New Zeeland Dollars, and American Dollars***. An Amount is represented by a decimal value and a currency. As an example, 40 US Dollar is represented as 40 **USD.**

4.2.4 Contract combinators

In this implementation, there are 8 combinators for contracts. Contract is defined using these 8 combinators. ***zero*** and ***one*** are the most primitive type of contracts according to this language. They are implemented as follows.

```
zero :: Terms
one :: Amount -> Terms
scale :: Obs Int -> Terms -> Terms
give :: Terms -> Terms
and :: Terms -> Terms -> Terms
or :: Terms -> Terms -> Terms
cond :: Obs Bool -> Terms -> Terms -> Terms
when :: Obs Bool -> Terms -> Terms
```

4.2.5 Observables and combinators for observables

```
newtype Obs a = Obs (Time -> a)
```

As explained in the language, observable is implemented as a function of time to a random variable. And for those observables set of combinators are defined. Those are,

```
konst :: a -> Obs a
konst k = Obs (\t -> k)

at :: Time -> Obs Bool
at t = Obs (\time -> (time == t))

lift2 :: (a -> b -> c) -> Obs a -> Obs b -> Obs c
lift2 f (Obs o1) (Obs o2) = Obs (\t -> f (o1 t) (o2 t))

date :: Obs Time
date = Obs (\t -> t::Time)
```

konst combinator creates an observable which returns a constant value at each time value. And ***at*** combinator creates a Boolean observable that returns that get true at a given time period.

4.3 Calendar evaluation model implementation

Calendar evaluation model is implemented based on the implementation of contract descriptive language specified in chapter 4.2. Two new data types are implemented as follows,

```
type Event = [Int]
type Calender = Obs Event
```

In this implementation, ***Event*** is defined as an array of integers. One integer represents a cash inflow or cash outflow. This can be changed according to the requirement. ***Calendar*** is an observable of Events. We have used the same implementation for Observables as it is in the Peyton Jones' language.

4.3.1 Combinators for Calendar

In the implementation, we have implemented 7 new combinators for calenders. As mentioned in chapter 4, we have 6 main combinators which are ***zeroCal, oneCal, scaleCal, zipCal, shift*** and ***giveCal***. When comes to implementation level, ***zipCal*** need to be implemented separately for ***and and or*** combinators.

```
zeroCal :: Calender
zeroCal = konst [0]

oneCal :: Amount -> Calender
oneCal k = konst [1]

scaleCal :: Obs Int -> Calender -> Calender
scaleCal o cal = lift2 mult o cal

zipCalOr :: Calender -> Calender -> Calender
zipCalOr cal1 cal2 = lift2 merge cal1 cal2

zipCalAnd :: Calender -> Calender -> Calender
zipCalAnd cal1 cal2 = lift2 add cal1 cal2

shift :: Calender -> Obs Bool -> Calender
shift cal (Obs o) = Obs (\time -> (if (o time) then (getValue cal time) else (getValue
zeroCal time)))

giveCal :: Calender -> Calender
giveCal cal = lift2 mult (konst (-1)) cal
```

When implementing those combinators, we have introduced following supporting functions.

4.3.1.1 *revobs*

revobs returns the complement of a Boolean observable. It takes a Boolean observable and for each time step it returns the opposite of the Boolean value.

```
revobs :: Obs Bool -> Obs Bool
revobs (Obs o) = Obs (\time -> (if (o time) then False else True))
```

4.3.1.2 *merge and rmdups*

merge function uses to merge two events together and return a single event. When merging duplicate values are removed. the ***rmdups*** function is used to remove duplicate values from the list.

```
rmdups :: (Ord a) => [a] -> [a]
rmdups = map head . L.group . L.sort

merge :: [Int] -> [Int] -> [Int]
merge xs     []     = rmdups (0:xs)
merge []     ys     = rmdups (0:ys)
merge (x:xs) (y:ys) = rmdups (x : y : merge xs ys)
```

4.3.1.3 *add*

add is used to combine two events and return a new event with transactions of both events and all combinations of transactions between those two events. As the **merge** function, duplicates are removed after combination.

```
add :: [Int] -> [Int] -> [Int]
add xs [] = rmdups (0:xs)
add [] ys = rmdups (0:ys)
```

```
add (x:xs) (y:ys) = rmdups ((x+y) : (merge (add (x:xs) ys)  (add xs (y:ys))))
```

*4.3.1.4 **mult***

This function is used to scale an event from a given value. And this function is used when implementing ***scaleCal*** and ***giveCal*** combinators.

```
mult :: Int -> [Int]-> [Int]
mult x ys = map (x *) ys
```

4.3.2 Contract evaluation semantics for Calendar

Proposed evaluation semantics parses a given contract and returns its relevant calendar. This evaluation semantics takes **Time** and **Terms** as parameters and returns a Calendar. Time is given because the returned calendar considers the given time as the starting point of the calendar.

```
evalCalenderAt :: Time -> Terms -> Calender
evalCalenderAt t = calender
  where
    calender Zero              = zeroCal
    calender (One k)           = oneCal k
    calender (Give c)          = giveCal (calender c)
    calender (o `Scale` c)     = scaleCal o (calender c)
    calender (c1 `And` c2)     = zipCalAnd (calender c1) (calender c2)
    calender (c1 `Or` c2)      = zipCalOr (calender c1) (calender c2)
    calender (Cond o c1 c2)    = zipCalAnd (shift (calender c1) o)
                                           (shift (calender c2) (revobs o))
    calender (When o c)        = shift (calender c) o
```

4.4 Valuate Stochastic Processes

We represented the stochastic process as an observable of distributions. Following Haskell code represents the implementation of ***Dist*** data type and other supporting functionalities.

```
data Dist = Dist Decimal Decimal

mean :: Dist -> Decimal
mean (Dist m sd) = m

std :: Dist -> Decimal
std (Dist m sd) = sd
```

4.4.1 Random Path Generation

One of the main challenges we faced was to create a random path generator. Problem with implementing a random path generator is that pure functional languages such as Haskell are inherently immutable. To overcome this problem we implemented random path generator as an inner function. ***interestWalk*** function create a single path for interest rate.

```
interestWalk :: Float -> [Float] -> [Float]
interestWalk = walk
  where walk currentIr (x:xs) = (currentIr * x) : (walk (currentIr * x) xs)
```

4.4.2 Implementing Interest Rate Paths

To generate several interest rate paths at ones we implemented the ***interestWalkAll*** function. ***interestWalk*** function is used to generate individual paths and they are stored in a multidimensional array.

```
interestWalkAll :: Float -> Int -> [[Float]]
interestWalkAll a = walkMore
  where
  walkMore 0 = []
  walkMore n = (walkMore (n-1)) ++ [(interestWalk a (map (/ 100)
                                                    (map unsafePerformIO x)))]
        where x = (getStdRandom (randomR (90, 110))) : x
```

Once this function called, generated values will remain same so the Interest rate process remains fixed. In this implementation, random numbers are taken from an infinite array of random numbers and the random walk is also an infinite array. Because of the lazy evaluation, these arrays won't evaluate until a particular value is requested. And generated values will remain fixed so that it won't recalculate them again.

4.4.3 Discounting Function

This function is used to discount a particular value. This function takes simulated interest rate paths as a parameter. ***discAll*** function returns a distribution of the present value of a future cash flow.

```
disc :: [[Float]] -> Float -> Int -> Int -> Float
disc intr p k = discin
  where
  discin 0 = p
  discin n = (discin (n-1))*(1 + (intr!!k!!(n-1)))

discAll :: [[Float]] -> Float -> Int -> Dist
discAll intr p n = Dist (discMean intr p n) (discVar intr p n)
```

4.4.4 Evaluation Semantics

Evaluation semantics for contract evaluation is the most important part of this implementation. This evaluation semantics converts a given contract in to a stochastic value process. ***evalP*** function recursively parse a given contract using bottom up evaluation.

```
evalTermsAtP :: Model -> Time -> Terms -> Obs Dist
evalTermsAtP m t = evalP
  where
    evalP Zero                = zeroP
    evalP (One amt)           = convertToP m t (mainCurrency m) $ One amt
    evalP (Give c)            = scaleP (Obs (\t -> (-1))) (evalP c)
    evalP (Zero `And` Zero)         = zeroP
    evalP (Zero `And` (One amt))    = oneP amt
    evalP ((One amt) `And` Zero)    = oneP amt
```

```
evalP (c1 `And` c2)                  = (evalP c1) `andP` (evalP c2)
evalP (c1 `Or`  c2)          = maxTP (getValue ((exchangeRate m) USD
                                        USD) t) t (evalP c1) (evalP c2)
evalP (Cond (Obs o) c1 c2)           = if (o t) then (evalP c1) else (evalP c2)
evalP (When (Obs o) c)               = if (o t) then (discAllP m t (evalP c))
                                                                  else zeroP
evalP (Scale (Obs s) (One (Amt amt cur))) = oneP $ Amt
                                                  (amt *. (realToFrac $ s t)) cur
```

4.4.5 Combinators for Stochastic Processes

To support stochastic process building, we implemented a set of combinators for stochastic processes. ***zeroP*** and ***oneP*** are the unit stochastic processes that use to evaluate zero contract and one contract. And other combinators are implemented to support the conversion of contracts to stochastic processes. Following is an implementation of those set of combinators.

```
zeroP :: Obs Dist
zeroP = (konst (Dist 0 0))

oneP :: Amount -> Obs Dist
oneP amt = konst (Dist (amountToDecimal amt) 0)

scaleP ::Obs Decimal -> Obs Dist -> Obs Dist
scaleP (Obs o) (Obs p) = Obs (\t -> (Dist ((mean (p t)) * (o t)) ((std (p t)) *
                                                                  (o t))))

andP :: Obs Dist -> Obs Dist -> Obs Dist
andP (Obs p1) (Obs p2) = Obs (\t -> (Dist ((mean (p1 t)) + (mean (p2 t)))
                                          ((std (p1 t)) + (std (p2 t))) ))
```

All these combinators return a stochastic process which is implemented as an observable of distribution.

Chapter 5 - Evaluation and Conclusion

In this chapter, example contract evaluations are done for models introduced in Chapter 4 as proof of concept. This chapter gives a conclusion for research questions and research objectives based on findings. Next, it describes some limitations in proposed models. Finally, this chapter concludes by introducing toward some future research and experiments possible.

5.1 Example Contract calendar Evaluation

Let's consider some example contracts from contract descriptive language and try to evaluate using proposed evaluation semantics. First, consider the following contract which is a simple zero coupon bond. It represents a cash outflow of ***100 USD*** at ***4th month***.

Representation for this contract using the contract descriptive language is as follows,

C1 = When (at $ 4 **Months**) (**Scale** (konst 100) $ **Give** $ **One USD**)

Figure 20 elaborate the conversion of this contract into calendar representation using calendar evaluation semantics. At the end of the parsing, evaluation semantics returns the following representation of a calendar,

shift (at 4) scaleCal (**Konst** 100) (giveCal calOne)

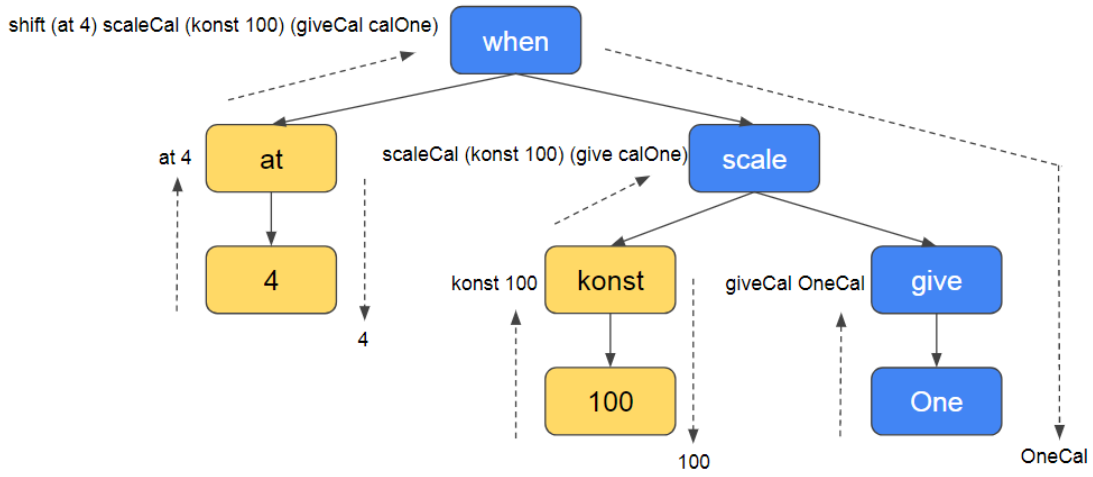

Figure 20: Contract Parsing for Calendar

This representation consists of Calender combinators and observables. Proposed evaluation semantics will evaluate this expression recursively. From the bottom of the recursive tree, it first evaluates ***One USD***. And the result would be a Calender which returns an event with value 1 at time 0 and an event with value 0 at every other time. then it will evaluate ***Give $ One USD*** Contract. It will return the negation of all events in the previous calendar. And then all values in all events will be scaled by 100. Finally, will get shifted to start from the third month of the relative timeline.

The next example is an American option. In this example, contract holder has the right to receive ***10 USD*** within ***2nd*** and the ***4th months***.

C2 = american (1 **Month**, 3 **Months**) (**Scale** (konst 10) $ **One USD**)

This contract will translate into simple combinators as follows,

When (**Obs Bool**) (**Scale** (konst 10) $ **One USD**)

After that ***(Scale (konst 10) $ One USD)*** will evaluate as it in the previous example. And ***when*** combinator will check whether the observable is true or not. If it is true then shifted calendar of ***(Scale (konst 10) $ One USD)*** will be returned. Otherwise, ***zeroCal*** will be returned. So the evaluated calendar will be,

The next example is a combination of previous two contracts.

```
C3 = And c2 c1
C3 = And (american (1 Month, 3 Months) (Scale (konst 10) $ One USD))
         (when (at $ 3 Months) (Scale (konst 100) $ Give $ One USD))
```

For this example, ***and*** combinator will combine calendars of both ***C1*** and ***C2*** with ***calZip*** combinator. And each result of those two calenders will get zip into one event.

5.1.1 Calendar Generation Model Evaluation

We gave this model to industry experts for the evaluation. According to their opinion, this model consists of important features that may support to manage contracts in the industry. They mainly recommended this model for contracts in actuarial services because contracts in that domain have long lifecycles.

5.2 Example Contract Monte Carlo Simulation

Consider the ***C1*** contract introduced in the previous section.

```
C1 = When (at $ 4 Months) (Scale (konst 100) $ Give $ One USD)
```

Let's convert this contract into stochastic a process using our Evaluation semantics. First ***One USD*** will be converted into a distribution with mean 1 and standard deviation 0. This is because the value is known. Then parsing the scale combinator scales the distribution with the given observable. After scaling it returns a distribution with mean 100 and standard deviation equal to 0.

To valuate, we need to have an interest rate model and future interest rate simulation. In our implementation, ***interestWalkAll*** function can be used.

```
m = interestWalkAll 0.01 50
```

M consists of 50 random paths for the interest rate assuming the current interest rate of 0.01. We can use this interest rate model to discount the previous contract value.

```
v = discAll m 100 4
```

discAll function returns the current value of a future cash flow. In this scenario, it returns the present value of ***C1***. This value is returned as a distribution of values. Output for this given example is a distribution with mean equals to 101.12 and the standard deviation is equals to 0.22.

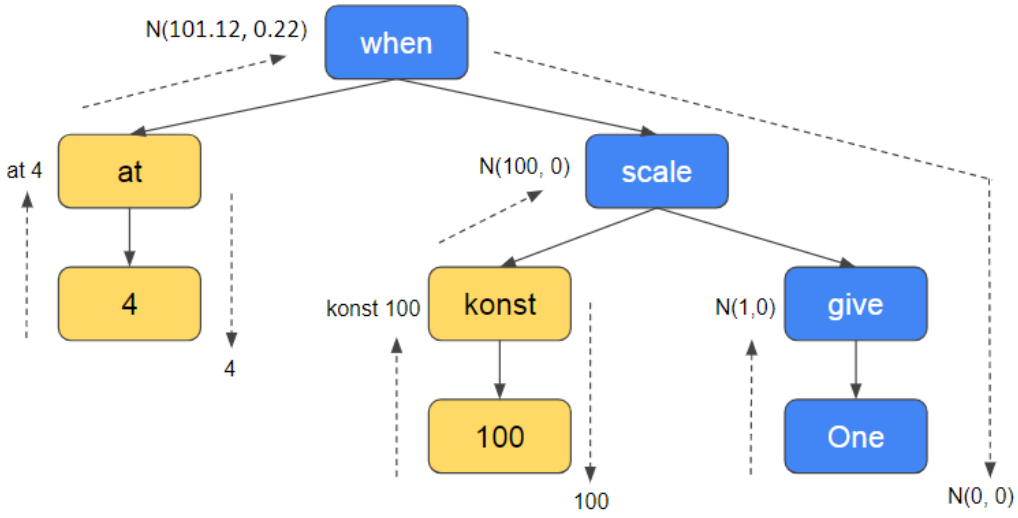

Figure 21: Contract Parsing for a Stochastic Process.

Figure 21 represents how parsing is done for a contract using the evaluation semantics. Monte Carlo simulation is done when parsing the ***when*** combinator.

5.3 Conclusion

Peyton Jones' Contract Descriptive Language is widely used in the financial sector. Our aim was to evaluate different value processes that can be applied to Peyton Jones' Contract Descriptive Language. First, we looked into the possibility of constructing a model to generate a calendar for a given contract represented in this language. The motivation for exploring this value process came from a keynote speech delivered by Jean-Marc Eber at the Domain-Specific Languages for Financial Systems (DSLFIN) 2013 workshop [10].

We gradually built up several models until it aligned with our calendar definitions. A set of evaluation semantics, that can be used to translate a given contract into its relevant calendar was introduced. To represent and build calendars we came up with a set of new combinators. We validated each of these combinators by aligning them with our calendar definitions. According to the industry experts, this model is suitable to generate a calendar for a contract, and they recommended this model mainly for actuarial services. Our final conclusion of the first part of the research is that it is possible to build a model to generate a calendar for a contract written in Peyton Jones' Contract Descriptive Language. And such model can be implemented and tested. And we concluded that such model is highly beneficial and have a high value in the industry.

As the second objective of this research, we created a model to simulate Monte Carlo valuation for a contract written in Peyton Jones' Contract Descriptive Language. To achieve this objective, we had to create a representation for stochastic processes. We represented a stochastic value process as an observable of normal distributions. In the process, we modified a set of combinators so that, they would support to combine stochastic processes. We designed an interest rate model and introduced a function to generate Monte Carlo simulation for future interest rate values. We modified the discounting function in Peyton Jones' Language, so that it uses the simulated interest rate paths. Without doing major changes to the existing model proposed by Peyton Jones et al. [13], we were able to create a model to do Monte Carlo simulation for a contract written in Peyton Jones' Language.

As the conclusion, this implies the power of Peyton Jones' Contract Descriptive Language in valuating contracts. Our findings further prove that Monte Carlo simulation can be used for valuation, instead of using lattice valuation method, without doing major changes to the existing language model.

5.4 Limitations

Proposed models only rely on the information provided by the contract representation of Peyton Jones' contract descriptive language. Because of this calendar model only yield timeline of possible cash flows for a given contract.

In the evaluation semantics which was introduced for stochastic processes will not work on American options. This is because of the uncertainty of their acquisition date. Industry experts are also recommended not follow the path of valuating American options with limited time constrains.

5.5 Future Work

In this research, we introduced several models for computing processing activities in Peyton Jones' contract descriptive language. Other than the tested activities there are many processing activities that can be evaluated for this language. Risk calculation is one such activity. Generating a model to calculate the risk would be highly beneficial.

Calendar model can be further improved by modifying the contract definition. Other than having cash flows, the contract can have details like acquisition date, quantity, long term or short term etc. These kinds of representations would benefit when it comes to the contract management. And in the calendar, these details can be used to represent detailed event plan rather than showing cash flows.

The stochastic process valuation can be done using a more complex Monte Carlo simulation method. In our model, we used a simple method that outlines the basic functionalities of Monte Carlo simulation. The proposed model is suitable for valuating American options by doing slight changes to the model. But in this research, we haven't tested for American options. It is an important future work to be done.

References

[1] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen, “Compositional specification of commercial contracts,” *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 6, pp. 485–516, 2006.

[2] RL McDonald; M Cassano; R Fahlenbrach, *Derivatives markets*, Vol. 2. Boston: Addison-Wesley, 2006.

[3] M. H. A. Davis, V. G. Panas, and T. Zariphopoulou, “European option pricing with transaction costs,” *SIAM J. Control Optim.*, vol. 31, no. 2, pp. 470–493, 1993.

[4] P. Boyle, M. Broadie, and P. Glasserman, “Monte Carlo methods for security pricing,” *J. Econ. Dyn. Control*, vol. 21, no. 8–9, pp. 1267–1321, 1997.

[5] J. Vecer, “A new PDE approach for pricing arithmetic average Asian options,” *J. Comput. Financ.*, vol. 4, pp. 1–9, 2001.

[6] P. Hudak, “Domain specific languages,” *Handb. Program. Lang. Vol. III Little Lang. Tools*, vol. III, pp. 39–60, 1998.

[7] S. P. Jones, J.-M. Eber, and J. Seward, “Composing Contracts : An Adventure in Financial Engineering,” *ACM SIGPLAN Not.*, 2000.

[8] K. Czarnecki, J. O’Donnell, J. Striegnitz, and W. Taha, “DSL Implementation in MetaOCaml, Template Haskell, and C++,” *Domain-Specific Progr. Gener.*, vol. 3016, pp. 51–72, 2004.

[9] J.-Q. Hu and M. C. Fu, “Sensitivity Analysis for Monte Carlo Simulation of Option Pricing,” *Probab. Eng. Informational Sci.*, vol. 9, no. 3, pp. 417–446, 1995.

[10] J.-M. Eber, “Beyond Valuation: Past, Present and Future of Domain Specific Languages for Finance Applications: Ten Years of DSL Development, Client Interaction, and Market Feedback at LexiFi,” 2014.

[11] B. Arnold, A. Van Deursen, and M. Res, “An algebraic specification of a language for describing financial products,” *ICSE-17 Workshop on Formal Methods Application in Software Engineering*. pp. 6–13, 1995.

[12] P. Bahr, J. Berthold, and M. Elsman, “Certified symbolic management of financial multi-party contracts,” *Proc. 20th ACM SIGPLAN Int. Conf. Funct. Program. - ICFP 2015*, vol.

92299, no. 10, pp. 315–327, 2015.

[13] S. L. Peyton Jones and J.-M. Eber, “How to Write a Financial Contract,” *The Fun of Programming*. 2003.

[14] A. Mediratta, “A generic domain specific language for financial contracts,” p. 46, 2007.

[15] D. Christiansen, K. Grue, and H. Niss, “An Actuarial Programming Language for Life Insurance and Pensions,” pp. 1–25, 2013.

[16] J. Ahnfelt-rønne and M. F. Werk, “Pricing composable contracts on the GP-GPU,” 2011.

[17] J.-M. Eber, “Describing, Manipulating and Pricing Financial Contracts: The MLFi Language,” vol. 2003, no. January, 2005.

[18] M. F. Werk, J. Ahnfelt–Rønne, and K. F. Larsen, “An Embedded DSL for Stochastic Processes,” *Proc. 1st ACM SIGPLAN Work. Funct. high-performance Comput.*, no. June 2007, pp. 93–101, 2012.

[19] M. Erwig and S. Kollmansberger, “FUNCTIONAL PEARLS: Probabilistic functional programming in Haskell,” *J. Funct. Program.*, vol. 16, no. 1, p. 21, 2005.

[20] L. C. G. Rogers, “Monte Carlo valuation of American options,” *Math. Financ.*, vol. 12, no. 3, pp. 271–286, 2002.

[21] R. E. Caflisch, “Chapter 1 MONTE CARLO SIMULATION FOR AMERICAN OPTIONS,” 2003.

Appendix A

Code: Language

```
module DSL where
-------------------------------------------------------------------------------
import Data.Decimal
import qualified Data.List as L
-------------------------------------------------------------------------------
import System.IO.Unsafe  -- be careful!
import System.Random

      -- Contract

-------------------------------------------------------------------------------

data Contract = Contract Name Terms deriving Show

name :: Contract -> Name
name (Contract n t) = n

terms :: Contract -> Terms
terms (Contract n t) = t

type Name = String
data Terms =
   Zero
   | One  Amount
   | Give Terms
   | And  Terms Terms
   | Or   Terms Terms
   | Cond   (Obs Bool)   Terms Terms
   | Scale  (Obs Int)    Terms
   | When   (Obs Bool)   Terms
   deriving Show
-------------------------------------------------------------------------------


      -- Time, Period, Random value, Observable

-------------------------------------------------------------------------------
data PeriodName = Month | Months

type Time   = Integer
type Period = Integer

instance Num (PeriodName -> Time) where
   fromInteger t Month = t::Time
   fromInteger t Months = t

newtype Obs a = Obs (Time -> a)

getValue :: Obs a -> Time -> a
getValue (Obs x) time = x time

instance Show a => Show (Obs a) where
   show (Obs obs) = "(Obs " ++ show (obs 0) ++ ")"
```

```
konst :: a -> Obs a
konst k = Obs (\t -> k)

at :: Time -> Obs Bool
at t = Obs (\time -> (time == t))

lift2 :: (a -> b -> c) -> Obs a -> Obs b -> Obs c
lift2 f (Obs o1) (Obs o2) = Obs (\t -> f (o1 t) (o2 t))

date :: Obs Time
date = Obs (\t -> t::Time)



-- Compare observables
(%<), (%<=), (%==), (%>=), (%>) :: Ord a => Obs a -> Obs a -> Obs Bool
(%<)  = lift2 (<)
(%>)  = lift2 (>)
(%==) = lift2 (==)
(%>=) = lift2 (>=)
(%<=) = lift2 (<=)

type Term          = [Time]
type PaymentSchedule = [Amount]
--------------------------------------------------------------------------------


    -- Currencies and amounts

--------------------------------------------------------------------------------
data Currency = AUD | NZD | USD deriving (Eq, Show)
data Amount = Amt Decimal Currency

instance Show Amount where
  show (Amt amt currency) = show amt ++ show currency

instance Num (Currency -> Amount) where
  fromInteger amt c = Amt (Decimal 0 amt) c

instance Num Amount where
  (-) (Amt a1 c1) (Amt a2 c2)
    | (c1 == c2) = Amt (a1-a2) c1
  (+) (Amt a1 c1) (Amt a2 c2)
    | (c1 == c2) = Amt (a1+a2) c1

instance Eq Amount where
  (==) (Amt a1 c1) (Amt a2 c2) = (a1==a2) && (c1==c2)

instance Ord Amount where
  compare (Amt a1 c1) (Amt a2 c2)
    | (c1 == c2) = compare a1 a2

instance Eq Terms where
  (==) (One a1) (One a2) = (a1 == a2)

instance Ord Terms where
  compare (One a1) (One a2) = compare a1 a2
  compare (One a1) Zero = compare a1 0
  compare Zero (One a2) = compare 0 a2
  compare Zero Zero = EQ

instance Eq Contract where
  (==) (Contract n1 t1) (Contract n2 t2) = (t1 == t2)

instance Ord Contract where
  compare (Contract n1 t1) (Contract n2 t2) = compare t1 t2
```

```
amountToDecimal (Amt a c) = a
-------------------------------------------------------------------------------


    -- Operations on Terms

-------------------------------------------------------------------------------
zero :: Terms
zero = Zero

one :: Amount -> Terms
one = One

scale :: Obs Int -> Terms -> Terms
scale = Scale

give :: Terms -> Terms
give = Give

and :: Terms -> Terms -> Terms
and = And

or :: Terms -> Terms -> Terms
or = Or

cond :: Obs Bool -> Terms -> Terms -> Terms
cond = Cond

when :: Obs Bool -> Terms -> Terms
when = When
-------------------------------------------------------------------------------
```

Code: Calendar Model

```
revobs :: Obs Bool -> Obs Bool
revobs (Obs o) = Obs (\time -> (if (o time) then False else True))

rmdups :: (Ord a) => [a] -> [a]
rmdups = map head . L.group . L.sort

merge :: [Int] -> [Int] -> [Int]
merge xs     []     = rmdups (0:xs)
merge []     ys     = rmdups (0:ys)
merge (x:xs) (y:ys) = rmdups (x : y : merge xs ys)

add :: [Int] -> [Int] -> [Int]
add xs [] = rmdups (0:xs)
add [] ys = rmdups (0:ys)
add (x:xs) (y:ys) = rmdups ((x+y) : (merge (add (x:xs) ys)  (add xs (y:ys))))

mult :: Int -> [Int]-> [Int]
mult x ys = map (x *) ys

--data types

type Event = [Int]
type Calender = Obs Event

--eval

zeroCal :: Calender
```

```
zeroCal = konst [0]

oneCal :: Amount -> Calender
oneCal k = konst [1]

scaleCal :: Obs Int -> Calender -> Calender
scaleCal o cal = lift2 mult o cal

zipCalOr :: Calender -> Calender -> Calender
zipCalOr cal1 cal2 = lift2 merge cal1 cal2

zipCalAnd :: Calender -> Calender -> Calender
zipCalAnd cal1 cal2 = lift2 add cal1 cal2

shift :: Calender -> Obs Bool -> Calender
shift cal (Obs o) = Obs (\time -> (if (o time) then (getValue cal time) else (getValue zeroCal time)))

giveCal :: Calender -> Calender
giveCal cal = lift2 mult (konst (-1)) cal

--contract eval calender

evalCalenderAt :: Time -> Terms -> Calender
evalCalenderAt t = calender
  where
    calender Zero              = zeroCal
    calender (One k)           = oneCal k
    calender (Give c)          = giveCal (calender c)
    calender (o `Scale` c)     = scaleCal o (calender c)
    calender (c1 `And` c2)     = zipCalAnd (calender c1) (calender c2)
    calender (c1 `Or` c2)      = zipCalOr (calender c1) (calender c2)
    calender (Cond o c1 c2)    = zipCalAnd (shift (calender c1) o) (shift (calender c2) (revobs o))
    calender (When o c)        = shift (calender c) o
```

Code: Monte Carlo Simulation

```
data Dist = Dist Decimal Decimal

mean :: Dist -> Decimal
mean (Dist m sd) = m

std :: Dist -> Decimal
std (Dist m sd) = sd

zeroP :: Obs Dist
zeroP = (konst (Dist 0 0))

oneP :: Amount -> Obs Dist
oneP amt = konst (Dist (amountToDecimal amt) 0)

convertToP :: Model -> Time -> Currency ->  Terms -> Obs Dist
convertToP m t c0 Zero = zeroP
convertToP m t c0 (One (Amt amt c))
  | (c0 == c)  = oneP $ Amt amt c0 -- No need to convert anything
  | otherwise  = oneP $ Amt (amt *. currentER) c0
  where
    -- Exchange Rates
    observableER = (exchangeRate m) c c0
    currentER    = realToFrac $ getValue observableER t

scaleP ::Obs Decimal -> Obs Dist -> Obs Dist
scaleP (Obs o) (Obs p) = Obs (\t -> (Dist ((mean (p t)) * (o t)) ((std (p t)) * (o t))))
```

```
andP :: Obs Dist -> Obs Dist -> Obs Dist
andP (Obs p1) (Obs p2) = Obs (\t -> (Dist ((mean (p1 t)) + (mean (p2 t))) ((std (p1 t)) + (std (p2 t))) ))

maxTP :: ExchangeRate -> Time -> Obs Dist -> Obs Dist -> Obs Dist
maxTP exchR t (Obs p1) (Obs p2) = if ((mean (p2 t)) > (mean (p1 t))) then (Obs p2) else (Obs p1)

evalTermsAtP :: Model -> Time -> Terms -> Obs Dist
evalTermsAtP m t = evalP
  where
    evalP Zero                = zeroP
    evalP (One amt)             = convertToP m t (mainCurrency m) $ One amt
    evalP (Give c)              = scaleP (Obs (\t -> (-1))) (evalP c)
    evalP (Zero `And` Zero)       = zeroP
    evalP (Zero `And` (One amt))   = oneP amt
    evalP ((One amt) `And` Zero)   = oneP amt
    evalP (c1 `And` c2)           = (evalP c1) `andP` (evalP c2)
    evalP (c1 `Or` c2)            = maxTP (getValue ((exchangeRate m) USD USD) t) t (evalP c1) (evalP c2)
    evalP (Cond (Obs o) c1 c2)    = if (o t) then (evalP c1) else (evalP c2)
--      evalP (When (Obs o) c)        = if (o t) then (discAll m t (evalP c)) else zeroP
    evalP (Scale (Obs s) (One (Amt amt cur))) = oneP $ Amt (amt *. (realToFrac $ s t)) cur

interestWalk :: Float -> [Float] -> [Float]
interestWalk = walk
  where walk currentIr (x:xs) = (currentIr * x) : (walk (currentIr * x) xs)

interestWalkAll :: Float -> Int -> [[Float]]
interestWalkAll a = walkMore
  where
  walkMore 0 = []
  walkMore n = (walkMore (n-1)) ++ [(interestWalk a (map (/ 100) (map unsafePerformIO x)))]
                                  where x = (getStdRandom (randomR (90, 110))) : x

disc :: [[Float]] -> Float -> Int -> Int -> Float
disc intr p k = discin
  where
  discin 0 = p
  discin n = (discin (n-1))*(1 + (intr!!k!!(n-1)))

discSum :: [[Float]] -> Float -> Int -> Int -> Float
discSum intr p 0 n = disc intr p 0 n
discSum intr p k n = (discSum intr p (k-1) n) + (disc intr p k n)

discMean :: [[Float]] -> Float -> Int -> Float
discMean intr p n = (discSum intr p ((length intr)-1) n)/(fromIntegral (length intr))

discVarSum :: [[Float]] -> Float -> Int -> Int -> Float
discVarSum intr p 0 n = 0
discVarSum intr p k n = (((disc intr p k n) - dmean) * ((disc intr p k n) - dmean)) + (discVarSum intr p (k-1) n)
  where dmean = discMean intr p n

discVar :: [[Float]] -> Float -> Int -> Float
discVar intr p n = (discVarSum intr p ((length intr)-1) n) / (fromIntegral ((length intr)-1))

discAll :: [[Float]] -> Float -> Int -> Dist
discAll intr p n = Dist (discMean intr p n) (discVar intr p n)
```