A GENERIC DOMAIN SPECIFIC LANGUAGE FOR FINANCIAL CONTRACTS

BY ANUPAM MEDIRATTA

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Assistant Professor Chung-chieh Shan

and approved by

_____

_____

_____

New Brunswick, New Jersey

October, 2007

ABSTRACT OF THE THESIS

A generic domain specific language for financial contracts

by Anupam Mediratta

Thesis Director: Assistant Professor Chung-chieh Shan

Financial contracts require management, such as valuation, scheduling and generating legal documents. The current approach for managing financial contracts is inefficient, for lack of a universal language for representing contracts. Peyton Jones et al.'s proposed such a language, even though contracts are diverse and new ones are introduced often. We verify that Peyton Jones et al.'s language is expressive enough by using it to represent two new contracts: credit default swap and power reverse dual currency swap. We demonstrate its advantage by using the same program to value all contracts represented. More generally, we need only one program for each contract management task.

Acknowledgements

I would like to thank my adviser Prof. Chung-chieh (Ken) Shan for guiding me throughout this work. The experience of interacting with him was very challenging. To catch up with his thought process, I was always required to stretch my intellectual abilities and in this process I did learn a bit (at least I think so). I am grateful to him for his guidance in approaching the work, presenting it and writing it.

I like to thank Prof Ren-Raw Chen from Business School at Rutgers also. He was kind enough to spend time with me and answer my naive questions about financial contracts. I would also like to thank a dear friend of mine, Mr Rahul Singh, who works with BNP Paribas bank in Singapore and was helpful enough to expose me to latest introduced contracts like PRDC.

I am grateful to Prof. Borgida and Prof. McCarty also, for agreeing to be in my thesis committee.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, I present a language to *represent* financial contracts. To represent a financial contract means to express its rights and obligations. This language was proposed by Peyton Jones et al. ([4]). I verify the *generic* nature of this language by representing two new contracts, which are not mentioned in their paper.[1]

The motivation of building such a language comes from the fact that the contracts traded in the market today are increasingly complex and diverse. The banks which deal with these contracts have to process them for various activities. The processing activities of a contract are valuation, generation of legal documents and other back-end activities. Among all these activities, valuation is the most important and critical one. This is because the banks want to correctly value a contract as soon as possible, so that they can make the most profitable trades. There exists a competition between *traders*, who trade contracts, and *quants*, who value contracts. Traders introduce new contracts so that they can profit because quants take time to value it correctly. The language can be seen as a tool to help quants to value a contract quickly.

Before building such a language, we have to overcome the following hurdles:

1. The variety of contracts is diverse, so it is challenging to come up with a precise language (or small set of *combinators*) that can represent (most of) them.[2]

2. The industry invents new contracts every now and then. For example, someone can compose existing contracts to create a new one. An example is the contract *swaption*, which is created as an *option* over *swaps*. Both of the latter contracts

[1] Generic in the sense that the language generalizes over a wide variety of contracts.

[2] I define combinators in chapter 4.

exist from before.

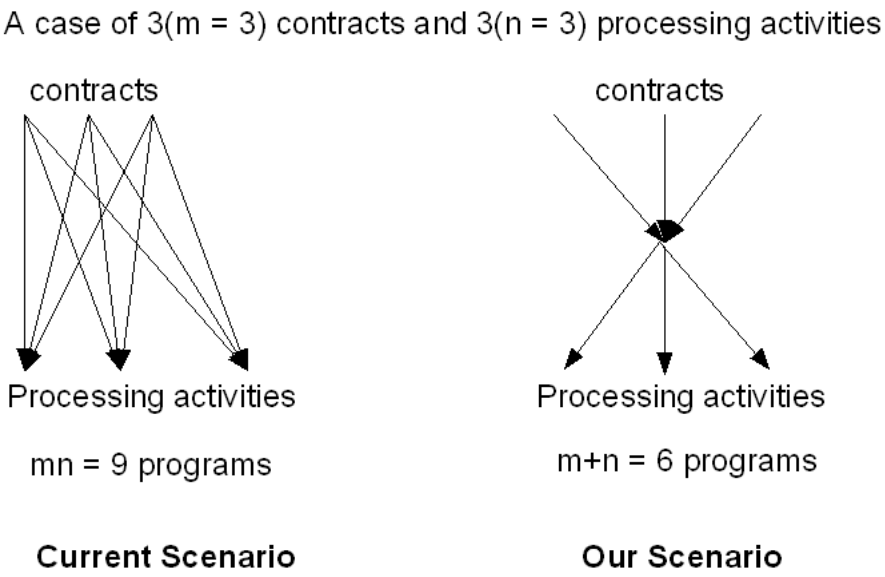

Figure 1.1: Motivation

The advantage of building such a language is shown in figure 1.1 through two scenarios. The job in both scenarios is to apply each of the n processing activities to each of the m contracts. In the current scenario, we have to write mn programs, one for each pair of contract and processing activity. If we use the language, which is our scenario in the figure, we will be able to do this job by writing only $m + n$ programs. The reason is we would write one program to represent a contract in the language. To represent m contracts we would write m programs. Once the contracts are represented in the language, to process all of them for one activity requires only one program. And similarly to process them for n activities would require n programs. Hence, in total we would write $m + n$ programs.

Since our work is an extension of Peyton Jones et al.'s paper, I summarize most of their work in chapter 4. For the rest of their work, I will credit them wherever I use it. They chose Haskell as a programming language to implement the contract language. I explain Haskell's basic syntax and the reasons for choosing it in chapter 3. I verify the generic nature of the contract language by representing two contracts: Credit Default

Swap (CDS) and Power Reverse Dual Currency Swap (PRDC). This is how I validate Peyton Jones et al.'s work. In chapter 5, I show how to represent these contracts in the language. To illustrate the advantage of the language, I explain in chapter 6 how a single program suffices to value any contract with an example of a CDS contract. Since Peyton Jones et al.'s software is publicly unavailable, our code is a good starting point for future research. I attach our code in Appendix B, C and D.

Since this thesis deals with financial terms and financial instruments, I introduce them in the next chapter.

Chapter 2

Finance 101

In this chapter, I explain the finance terms used commonly in this thesis. I begin by defining a financial contract. From here onwards whenever I write contract it means a financial contract. A contract usually involves three parties, the *buyer*, the *seller* and the *issuer*. The *buyer* is the one who buys the contract; the *seller* is the one who owns it and then sells it for its value; the *issuer* is the one who had issued or introduced the contract. Now it can be the case that the issuer and the seller are the same entities. A contract has a set of rights and obligations for both its buyer and the issuer. The rights of the buyer are the obligations of the issuer and vice versa. The seller has no rights and obligations once he has sold the contract. For instance, Goldman Sachs (GS) is the issuer of its bond. If Henry owns this bond, he is the buyer of the contract. So, for this contract, GS's rights are Henry's obligations and vice versa. If Henry chooses to sell this bond to Anna, she is the buyer and he is the seller. By paying the market value of the contract, she buys all its rights and obligations from him. Before I define a contract, I need to define a *cash flow*.

Definition 2.1 *A* cash flow *is a transfer of money specified by its amount and date.*[1]

An example of a cash flow is ($100, 12/31/2007). It means a transfer of $100 takes place on Dec. 31, 2007 between given two parties.

Definition 2.2 *A* financial contract *is a set of cash flows.*

[1] *More generally, there are conditional cash flows in which the transfer of money occurs only if some particular condition is true. I will show an instance of it, when I define CDS.*

This set of cash flows are the rights and obligations for the buyer.[2] For each cash flow of a contract, if the amount is positive, then the buyer of the contract receives that amount, else she has to pay it to the issuer.

A hypothetical contract is, receive \$100 at the end of year 2007 and pay \$110 at the end of year 2008. It can be written as: $[(\$100, 12/31/2007), (-\$110, 12/31/2008)]$.

Definition 2.3 *The* maturity *of a contract is the last date off all the dates of its cash flows.*[3]

The maturity of the above hypothetical contract is 12/31/2008. If the maturity is infinite then the corresponding contract never matures. Since there are no cash flows after the maturity of a contract, the contract offers no opportunity to make money after it matures. Hence, it is not worth trading then.

Definition 2.4 *A contract is* void *if it has no future cash flows associated with it.*

Another relevant term is *default*:

Definition 2.5 *The debtor* defaults *when he has not met the obligations of the debt contract.*

For instance, in the case of a bond (See Definition 2.6), if the bond issuer fails to make the payments to the buyer of the bond as per his obligations, then the bond issuer *defaults* on this criterion.

2.1 Financial Instruments

Now, I define some financial instruments (or contracts), which will be used in the following chapters. I begin with a bond whose definition is taken from http://en.wikipedia.org/wiki/Bond_%28finance%29:

---

[2]Recall that rights and obligations for the issuer are vice versa of those of the buyer.

[3]*Peyton Jones et al. in their paper use the term horizon instead of maturity.*

Definition 2.6 *A* bond *is a contract in which the issuer owes the buyer a debt and is obliged to repay the principal and interest (the coupon) at a later date, termed maturity.*

The main contract of this thesis, Credit Default Swap, is similar to an insurance policy. The latter contract works in the following way: the buyer of the policy is the protection buyer (buyer) and the insurance company is the protection seller (seller). Till there is no eventuality, the buyer pays to the seller at regular intervals of time, a fixed amount of money which is called premium or *spread*. Otherwise, the seller pays to the buyer a fixed amount which is called *nominal* and the contract becomes void. Now a CDS differs from the above contract in following ways: First, the condition of who pays is whether a company has defaulted or not. This company is called as the reference company in the definition of a CDS. Its instance is Goldman Sachs (GS). Second, in contrast to an insurance policy where the buyer secures himself from a possible eventuality, the purpose of buyer of a CDS may not be to secure himself but to bet on the default of the company. I present the definition of a CDS which has been slightly modified from the one at Wikipedia [8].

Definition 2.7 *A* Credit Default Swap *is an agreement between a protection buyer and a protection seller whereby the buyer pays a periodic fee in return for a contingent payment by the seller upon a default happening in the reference company.*

For ease of explanation the nominal value for CDS is assumed to be equal to $1.

Chapter 3

Haskell

The language of implementation by both, Peyton Jones et. al. and I, is Haskell. Wikipedia says: "Haskell is a standardized purely functional programming language with non-strict semantics, named after the logician Haskell Curry." Purely functional puts a restriction that the variables in a program written in Haskell will have immutable values. Non-strict semantics mean that Haskell will allow lazy evaluation, which I discuss towards the end of this chapter. In order to understand the representation of contracts in the contract language, the reader needs to understand some basic syntax and types in Haskell. I next illustrate these features and then argue why Haskell is a good choice to implement this work.

3.1 Computing Fibonacci numbers in Haskell

To illustrate how to write a function in Haskell, I write a function to compute Fibonacci Numbers.

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

The function definition for fibonacci consists of three declarations or rules. Whenever this function is called, one of the three rules will be executed. Interpret each rule in the following way: on the left hand side (lhs) of the symbol = we write the function name followed by the arguments it takes, and on the right hand side (rhs) we write the body of the rule. The value calculated in this body is returned as a result of the execution of this rule and hence the execution of the function. I can make calls to

functions in the body of a rule just as I do in the body of the the rule for *fibonacci n.*[1] This is how I will define functions in the chapters ahead.

3.2 Types in Haskell

Like there are data types in any programming language, there are types for data in Haskell also. Atomic data like 'a' and 5 have the following built in types:

'a' :: Char,
5 :: (Num t) ⇒ t.

In English these definitions mean that 'a' is of type char and 5 can be of any numeric type like Int or Double.

Another relevant data type is a list. Its purpose is to store a collection of data, each of which has the same type. For instance, ['a', 'b'] has the following type:

['a', 'b'] :: [Char].

[Char] means that this list is a collection of data of type Char. Similarly there are types for functions also. The type of fibonacci is,

fibonacci :: Int → Int.

It means that fibonacci maps an Int value to another Int value. We can have functions which map multiple data to another data. For instance,

func :: a → b → c

The function *func* maps data of type a and type b together, to data of type c, where a, b and c are any types. Similarly, if the type signature of a function contains n types separated by →, then the n^{th} type is the return type and the rest of n-1 types are the input data types.

---

[1] If you are familiar with rule based languages like Prolog, then it is easy to see that Haskell's notation resembles them.

3.3 Why Haskell?

The reasons for choosing Haskell to implement the language for financial contracts are the following:

1. Haskell is a declarative language, in which we specify rules, like I did for the function fibonacci. The contract language also specifies rules or declarations to construct data of type *Contract* and *Observable*. These rules are collectively called *combinator* library. I illustrate it in next chapter 4. So, since both the languages are declarative in nature, it is much easier to implement the latter language in Haskell than the OO languages like C++ and Java. In chapter 7, I elaborate on how to implement the contract language in Java and then compare the ease of implementation in both cases.

2. Since Haskell has non-strict semantics, it supports lazy evaluation, which is very helpful in evaluating data structures used to implement the contract language. To illustrate lazy evaluation, I introduce two built in Haskell functions and then combine them. First function is *take*:

   take :: Int $\rightarrow [a] \rightarrow [a]$

   This type signature means that it takes two arguments, an integer, say n, and a list. It returns a list of the same type as the one in argument. The purpose of the function is to trim the argument list to its first n elements. Second function is *[1..]*, it returns a list containing all natural numbers. Since the length of this list is infinite, it takes forever (infinite time) to compute this list. However, when I combine the above two functions like this:

   take 3 [1..],

   it returns [1,2,3] at the next instant. The reason for this is, because I am asking for only first three elements, Haskell does not compute the whole list, rather only the first three elements. The moral is that lazy evaluation means that only the

needed amount of computation is done. I will demonstrate in chapter 6 how useful it is in calculating the value of a contract.

3. Built-in Characteristics: Haskell has built-in data structures like lists, functions over lists and other mathematical functions. For instance, *take* (to trim a list), *++* (to append two lists), *zipWith* (to combine the corresponding elements of two lists using a mathematical operator), so on and so forth. They come in very handy in implementing the valuation of contracts (Please see Implementation in Appendix C to know how frequently they are used).

Chapter 4

Prior Work

In this chapter I will summarize the language proposed by Peyton Jones et al. [4]. This information will serve two purposes: first, it will provide the necessary background knowledge for what I say in next chapters; second, it will distinguish the prior work from my work.

Peyton Jones et al. have proposed a language which consists of *combinator* libraries for *observables* and *contracts*. The purpose of these libraries is to represent financial contracts by representing their cash flows and maturity. The amount of a cash flow in a contract usually depends on time varying quantities, which are called *observables* in the contract language.

Definition 4.1 *An* observable *is a time-varying quantity like an interest rate, exchange rate or today's date.*

In the contract language, the observables are defined as data whose type is *Obs a*, where a can be any type. For instance, the observable for interest rate has the type *Obs Double* because interest rate can take value of type Double. To construct observables Peyton Jones et al. define a combinator library for observables. This library is a set of *combinators for observables.*

Definition 4.2 *A* combinator for observables *is a construct or a function whose return type is always* Obs a*, where a can be any type.*

I present the combinators proposed by Peyton Jones et al. in table 4.1. In the table, the combinator *time* returns a data of type *Obs days*, where type of *days* is Int. The reader can confirm that all combinators have return type of *Obs a*, where a is any type. The description in this table has been taken as it is from Peyton Jones et al.[4].

konst :: a → Obs a (konst x) is an observable that has value x at any time. time :: Date → Obs days The value of observable (time t) at time s is the number of days between s and t, positive if s is later than t. lift :: Obs(a → b) → Obs a → Obs b (lift f o) is the observable whose value is the result of applying f to the value of the observable o lift2 :: Obs (a → b → c) → Obs a → Obs b → Obs c (lift2 f o1 o2) is the observable whose value is the result of applying f to the values of the observables o1 and o2.

Table 4.1: Combinators for Observables

To construct a financial contract, Peyton Jones et al. introduce another data whose type is *Contract*. This effectively means that in the contract language a financial contract is represented as a data of type Contract. Like observables, Contract also has a corresponding combinator library which is a set of *combinators for Contract*.

Definition 4.3 *A* combinator for Contract *is a construct or a function whose return type is always Contract.*

Table 4.2 shows the description of a subset of the combinators for contracts. We can see that all the combinators are functions except *one* because it doesn't take any argument. One is in fact a special function which acts as a seed to construct other contracts. This description, like that for the observables, is also taken as it is from the Peyton Jones et al.'s paper.

In the next chapter I present simple examples which use these combinators. Using these examples, I construct a CDS contract.

one :: Contract one is a contract that immediately pays the buyer $1. The contract has infinite maturity i.e. it never matures. scale :: Obs Double → Contract → Contract If you buy (scale o c), then you buy 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. truncate :: Date → Contract → Contract (truncate t c) is exactly like c except that it expires at the earlier of t and the maturity of c. and :: Contract → Contract → Contract If you acquire ('and' c1 c2) then you immediately acquire both c1 (unless it has matured) and c2 (unless it has matured). The composite contract matures when both c1 and c2 mature. get :: Contract → Contract If you acquire (get c) then you must acquire c at c's maturity. The composite contract matures at the same moment that c matures. give :: Contract → Contract To buy (give c) means to sell c. The buyer's rights are c's obligations and her obligations are c's rights.

Table 4.2: Combinators for Contracts

Chapter 5

Representation of Contracts

In this chapter I represent CDS and PRDC in the contract language. In this process, I detail the combinators in table 4.1 and 4.2. I also describe the combinators I introduced to be able to represent the above contracts. I choose CDS because it is not mentioned by Peyton Jones et al. and has a huge market.[1] I choose PRDC because it is completely different from CDS and a relatively new contract.[2] By representing these contracts I validate Peyton-Jones et al.'s claims, that the language is generic for both existing and new contracts.

5.1 CDS

To represent a CDS contract I introduce an observable combinator *dflt*. Its type is given by

dflt :: Int → Obs Bool

It takes an Int as an argument which is the unique identifier of a company, and returns a Bool value. If the returned value is true, then it implies that the corresponding company has defaulted, otherwise it has not. Let 0 be the identifier for Goldman Sachs (GS), then *dflt 0* is the observable (of type Bool) for defaulting of GS.[3]

---

[1] According to [3]: "The British Bankers Association (BBA) and the International Swaps and Derivatives Association (ISDA) estimate that the (CDS) market has grown from $180 billion in notional amount in 1997 to $5 trillion by 2004."

[2] According to http://www.risklatte.com/exotics/hotNotes/hotNotes021.php: "Since early 2004 Power Reverse Dual Currency (PRDC) swaps have become very popular with the Japanese investors, especially in the Dollar-Yen market." I am not sure when PRDC was invented but this source tells that the contract was not very popular before 2004.

[3] It was suggested by Prof. Alex that an alternative way of implementing *dflt* is to use the *ticker symbol* of a company, which is already unique rather than creating another unique integer identifier

In a CDS contract, the buyer pays a regular spread (in case of no default) and the seller pays the nominal value (in case of default). The contract becomes void when the contract matures or the company defaults, whichever is earlier. The time to maturity is divided into periods, at the end of each of which, either the buyer pays the spread or the seller pays the nominal. Of course, if the latter happens then the contract becomes void. To represent a CDS contract in terms of combinators I break the contract into elementary contracts, one for each period. I next break each elementary contract into two further elementary contracts: **Receive Nominal**, for the receipt of the nominal value and **Pay Spread**, for the payment of spread.

Receive Nominal: I begin by representing the contract for the receipt of the nominal value. To do this, let us define a made–up contract, which says that if GS defaults, then the buyer of this contract gets $1 right away, otherwise nothing happens. This contract has infinite maturity. I define a function called *receive_nominal* which takes maturity as an argument and returns this made-up contract with this maturity. The purpose of defining such a function is, I can use this contract in constructing more complicated contracts by calling this function. Similarly, for each contract I will construct, I will define the corresponding function along with its arguments. One of the rules of the function *receive_nominal* which constructs the above contract is defined below. In this rule, I use a variable called *inf* whose value is a very large number signifying infinity.

```
inf = 123456789
boolToDouble b1 = if(b1 == True)
                  then 1
                  else 0
receive_nominal inf = scale (lift boolToDouble (dflt 0)) one
```

In the body of the rule for receive_nominal, I have used few combinators. The combinator *one* takes no argument and returns a contract, which can be exchanged

---

(as I am doing). Since it involves significant changes in the current code, I have kept it as a future improvement.

for $1 any time (which means infinite maturity). The combinator *dflt 0* returns an observable, whose value is True (if Goldman has defaulted) or False (otherwise). The combinator *lift* lifts an operator over observables, in this case it applies the operator *boolToDouble* to *dflt 0*. By doing this, it converts an observable of type *Obs Bool* to another observable of type *Obs Double*. The combinator *scale* takes two arguments: an observable and a contract, which in this case are *lift boolToDouble (dflt 0)* and *one* respectively. It then multiplies the cash flows of the contract with the value of the observable. The maturity of the resulting contract is same as that of the contract in the argument (which in this case is *one*). Since there is no combinator preceding *scale*, *receive_nominal inf* returns the contract returned by *scale*. The value of this contract is $1 in case of default and 0 otherwise; its maturity is equal to infinity.

The type definitions of above methods are

```
boolToDouble :: Bool → Double
receive_nominal :: Int → Contract
```

Like this, **all functions** I define in this section have the **same type** as that of *receive_nominal* unless otherwise specified.

Now, I extend the above definition to the contract whose maturity is equal to one period. Before this, let us assume the variable *currentDate*'s value is today's date and *period_length* is the length of the period in days. I define another function *addDate*, which adds given number of days to a given date, and returns the resulting date according to some calendar.

```
addDate :: Date → days → Date
nextDate :: Int → Date
nextDate 1 = addDate currentDate (1*period_length)
receive_nominal 1 = truncate (nextDate 1) (receive_nominal inf)
```

In the body of receive_nominal 1, I use the contract combinator *truncate*. It takes two arguments: a date which in this case is *nextDate 1* and a contract which in this case is *receive_nominal inf*. The combinator trims the maturity of the argument contract to

the argument date. Next, I similarly define *receive_nominal n* which returns a similar contract to the above contract, but it matures after n periods from today.

```
nextDate n = addDate currentDate (n*period_length)
receive_nominal n = truncate (nextDate n) (receive_nominal inf)
```

The buyer of *receive_nominal n* gets the nominal as soon as GS defaults. Whereas, in CDS contract, the buyer will get the money only at the end of the period in which default occurs, not anytime before. To remove this difference we use the contract combinator *get*. The purpose of *get* is to fix the time of a cash flow, like in this case I want the cash flow to occur only at the end of the period. This combinator works as follows, to acquire *get c* means to acquire *c* at its maturity. Similarly, to acquire *get (receive_nominal n)* means to acquire *receive_nominal n* at its maturity, which is at the end of n^{th} period. Therefore now if there is a default before the end of n^{th} period, the buyer of *get (receive_nominal n)* receives the nominal value only at the end of n^{th} period because *receive_nominal n* is not acquired earlier.

```
get_receive_nominal n = get (receive_nominal n)
```

So, the contract returned by the above function is corresponding to the receipt of nominal at the end of n^{th} period. Now I move on to represent the contract corresponding to the payment of spread.

Pay Spread: The contract corresponding to paying spread differs from receiving nominal in three ways: First, the condition of payment is that the company has not defaulted. Second, the value of payment is equal to *spread* and not nominal. Third, the buyer of the CDS pays the spread rather than receiving it.

I now explain how to address these differences. To take care of the first difference, I define *not_receive_nominal inf* which is similar to *receive_nominal inf*. In the body, I use the observable combinator *lift2*. It lifts a binary operator like subtraction (-) over observables. .

```
not_receive_nominal inf = scale (lift2 (-) (konst 1) (lift boolToDouble (dflt 0) ) ) one
```

From what we know till now, I can say that the value of *konst 1* is 1 all the time and the value of *lift boolToDouble (dflt 0)* is either 1 or 0. The result of the observable given by (lift2 (-) (konst 1) (lift boolToDouble (dflt 0))) is the difference of 1 and (lift boolToDouble (dflt 0)). It is easy to see that this difference is 1 when there is no default and 0 otherwise.

The steps to construct *get_not_receive_nominal n* from *not_receive_nominal inf* are same as those of constructing *get_receive_nominal n* from *receive_nominal inf*, so I mention these steps without any explanation:

```
not_receive_nominal n = truncate (nextDate n) (not_receive_nominal inf)
get_not_receive_nominal n = get (not_receive_nominal n)
```

To overcome the second difference, I *scale* the contract returned by *get_not_receive_nominal 1* by an observable, which takes constant value equal to *spread*. To address the third difference, I use another contract combinator called *give*; it takes a contract as an argument and interchanges the rights and obligations for the buyer (hence for the seller also). This essentially means that the buyer of *give c* sells *c*. I define the payment of spread for one period as

```
pay_spread 1 = give (scale (konst spread) (get_not_receive_nominal 1))
```

The buyer of the contract returned by *pay_spread 1* pays the spread at the end of 1^{st} period, if GS does not default. Similarly, for the payment of spread at the end of n^th period,

```
pay_spread n = give (scale (konst spread) (get_not_receive_nominal n)).
```

1 Period CDS: I combine the two kinds of contracts defined above to write a one period CDS.

```
cds 1 = and (get_receive_nominal 1) (pay_spread 1)
```

In the body of *cds 1*, I use the contract combinator called *and*. It takes two contracts as arguments and returns a contract which combines the rights and obligations of both

the contracts respectively. Hence, the contract returned by *cds 1* means at the end of the period, its buyer receives the nominal ($1) if the default has occurred, otherwise she pays the *spread*, which is what a 1 period CDS contract specifies. The maturity (mat) of the contract returned by *and c1 c2* is *max (mat(c1), mat(c2))*, so the maturity of *cds 1* is 1 period.[4]

2 Period CDS: I extend the definition of one period CDS to two period CDS as:

```
cds 2 = and (cds 1) (and (get_receive_nominal 2) (pay_spread 2))
```

The two period CDS contract is an *and* of two contracts: one period CDS and *(and (get_receive_nominal 2) (pay-spread 2))*. Hence, cds 2 implies that at the end of each period: 1 and 2, the buyer pays the spread in case default has not occurred and gets the nominal otherwise. This definition does not take care of the fact that the contract becomes void at default. So, the buyer of *cds 2* will receive nominal twice if there is a default during the first period.[5] Since this representation is not exactly what I want, I define another contract which nullifies the excess money the buyer of *cds 2* is getting. This contract is called *excess 2*. It will correspond to the situation of default in first period and payment of nominal at the end of the second period. I negate the value of this contract by using the combinator *give* and then add *excess 2* to *cds 2*.

```
excess 2 = give (get (truncate (nextDate 1) (scale (lift boolToDouble (dflt 0))
                (get (truncate (nextDate 2) one)) )))
```

Since I have not introduced any new combinator in this definition, I leave its reasoning up to the reader as a fun exercise. But this exercise requires the knowledge of valuation of combinators, which I cover in the next chapter. I redefine *cds 2* as:

```
cds 2 = and (and (cds 1) (and (get_receive_nominal 2) (pay_spread 2))) (excess 2)
```

N Period CDS: Now by generalizing over above definitions, I can represent *cds n*. Before that I define *excess n* as:

---

[4]The function max is an built in function in Haskell which returns the greater of the two arguments.

[5]The underlying assumption in this statement is that the company which defaults remains defaulted in future.

```
excess n = give (get (truncate (nextDate (n-1)) (scale (lift boolToDouble(dflt 0))
                (get (truncate (nextDate n) one)) )))
```

The contract *excess n* is a generalized form of *excess 2*. It means payment of nominal at the end of n^{th} period, when default occurs by the end of $n-1^{th}$ period. I can construct *cds n* from *cds n-1* as:

```
cds n = and (and (cds (n-1)) (and (get_receive_nominal n) (pay_spread n))) (excess n)
```

In this way I can iteratively represent *cds n*, starting from *cds 1*. Since I am using the combinator *and* to combine contracts corresponding to each period, the maturity of *cds n* is n years.

5.2 Representation of Power Reverse Dual Currency (PRDC) Swap

In this section I explain what PRDC is and how I can represent it using the combinators. PRDC is a kind of a bond such that the its buyer receives a fixed amount (like principal) at the maturity and a coupon amount after every fixed period of time (like interest). But, the coupon payments of PRDC depend on the current foreign exchange (FX) rate between two given economies and not rate of interest. Typically a coupon payment looks like this:

max(current FX rate - Designated FX rate, 0) $\times$ A

In this payment, *current FX rate* (FX) is the foreign exchange rate between two economies, *Designated FX rate* (DFX) and A (A) are constant values. To represent this coupon payment in the contract language, I introduce an observable combinator which will represent FX.

```
fx :: Double → Obs Double
```

The combinator *fx* takes the current foreign exchange rate, which is a Double quantity and maps it to its values in future. DFX and A are represented as *konst dfx* and *konst a* respectively.

To write coupon payment for a period, I use *dol–yen* as a variable whose value is equal to the current Dollar to Yen exchange rate. Also, I use a built-in function in Haskell, *max* which returns the greater of the two arguments.

```
prdc_coupon 1 = truncate (nextDate 1) (scale (lift2 max (lift2 (-) (fx dol–yen)
                    (konst dfx)) (konst 0)) (scale (konst a) one))
```

Similarly, I can define coupon payments for all the periods. Since they are trivial extension of *prdc_coupon 1*, I omit them. After representing this contract, I have validated Peyton Jones et al.'s claims in another setting.

Conclusion: It might appear that the combinators were designed keeping these contracts in mind, but this is not the case. The combinator library was proposed by Peyton-Jones et al. in the year 2000. This paper does not mention CDS contract in any form and PRDC was not even popular at that time. Hence, I believe choosing these contracts to test the representative nature of the language makes sense. To model these contracts the only addition I have done to the language is to add combinators for observables. This is a very reasonable addition because different contracts depend on different time varying quantities. Hence, I believe I can represent different kinds of present and future contracts by reasonable additions to the current set of combinators.

Chapter 6

Valuation

Once the contracts are represented, the task of processing them becomes simpler than treating each kind of contract independently. I illustrate this by showing how to value a contract represented in the language.

To understand Valuation we need to know *Value Process*, which Peyton Jones et al. define as

Definition 6.1 *A* Value Process (VP) *is a partial function or procedure which takes time as an argument and returns a random variable (RV) corresponding to that time.*

VP :: time → RV

In implementation, a *VP* is a data structure which stores the value of contracts and observables for present and future times. But for future times, these (predicted) values are random variable and not fixed values. Hence, the return type of a VP is a random variable. Also, VP is a partial function because the value of a contract is undefined for times after maturity.

6.1 Value a Contract

In this sub–section, I show how to value a contract represented in the contract language. In the absence of any language we have to write a separate valuation program for each kind of contract. The advantage of using the language is that now we need to write only one program which values any represented contract.

To value the contracts and observables, Peyton Jones et al. define valuation functions *evalC* and *evalO* as:

evalC :: Model → Contract → VP

evalO :: Model → Obs a → VP

The function *evalC* (*evalO*) takes two arguments, a Model and a contract (observable) and returns the VP corresponding to the value of the contract (observable).

Lets digress a little to learn about data of type Model. This data contains information of a financial model. This information is about how the value of each observable will evolve in future. There are several models used in the industry but I have used only lattice models in my implementation. This does not reflect any restriction on the usability of the language because the model is external to the language and hence this approach is not model specific. However, the implementations of *evalC* and *evalO* are model dependent, but I see no problems in writing these methods for other models also.

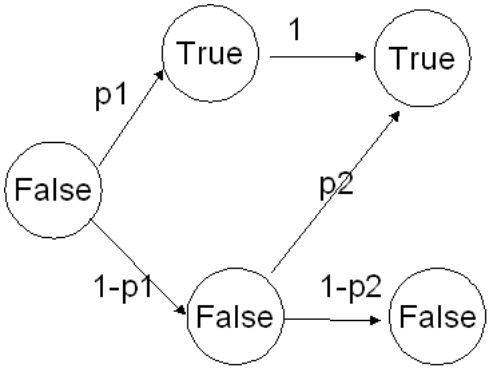

Figure 6.1: A Default Model

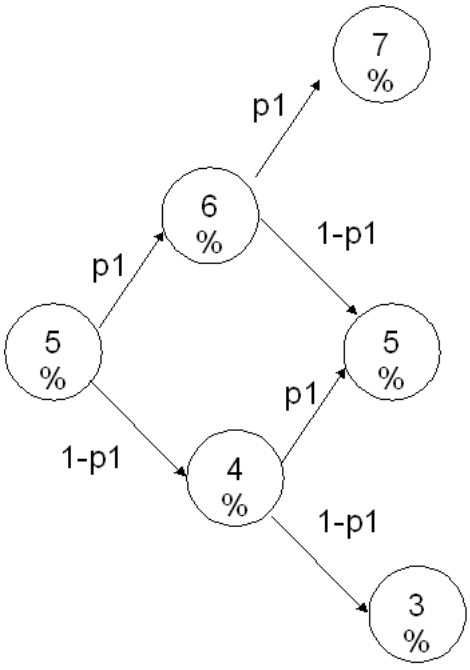

Figure 6.2: An Interest Rate Model

Figure 6.1 and 6.2 show the lattice models I have used to model the default status and the interest rate respectively. The default model means the following: initially the value of *dflt 0* is false, at the end of first period the probability of *dflt 0* being True is p1 and being False is 1-p1. Once its value is True, it will remain True with certainty otherwise it will again transition to True with p2 and False with 1-p2. Similarly, the

model can be extended to more than 2 periods. I believe this explanation is sufficient to understand the interest rate model also. A more elaborate explanation of the lattice models and how do we value a contract using them is in the appendix A. The interest rate lattice model is very common in finance and is mentioned in Peyton Jones et al.[4], whereas I have designed the default model. In order to represent the combined information, I multiply these models. The multiplication is accomplished by taking cartesian product of the snapshots of two models at the end of each period. For instance, at the end of first period *dflt 0* can take any of the values in this set: {*True, False*} and similarly interest rate can take values from {*6%, 4%*} , their cartesian product is: {*(6%, True), (6%, False), (4%, True), (4%, False)*}.

The digression is over and now we return to our original goal of writing one program to value any contract. The trick is to implement the instances of *evalC* and *evalO* for all the combinators for *Contract* and *Obs a* respectively. The implementation depends on what data structure you choose to represent VP and Model. You can see appendix C to see one instance of implementation for our choice of data structures. We list here (figure 6.3) all the semantics for applying evalC and evalO on combinators. They can be appropriately implemented depending on the data structures you choose.

```
evalO m (Lift2 op o1 o2) = (evalO m o1) op (evalO m o2)
evalO m (Lift op o1)     = op (evalO m o1)
evalC m (And c1 c2)      = (evalC m c1) + (evalC m c2)
evalC m (Or c1 c2)       = max (evalC m c1) (evalC m c2)
evalC m (Scale o c)      = (evalO m o) * (evalC m c)
evalC m (Give c)         = - (evalC m c)
evalC m (Truncate d c)   = trim (evalC m c) d
evalC m (Get c)          = disc (evalC m c)
evalC m One              = valueprocess_1
evalO m (Konst d)        = valueprocess_d
evalO m (Default i)      = returnVP m i
evalO m (Time d)         = returnTimeDiff m d
```

Figure 6.3: Semantics for the implementation of evalC and evalO

The semantics for the combinators proposed by Peyton Jones et al. are taken from their paper. I will elaborate on last six semantic rules because the ones before these

are fairly easy to understand just by looking at. When we apply evalC to Truncate d c, we evaluate the contract c and ignore its values after the date d. The function *trim* will chop the values which are to be ignored. When evalC is applied to Get c, we need to evaluate c, find c's value at maturity and discount this value back to earlier periods till current time. In the body of the corresponding rule, *disc* does this job when it gets VP corresponding to the value of c. Rest of the instances of evalC and evalO are base cases because they do not in turn call evalC or evalO. When evalC is applied to One, the return value should be a VP such that this VP maps each time point to a RV, which takes value 1 with certainty. Similarly, when evalO is applied to (Konst d), result should be same as *evalC m One* except that now the RV should take value d. When evalO is applied to *Default i* it should return a VP which maps each time point to a RV, whose probability distribution should be same as that of the default till that time point. The application of evalO to *time d* returns a VP such that each time is mapped to a difference between that time and the date d.

This is the point where I show the real advantage of the contract language. To value any data of type Contract, we apply the method *evalC* to it, the method percolates down to the Contract's constituent combinators. I now show this next for the Contract *cds 1*. Since we have already written evalC and evalO for all the combinators, a call to any of them will return a VP which will be appropriately combined with other VPs.

```
evalC m (cds 1) = (evalC m (get_receive_nominal 1)) + (evalC m (pay_spread 1))
evalC m (get_receive_nominal 1) = disc (evalC m receive_nominal 1)
evalC m (receive_nominal 1) =trim (evalC m (receive_nominal inf)) (nextDate 1)
evalC m (receive_nominal inf) = (bool2Double (evalO m (dflt 0))) * (evalC m one)
evalC m (pay_spread 1) = -(evalO m (konst spread)) *
                                    (evalC m (get_not_receive_nominal 1))
evalC m (get_not_receive_nominal 1) = disc (evalC m (not_receive_nominal 1))
evalC m (not_receive_nominal 1) = trim (evalC m (not_receive_nominal inf))
                                    (nextDate 1)
```

```
evalC m (not_receive_nominal inf) = ((evalO m (konst 1)) -
                                     (bool2Double (evalO m (dflt 0)))) *
                                     (evalC m one)
```

Among these instances of applications of *evalC* and *evalO*, the fourth and the last instances from the top are evaluated first because their *rhs* consists of evaluation of combinators only. The resultant values percolate up and are used to evaluate the rest of the instances. This is how *cds 1* gets evaluated; similarly, *cds n*, *PRDC* or any other contract will also get evaluated.

Next I present a discussion on some of the issues regarding the choice of combinators and the implementation language.

Chapter 7

Discussion

In this section I will discuss some of the issues regarding the choices I made to implement this work.

First issue is the way in which the combinator *dflt* is used in constructing the CDS contract. While constructing *cds 1*, what we wanted was to be able to pick one contract from two contracts: *receive_nominal 1* and *pay_spread 1* depending on the value of *dflt 0*. To accomplish this, we apply different operations on this observable and then scale these two contracts with the modified obersvable values to get the desired contract. A cleaner way would be using a combinator which is of the form if–then–else. It checks the value an observable of type Bool and accordingly returns one of the two contracts. Peyton Jones et al. must have realized this need, that is why in their later work have added a combinator which exactly does what I just wrote.[1]. The combinator is called *cond* and is defined below.

cond :: Obs Bool → Contract → Contract → Contract

This combinator values a boolean observable say *dflt 0* and if it is true, returns the first contract, else the second. We can use this to write *cds 1* as:

```
cds 1 = cond (dflt 0)(get (truncate (nextDate 1) one))
                     (give (scale spread (get (truncate (nextDate 1) one))))
```

This is clearly a cleaner representation of *cds 1* than the one given in chapter 5.

Second is more of a philosophical issue that, why do these combinators work and are there some better alternative combinators. As a matter of fact, Peyton Jones et al.

[1]This work is available as a chapter called *How to write a financial contract* in the book "The Fun of Programming".

have suggested the future work as expanding the set of their combinators so that we can represent wider set of contracts. They (and also I) have shown that these combinators can already represent different kind of contracts. But, there is a possibility that there are alternative combinators, which are as (or more) primitive as (than) these and can represent even wider variety of contracts. But, such an alternative does not exists to our knowledge, so I find it hard to debate. Hence, I see these combinators as a good starting point to build a combinator library which can represent all the contracts.

Third issue is about the choice of language to implement the contract language. Peyton Jones et al. choose Haskell, but it can be done in any OO language like Java or C++. I explain one possible class design for this application in figure 7.1.

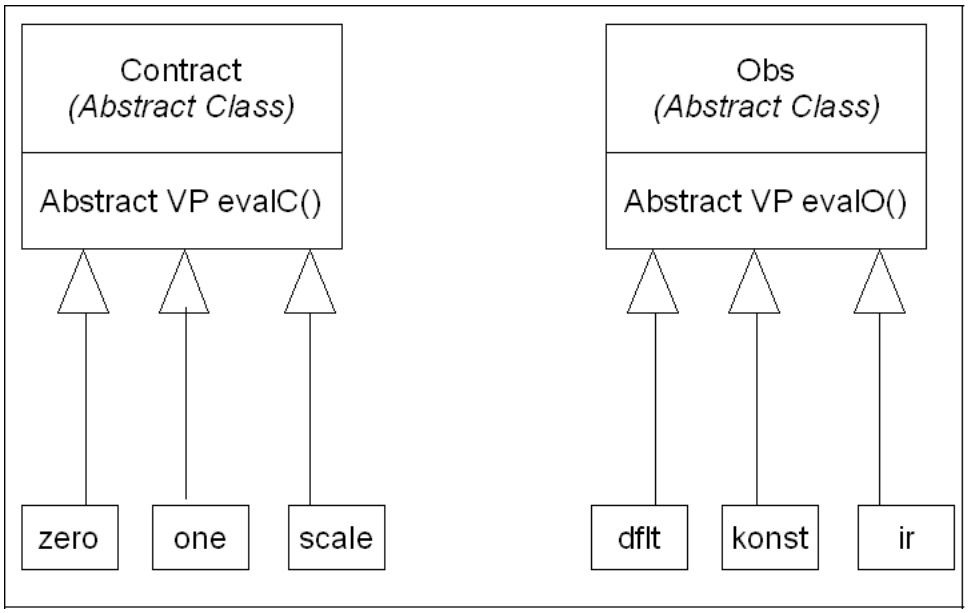

Figure 7.1: UML Class Diagram for the Contract Language

As shown in the UML class diagram, I model *Contract* and *Obs* as abstract classes, and each of their combinator extends their corresponding abstract class. The abstract classes have *evalC* and *evalO* respectively as abstract methods, which each combinator has to implement. These abstract methods return VP as the value of the combinator.

In this scenario, I have to make new class for each combinator, whereas in Haskell, I just had to add one line in the definition of the data types *Contract* or *Obs*. Haskell

being declarative provides this advantage in implementation.

Next I present the related work to the field of domain specific languages applied to finance.

Chapter 8

Related Work

The idea of building a domain specific language (*DSL*) is a well researched area. The paper by Deursen et al. ([7]) surveys the relevant literature and presents example *DSLs*. It also presents the risks and benefits that come with building a *DSL*. We see this work as a good starting point for someone who intends to acquire introductory and broad knowledge about the topic of *DSL*.

The paper by Deursen and Klint [6] is more focused and discusses in detail the issues involved with building a *DSL*. It takes the case of RISLA, a *DSL* which was built by a Dutch bank called *Mees Pierson* to deal with interest rate products. The paper addresses two challenges: first, describes a meta environment *ASF+SDF* (Algebraic Specification Formalism + Syntax Definition Formalism) to support the development and maintenance of domain specific languages; second, provides a solution to the problem of how different applications built for different *DSLs* can talk to each other. As a solution, they propose a *ToolBus* architecture which can be used to coordinate different domain specific languages.

An an alternative approach to building a DSL, which achieves the same goals is to build an object oriented framework. Unlike the case of RISLA, it does not build an environment to support the development and maintenance of *dsls*, rather it uses the available technologies like modern user interfaces, domain-specific frameworks and design patterns to build banking applications [5]. An instance of such a framework is ET++ SwapsManager, which was built to value the swaps at the Union Bank of Switzerland [1].

Peyton Jones et al.'s approach falls in the category of building a domain specific

language, even though it is not quite the same as RISLA. Peyton Jones et al.'s combinator library is applicable to a wider variety of financial instruments, whereas RISLA was built only for interest rate products. I believe the reason for this limitation is that RISLA has no concept of observables. So, even though the definition of contract is same for both RISLA and Peyton Jones et al.'s language, RISLA cannot represent contingent cash flows, which occur in contracts like a CDS. Another difference is that the RISLA system is heterogeneous in the sense it is a meta level language which uses COBOL libraries underneath. In contrast, the combinator library described in this thesis is a homogeneous system because both the language and the libraries are written in the same language (Haskell). Intuitively, I believe that the homogeneous design is easier to maintain and extend. On the other hand, both RISLA and the combinator library are modular in nature. Both of them define primitive contracts which can be combined to create complex contracts.

Chapter 9

Conclusion and Future Work

In this thesis we have tried to educate the reader that Peyton Jones et al.'s approach to build a domain specific language to represent financial contracts, is indeed an effective approach. This is because the language is very generic in nature and hence can represent existing and evolving contracts. We showed this by representing CDS and PRDC. The most significant advantage of building such a language is that now the number of computer programs required to process different contracts is much less $(m + n)$ than otherwise(mn).

There have been a few attempts in this direction and there is a lot to be done so that this technology can actually be used. We see the following things as a good starting point for this:

1. Extending the system built by us (my adviser and I) so that it can represent and value all the traded contracts in any one domain say credit risk domain. For this we will need to enumerate all the observables, which can appear in the definition of contracts of this domain. Any mathematical combination of these observables will be automatically taken care of because we have mathematical operators as combinators. Now optimistically, we should be able to represent all the existing and yet to come contracts from credit risk domain. Like this, we can extend it to other domains like interest rate derivatives and forex derivatives.

2. Educating the user about how to use this system is a concern. He has to be well versed with the combinators to use them correctly and efficiently. To alleviate this problem, we see one possible solution as building a UI which can guide the user in coming up with the correct definition of the contract. Hence, we see this

as a useful feature because it will ease the job of the user and make the tool easily usable.

Appendix A

Finance 102

In this part we illustrate how we use a financial model, specifically a lattice model to value a financial contract, specifically a CDS. A typical one period CDS corresponding to Goldman Sachs (GS) from the buyer's perspective looks like:

CDS_1: Both of:

C_1: On December 31,2008, if GS has defaulted, Receive \$1.

C_2: On December 31,2008, if GS has not defaulted, Pay \$0.1.

The (present) value of a CDS (or any other contract) is the sum of the *expected* value of each cash flow when *discounted* back to present time[1]. The expected value of a cash flow is product of, value of the cash flow and the probability that it will occur. The discounted value of a cash flow is the equivalent present value assuming risk–free interest rate as the rate of return. For instance, consider a hypothetical contract in which the buyer receives \$1 after one year with probability 0.3. Assuming the rate of interest to be 5%, the present value of the contract will be:

value1 = $\frac{(0.3\times\$1)}{1+0.05}$

value2 = exp(-0.05)×0.3×\$1

If we assume discrete compounding, value1 is the present value and if we have continuous compounding, value2 is the present value.

To calculate the value of any contract we need to predict its future cash flows which in turn depend on how the value of observables will evolve in future. In the

[1]To learn more about financial instruments and their valuation techniques the reader should refer to "Options, Futures and Other Derivatives" by John C. HULL[2]

case of CDS the observables are default status of the company and the interest rate. Financial models are used to predict these values. We have used lattice model in our implementation and we now explain this model in more detail.

Figures 6.1 and 6.2 show the lattice models for the prediction of default and interest rate respectively. The starting point for each model is the left most circle which contains the present value of the observable. The arrows emanating from this point are corresponding to possible transitions to values, which the observable might take by the end of the period. The value labeling each arrow is the probability of that transition. For instance, Fig 6.1 represents the default model in which the initial value of the observable *Default 0* is *False*. There are two arrows emanating from this value, one goes to the value *True* and the other to *False*. The probability of first transition is *p1* and that of second is *1-p1*. This implies that the probability of default during the first period is *p1*. From the value *True*, the only possible transition is to the same value. This is because once the company has defaulted it remains so with probability equal to 1. From the value *False*, there are similar transitions as there were from the initial value *False*. That is how the model is constructed.

If we apply the same reasoning to the interest rate model in Fig 6.2, we can deduce that the initial interest rate is 5% and in all periods it increases by 1% with probability p1 and decreases by the same amount with probability $1 - p1$.

This information allows us to value CDS_1:

eVal = (p1*1.00 - (1-p1)*0.1)
dVal = exp(-0.05)×eVal

In these equations $eVal$ is the expected value of the cash flows and $dVal$ is the price of CDS_1 assuming continuous compounding.

We can extend this procedure to value any other CDS contract or any other contract for that matter.

Appendix B

Code: Language

```
{-# OPTIONS -fglasgow-exts #-}
module Language2 where

import GHC.Num
import GHC.Real
import GHC.Base
import List

type TimeI = Int
type Time = Double

-- Contract Definition
data Contract a =
        Zero
        | One
        | Give Contract
        | And Contract Contract
        | Or Contract Contract
        | Truncate TimeI Contract
        | Scale (Obs Double a) Contract
        | Get Contract
        deriving (Show)

-- interface functions
zero = Zero
one = One
give = Give
and = And
or = Or
truncate = Truncate
scale = Scale
get = Get
anytime = Anytime

data Obs a where
        Konst :: a -> Obs a
        Apply :: Obs (b -> a) -> Obs b -> Obs a
        DiffTime :: Double -> Obs Double
        IR :: Double -> Obs Double
        Default :: Int -> Obs Bool

instance Show (Obs a)

-- interface functions
konst = Konst
apply = Apply
```

```
diffTime = DiffTime
dflt = Default

-- lift functions
lift f o1 = Apply (Konst f) o1
lift2 f o1 o2 = Apply (Apply (Konst f) o1) o2
```

Appendix C

Code: Implementation of Evaluation

```
module Implementation4 where

import Language2
import Data.Array.IArray


-- date types, constants
type Probability = Double
type IR = Double
type Val = Double
type Spread = Double
type Default = Double
type TProb = (Double, Int)
type State = (Double, [Bool])
type Dist = Array Int (State, [TProb])
type Model = [Dist]

--length' :: Array i e -> Int
length' arr1 = let bds = bounds arr1
               in (snd bds) - (fst bds) + 1

ir :: IR
ir = 0.05

dir :: IR
dir = 0.01

threshold :: Double
threshold = 0.00001

thisyear :: Time
thisyear = 2006

thisyearI :: TimeI
thisyearI = 2006

prob :: Probability
prob = 0.5

getIR :: State -> Double
getIR (x,_) = x
getDef :: State -> [Bool]
getDef (_,x) = x

getP :: TProb -> Double
getP (x,_) = x
```

```
getSt :: TProb -> Int
getSt (_,x) = x


tmpIRState :: State
tmpIRState = (ir, [])

tmpIRM :: Dist
tmpIRM = array (1,1) [(1,(tmpIRState, []))]

tmpDState :: State
tmpDState = (0, [False])

tmpDM :: Dist
tmpDM = listArray (1,1) [(tmpDState, [])]


-- build Interest Rate Model
buildInterestRateModel :: Probability -> IR -> Model
buildInterestRateModel p1 dir1 = let tp = [(p1, 1), (1-p1, 0)]
                                 in let lt = tmpIRM ! 1
                                    in let st = fst lt
                                       in let arr1 = (array (1,1) [(1,(st, tp))])
                                          in arr1 : (buildInterestRateModel' arr1 p1 dir1)

buildInterestRateModel' :: Dist -> Probability -> IR -> Model
buildInterestRateModel' dist1 p1 dir1 = let lt = elems dist1
                                        in let lt1 = reverse (take (length lt) [0..] )
                                           in let newLt = shrink (concat (zipWith (\x y ->  [((getIR (fst x) - dir1, get!
!Def (fst x) ), [(p1, y+1), (1-p1, y)] ), ((getIR (fst x) + dir1, getDef (fst x) ), [(p1, y+2), (1-p1, y+1)] )] ) lt lt1 !
!) )
                                              in let len = length newLt
                                                 in let newArr = listArray (1, len) newLt
                                                    in newArr : (buildInterestRateModel' newArr p1 dir1)


azipWith :: Ix i => (a -> b -> c) -> Array i a -> Array i b -> Array i c
azipWith f a b | bounds a == bounds b
        = listArray (bounds a) (zipWith f (elems a) (elems b))



buildDefaultModel :: [Probability] -> Model
buildDefaultModel [] = []
buildDefaultModel pl = (getInitModel pl) : buildDefaultModel' pl

buildDefaultModel' :: [Probability] -> Model
buildDefaultModel' pl = getDModel pl 0 : buildDefaultModel' pl

getInitModel :: [Probability] -> Dist
getInitModel pl = let st = (0, take (length pl) (repeat False))
                  in let tp = getTProbs 0 pl (2^(length pl) - 1)
                     in array (1,1) [(1, (st,tp))]

getTProbs :: Int -> [Probability] -> Int -> [TProb]
getTProbs sr pl dt = if (dt < 0)
                        then []
                        else if (not (validTransition sr dt pl))
```

```
                                then (0,dt): getTProbs sr pl (dt-1)
                                else let pl' = zipWith (\x y -> if x == True
                                                                  then 1
                                                                  else y) (getBoolean sr (length pl)) pl
                                     in ((foldl1 (*) (zipWith (\x y -> if (y == True)
                                                                  then x
                                                                  else 1-x) pl' (getBoolean dt (length pl)) ) ), d!
!t) : getTProbs sr pl (dt-1)


 validTransition :: Int -> Int -> [Probability] -> Bool
 validTransition sr dt pl = let p = zipWith (\x y -> if (x == True && y == False)
                                                    then False
                                                    else True) (getBoolean sr (length pl)) (getBoolean dt (length pl!
!))
                            in (not (elem False p) )

 getBoolean :: Int -> Int -> [Bool]
 getBoolean l1 ln = let l2 = getBoolean' l1
                    in (take (ln - (length l2)) (repeat False) ) ++ l2

 getBoolean' :: Int -> [Bool]
 getBoolean' l1 = if (l1 == 0)
                        then [False]
                        else reverse (getBoolean'' l1)

 getBoolean'' :: Int -> [Bool]
 getBoolean'' l1 = if (l1 == 0)
                        then []
                        else if (mod l1 2 == 1 )
                                then True : getBoolean'' (quot l1 2)
                                else False : getBoolean'' (quot l1 2)

 getDModel :: [Probability] -> Int -> Dist
 getDModel pl l1 = let dst = getDModel' pl l1
                   in let len = length dst
                      in listArray (1, len) dst

 getDModel' :: [Probability] -> Int -> [(State, [TProb])]
 getDModel' pl l1 = if (l1 > (2^(length pl) - 1))
                        then []
                        else let st = (0, getBoolean l1 (length pl))
                             in let tp = getTProbs l1 pl (2^(length pl) - 1)
                                in (st,tp) : getDModel' pl (l1+1)


 -- shrink for building interest rate model
 shrink :: [(State, [TProb])] -> [(State, [TProb])]
 shrink (l1:l2:lt) = let l1t = getIR (fst l1)
                         l2t = getIR (fst l2)
                     in if ((abs (l1t - l2t)) < threshold)
                         then shrink (l1: lt)
                         else l1:(shrink (l2:lt))
 shrink (l1:[]) = [l1]

 multiplyIntDefModels :: Model -> Model -> Model
 multiplyIntDefModels [] md2 = md2
 multiplyIntDefModels md1 [] = md1
```

```
multiplyIntDefModels md1 md2 = zipWith (\x y -> let lx = elems x
                                                    ly = elems y
                                                in let newlt = concat (map (\z -> let sz = getIR (fst z)
                                                                                      tz = snd z
                                                                                  in map (\p -> let sp = getDef (fst p)
                                                                                                    tp = snd p
                                                                                                in let newtp = mulTransi!
!tionProbs tz tp
                                                                                                       newst = (sz,sp)
                                                                                                   in (newst,newtp) ) ly!
!) lx )
                                                    in let len = length newlt
                                                       in (listArray (1, len) newlt) ) md1 md2

mulTransitionProbs :: [TProb] -> [TProb] -> [TProb]
mulTransitionProbs [] tp2 = tp2
mulTransitionProbs tp1 [] = tp1
mulTransitionProbs tp1 tp2 = concat (map (\x -> let p1 = getP x
                                                    st1 = map (\x -> if (x == True)
                                                                     then 1
                                                                     else 0) (getBoolean' (getSt x))
                                                in map (\y -> let p2 = getP y
                                                                  st2 = map (\x -> if (x == True)
                                                                                   then 1
                                                                                   else 0) (getBoolean' (getSt y))
                                                              in let newP = p1*p2
                                                                     tmpst = st1 ++ st2
                                                                 in let newSt = foldl1 (+) (zipWith (*) (reverse tmpst) !
!(scanl1 (*) (1:(repeat 2))) )
                                                                    in (newP, newSt) ) tp2) tp1)

-- helper method to build the final Model
buildModel :: Probability -> IR -> [Probability] -> Model
buildModel p1 ir2 pl = multiplyIntDefModels (buildInterestRateModel p1 ir2) (buildDefaultModel pl)


-- shrink for building interest rate model
{-shrink :: [(State, [TProb])] -> [(State, [TProb])]
shrink (l1:l2:lt) = let l1t = getIR (fst l1)
                        l2t = getIR (fst l2)
                    in if ((l1t - l2t) < threshold)
                        then shrink (l1) : lt
                        else l1:(shrink (l2:lt))
shrink (l1:[]) = [l1]
-}

-- Value Process is now the distribution and not expected value
type ValueProcess a = [RandomV a]
type RandomV a = [a]

-- finds the expected Value for each time point.
findExpValueProcess :: Model -> ValueProcess Double -> [Double]
findExpValueProcess md vp = let pr = getReachableProb md
                            in zipWith (\x y -> foldr1 (+) (zipWith  (\p q -> p* (lookup' q x) ) y [0..] ) ) pr vp

getReachableProb :: Model -> [[(Int, Probability)]]
getReachableProb (m1:md) = take (length' m1) (repeat (0,1)) : getReachableProb' (m1:md) (repeat (0,1))
getReachableProb' (m1:md) lt = let newLt = sumLists (zipWith (\x z -> let y = lookup' z lt
```

```
                                                                    in map (\p -> let tmpP = y*(getP p)
                                                                                      tmpS = getSt p
                                                                                  in (tmpS, tmpP) ) (snd x) ) (elems m!
!1) [0..])
                              in newLt : getReachableProb' md newLt


sumLists :: [[(Int, Double)]] -> [(Int, Double)]
sumLists (l1:[]) = l1
sumLists (l1:l2:lt) = sumLists ((concat (zipWith' l1 l1 l2 l2) ):lt)

{-sumLists (l1:l2:lt) = sumLists ((concat (zipWith' (\x y -> let tmpX = (lookup' (fst x) l2)
                                                                 tmpY = (lookup' (fst y) l1)
                                                             in if (not (tmpX == -1))
                                                                  then if (not (tmpY == -1) )
                                                                          then [(fst x, (snd x + tmpX))]
                                                                          else ([(fst x, (snd x + tmpX)), y])
                                                                  else if (not (tmpY == -1))
                                                                          then [x]
                                                                          else ([x,y])) l1 l2) ):lt)
-}

--zipWith' :: (a -> a -> [a]) -> a -> a -> [a]
zipWith' [] _ [] _ = []
zipWith' [] l1 (y:lt2) l2 = let tmpY = (lookup' (fst y) l1)
                            in if (tmpY == -1)
                                then [y] : (zipWith' [] l1 lt2 l2)
                                else zipWith' [] l1 lt2 l2
zipWith' (x:lt1) l1 [] l2 = let tmpX = (lookup' (fst x) l2)
                            in if (not (tmpX == -1))
                                then [(fst x, (snd x + tmpX))] : (zipWith' lt1 l1 [] l2)
                                else [x] : (zipWith' lt1 l1 [] l2)
zipWith' (x:lt1) l1  (y:lt2) l2 = let tmpX = (lookup' (fst x) l2)
                                      tmpY = (lookup' (fst y) l1)
                                  in if (not (tmpX == -1))
                                        then if (not (tmpY == -1) )
                                                then [(fst x, (snd x + tmpX))] : (zipWith' lt1 l1 lt2 l2)
                                                else [(fst x, (snd x + tmpX)), y] : (zipWith' lt1 l1 lt2 l2)
                                        else if (not (tmpY == -1))
                                                then [x] : (zipWith' lt1 l1 lt2 l2)
                                                else [x,y] : (zipWith' lt1 l1 lt2 l2)


lookup'           :: Int -> [(Int,Double)] -> Double
lookup' key []    =  -1
lookup' key ((x,y):xys)
     | key == x   =  y
     | otherwise  =  lookup' key xys


getTValueProcess :: Model -> Double -> ValueProcess Double
getTValueProcess m1 d1 = let t3 = convertDateToTime d1
                             vp2 = incValueProcess m1 1.0
                         in map (map (\x -> x - t3)) vp2

incValueProcess :: Model -> Double -> ValueProcess Double
incValueProcess (m1:md) i = [i|x<-[1..(length' m1)]] : incValueProcess md (i+1)
```

```
andValueProcesses :: ValueProcess Double -> ValueProcess Double -> ValueProcess Double
andValueProcesses [] vp3 = vp3
andValueProcesses vp4 [] = vp4
andValueProcesses (v1:vp1) (v2:vp2) = (zipWith (+) v1 v2) : (andValueProcesses vp1 vp2)


-- evaluate procedures for different contracts
evalC :: Model -> Contract -> ValueProcess Double
evalC m (Zero) = map (\x -> [0.0|y<-[1..(length' x)]]) m
evalC m (One) = map (\x -> [1.0|y<-[1..(length' x)]]) m
evalC m (Give c1) = map (map negate) (evalC m c1)
evalC m (And c1 c2) = andValueProcesses (evalC m c1) (evalC m c2)
evalC m (Or c1 c2) = zipWith (zipWith max) (evalC m c1) (evalC m c2)
evalC m (Truncate d0 c1) = let t1 = convertDateToTimeInt d0
                               vp1 = evalC m c1
                           in take t1 vp1

evalC m (Then c1 c2) =  let vp1 = evalC m c1
                            vp2 = evalC m c2
                        in if length vp1 < length vp2
                                then vp1++drop (length vp1) vp2
                                else vp1
evalC m (Scale o1 c1) = zipWith (zipWith (*)) (evalC m c1) (evalO m o1)

evalC m (Get c1) = let vp1 = evalC m c1
                   in let l1 = last vp1
                      in reverse (discountRandomVar m l1 ((length vp1) - 1))

discountRandomVar :: Model -> RandomV Double -> Int -> ValueProcess Double
discountRandomVar md rv t1 = if (t1 == 0)
                               then [rv]
                               else (rv : (discountRandomVar md (discountRV md rv t1) (t1-1)))

discountRV :: Model -> RandomV Double -> Int -> RandomV Double
discountRV md rv1 t1 = let md1 = elems (md !! t1)
                           md2 = elems (md !! (t1-1))
                       in map (\y -> let prob = snd y
                                     in foldl1 (+) (map (\z -> let ind = snd z
                                                               in ((fst z)*(rv1 !! ind)/(1 + getIR (fst (md1 !! ind)) ))!
!) prob) ) md2


--finds the discount factor
--discountValueProcess :: Model a -> ValueProcess -> Val -> TimeI -> ValueProcess
{- # discountValueProcess m1 vp2 l1 t1 = if t1 < 0
                                          then []
                                          else let l2 = getDiscountedValues m1 l1 vp2 t1
                                               in l2 : discountValueProcess m1 vp2 (foldl1 (+) (zipWith (*) (map (!
!\x -> (getP x) ) (m1 !! t1) ) l2 ) ) (t1-1)


--getDiscountedValues :: Model a -> Val -> ValueProcess -> TimeI -> [Double]
getDiscountedValues m1 v1 vp1 t1 = map ((*) v1) (map ((/) 1.0) ((zipWith (zipWith (+)) vp1 (evalC m1 One)) !! t1))
-}

convertDateToTime :: Time -> Time
convertDateToTime y1 = (y1 - thisyear + 1)
```

```
convertDateToTimeInt :: TimeI -> Int
convertDateToTimeInt y1 = (y1 - thisyearI + 1)


--evaluation of observables
evalO :: Model -> Obs a -> ValueProcess a
evalO m (Konst d) = map (\x -> [d|y<-[1..(length' x)]]) m
evalO m (DiffTime d) = getTValueProcess m d
evalO m (Default i) = map (\x -> let lt = elems x
                                 in map (\y -> if ( ((getDef (fst y)) !! i) == True )
                                               then True
                                               else False) lt) m
evalO m (IR intr) = map (\x -> let lt = elems x
                               in map (\y -> getIR (fst y) ) lt) m
evalO m (Apply o1 o2) = zipWith (zipWith (\x y -> x y)) (evalO m o1) (evalO m o2)
```

Appendix D

Code: Implementation and Valuation of CDS

```
module Cds2 where

import Implementation4
import Language2


-- helper function to value a CDS for notional = $1, it is ~ 0 in all cases.
creditDefault' :: TimeI -> Spread -> ValueProcess Double
creditDefault' t1 s1 = let p1 = s1/(s1+1)
                       in let m1 = buildModel prob dir [p1]
                          in eval m1 (cds t1 s1)

creditDefault :: TimeI -> Spread -> ValueProcess Double
creditDefault t1 s1 = let p1 = s1/(s1+1)
                      in let m1 = buildModel prob dir [p1]
                         in eval m1 (returnCDS 1 t1 s1)



-- returnCDS: returns the cds contract in the specified language.
--returnCDS t1 s1 = Get (returnCDS' 1 t1 s1)
returnCDS :: TimeI -> TimeI -> Spread -> Contract
returnCDS t1 t2 s1 = if t1 > t2
                       then (Language2.truncate (thisyearI+t2) Zero)
                       else Language2.and (Language2.and (Language2.and (getCDSGet t1) (getCDSGive t1!
! s1)) (subExcess t1)) (returnCDS (t1+1) t2 s1)

getCDSGet :: TimeI -> Contract
getCDSGet t1 = get (Language2.truncate (thisyearI+t1) (scale (lift boolToDouble (dflt 0)) one))

getCDSGive :: TimeI -> Spread -> Contract
getCDSGive t1 sp = give (scale (Konst sp) (get (Language2.truncate (thisyearI+t1) (scale (lift2 (-) (Konst 1) (lif!
!t boolToDouble (dflt 0))) one))))



subExcess :: TimeI -> Contract
subExcess t1 = give (get (Language2.truncate (thisyearI+t1-1) (scale (lift boolToDouble (dflt 0)) (get (Impleme!
!ntation4.truncate (thisyearI+t1) one)) )))
subExcess' t1 = give (get (Language2.truncate (thisyearI+t1-1)  (get (Language2.truncate (thisyearI+t1) (sca!
!le (lift boolToDouble (dflt 0)) one)) )))

boolToDouble :: Bool -> Double
boolToDouble b1 = if (b1 == True)
                       then 1
                       else 0
```

References

[1] Thomas Eggenschwiler and Erich Gamma. Et++swapsmanager: Using object technology in the financial engineering domain. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 166–177, October 1992.

[2] John C. Hull. Options, futures and other derivatives, prentice hall.

[3] Mike Jakola. Credit default swap index options at: http://www.kellogg.northwestern.edu/research/fimrc/papers/jakola.pdf.

[4] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). *In ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 280–292, 2000.

[5] Arie van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *Proceedings Smalltalk and Java in Industry and Academia*, pages 35–39, September 1997.

[6] Arie van Deursen and Paul Klint. Little languages: Little maintenance? In Samuel Kamin, editor, *First ACM-SIGPLAN Workshop on Domain-Specific Languages; DSL'97*, pages 109–127, January 1997. Technical report, University of Illinois, Department of Computer Science.

[7] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[8] Wikipedia. Credit default swap: http://en.wikipedia.org/wiki/credit_default_swap.