UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor Thesis

Kasper Nicolaj Schiller

A Domain Specific Language for Financial Contracts

BSc programme in Computer Science & Economics

Supervisor: Martin Elsman

Submission: 12th of June, 2023

Abstract

In the past decades, the financial derivatives market has experienced a rapid change, resulting in a necessity to not only manage and valuate existing derivatives, but also to minimise time to market for new, exotic derivatives. In this thesis, we establish a framework capable of expressing, managing and pricing a wide range of financial instruments. We do this by constructing a DSL deeply embedded in $F\#$ which employs a set of constructors grounded with logical principles, enabling the user to express a wide variety of financial contracts. We supply with functions that allow for management of contracts expressed within the language, and employ methods from continuous time finance for pricing.

Contents

1 Introduction **1**

1.1 Outline . . . 2

2 Background **2**

2.1 Financial Contracts . . . 2

2.2 Pricing Derivatives . . . 3

2.3 Domain Specific Languages . . . 5

2.4 Denotational Semantics . . . 5

3 Design and Implementation of the DSL **6**

3.1 Related work . . . 7

3.2 Language Choice . . . 7

3.3 Time Representation . . . 8

3.4 Data Types and Primitives . . . 9

3.5 Logic . . . 11

3.6 Expressing Contracts . . . 11

4 Managing Contracts **15**

4.1 Related Work . . . 16

4.2 Design . . . 16

4.2.1 Information Functions . . . 16

4.2.2 Manipulating Contracts . . . 20

4.3 Implementation . . . 25

5 Evaluating Contracts **26**

5.1 Related Work . . . 26

5.2 Design . . . 26

5.2.1 Evaluation Functions . . . 26

5.2.2 Wiener Processes . . . 29

5.2.3 Pricing Options . . . 29

5.2.4 The Path-Dependent Case . . . 31

5.3 Implementation 31

6 Testing **34**

6.1 Introduction to xUnit and FsUnit 34

6.2 Contract Management 35

6.3 Pricing 36

6.3.1 Currency Evaluation 36

6.3.2 Obs Evaluation 36

6.3.3 Simulations 37

6.3.4 Contract Evaluation 38

7 Evaluation of the DSL **39**

7.1 Logical Principles 39

7.2 Demonstration 41

8 Future Work **44**

9 Conclusion **46**

1 Introduction

In the past decades, the financial derivatives market has experienced a remarkable evolution and today, the aggregate value of the assets linked to existing derivative transactions is several times larger than the world GDP (Hull, 2018, p. 1). These contracts consist of an infinite amount of variations and permutations, each with its own unique set of terms and conditions.

Institutions, banks, asset managers and other entities enter contracts like these today, not only to speculate in the market, but also to adjust risk. In fact, risk emerges in circumstances where insurance is not necessarily accessible. For instance, a manufacturing company which production relies heavily on a particular commodity, such as copper, may want to buy a security whose value will rise if copper prices rise. This way, they can protect themselves against the losses they may experience with the rising prices of copper, also known as hedging (Shreve, 2004, p. XVI). Entities that are involved in such trades need not only to know how much their contracts are worth currently, but they also have to define these contracts in a system and manage them.

In this thesis, the aim is to develop a program with the capability of not only generating terms and conditions of financial contracts, but also manage and price them according to selected logic principles and pricing models. In addition, we validate the design and test the implementation of the program.

We do this by developing a language deeply embedded in F# tailored for the financial domain, also known as an embedded Domain Specific Language (DSL) (Hudak, 2012, p. 1). We build the DSL on a set of algebraic data types and constructors, so that we are be able to create complex contracts while maintaining clarity. In addition, we will be able to create new derivatives as the financial market evolves. We supply with functions that operate on the algebraic data types which will allow for management and pricing of contracts. The design is validated by proving that logic principles of contracts do hold when pricing contracts, and the implementation is tested with unit black box testing.

1.1 Outline

In general, this thesis is divided into 4 main parts.

In the first part, sections 2 and 3, we present theory and design decisions that have influenced the development of the the DSL. Additionally, we implement the language in $F\#$, and demonstrate that this implementation can indeed be used to express various financial contracts.

The second part, section 4, presents, designs and implements functions that have the purpose of managing financial contracts expressed within the language.

In the third part, section 5, we create functionality for pricing financial contracts within the language based on logic principles introduced in the first part of the thesis.

Finally, we test the implementation and validate the design of the DSL and its functionalities in sections 6 and 7.

After the main parts, possibilities for future work are discussed and the results are concluded.

2 Background

This section aims to provide an overview of the theoretical foundations of the thesis. We begin with a concise introduction of the concept of financial contracts in section 2.1. In section 2.2, we provide a deeper introduction to continuous time finance and its applications in pricing derivatives. Section 2.3 gives a brief introduction to what a DSL is. Lastly, in section 2.4, we present the syntax of the semantic notation which will be used when designing functions in this thesis.

2.1 Financial Contracts

When we refer to the term *financial contract* or *financial instrument*, it means an agreement among parties to exchange a certain amount of currency. One type of contract is a *derivative*, which is a contract which outcome is determined by a value which can be observed and agreed upon by the parties involved (Hull, 2018, p. 1), also known as an observable. Examples of such observables include interest rates, stock prices, and even the weather on Mars on a specific date.

2.2 Pricing Derivatives

To price derivatives, we will rely on theory from continuous time finance and simulation of stochastic processes. In the case of a non-dividend paying stock, we assume that its price follows the stochastic process

$$S_t = S_0 e^{\left(\mu - \frac{1}{2}\sigma^2\right)t + \sigma W_t} \tag{1}$$

which is also known as a Geometric Brownian Motion. Here, S_0 is the current price of the stock, μ is the yearly drift, that is, the expected average growth over a year, and σ is the yearly volatility of the stock (Björk, 2009, p. 69). The term W_t refers to a Wiener Process, also known as a Brownian Motion, at time t, which is a stochastic process with the following properties

1. The process starts at zero, meaning $W(0) = 0$.
2. The future behaviour of the process does not depend on the past. In other words, the increments of the process between time points are independent from each other, that is, for $r < s \leq t < u$, we have $W(u) - W(t) \perp\!\!\!\perp W(s) - W(r)$.
3. The increments are normally distributed, that is $W(t) - W(s) \sim \mathcal{N}(0, t-s)$ for $s < t$.
4. W has continuous trajectories.

(Björk, 2009, p. 40). We can utilize the GBM to price a derivative that has a stock as an underlying by solving the expected payoff discounted to the present value, which can be expressed as

$$e^{-rT} E\left[f(S_T)\right] \tag{2}$$

where f is the payoff function for the derivative, also known as the payment to the holder at expiration, and e^{-rT} is a discount function used to calculate the present value of the payoff. In general, we operate with 4 types of ways to compute the expected payoff of a derivative:

Explicit Solutions. Sometimes, the term $E\left[f(S_T)\right]$ can be solved analytically by probability theory. This is the case with a forward option, which is exercised at maturity and pays $S_T - K$ (can be negative), where K is the negotiated strike. This makes the option fairly easy to price. Recall that the moment generating function for a normal random variable $Z \sim N(\mu, \sigma^2)$ is given by

$$E\left(e^{sZ}\right) = e^{\mu s + \frac{\sigma^2 s^2}{2}} \tag{3}$$

where $s \in \mathbb{R}$ (Bulmer, 1979, p. 111). We have $W_t \sim N(0,t)$ by property 3 of the wiener process. Thus, the theoretical price of the forward option is

$$\begin{aligned}
e^{-rT}\left[E[S_T - K]\right] &= e^{-rT}\left[E[S_T] - K\right] &(4)\\
&= e^{-rT}\left[E[S_0 e^{\left(\mu-\frac{1}{2}\sigma^2\right)T+\sigma W_T}] - K\right] &(5)\\
&= e^{-rT}\left[E\left[S_0 e^{\left(\mu-\frac{1}{2}\sigma^2\right)T}\right]\cdot E\left[e^{\sigma W_T}\right] - K\right] &(6)\\
&= e^{-rT}\left[S_0 e^{\left(\mu-\frac{1}{2}\sigma^2\right)T}\cdot e^{0\cdot\sigma+\frac{T\sigma^2}{2}} - K\right] &(7)\\
&= e^{-rT}\left[S_0 e^{\left(\mu-\frac{1}{2}\sigma^2\right)T+\frac{1}{2}T\sigma^2} - K\right] &(8)\\
&= e^{-rT}\left[S_0 e^{\mu T} - K\right] &(9)\\
&= S_0 e^{(\mu-r)T} - e^{-rT}K &(10)
\end{aligned}$$

assuming that the underlying stock follows a GBM.

Monte Carlo Simulation. We can also price a derivative by simulating the paths of the GBM N times and aggregating the results of the payoff functions by means of averaging. The law of large numbers ensures that this estimate converges to the actual expected value of the payoff as N increases (Glasserman, 2003, p. 1). This method is especially relevant when working with options that do not have a closed form solution and require simulation. But even when working with options that do have a closed form payoff solution, it provides a general framework to price them all at once, which is highly useful.

Partial Differential Equations, which are solved either analytically or numerically. They are derived from abstract mathematical and probability theory under specific assumptions, such as that a stock price follows the GBM, no arbitrage, or that people are risk-neutral and only wishes to maximize profitability. An example of a PDE is the Black Scholes equation, which gives a closed form solution for the price of a call or put option, with the assumption that the drift of a stock is equal to the interest rate (no arbitrage) (Shreve, 2004, ch. 4.5).

Binomial Models, also known as lattice trees. These are discrete time models which utilize the principles of valuation and risk to price options. This method typically consists of drawing a tree of the possible outcomes of the option, assuming a certain probability for each outcome and solving the tree with backward induction (Cox et al., 1979).

In this thesis, we will utilize Monte Carlo methods when valuating derivatives and compare the result to the closed form solutions or PDE if it exists. While this method has certain drawbacks in terms of performance, it provides a general framework for pricing contracts. In addition, it allows us to comprehend from abstract mathematical theory and focus on the implementation and design of the DSL.

2.3 Domain Specific Languages

A Domain Specific Language (DSL) is characterized as a language which is tailored to a specific domain. Usually, they are targeted at end users or domain experts who are not experts in programming (Hudak, 2012, p. 1). In general, there are two types of DSLs we consider:

External DSLs. These languages are build in a separate language to the main programming language it is represented in (Fowler et al., 2010, p. 15). Depending on the implementation, the developer may have to implement their own compiler and parser for deployment.

Embedded (internal) DSLs. Here, the language is built, designed and implemented within its host language, thereby inheriting its constructs, syntax and compiler (Hudak, 2012, p. 2). One way to develop such an DSL is by considering a set of data types with belonging *primitives* or *constructors*, and implementing the language with foundation in these.

2.4 Denotational Semantics

In sections 4 and 5, we will be considering the design and implementation of various functions on abstract data types which sometimes consist of recursive pattern matching. In the design phase, we will try to abstract from the implementation in $F\#$ by utilizing the notation of denotational semantics to define these functions. We use syntax inspired by (Filinski, 2023, ch. 7).

For every constructor q in a data type, we denote $\mathcal{Q}[\![q]\!]$ to define the semantic meaning of a function $\mathcal{Q}$. We denote inputs that are not constructors in pattern matching by a subscript or superscript. That is, we denote the first input by a subscript, the next by a superscript, the next by a subscript, and so on. An example is presented in 1. In addition, we let λ denote an anonymous function, so that $\lambda x.\sqrt{x}$ denotes a function $f(x) = \sqrt{x}$. We also borrow notation

$$
\begin{aligned}
Bool &:: True \mid False \\
\mathcal{T} &: (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to Bool \to \mathbb{R} \\
\mathcal{T}[\![True]\!]^{g}_{f,x} &= f(x) \\
\mathcal{T}[\![False]\!]^{g}_{f,x} &= g(x)
\end{aligned}
$$

Figure 1: A concrete example

from functional languages when denoting operations on lists. We could write

$$Concat\left(map\left(\lambda x.x^2\ \ [\mathbb{R}_1, \ldots \mathbb{R}_n]\right)\right)$$

to express that the anonymous function x^2 is being mapped to each element in a float list, and then returned as a single list by the $Concat$ operator. Of course, denoting a float by a real number is not quite correct, since a float does not have infinite precision: this is just for mathematical notation purposes. We denote @ to be the concatenation operator and reserve $\mathcal{S}$ for the string type.

3 Design and Implementation of the DSL

When designing a domain-specific language, several key decisions must be made to ensure that the language is expressive, efficient, and user-friendly. In this section, the main design decisions and limitations that shaped the development of the DSL are presented. In section 3.1, we explore the existing work already done in the field of domain specific languages, especially within financial contracts. Section 3.2 presents and reasons the programming language and paradigm chosen. Section 3.3 evaluates the design decision behind representing time in contracts. In 3.4, we present the data types and primitives that will shape the design of the language. In section 3.5, we outline some of the logical requirements for contracts to ensure the validity of the contracts expressed in the language. In 3.6, we show how to express contracts in the DSL based on a implementation in F#.

3.1 Related work

There are several examples of work in the area of designing languages for the financial domain, often embedded in functional programming languages. One example is a combinator library in Haskell which defines a set of combinators that is able to describe unforeseen contracts as the market evolves. In addition, the language allows for *processing* of contracts, which is essentially simulating the value of a contract (Peyton et al., 2000, p. 280). Another example is the design and implementation of the DSL called SPL. This language, embedded in Haskell, was designed for specifying stochastic processes. The implementation focused on performing Monte Carlo simulations using GPGPU (Werk et al., 2012, p. 1).

Several companies in the financial sector also work with DSLs within their domain. This includes SimCorp, who have developed a DSL for use in their module XpressInstruments among other things, which allows to express, manage and price a wide range of financial contracts, including exotic derivatives (XpressInstruments User Manual, 2008).

In addition, a number of financial institutions have acknowledged the benefits of functional programming. Examples include Jane Street Capital, a electronic trading firm, who use OCaml as their primary programming language (Minsky et al., 2008).

Beyond the realm of finance, DSLs are employed in various other domains. Well known examples are SQL, a language used for managing and manipulating databases and MATLAB, which has its domain in science and engineering.

3.2 Language Choice

Our DSL is deeply embedded in F# and is designed to be able to describe, manage and price financial contracts. F# is a functional first programming language developed by Microsoft for the .NET platform, and is available as open source for many operation systems (Sporring, 2020, p. 12).

Functional programming as a paradigm was chosen due to several factors. Functional languages have a declarative style, that is, the focus is on expressing what needs to be done, rather than how it should be done. Also, higher order functions, pattern matching, and other constructs help improve readability of the code. This way we can abstract away low-level- implementation details and allow for non-programmers to validate both the contract definitions and the functions used to manage and price them.

3.3 Time Representation

One critial design decision in implementing our DSL is the time representation. Most, if not every contract will be time dependent. In general, there are two approaches one can take in a financial DSL in this regard.

Relative time: Here, is time is represented as a relative quantity. For example, an observable for Apple's stock price in 10 days could be expressed by Stock("AAPL" 10). Todays price would be Stock("AAPL", 0). This approach is more intuitive for users who need to model financial instruments over a fixed time horizon, such as options with a specific expiration date. Relative time also simplifies calculations and comparisons, as it avoids dealing with specific dates and their associated complexities like weekends and holidays.

Absolute time: In SimCorp's DSL, time is represented using absolute dates, such as Stock("AAPL", "2023-07-31") (Elsman, 2010). This corresponds with the design decision done in (Peyton et al., 2000), where the authors define a primitive:

$$time :: Date \to Obs\ Days$$

which lets the author perform arithmetic operations on dates. This allows them to define a contract from an absolute date, and then convert that date to a relative date to allow for operations on the contract.

Absolute time can be more natural for users who need to express financial instruments with reference to specific calendar dates, such as bonds with coupon payments on specific days. The approach also allows for more accurate modeling of real-world market behavior, as it can take into account the exact timing of events and their potential impact on financial instruments. In addition, it can make the contracts more readable.

In this project, we have chosen relative time representation and 365 days in a year. The reason for this is that it is easier to work with, as it avoids dealing with specific dates and their associated complexities like weekends, and holidays. This allows for more straightforward calculations and comparisons, which can lead to easier maintenance of the DSL. While the absolute time representation may offer certain advantages in modeling financial instruments with reference to specific calendar dates, the ease of development provided by the relative time approach was the primary factor in the decision.

3.4 Data Types and Primitives

To maintain simplicity and ease of use while still being able to provide an expressive language for describing financial contracts, we have a set of fundamental data types and primtivies that should be created. Combining these, we should be able to create various financial contracts. They are inspired by related work related work in the field, in particular (Peyton et al., 2000) and (Elsman, 2010). First, we present the data types and primitives, and then we go over them by expressing contracts.

The chosen data types include:

- Currency: A type that represents the currency used in the contract (e.g., USD, EUR, GBP, DKK).
- Observable values (Obs): A type that represents values which two parties agree upon. This is a quantity that can be time-varying. Examples of such values are the market price of an underlying. If a contracts terms and conditions depend on this price, it is crucial that the two parties are able to determine and agree on this value. However, the observable value can also contain uncertainty: contracts can depend on values that are not fully known yet, such as the average temperature in the London area next week. In addition, the data type should also be able to perform arithmetic operations.
- Contract: A type that represents a financial contract. We should be able to define a varitey of contracts using this data type.

Now, we need to consider of primitives for the data types in order to be able to define financial contracts.

The Currency type is straightforward, representing a unit of currency such as USD, CNY or DKK. In this case, the primitives for the Currency data types are defined as the specific currencies that are desired to be included in financial contracts in our language.

For the Obs data type, we must be able to represent a value that is agreed upon by two parties and perform arithmetic operations on it. There are several ways to do this. In

(Peyton et al., 2000), they enable the use of standard mathematical operators and functions on observables of numeric types by creating generic primitives, such as

$$lift :: (a \to b) \to \textit{Obs}\ a \to \textit{Obs}\ b \tag{11}$$

where $(\textit{lift}\, f\, o)$ is the result of applying the function f to the observable o (Peyton et al., 2000, p. 284). Such a primitive provides generality. However, we can also implement it as in SimCorps DSL, where we have one primitive for each arithmetic operation (Elsman, 2010, p. 8). This design offers explicitness and fine-grained control, however not as much freedom. For now, we have decided to follow SimCorps implementation of arithmetic operations. Finally, we define the primitive $Underlying$, which lets us obtain a underlying value at a specific time relative to today. Every primitive for the Obs data type that will be considered a part of the DSL are presented in Figure 2.

Obs Primitives
Value :: *double* → *Obs* $Value(r)$ associates r with an Obs type.
Underlying :: *string* → *int* → *Obs* $Underlying(s, t)$ returns the value representing the underlying s at time t.
Mul :: *Obs* → *Obs* → *Obs* $Mul(o_1, o_2)$ represents the multiple of o_1 and o_1.
Add :: *Obs* → *Obs* → *Obs* $Add(o_1, o_2)$ represents addition of $o1$ and $o2$.
Sub :: *Obs* → *Obs* → *Obs* $Sub(o_1, o_2)$ represents subtraction on o_1 and o_2.
Max :: *Obs* → *Obs* → *Obs* $Max(o_1, o_2)$ represents the maximum value of o_1 and o_2.

Figure 2: Primitives on the Obs data type

Now we turn to the main data type of the program: Contract. The goal of primitives of this type are to let us be able to describe new, unforeseen contracts by combining them: not have one primitive for each type of contract. The primitives considered are presented in Figure 3.

3.5 Logic

When reasoning about contracts in our DSL, there are some logic principles that we want to make sure applies when expressing, managing and pricing contrats. One inequality that must hold is that if c_1 and c_2 are contracts, then

$$Give(Or(c_1, c_2)) \neq Or(Give(c_1), Give(c_2)) \tag{12}$$

since in the first contract contract the counterparty has the right to choose between c_1 and c_2, and in the second contract the holder has the choice (Peyton et al., 2000, p. 288). We also introduce the following logic principles our the language. Assume that c denotes a contract and o denotes an observable, then the following must hold at all times

$$Acquire(t_1, Acquire(t_2, \ldots, Acquire(t_n, c)) = Acquire(t_1 + t_2 + \ldots + t_n, c) \tag{13}$$

$$Give(Give(c)) = c \tag{14}$$

$$Scale(o_1, Scale(o_2, \ldots, Scale(o_n, c))) = Scale(o_1 \cdot o_2 \cdot \ldots \cdot o_n, c) \tag{15}$$

$$Acquire(0, c) = c \tag{16}$$

$$Scale(Value\ 0.0, c) = Scale(Value\ o, All[]) \tag{17}$$

In addition, when we denote $Acquire(t, c)$, the time elements of the contract c are not directed towards the present, but rather projected into the future by a duration of time t, since we are working with absolute time. An example where this is important is shown in Listing 4.

When expressing and managing contracts, we will *assume* that these logic principles hold. When pricing contracts, we will *make them* hold.

3.6 Expressing Contracts

Using the building blocks from the DSL, we can express various types of instruments that can be used to create contracts, shown in the file $src/Instruments.fs$ and presented in this section. We begin by implementing the presented data types in $F\#$ as discriminated unions as shown in the file $src/Domain.fs$.

The first instrument we want to describe is the movement of a certain amount of currency on a specific date. We define the function $flow$ for this purpose. This function can be used to define contracts that are deterministic: that is, contracts where we have full information about

Contract Primitives
One :: *Currency* $\rightarrow$ *Contract* *One(ccy)* is a contract which consists of one unit of ccy.
Scale :: *Obs* $\rightarrow$ *Contract* $\rightarrow$ *Contract* *Scale(o, c)* is a contract that consists of o units of the contract c.
All :: *Contract List* $\rightarrow$ *Contract* $All([c_1, c_2, \dots c_n])$, $n \geq 1$, is a contract which consists of the contracts $c_1, c_2 \dots c_n$.
Acquire :: int $\rightarrow$ *Contract* $\rightarrow$ *Contract* *Acquire(t,c)* means to acquire the contract c at time t.
Give :: *Contract* $\rightarrow$ *Contract* *Give(c)* is a contract that consists of the counterparty acquiring c.
Or :: *Contract* $\rightarrow$ *Contract* $\rightarrow$ *Contract* $Or(c_1, c_2)$ gives the holder the right to acquire either c_1 or c_2 immediately.

Figure 3: Primitives on the Contract data type

the outcome as we enter the contract. A zero coupon bond is one of the simplest examples of such a contract. This bond pays no dividends and is redeemed for its full face value at maturity. A function that expresses this instrument is shown in Listing 1 along with the $flow$ function.

```
// A function for determninistic flows
let flow(t: int) (v: double) (c: Currency) : Contract
    = Acquire(t, Scale(Value v, One c))
// A zero coupon bond
let zcb (maturity : int) (face : float) (ccy : Currency) : Contract =
  flow maturity face ccy
```

Listing 1: A Zero Coupon Bond

We can also express instruments with multiple flows using the *All* constructor. Conceive that the user wants to buy a coupon bond, which pays interest until maturity, at which point the face value is paid. We can express such an asset as seen in Listing 2. The function begins by generating a sequence of dates for the interest payments. Then it uses this list to create a Contract List, which is a list of flows containing interest payments at their day of payment. It then use the All constructor to make this into a Contract containing these flows, along with the flow regarding the payment of the face value at maturity.

```
let cb (T : int) (face : float) (rate : float) (yearlyFreq : float)
    (ccy : Currency) : Contract =
    let dates = int(yearlyFreq * 365.)
    let rateDates = [dates .. dates .. T] |> List.ofSeq
    let couponFlows : Contract List =
        List.map (fun x -> flow x (face * rate / yearlyFreq) ccy) rateDates
    let faceFlow = zcb T face ccy
    All(faceFlow :: couponFlows)
```

Listing 2: A Coupon Bond

Transactions can also be executed in the opposite direction using the Give constructor. The Give constructor is essentially just a contract in the opposite direction. Let us utilize this constructor as we define a cross currency swap. This is a derivative in which two parties agree to exchange interest payments and principal amounts denominated in different currencies. Assuming that the principal amount paid at the beginning and maturity of the contract are the same, we can define a currency swap as in 3. Here, we assume that the holder of the contract is party A.

Let us now define a European call option. This is an option where the buyer has the right, but not the obligation, to acquire stock at maturity in exchange for strike price negotiated at the acquisition. The payoff for a call option is

$$C_T = (S_T - K)^+ \tag{18}$$

where S_T is the stock price at maturity, K is the strike, and x^+ is a notation for $max(x, 0)$. The reasoning behind the payoff function is that if S_T is greater than K, the buyer will receive a payoff of $S_T - K$, otherwise the payoff will be 0 since the option is not exercised. Using this formula, we can define a call option in two ways as shown 4: that is, either from its payoff

```
let ccSwap (T : int) (ccA : Currency) (ccB : Currency) (nA : float)
(nB : float) (rA : float) (rB : float) (yearlyFreq : float) : Contract =
    let dates = int(yearlyFreq * 365.)
    let rateDates = [365 / dates .. 365  .. T]
    let interestPaymentsA
        = List.map (fun x ->
                          Give(flow x (nA * rA / yearlyFreq) ccA)) rateDates
    let interestPaymentsB
        = List.map (fun x ->
                          flow x (nB * rB / yearlyFreq) ccB) rateDates
    All([
        Give(flow 0 nA ccA); // Inception: give nA units of currency ccA.
        flow 0 nB ccB; // Inception: acquire nB units of currency ccB.
        All(interestPaymentsA); // Pay interest at intervals
        All(interestPaymentsB); // Receive interest at intervals
        Give(flow T nB ccB); // At maturity, pay back nB units of ccB.
        flow T nA ccA // At maturity, acquire back nA units of ccA.
    ])
```

Listing 3: A Cross Currency Swap

function or from its definition using the Or constructor. Upon having a look at this listing, it may seem odd that we have defined the payoff function to depend on the underlying stock at time 0, but recall the last point in 3.5: when we denote $Acquire(T, c)$, the time elements of c are shifted into the future by a duration of time T.

Now, assume that we have defined a function $europeanPut$ which corresponds to the second option in 4 but with the choice being $(K - S_T)^+$. Then we can define a Chooser Option, which gives the holder the right to decide whether to obtain a put or call option at time t as in Listing 5.

It is evident that there is an arbitrary amount of more contracts to introduce within the language, but these contracts showcase how the constructors can be utilized to define a wide range of contracts.

```
// A european call expressed from its payoff function
let europeanCall1 (T : int) (stock : string) (strike : float )
    (ccy : Currency) : Contract =
    let payoff : Obs =
        Max(Value 0.0,
            Sub(Underlying(stock, 0),
                Value strike))
    Acquire(T, Scale(payoff, One ccy))
// A european call expressed from its definition
let europeanCall2 (T : int) (stock : string) (strike : float )
    (ccy : Currency) : Contract =
    let c : Contract = Or(
                        Scale(Value 0.0, One ccy),
                        Scale(Sub(Underlying(stock, 0),Value strike),
                            One ccy))
    Acquire(T, c)
```

Listing 4: Two ways to express a European Call Option in the DSL

```
let chooserOption(t : int) (T : int) (stock : string) (strike : float)
    (ccy : Currency) : Contract =
    let ec = europeanCall2 T stock strike ccy
    let ep = europeanPut T stock strike ccy
    Acquire(t, Or(ec, ep)) // the maturity of ec or ep is then t+T.
```

Listing 5: A Chooser Option

4 Managing Contracts

In this section, we will be considering how to manage contracts in the language. 4.1 introduces the work that serves as a main inspiration for some of the management functions created. In section 4.2, we design functions that are able to 1) give information on contracts (section 4.2.1) and 2) manipulate contracts (section 4.2.2). The functions are designed using the semantics syntax presented in 2.4 with logic principles in mind from 3.5. In 4.3, we present the implementation of the functions.

4.1 Related Work

In the context of managing contracts, the work done in (Elsman, 2010) will serve as a source of inspiration both on getting information on contracts and managing them. Notably, the implementations in (Elsman, 2010) of the functions $simplifyObs$, $simplify$, $advance$, and $cashflows0$ in Standard ML has served as a valuable reference for the design of the functions $simplfyObs$, $simplify$, $advance$ and $flows$ in this section. However, our design of these functions differs significantly due to the fact that we are working with absolute dates, an extra constructor and we have extra logical principles such as (13).

4.2 Design

In Listing 4 we present functions that we consider to implement to ease the management of contracts in the language. The first 4 functions provide information on a contract, and the last 3 functions manipulate contracts. To begin, we use the semantic syntax presented in 2.4 to formally describe the data types and their primitives from section 3.4. The result is shown in Listing 5.

4.2.1 Information Functions

The purpose of functions in this section is to provide information about contracts. Every function in this section assumes that the input contract has been simplified according to the logical principles in section 3.5, which can be done using the $simplify$ function presented in section 4.2.2.

Starting with $maturity$, the aim of this function is to be able to retrieve the relative date of which a contract matures. Semantics for a possible solution is presented in Listing 6, where we per design define the maturity date to be the last flow of the contract.

It is also useful to know which flows a contract consists of. To start, we construct the data type $flow$ that has 4 constructors: Uncertain, Certain, Choose and Causal. A Uncertain flow is a flow which is dependent on an underlying we do not know the value of yet. A Choose flow is a flow where the holder can choose between contracts, such as the case with $Or(c_1, c_2)$.

Functions for Contract Management
maturity :: *Contract* $\rightarrow$ int *maturity(c)* returns the maturity date of c.
flows :: *Contract* $\rightarrow (int \times flow)\ List$ flows(c) returns a list containing the the time and value (if certain) of the incoming flows from the contract and whether they are certain or uncertain.
causal :: *Contract* $\rightarrow (int \times flow)\ List$ causal(c) returns the flows of c which are causal, at which time they are executed and at which time in the future the flows are dependent on. A causal flow is a flow which we do not know the value of as it is executed.
underlyings :: *Contract* $\rightarrow (\mathcal{S} \times int)\ List$ underlyings(c) returns the name and time of the underlyings that c is dependent on.
simplify :: $((string \times int) \rightarrow \mathbb{R}) \rightarrow$ *Contract* $\rightarrow$ *Contract* *simplify E c* simplifies the contract c given the environment E.
advance :: $((string \times int) \rightarrow \mathbb{R}) \rightarrow$ Contract $\rightarrow int \rightarrow$ Contract *advance E c d* advances the contract c with d days given the environment E.
choose :: (Contract $\rightarrow \mathbb{R} + \perp$) $\rightarrow$ Contract $\rightarrow$ Contract *choose f c* makes a choice on the $Or(c_1, c_2)$ constructor based on the expected payoff of c_1 and c_2 according to f. If f returns $\perp$ for either c_1 or c_2, it returns $Or(c_1, c_2)$.

Figure 4: Functions for Contract Management

Semantics for a solution is shown in Listing 7. We base the design of the solution on the following: The main function $flows$ uses a helper function which takes as input an int, float Option and a contract, which $flows$ runs with the inputs 0, $Some(1.0)$ and c, where c is the input contract to $flows$. The first argument of the helper function keeps track of the time at which the flow is executed, and the second argument tells us whether it is a certain or uncertain flow. A simple example of running the function on a contract is shown in the following. We

$$ccy := USD \mid EUR \mid GBP \mid DKK \mid \ldots$$

$$\begin{aligned} k &:= Value(\mathbb{R}) \mid Underlying\,(\mathcal{S}, int) \mid k_1 \times k_2 \mid k_1 + k_2 \mid k_1 - k_2 \\ &\mid max(k_1, k_2) \end{aligned}$$

$$\begin{aligned} c &:= One(ccy) \mid Scale(k, c) \mid All\,([c_1 \ldots c_n])\,,\ n \geq 1 \mid Acquire(int, c) \mid Give(c) \\ &\mid Or(c_1, c_2) \end{aligned}$$

Figure 5: Semantic notation for the data types Currency (ccy), Obs (k) and Contract (c).

$$\begin{aligned}
\mathcal{M} &: c \rightarrow int \\
\mathcal{Q} &: int \rightarrow Contract \rightarrow int \\
\mathcal{M} &= \mathcal{Q}[\![c]\!]_0 \\
\mathcal{Q}[\![One(ccy)]\!]_i &= i \\
\mathcal{Q}[\![Scale(k, c)]\!]_i &= \mathcal{Q}[\![c]\!]_i \\
\mathcal{Q}[\![Give(c)]\!]_i &= \mathcal{Q}[\![c]\!]_i \\
\mathcal{Q}[\![Acquire\,(t, c)]\!]_i &= \mathcal{Q}[\![c]\!]_{t+i} \\
\mathcal{Q}[\![All\,([])]\!]_i &= i \\
\mathcal{Q}[\![All([c_1 \ldots c_n])]\!]_i &= max\,(\mathcal{Q}[\![c_1]\!]_i, \ldots, \mathcal{Q}[\![c_n]\!]_i) \\
\mathcal{Q}[\![Or(c_1, c_2)]\!]_i &= max\,(\mathcal{Q}[\![c_1]\!]_i, \mathcal{Q}[\![c_2]\!]_i)
\end{aligned}$$

Figure 6: Function $\mathcal{M}$ (maturity).

define

$$c_1 = Acquire(10, Scale(Underlying(\text{"}MSFT\text{"}, 0), One\ DKK)) \tag{19}$$

$$c_2 = Acquire(2, Scale(Value\ 10.0, One\ DKK)) \tag{20}$$

and show that having the function process the input $All[c_1, c_2]$ yields the result in (21).

$$\begin{aligned}
\mathcal{F}[\![All[c_1, c_2]]\!] &= \mathcal{K}[\![All[c_1, c_2]]\!]_0^{Some(1.0)} \\
&= Concat\left(\mathcal{K}[\![c_1]\!]_0^{Some(1.0)}, \mathcal{K}[\![c_2]\!]_0^{Some(1.0)}\right) \\
&= Concat\left(\mathcal{K}[\![Scale(Underlying("MSFT", 0), One\ DKK)]\!]_{10}^{Some1.0}, \mathcal{K}[\![c_2]\!]_0^{Some(1.0)}\right) \\
&= Concat\left(\mathcal{K}[\![One\ DKK]\!]_{10}^{None}, \mathcal{K}[\![c_2]\!]_0^{Some(1.0)}\right) \\
&= Concat\left([(10, Uncertain)], \mathcal{K}[\![c_2]\!]_0^{Some(1.0)}\right) \\
&= Concat\left([(10, Uncertain)], \mathcal{K}[\![Scale(Value\ 10.0, One\ DKK]\!]_2^{Some(1.0)}\right) \\
&= Concat\left([(10, Uncertain)], \mathcal{K}[\![One\ DKK]\!]_2^{Some(10.0)}\right) \\
&= Concat\left([(10, Uncertain)], [(2, Certain(10.0, DKK))]\right) \\
&= [(10, Uncertain), (2, Certain(10.0,\ DKK))] \qquad (21)
\end{aligned}$$

The $Causal$ flow constructor is used for the function $causal$. We define a flow to be causal if it contains at least one flow of the form

$$Acquire(t, Underlying("foo", d), One\ ccy) \quad d > 0 \tag{22}$$

that is, we do not know the flow at the time it is executed. Semantics for a solution is shown in 8. The function outputs $[t,\ Causal(d)]$ given the input in (22), assuming that d and t are integers and $d > 0$.

We also define the function $underlyings$. The purpose of this function is to return the underlyings and their time stamp which a contract depends on. The functions semantics are shown in Figure 9.

$$
\begin{aligned}
f &:= \mathit{Uncertain} \mid \mathit{Certain}(\mathbb{R} \times \mathit{currency}) \\
&\quad \mid \mathit{Choose}((\mathit{int} \times \mathit{flow})\ \mathit{List} \times (\mathit{int} \times \mathit{flow})\ \mathit{List}) \\
&\quad \mid \mathit{Causal}(\mathit{int}) \\
\mathcal{F} &: c \rightarrow (\mathit{int} \times \mathit{flow})\ \mathit{List} \\
\mathcal{K} &: \mathit{int} \rightarrow \mathbb{R}\ \mathit{Option} \rightarrow c \rightarrow (\mathit{int} \times f)\ \mathit{List} \\
\mathcal{F} &= \mathcal{K}[\![c]\!]_0^{Some(1.0)} \\
\mathcal{K}[\![One(ccy)]\!]_t^s &= \begin{cases} [(t, Uncertain)] & if\ s = None \\ [(t, Certain(s, ccy))] & if\ Some\ s \end{cases} \\
\mathcal{K}[\![Scale(Value\ a, c)]\!]_t^s &= \begin{cases} \mathcal{K}[\![c]\!]_t^{None} & if\ s = None \\ \mathcal{K}[\![c]\!]_t^{Some(s \cdot a)} & if\ Some\ s \end{cases} \\
\mathcal{K}[\![Scale(\_, c)]\!]_t^s &= \mathcal{K}[\![c]\!]_t^{None} \\
\mathcal{K}[\![Acquire(t', c)]\!]_t^s &= \mathcal{K}[\![c]\!]_{t+t'}^s \\
\mathcal{K}[\![Give(c)]\!]_t^s &= \begin{cases} \mathcal{K}[\![c]\!]_t^{None} & if\ s = None \\ \mathcal{K}[\![c]\!]_t^{Some(-s)} & if\ Some\ s \end{cases} \\
\mathcal{K}[\![All([c_1, c_2, \ldots c_n])]\!]_t^s &= Concat(map(\lambda x.\mathcal{K}[\![x]\!]_t^s\ [c_1 \ldots c_n])) \\
\mathcal{K}[\![Or(c_1, c_2)]\!]_t^s &= [t, Choose\,(\mathcal{K}[\![c_1]\!]_t^s, \mathcal{K}[\![c_2]\!]_t^s)]
\end{aligned}
$$

Figure 7: Function $\mathcal{F}$ (flows).

$$
\begin{aligned}
\mathcal{C} &: \mathit{Contract} \to (\mathit{int} \times \mathit{flow})\ \mathit{List} \\
\mathcal{L} &: \mathit{int} \to \mathit{int} \to s \to \mathit{Contract} \\
&\quad \to (\mathit{int} \times \mathit{flow})\ \mathit{List} \\
\mathcal{C} &= \mathcal{L}[\![c]\!]^{0}_{0,None} \\
\mathcal{L}[\![One(ccy)]\!]^{d}_{t,s} &= \begin{cases} [] & if\ s = None \\ [t, Causal(d)] & if\ Some\ s \end{cases} \\
\mathcal{L}[\![Scale(Underlying(\_, t'), c)]\!]^{d}_{t,s} &= \begin{cases} \mathcal{L}[\![c]\!]^{d+t'}_{t,Some(1.0)} & if\ t > 0 \\ \mathcal{L}[\![c]\!]^{d+t'}_{t,None} & if\ t \leq 0 \end{cases} \\
\mathcal{L}[\![Scale(\_, c)]\!]^{d}_{t,s} &= \mathcal{L}[\![t, d, None, c]\!] \\
\mathcal{L}[\![Acquire(t', c)]\!]^{d}_{t,s} &= \mathcal{L}[\![c]\!]^{d+t'}_{t+t',s} \\
\mathcal{L}[\![All\ ([c_1, \ldots c_n])]\!]^{d}_{t,s} &= Concat\left(map\left(\lambda x.\mathcal{L}[\![x]\!]^{d}_{t,s}\ [c_1, \ldots c_n]\right)\right) \\
\mathcal{L}[\![Give(c)]\!]^{d}_{t,s} &= \mathcal{L}[\![c]\!]^{d}_{t,s} \\
\mathcal{L}[\![Or(c_1, c_2)]\!]^{d}_{t,s} &= \mathcal{L}[\![c_1]\!]^{d}_{t,s}\ @\ \mathcal{L}[\![c_2]\!]^{d}_{t,s}
\end{aligned}
$$

Figure 8: Function $\mathcal{C}$ (causal).

4.2.2 Manipulating Contracts

The functions in this section serve the purpose of changing contracts.

We want to be able to *simplify* a contract. To explain the problem, let us start out with a few examples. Conceive that the user has the following contract in the portfolio

$$
\begin{aligned}
&\text{Acquire}(d, \\
&\quad \text{Scale}(\text{Value}(100.0), \\
&\quad\quad \text{Scale}(\text{Value}(200.0), \\
&\quad\quad\quad \text{Scale}(\text{Value}(300.0), One\ DKK))))
\end{aligned}
$$

This contract is equivalent to the following according to (15).

$$
\begin{aligned}
&\text{Acquire}(d, \\
&\quad \text{Scale}(\text{Value}(6.000.000.0), One\ DKK))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{U} &: c \to (\mathcal{S} \times int)\ List \\
\mathcal{Y} &: int \to k \to (\mathcal{S} \times int)\ \textbf{\textit{List}} \\
\mathcal{H} &: int \to c \to (\mathcal{S} \times int)\ \textbf{\textit{List}} \\
\mathcal{U} &= Distinct\,(\mathcal{H}[\![0, c]\!]) \\
\mathcal{Y}[\![Value(k)]\!]_t &= [] \\
\mathcal{Y}[\![Underlying(s, t')]\!]_t &= [s, t + t'] \\
\mathcal{Y}[\![k_1 \pm k_2]\!]_t &= \mathcal{Y}[\![k_1]\!]_t \mathbin{@} \mathcal{Y}[\![t, k_2]\!] \\
\mathcal{Y}[\![k_1 \times k_2]\!]_t &= \mathcal{Y}[\![k_1]\!]_t \mathbin{@} \mathcal{Y}[\![k_2]\!]_t \\
\mathcal{Y}[\![max(k_1, k_2)]\!]_t &= \mathcal{Y}[\![k_1]\!]_t \mathbin{@} \mathcal{Y}[\![k_2]\!]_t \\
\mathcal{H}[\![One(ccy)]\!]_t &= [] \\
\mathcal{H}[\![Scale(k, c)]\!]_t &= \mathcal{Y}[\![k]\!]_t \mathbin{@} \mathcal{H}[\![c]\!]_t \\
\mathcal{H}[\![All([c_1, \ldots c_n])]\!]_t &= Concat\,(map\,(\lambda x.\mathcal{H}[\![x]\!]_t\ [c_1, \ldots c_n])) \\
\mathcal{H}[\![Acquire(t', c)]\!]_t &= \mathcal{Y}[\![c]\!]_{t+t'} \\
\mathcal{H}[\![Give(c)]\!]_t &= \mathcal{Y}[\![c]\!]_t \\
\mathcal{H}[\![Or(c_1, c_2)]\!]_t &= \mathcal{Y}[\![c_1]\!]_t \mathbin{@} \mathcal{Y}[\![c_2]\!]_t
\end{aligned}
$$

Figure 9: function $\mathcal{U}$ (underlyings)

since $100 \cdot 200 \cdot 300 = 6.000.000$. Also, if we have

$$Acquire(d, Underlying(\text{"}foo\text{"}, d), One\ DKK) \quad d = 0$$

we should be able to solve the Acquire and Underlying cases since $d = 0$ for both constructors. Let us further assume that the stock "foo" has a market price of 100 USD today according to the environment E, the contract should be simplified to

$$
\begin{aligned}
&Acquire(d, Value(100.0), One\ DKK) \\
&= Scale(Value(100), One\ DKK)
\end{aligned}
$$

according to (16) We also want to be able to simplify contracts according to principles (14) and (17). To do this, we define two functions.

$simplifyObs$, which semantics are shown in Listing 10. The design is based on the following: we try to evaluate the observable input, which includes checking if we have the price for the underlying. If we catch an error, we try to simplify each observable recursively until we are not able to simplify further.

$simplify$, which semantics are shown in Listing 11. As expected, we have propagaters for $Scale$ and $Give$, and we remove $Scale$ cases that have an empty Obs or Contract. In addition, we check if it is possible to propagate the Scale constructor by multiplying them its observable values. For the give case, we also check for flows that cancel out with each other, according to (14). Other than that, we follow the logical principles stated in Section 3.5.

$$
\begin{aligned}
E &: (\mathcal{S} \times int) \to \mathbb{R} + \bot \\
g &: \mathbb{R} \to \mathbb{R} \to \mathbb{R} \\
\pi &: k \times k \to k \\
\Pi &: g \to \pi \to k \to k \to k \\
\mathcal{B} &: E \to k \to k \\
\Pi[\![k_1, k_2]\!]^{\pi}_{g} &= \begin{cases} Value(g(r_1, r_2)) & if\ (\mathcal{B}[\![k_1]\!]_E, \mathcal{B}[\![k_2]\!]_E) = (Value(r_1), Value(r_2)) \\ \pi(k_1, k_2) & if\ (\mathcal{B}[\![k_1]\!]_E, \mathcal{B}[\![k_2]\!]_E) = (k_1, k_2) \end{cases} \\
\mathcal{B}[\![Value(r)]\!]_E &= Value(r) \\
\mathcal{B}[\![Underlying(s,t)]\!]_E &= \begin{cases} Value(r) & if\ r = E(s,t) \\ Underlying(s,t) & if\ r = \bot \end{cases} \\
\mathcal{B}[\![k_1, k_2]\!]_E &= \Pi[\![k_1, k_2]\!]^{(\lambda x,y.x*y)}_{Mul} \\
\mathcal{B}[\![k_1, k_2]\!]_E &= \Pi[\![k_1, k_2]\!]^{(\lambda x,y.x+y)}_{Add} \\
\mathcal{B}[\![k_1, k_2]\!]_E &= \Pi[\![k_1, k_2]\!]^{(\lambda x,y.x-y)}_{Sub} \\
\mathcal{B}[\![k_1, k_2]\!]_E &= \Pi[\![k_1, k_2]\!]^{max}_{Max}
\end{aligned}
$$

Figure 10: Function $\mathcal{B}$ (simplifyObs).

Now, let us move on to advancing contracts. To advance a contract means to move it d days forward in time. As an example, conceive that today we have the following option in our portfolio

```
Acquire(1,
  Scale(Max(Value 10.0,
    Sub(Underlying("ALMB", 0), Value 100.0)), One DKK))
```

$$
\begin{aligned}
\mathcal{V} &: E \to c \to c \\
\mathcal{X} &: \mathit{int} \to E \to c \to c \\
\mathcal{P} &: \mathit{int} \to E \to c \to c \\
\mathcal{V} &= \mathcal{X}\llbracket c \rrbracket_E^0 \\
\mathcal{X}\llbracket \mathit{All}([c]) \rrbracket_t^E &= \mathcal{X}\llbracket c \rrbracket_t^E \\
\mathcal{X}\llbracket \mathit{All}([c_1, c_2, \dots c_n]) \rrbracket_t^E &= \mathit{All}\left(\left[\mathcal{X}\llbracket c_1 \rrbracket_t^E, \mathcal{X}\llbracket c_2 \rrbracket_t^E \dots \mathcal{X}\llbracket c_n \rrbracket_t^E \right] \right) \\
\mathcal{X}\llbracket \mathit{One}(c) \rrbracket_t^E &= c \\
\mathcal{X}\llbracket \mathit{Acquire}(t_1, \mathit{Acquire}(t_2, c)) \rrbracket_t^E &= \mathcal{X}\llbracket \mathit{Acquire}(t_1 + t_2, c) \rrbracket_t^E \\
\mathcal{X}\llbracket \mathit{Acquire}(t', c) \rrbracket_t^E &= \begin{cases} \mathcal{X}\llbracket c \rrbracket_{t+t'}^E & \textit{if}\, t' \leq 0 \\ \mathit{Acquire}\left(t', \mathcal{X}\llbracket c \rrbracket_{t+t'}^E\right) & \textit{otherwise} \end{cases} \\
\mathcal{X}\llbracket \mathit{Scale}(k, c) \rrbracket_t^E &= \mathcal{P}\left\llbracket \mathit{Scale}\left(\mathcal{B}\llbracket t, k \rrbracket_t^E, \mathcal{X}\llbracket c \rrbracket_t^E \right) \right\rrbracket_t^E \\
\mathcal{P}\llbracket \mathit{Scale}(k_1, \mathit{Scale}(k_2, c)) \rrbracket_t^E &= \mathit{Scale}\left(\mathcal{B}\left\llbracket \mathit{Mul}(\mathcal{B}\llbracket k_1 \rrbracket_t^E, \mathcal{B}\llbracket k_2 \rrbracket_t^E) \right\rrbracket_t^E, c \right) \\
\mathcal{P}\llbracket \mathit{Scale}(k, \mathit{All}([])) \rrbracket_t^E &= \mathit{All}\,([]) \\
\mathcal{P}\llbracket \mathit{Scale}(k, c) \rrbracket_t^E &= \begin{cases} \mathit{All}\,([]) & \textit{if}\, \mathcal{B}\llbracket k \rrbracket_t^E = Value(0.0) \\ \mathit{Scale}(\mathcal{B}\llbracket k \rrbracket_t^E, c) & \textit{otherwise} \end{cases} \\
\mathcal{P}\llbracket \_ \rrbracket_t^E &= \mathit{Scale}\left(\mathcal{B}\llbracket k \rrbracket_E, c\right) \\
\mathcal{X}\llbracket \mathit{Give}(c) \rrbracket_t^E &= \begin{cases} \mathcal{X}\llbracket c \rrbracket_t^E & if\ Give(\mathcal{X}\llbracket c \rrbracket_t^E) = Give(Give(c')) \\ Give(\mathcal{X}\llbracket c \rrbracket_t^E) & otherwise \end{cases} \\
\mathcal{X}\llbracket Or(c_1, c_2) \rrbracket_t^E &= \mathit{Or}\left(\mathcal{X}\llbracket c_1 \rrbracket_t^E, \mathcal{X}\llbracket c_2 \rrbracket_t^E \right)
\end{aligned}
$$

Figure 11: Function $\mathcal{V}$ (simplify).

We want to update the contract as days go by. Assume that now one day has gone by and the price of the stock $ALMB$ today is 150. Then the contract should update to

```
Acquire(0,
  Scale(Max(Value 0.0,
    Sub(Underlying("ALMB", 0), Value 100.0)), One DKK))
↓
Scale(Max(Value 0.0,
  Sub(Value 150.0, Value 100.0)), One DKK)
↓
Scale(Value 50.0, One DKK)
```

The *advance* function is responsible for doing the first step: that is, subtracting the relative date of acquisition d. The simplify function is responsible for doing the last steps: that is simplifying the contract after advancement. A possible design for this function is shown in Listing 12, where we advance the contract by a helper function $\mathcal{F}$ and then run $simplify$ on the output of $\mathcal{F}$.

$$
\begin{aligned}
\mathcal{A} &: ((\mathit{string} \times \mathit{int}) \to \mathbb{R} + \bot) \to \mathit{int} \to \mathit{Contract} \to \mathit{Contract} \\
\mathcal{F} &: \mathit{int} \to \mathit{Contract} \to \mathit{Contract} \\
\mathcal{A}[\![c]\!]_d^E &= \mathcal{V}[\![\mathcal{F}[\![c]\!]_d^E]\!]_E \\
\mathcal{F}[\![\mathit{One}(\mathit{ccy})]\!]_d^E &= \mathit{One}(\mathit{ccy}) \\
\mathcal{F}[\![\mathit{Scale}(k, c)]\!]_d^E &= \mathit{Scale}(k, \mathcal{F}[\![c]\!]_d^E) \\
\mathcal{F}[\![\mathit{All}\,([c_1, \dots c_n])]\!]_d^E &= \mathit{All}\left(\mathit{map}\left(\lambda x.\mathcal{F}[\![x]\!]_d^E\ [c_1, \dots c_n]\right)\right) \\
\mathcal{F}[\![\mathit{Acquire}(t, c)]\!]_d^E &= \mathit{Acquire}\left(t - d, \mathcal{F}[\![c]\!]_d^E\right) \\
\mathcal{F}[\![\mathit{Give}(c)]\!]_d^E &= \mathcal{F}[\![c]\!]_d^E \\
\mathcal{F}[\![\mathit{Or}(c_1, c_2)]\!]_d^E &= \mathit{Or}\left(\mathcal{F}[\![c_1]\!]_d^E, \mathcal{F}[\![c_2]\!]_d^E\right)
\end{aligned}
$$

Figure 12: Function $\mathcal{A}$ (advance).

To end this section we present the design of the *choose* function in Listing 13. Here, the notation $((-) \circ f)$ means applying the function $(-)$ to f. Recall that a contract $Or(c_1, c_2)$

is a contract where the holder must immediately make a choice either have c_1 or c_2 in the portfolio. The $choose$ function takes as input a valuation function f that calculates the value of a contract. We then solve the Or constructor by applying f to the contract inputs in it, and return the contract which has the largest payoff. If f returns $\perp$ for any of the inputs c_1 and c_2, we return $Or(c_1, c_2)$ and thus it is not solved. This may happen if the contract contains causal flows and f does not include prediction of future valuation of contracts, or in general if f does not have the necessary information to compute the valuation of the contract. In section 7.2, we show how this function can be utilized to make a choice on an option, based on the payoff function.

$$
\begin{aligned}
\mathcal{Z} &:: (c \to \mathbb{R}) \to c \\
\mathcal{Z}[\![One(ccy)]\!]_f &= One(ccy) \\
\mathcal{Z}[\![Scale(o, c)]\!]_f &= Scale\,(o, \mathcal{Z}[\![c]\!]_f) \\
\mathcal{Z}[\![All([c_1, \dots c_n])]\!]_f &= All\,(map\,(\lambda x.\mathcal{Z}[\![x]\!]_f\ [c_1, \dots c_n])) \\
\mathcal{Z}[\![Acquire(t, c)]\!]_f &= \begin{cases} Acquire\,(t, \mathcal{Z}[\![c]\!]_f) & if\ t = 0 \\ Acquire\,(t, c) & if\ t \neq 0 \end{cases} \\
\mathcal{Z}[\![Give(c)]\!]_f &= Give\left(\mathcal{Z}[\![c]\!]_{((-)\circ f)}\right) \\
\mathcal{Z}[\![Or(c_1, c_2)]\!]_f &= \begin{cases} Or(\mathcal{Z}[\![c_1]\!]_f, \mathcal{Z}[\![c_2]\!]_f) & if\ (f(\mathcal{Z}[\![c_1]\!]_f), f(\mathcal{Z}[\![c_2]\!]_f)) = \perp \\ \mathcal{Z}[\![c_1]\!]_f & if\ f(\mathcal{Z}[\![c_1]\!]_f) \geq f(\mathcal{Z}[\![c_2]\!]_f) \\ \mathcal{Z}[\![c_2]\!]_f & if\ f(\mathcal{Z}[\![c_1]\!]_f) < f(\mathcal{Z}[\![c_2]\!]_f) \end{cases}
\end{aligned}
$$

Figure 13: function $\mathcal{Z}$ (choose)

4.3 Implementation

All the functions presented in this section have been implemented in $F\#$ and can be found in the file *src/Management.fs*. We perform testing in section 6.2. A demonstration of the usefulness of a selected set of the functions are presented in section 7.2. In addition, the *underlyings* function is utilized when simulating the value of a contract in section 5.3.

5 Evaluating Contracts

In this section, we present a way of $evaluating$ contracts, that is, valuing them. We start by exploring related work in the field in section 5.1, where we introduce the literature that have shaped the pricing functions created. We make sure that the logic principles presented in section 3.5 holds by introducing evaluation functions in 5.2. In this section, we also show an approach for discretization of the continuous time finance models presented in 2.2 so that we can simulate the value of a contract which has as stock as an underlying. In 5.3, we present an implementation of the evaluation and simulation functions and combine them to price a European call option, and show that the value we get indeed is correct according to the Black Scholes formula.

5.1 Related Work

In the context of evaluating contracts, the work done in (Peyton et al., 2000) on $processing$ contracts will serve as a primary source of inspiration for the design phase of this section.

In terms of financial theory and pricing the underlying asset of a derivative, notable resources include (Hull, 2018) and (Shreve, 2004), which both offer a introduction and in-depth explanation of valuing financial assets and derivatives.

Additionally, our simulation technique will utilize Monte Carlo Methods. In this regard, (Glasserman, 2003) and (Werk et al., 2012) will be of inspiration as to how to simulate the price of a derivative on a computer.

5.2 Design

5.2.1 Evaluation Functions

The evaluation functions enable the user to evaluate financial contracts and observables when defined. This will act as our basis for valuing derivatives.

Listing 14 illustrates the evaluation process for the Currency data type. This function takes as input a exchange rate function and the currency to evaluate and provides a real number as output, representing the exchange rate of that currency in relation to USD.

In Listing 15, the evaluation process for the Observable data type is illustrated. The purpose of this function is to perform arithmetic operations on observables or return a specific asset price. The evaluation function takes as input an operation to the input variable, along with a function E which has the same definition as in Listing 10, that is, it is a environment and accepts a string representing a asset symbol and an integer indicating the date to fetch the corresponding asset price. The evaluation returns the observable as a floating-point value. In addition, we extend the standard arithmetic operators to be able to handle $\perp$. If one of the inputs are $\perp$, the function returns a $\perp$. This is relevant because the E function may not be able to fetch the asset price.

The evaluation process for the Contract data type is shown in Listing 16. This function takes as input a Contract to evaluate the price of, a discount function I, and the function E known from the Observable evaluation function. The discount function I is provided by the user, and it is used to calculate the present value of a flow. Notice here that the time of the stock price is 'shoot' in time according to t in the $Acquire$ constructor. This is due to what we presented earlier when expressing contracts in the DSL: If we have $Acquire(10, Underlying("AAPL", 0))$, we wish to evaluate the stock price at time 10, not at time 0. Important, however, is that this is not the case with the interest rate. This function should not be shoot in time according to time t. Conceive that we have the following contract in the portfolio

Acquire(10, Acquire(2, Scale(Value 100.0, One USD)))

Then if we made a similar anonymous function for I and shot it in time according to t, then we would get the price $I(10) \cdot I(12) \cdot (100.0\ USD)$. However, this is wrong: then we would be discounting with time 22 when the flow is executed at time 12.

$$\mathcal{C} : (ccy \to \mathbb{R}) \to ccy \to \mathbb{R}$$
$$\begin{aligned} C[\![USD]\!]_f &= 1.0 \\ C[\![EUR]\!]_f &= f(EUR) \\ C[\![GBP]\!]_f &= f(GBP) \\ C[\![DKK]\!]_f &= f(DKK) \end{aligned}$$

Figure 14: Evaluation of currencies

$$
\begin{aligned}
\Omega : E \to k \to \mathbb{R} + \bot \\
\Omega[\![Value(k)]\!]_E &= k \\
\Omega[\![Underlying(s,n)]\!]_E &= E(s,n) \\
\Omega[\![k_1 \times k_2]\!]_E &= \Omega[\![k_1]\!]_E \times \Omega[\![k_2]\!]_E \\
\Omega[\![k_1 + k_2]\!]_E &= \Omega[\![k_1]\!]_E + \Omega[\![k_2]\!]_E \\
\Omega[\![k_1 - k_2]\!]_E &= \Omega[\![k_1]\!]_E - \Omega[\![k_2]\!]_E \\
\Omega[\![max(k_1,k_2)]\!]_E &= max\left(\Omega[\![k_1]\!]_E, \Omega[\![k_2]\!]_E\right)
\end{aligned}
$$

Figure 15: Evaluation of observables

$$
\begin{aligned}
I &: int \to \mathbb{R} \\
f &: ccy \to \mathbb{R} \\
\Sigma &: I \to E \to f \to c \to \mathbb{R} \\
\Sigma[\![One(ccy)]\!]^I_{E,f} &= \mathcal{C}[\![ccy]\!]_f \\
\Sigma[\![Scale(k,c)]\!]^I_{E,f} &= \Omega[\![k]\!]_E \times \Sigma[\![c]\!]^I_{E,f} \\
\Sigma[\![Acquire(t,c)]\!]^I_{E,f} &= I(t) \times \Sigma[\![c]\!]^I_{(\lambda(s,m).E(s,m+t)),f} \\
\Sigma[\![All\,([])]\!]^I_{E,f} &= 0.0 \\
\Sigma[\![All\,([c_1,\ldots,c_n])]\!]^I_{E,f} &= \Sigma[\![c_1]\!]^I_{E,f} + \ldots + \Sigma[\![c_n]\!]^I_{E,f} + 0, n \geq 1 \\
\Sigma[\![Give(c)]\!]^I_{E,f} &= -\Sigma[\![c]\!]^I_{E,f} \\
\Sigma[\![Or\,(c_1,c_2)]\!]^I_{E,f} &= max\left(\Sigma[\![c_1]\!]^I_{E,f}, \Sigma[\![c_2]\!]^I_{E,f}\right)
\end{aligned}
$$

Figure 16: Evaluation of contracts

The only stochastic part of the DSL as of now are the asset prices, that is the Underlying constructor, or the environment E. Thus, we are only interested in simulating a contract if it is dependent on the price of one or more stochastic underlyings: otherwise, it should be sufficient to evaluate it once. To simulate the contract, we consider the design of concepts from mathematical finance presented in section 2.2.

5.2.2 Wiener Processes

We want to design the Wiener Process presented in 2.2 so that we can simulate it. However, anyone that has worked with simulation on a computer will know that we are not able to fulfill the fourth requirement of a the process: that is, it has continuous trajectories. In definition, the wiener process can move in an infinitely small amount of time, but on a computer we are only able to approximate this by taking discrete steps in time. Therefore, when modelling this process, we simulate it with time increments rather than its continuous trajectories.

Thus, we narrow the wiener process down to the following discretized function. Let $t > 0$ and $\Delta t \geq 0$. We define the iterative process

$$W_0 = 0 \tag{23}$$

$$W_{t+\Delta t} = W_t + \mathcal{N}_{t+\Delta t}\sqrt{\Delta t} \tag{24}$$

where $\mathcal{N}_1, \mathcal{N}_2, ... \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1)$ and Δt is a time increment of our choice (Werk et al., 2012, p. 93). Now, let us check that this function is a valid approximation for a Wiener Process by checking the properties listed earlier. The property $W_0 = 0$ is satisfied and follows from the definition. It is evident that property 2 follows as well. The increments of the process between time points cannot be dependent since they are only dependent on normal distributions that are independent, that is $\mathcal{N}_1, \mathcal{N}_2, ... \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1)$. It follows that for $t > s$, we have $W_t - W_s \sim N(0, t-s)$, which corresponds to property 3. This is easy to see due to the fact that $\sqrt{\Delta t}\mathcal{N} = N(0, \Delta t)$ and Δt is the time between t and s.

It is therefore concluded that the iterative process defined in (23) and (24) possess the properties of the wiener process at the discrete time increments. The process is plotted in Figure 17.

5.2.3 Pricing Options

When pricing underlying stocks, we will assume that a stock price follows the GBM in (1). Since the design of the Wiener Process has been presented, we are now able to simulate this process. Recall that the payoff of a Call Option is $(S_T - K, 0)^+$. We substitute in (1) and get the payoff function

$$C(T) = \left(S_0 e^{\left(u - \frac{1}{2}\sigma^2\right)T + \sigma W_T} - K\right)^+ \tag{25}$$

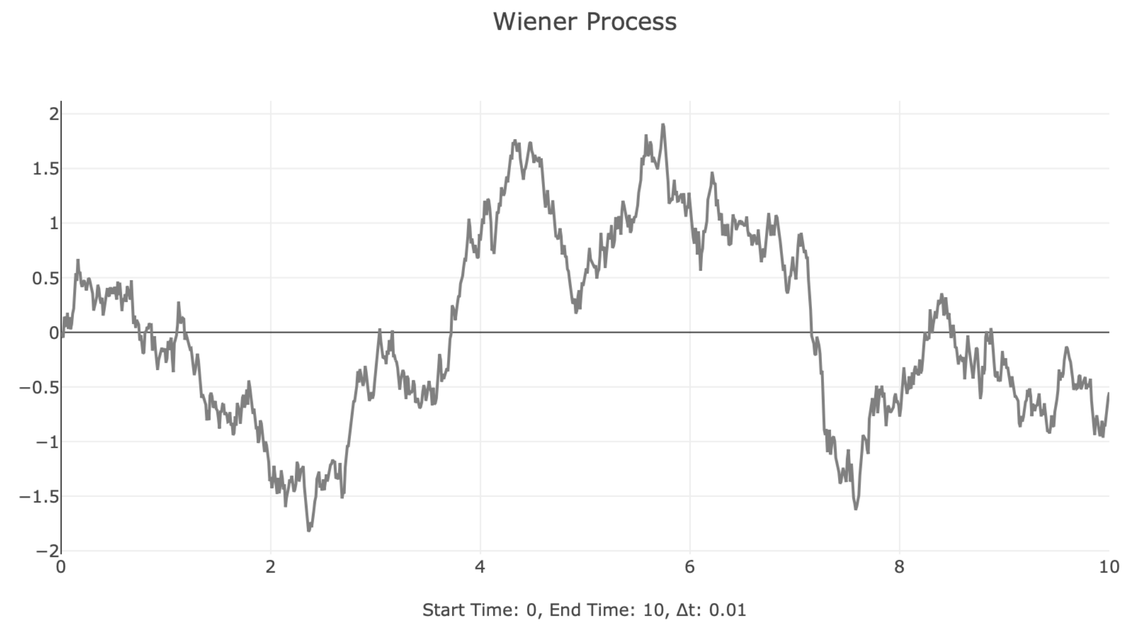

Figure 17: A simulation of a Wiener Process

where we have $\Delta t = T$ since the option is not path-dependent, so we have $W_T = \sqrt{T}\mathcal{N}$ by (24).

Now, the simulation process becomes evident. We do as in (Glasserman, 2003, p. 5), that is we simulate the stocks price at T, doing so for a substantial number of iterations, denoted as N. For each iteration, we compute the expected payoff of the call option at maturity. Subsequently, each of these payoffs are discounted to its present value using the discount function I, and we take the average of the simulations

$$V_{Call}(T) = I(T) \times \frac{1}{N}\sum_{i=1}^{N} C_i(T) \tag{26}$$

In the general case we have

$$V(T) = I(T) \times \frac{1}{N}\sum_{i=1}^{N} f\left(\begin{bmatrix} S_i^0(T) \\ S_i^1(T) \\ \vdots \\ S_i^k(T) \end{bmatrix}\right) \tag{27}$$

where f is the payoff function and k denotes a specific stock. This is the essence of Monte Carlo simulation (Glasserman, 2003, p. 5).

5.2.4 The Path-Dependent Case

When simulating a non path-dependent option, we are lucky enough to only need the price of the stock at the maturity date. However, for options like an Asian, which payoff is the average of the underlyings price in a specific time interval, we must simulate the whole stock trajectory. We can do this by redefining the GBM as in (Glasserman, 2003, p. 8) so that we have

$$S_{t+1} = S_t \cdot e^{\left(\mu - \frac{1}{2}\sigma^2\right)\Delta t + \sigma \mathcal{N} \Delta t} \tag{28}$$

Then we can simulate the price option which path depends on the price from t to T by utilizing monte carlo simulation in the following way

$$V(t,T) = I(T) \times \frac{1}{N} \sum_{i=1}^{N} f\left(S_i(t), S_i(t+\Delta t), \ldots, S_i(T)\right) \tag{29}$$

where $f(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}$ for the asian option. Generally, we can also do this with multiple stocks as seen below

$$V(t,T) = I(T) \times \frac{1}{N} \sum_{i=1}^{N} f\left(\begin{bmatrix} S_i^1(t), S_i^1(t+\Delta t), \ldots, S_i^1(T) \\ S_i^2(t), S_i^2(t+\Delta t), \ldots, S_i^2(T) \\ \vdots \\ S_i^k(t), S_i^k(t+\Delta t), \ldots, S_i^k(T) \end{bmatrix}\right) \tag{30}$$

An example of the simulation process for the Asian Option is shown in Figure 18. Here, we plot traces of the GBM presented in (28) along with the payoff function for the option.

5.3 Implementation

In our DSL, we facilitate the execution of the simulation process and follow the established simulation procedure. The implementation of the evaluation functions can be found in the file $src/Evaluations.fs$.

We begin by simulating every underlying that the contract depends on and then evaluate the contract using the evaluation functions provided in 5.2.1. To do this, we must first receive the name of the stocks the time at which their respective prices impact value of the contract. We do this by utilizing the $underlyings$ function defined in Figure 9.

We implement the Wiener Process and the GBM described in equation (25), by utilizing the $FSharp.Stats$ library to generate i.i.d. normal random variables. We define a function

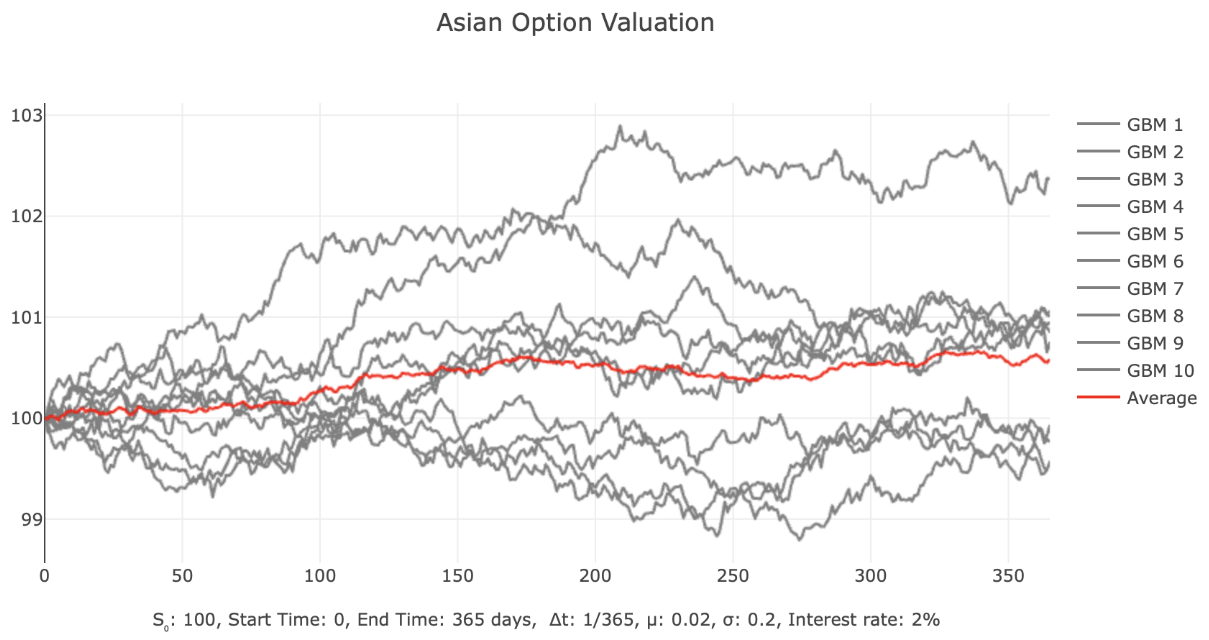

Figure 18: Pricing an Asian Option

$simStock$, which retrieves the needed stocks parameters from a XML file. We also define the function

$$\text{makeE} :: (\text{string} \times \text{int}) \text{ list} \rightarrow \text{Map<(string} \times \text{int), float>}$$

in $F\#$, which takes as input a list of stock names and a time for which we need to retrieve the price of the respective underlying and simulates the stock prices we need to price the contract. The implementations for these functions can be found in the file $src/Simulations.fs$. The final simulation function is shown in Listing 6.

We can give this function the second European call option defined in Listing 4 as input and perform the simulation. Let us assume a constant interest rate $r = 0.02$, a discount function $I(T) = e^{-r\frac{T}{365}}$ where T is in days and the the parameters for the AAPL stock to be $S_0 = 100$, $\mu = 0.02$, $\sigma = 0.05$. We get the output $5.05...$ as seen in Listing 7.

To consider if this is a valid output, we calculate the expected value of the call option, that is $e^{-rT}E[(S_T - K)^+]$. This expectation can be computed using the Black Scholes formula for a call option, defined as

$$BS(S, \sigma, T, r, K) = S\Phi\left(d_1(S, \sigma, T, r, K)\right) - e^{-rT}K\Phi\left(d_2(S, \sigma, T, r, K)\right) \tag{31}$$

```
let simulateContract (sims : int) (c : Contract) : float =
    let underlyings : (string * int) list = underlyings c
    let evaluations : float list =
        [for _ in 1..sims ->
            let resultMap = makeE underlyings
            let E(s,t) : float = Map.find(s, t) resultMap
            let res = evalc f I E c
            res]
    evaluations |> List.average
```

Listing 6: Simulating the price of a contract

```
// European call with T = 10 days, underlying APPL, K = 95 and
// payment in USD
let ec1 = europeanCall2 10 "AAPL" 95. USD
simulateContract 100_000 ec1
> 5.051929509
```

Listing 7: Pricing a European Call Option

where $d_1 = \frac{log(S/K)+\left(r+\frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}$, $d_2 = \frac{log(S/K)+\left(r-\frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}$ and Φ is the standard cumulative distribution (Glasserman, 2003, p. 5). We can supply this formula with the input variables of the call option where $T = \frac{10}{365}$ since the input T in the BS model is numerated in years and our DSL use days. We obtain

$$BS(100, 0.05, \frac{10}{365}, 0.02, 95) = 5.052041... \tag{32}$$

calculated by hand. This result is equal to the value obtained by simulation by 2 decimals, so we consider this a valid result. Further tests are done in the next section of the thesis.

As of now, we do not have a constructor to define path-dependent options, and thus we will not be able to simulate the Asian option. However, the process in (28) have been implemented and can be found in the file $src/Simulations.fs$. Thus, all we need is a new constructor to be designed and implemented. This is left for future work.

6 Testing

What we want to achieve with this section is to make sure that implementations in 4 and 5 are working as intended and try to find mistakes where they may occur. To do this, we have conducted unit testing where it is possible. We use black box testing, that is, we meticulously test each component of the program independently to ensure that every feature performs as expected. We do this by doing a through testing of each function for expected outputs. We have created a separate *.NET* project for this, which can be found in the Tests folder. The *README* file explains how to run every test in this section. In total, we have 175 test cases in which 175 passed and 0 failed.

In section 6.1 we introduce the unit testing framework that will be used to perform black box testing. 6.2 provides a summary of the tests performed for contract management. 6.3 gives an explanation of how the evaluation functions and simulation implementation were tested.

6.1 Introduction to xUnit and FsUnit

The chosen framework is xUnit, which, in combination with the FsUnit library, allows for idiomatic support. To introduce the framework, let us start with an example of testing a simple function as seen in Listing 8. Here, the square function is just a simple mathematical operation

```
let square(x : float)
    = x**2.0
[<Theory>]
[<InlineData(1.0, 1.0)>]
[<InlineData(5.0, 25)>]
[<InlineData(-5.7, 32.49)>]
let ``test square``(input : float) (expected : float) : unit =
    square(input) |> should (equalWithin 1e-7) expectedOutput
```

Listing 8: Testing the x^2 function in xUnit with FsUnit support

that takes a floating-point number as input and outputs the square of that number. The ‘Test square function‘ is a test method which takes two floats as input: the input to the function and the expected output of the function. It then runs the function on the input and checks if the output of that function matches with the expected output. This is done with help of the FsUnit library. In more depth, the result of the square function is passed to the 'should' function from this library, using the forward pipe operator $| >$. The *equalWithin* function is applied to specify the tolerance level, which in this case is 10^{-7}.

The test case is defined with the <*Theory*> attribute, which indicates that it is a parameterized test. This allows us to run the same test for different inputs and outputs.

The <*InlineData*> attribute is used to provide multiple sets of input and expected output values for the test. Each set of values represent a separate test case for the parameterized test.

One limitation that was encountered with the InlineData is that it is only designed to accommodate input values of primitive types. When working with custom data types such as Currency, Observable or Contract, this attribute may not be sufficient. Thus, when working with these types, we will be using the MemberData attribute, which enables the use of an object list as input. This object list may contain custom data types at its indices, allowing for the accommodation of these custom data types and objects as input values in parameterized tests.

6.2 Contract Management

In this section, we present independent and separated tests of the functions in Section 4.

The function $maturity$ was tested with the code shown in Listing 9. As seen in the code, the test case lists are defined as an input parameter for the MemberData attribute. The obj[] type in the lists represents an array of object values, which are passed as arguments to the functions test methods. Other than that, the test works exactly as with the InlineData attribute in the $square$ function example.

We have similar tests for every other function in the section, which can be found in the file $Tests/Management$.

All 42 test cases passed and no errors were uncovered.

```
let maturityTestCases : List<obj[]> =
    [   [| Acquire(10, Scale(Underlying("AAPL", 2), One USD)); 10 |]
        [| europeanCall2 17 "DIKU" 100.0 DKK ; 17 |]
        [| ccSwap 10 DKK USD 0.7 0.3 0.2 0.1 0.5; 10 |]
        [| Give(Acquire(10, Acquire(10, One DKK))); 20 |]
        [| Or(Acquire(10, One DKK), Acquire(20, One DKK)); 20 |]
    ]
[<Theory>]
[<MemberData(nameof(maturityTestCases))>]
let ``Assert maturity date to contract``(c: Contract, expected : int) =
    maturity c |> should equal expected
```

Listing 9: Testing the $maturity$ function.

6.3 Pricing

6.3.1 Currency Evaluation

We test the currency evaluation function with 8 test cases which can be found in the file $Tests/EvaluationTests/currencyTests$. Here, we define two functions, f and g, and evaluate them on 4 different currencies and make sure that the $evalccy$ function evaluates the input according to these functions when they are given as input.

This is a rather small test, but all we need to do here is to make sure the evalccy function is evaluating exchange rates according to which exchange rates the user decides from the exchange rate function. The test cases can be found in the file *Tests/EvaluationTests/contractTests*.

All 8 test cases passed, and no errors were uncovered.

6.3.2 Obs Evaluation

Moving on to observable evaluation testing, the same features from xUnit and FsUnit were used as in the currency evaluation. We conduct 68 test cases, where we first test each constructor independently and then combine them. The test cases can be found in the file file *Tests/EvaluationTests/ObsTests*. We get 68 passed tests and 0 failed.

6.3.3 Simulations

The tests in this section can be found in the file $Tests/SimulationTests$.

In order to test the simulation of contracts, we must first test that the stochastic processes that we have implemented in the program are working as expected. To test if the implementation of the wiener process is correct, we test property 3 of the Wiener Process: that is, the increments of the process are normally distributed so that $W(t) - W(s) \sim \mathcal{N}(0, t - s)$ for $s < t$. We do this by the following:

1. Simulate the Wiener Process N times from 0 to T with step size dt, resulting in a matrix $\boldsymbol{X} \in \mathbb{R}^{N \times \frac{T}{dt}}$
2. For every row in $\boldsymbol{X}$, we subtract the prior value resulting in a new matrix $\boldsymbol{Y} \in \mathbb{R}^{N \times \left(\frac{T}{dt} - 1\right)}$
3. For every column in $\boldsymbol{Y}$, we compute the distribution of the simulated values at each time step, resulting in the matrix $\boldsymbol{Z} \in (\mathbb{R} \times \mathbb{R})^{1 \times \left(\frac{T}{dt} - 1\right)}$, where $\mathbb{R} \times \mathbb{R}$ denotes a 2-tuple of floats. The the tuple entries indicate the average and variance of the simulations at each time step.
4. For each tuple in $\boldsymbol{Z}$, we check that it has the form $(0, dt)$, corresponding to the distribution $N(0, dt)$. This distribution corresponds to property 3, since $t - s = dt$ if s is the time step after t and dt is the size of the increments.

We compute 5 different tests for the Wiener Process with different end times and time increments. For smaller increments we set $N = 100.000$, and for larger increments we have $N = 1.000.000$. We only test of values where T is a multiplier of dt since this is a requirement. All tests are passed with a tolerance of 10^{-2} on the distribution tests.

For the path-dependent GBM, we can utilize the fact that we have shown in (10) that $E[S_t] = S_0 e^{t\mu}$. We simulate $N = 100.000$ GBM paths for different inputs it and check for each $t \in [0, T]$ that we have gotten the expected average by with tolerance 10^{-2}. For 4 different tests, no errors were uncovered.

We test the non-path dependent GBM in a similiar way: We simulate the GBM N times at time T, compute the average and then compute the expected value $E[S_T]$. Then, we compare the results with a tolerance of 10^{-2}. For 4 different tests, no errors were uncovered.

In total, we get 13 passed tests in this section and 0 failed.

6.3.4 Contract Evaluation

The tests in this section can be found in the file $Tests/EvaluationTests/ContractTests$.

To test the contract evaluation function, we evaluate contracts that can be priced analytically and compare the results. To do this, we begin by defining a dummy currency evaluation function and environment seen in Listing 10, and assume a interest rate of 0.02.

```
let cur (c : Currency) : float =
    match c with
    | USD -> 1.0
    | EUR -> 1.10
    | GBP -> 1.24
    | DKK -> 0.15

let testE (s: string, t: int) =
    match s, t with
    | "AAPL", 0 -> 180.42
    | "GOOG", 0 -> 129.44
    | "MSFT", 0 -> 333.61
    | _ -> failwith "price not found"
```

Listing 10: Dummies for currencies and stock prices.

For each constructor, we check that they yield the correct evaluation of the contract according to the expected value. In addition, we check that if we try to evaluate a contract which depends on a underlying which we do not know the price of, the evaluation function returns $\perp$.

Also, we check that **every** logic principle presented in Section 3.5 holds for our evaluation functions.

Moving on to testing the $simulateContract$ function, we begin by checking that it yields the same result as the evalc function for contracts that are non-stochastic; that is, they do not depend on an underlying. The next step involved simulating the acquisition of contracts that have an underlying and check that the solution was correct according to the expected value of the GBM.

As an extra test, we check that our simulation method is able to replicate the Put-Call Parity, which states that the the following always holds for non-arbitrage markets

$$\begin{aligned} C(T,S_T)+Ke^{-rT} &= P(T,S_T)+S_0 \qquad (33) \\ C(T,S_T)+Ke^{-rT}-P(T,S_T)-S_0 &= 0 \end{aligned}$$

where C is the price of the call at time T, P is the price of the put, K is the strike of both options and S_0 is the current price of the stock that both options have as an underlying (Hull, 2018, p. 242). We do this by defining a function which expresses a put option with the same definition as the second European option in Listing 4 but where the choice lies between $K - S_T$ and 0. The definition of the put be found in the $src/Instruments.fs$ file. We simulate both options and check if (33) holds.

In total, 40 test cases were covered in this section. Every test passed.

7 Evaluation of the DSL

In this section, we evaluate the DSL and demonstrate that it lives up to the objectives of the thesis, which are stated in section 1. We begin by showing in section 7.1 that the logical principles listed in section 3.5 holds. In section 7.2, we show that the $F\#$ implementations in sections 3, 4 and 5 can be combined to express, manage and price selected contracts.

7.1 Logical Principles

Let us start this section by checking whether the logical principles we outlined in 3.5, hold according to the design of the evaluation functions we have implemented. Beginning with (12), we can see that the inequality holds since

$$\begin{aligned} \Sigma[\![Give(Or(c_1,c_2))]\!]^I_{E,f} &= -\Sigma[\![Or(c_1,c_2)]\!]^I_{E,f} \\ &= -max(\Sigma[\![c_1]\!]^I_{E,f}, \Sigma[\![c_2]\!]^I_{E,f}) \qquad (34) \end{aligned}$$

and

$$\begin{aligned} \Sigma[\![Or(Give(c_1),Give(c_2))]\!]^I_{E,f} &= max\left(\Sigma[\![Give(c_1)]\!]^I_{E,f}, \Sigma[\![Give(c_2)]\!]^I_{E,f}\right) \\ &= max\left(-\Sigma[\![c_1]\!]^I_{E,f}, -\Sigma[\![c_1]\!]^I_{E,f}\right) \qquad (35) \end{aligned}$$

where we have that (34) $\neq$ (35) since $-max(a, b) \neq max(-a, -b)$ (Peyton et al., 2000, p. 288). To show that (13) holds, define the contract

$$c_1 = Acquire(t_1, Acquire(t_2, \ldots, Acquire(t_n, c))), \tag{36}$$

assume that $I(t)$ has the form e^{-rt} and use that $\prod_{i=1}^{n} e^{-rt_i} = e^{-r(t_1+t_2+\ldots+t_n)}$ to get

$$\begin{aligned}
\Sigma[\![c_1]\!]^I_{E,f} &= I(t_1) \times \Sigma[\![Acquire(t_2, \ldots, Acquire(t_n, c))]\!]^I_{(\lambda(s,m).E(s,m+t_1)),f} \\
&= I(t_1) \times I(t_2) \times \Sigma[\![Acquire(t_3, \ldots, Acquire(t_n, c))]\!]^I_{(\lambda(s,m).E(s,m+t_1+t_2)),f} \\
&\vdots \\
&= I(t_1) \times I(t_2) \times \ldots \times I(t_n) \times \Sigma[\![c]\!]^I_{(\lambda(s,m).E(s,m+t_1+t_2+\ldots+t_n)),f} \\
&= I(t_1 + t_2 + \ldots + t_n) \times \Sigma[\![c]\!]^I_{(\lambda(s,m).E(s,m+t_1+t_2+\ldots+t_n)),f} \\
&= \Sigma[\![Acquire(t_1 + t_2 + \ldots + t_n, c)]\!]^I_{E,f}
\end{aligned} \tag{37}$$

(14) follows from

$$\begin{aligned}
\Sigma[\![Give(Give(c))]\!]^I_{E,f} &= -\Sigma[\![Give(c)]\!]^I_{E,f} \\
&= \Sigma[\![c]\!]^I_{E,f}
\end{aligned} \tag{38}$$

And it is evident that (15) holds as

$$\begin{aligned}
&\Sigma[\![Scale(o_1, Scale(o_2, \ldots, Scale(o_n, c)))]\!]^I_{E,f} \\
&= \Omega[\![o_1]\!]_E \times \Sigma[\![Scale(o_2, \ldots, Scale(o_n, c))]\!]^I_{E,f} \\
&= \Omega[\![o_1]\!]_E \times \Omega[\![o_2]\!]_E \times \Sigma[\![Scale(o_3, \ldots, Scale(o_n, c))]\!]^I_{E,f} \\
&= \Omega[\![o_1]\!]_E \times \Omega[\![o_2]\!]_E \times \ldots \times \Omega[\![o_n]\!]_E \times \Sigma[\![c]\!]^I_{E,f} \\
&= \Omega[\![o_1]\!]_E \times \Omega[\![o_2]\!]_E \times \ldots \times \Omega[\![o_n]\!]_E \times \Sigma[\![c]\!]^I_{E,f} \\
&= \Omega[\![o_1 \cdot o_2 \cdot \ldots \cdot o_n]\!]_E \times \Sigma[\![c]\!]^I_{E,f} \\
&= \Sigma[\![Scale(o_1 \cdot o_2 \cdot \ldots \cdot o_n, c)]\!]^I_{E,f}
\end{aligned} \tag{39}$$

We can also show that (16) holds by the following

$$\begin{aligned}
\Sigma[\![Acquire(0, c)]\!]^I_{E,f} &= I(0) \times \Sigma[\![c]\!]^I_{\lambda(s,m).E(s,m+0)),f} \\
&= \Sigma[\![c]\!]^I_{E,f}
\end{aligned} \tag{40}$$

where we use that $I(0) = e^{-r \cdot 0} = 1$. In addition, one can show (17) by the following. Recall $\Sigma[\![All([])]\!]^f_{E,f} = 0.0$ from the definition of the contract evaluation function and see that

$$\begin{aligned}\Sigma[\![Scale(Value\ 0.0, c)]\!]^I_{E,f} &= \Omega[\![Value\ 0.0]\!]_E \times \Sigma[\![c]\!]^I_{E,f} \\ &= 0.0 \times \Sigma[\![c]\!]^I_{E,f} \\ &= 0.0 \end{aligned} \tag{41}$$

which shows that (17) indeed holds.

These results show that we have successfully incorporated the logical requirements from 3.5 in the design of the DSL. In addition, the test results of the implementation in 6.3.4 did not show any errors when checking the logical requirements.

7.2 Demonstration

Conceive that you are a bank which has acquired a portfolio which consists of a European call option, a zero coupon bond and a chooser option, and recall the definition of these contracts from section 3.6 which we use to express the portfolio as seen in Listing 11.

```
let c = All[
europeanCall2 1 "AAPL" 90.0 USD
zcb 1 500.0 USD
chooserOption 1 10 "AAPL" 120.0 USD]
```

Listing 11: A portfolio consisting of a European call, zcb and chooser option.

Now, one day has gone by and we want to update the contract. We utilize the Advance function from section 4 and get the contract shown in Listing 12. It is evident that the first call option matured today, so we have to choose between getting nothing or the price of the AAPL stock today subtracted by the strike. The choice optimal choice should be clear since the stock has a price larger than the strike according to the environment in Listing 12. We can combine the *choose* function from section 4 and the evaluation function from section 5 to get the contract seen in Listing 13. We have now simplified the contract as much as possible and made a choice. Now, we want to get the value or price of the contract. We utilize the $simulationContract$

```
// Environment
let E(s,t) : float =
    match (s,t) with
    | ("AAPL", 0) -> 100.0 // Assume S_0 = 100.0 for the AAPL stock
    | _ -> failwith "price not found"
    // Advance on c
advance E c 1
> All[ // the new portfolio
    Or(All [], // the first call: (S_T - K)^+
        Scale (Sub (Underlying ("AAPL", 0), Value 90.0), One USD))
    Scale(Value 500.0, One USD) // zcb have been advance and simplified
    Or( // The chooser option
        Acquire(9, // call in the chooser
        Or (All [],
            Scale (Sub (Underlying ("AAPL", 0), Value 120.0), One USD))),
            Acquire(9, // put in the chooser
                Or (All [],
                    Scale (Sub (Value 120.0, Underlying ("AAPL", 0)),
                        One USD))))
    ]
```

Listing 12: Utilizing advance on the portfolio in Listing 11

function from section 5 and get obtain the result $529.938...$ as seen in Listing 14. The portfolio value was simulated under the assumption that $\mu = 0.02$ and $\sigma = 0.05$ for the AAPL stock, which can be found in the *stock_data.xml* file. It is evident that this price is correct since the chooser option has the theoretical value

$$\begin{aligned} max(call, put) &= c + \left(Ke^{-rT} - S_0\right)^+ \\ &= BS\left(120, 0.02, \frac{10}{365}, 0.02, K\right) + \left(120e^{-0.02\frac{10}{365}} - 100\right)^+ \\ &\approx 0 + 19.93426... = 19.93426... \end{aligned} \tag{42}$$

(Hull, 2018, p. 603-604) and thus the theoretical value of the portolio follows by

$$10 + 500 + 19.93426... = 529.93426... \tag{43}$$

```
let c1 = advance E c 1
let f (cur : Currency) : float = // assume a constant exchange rate
    match cur with
    | USD -> 1.0
    | EUR -> 1.10
    | GBP -> 1.24
    | DKK -> 0.15
let I (t : int) : float =
    let r = 0.02 // assume constant interest rate
    exp(-r/365. * (float t)) // e^{-rt}
choose (evalc f I E) c1
> All[ // the new portfolio
    Scale(Value 10.0, One USD) // coresponds to (100 - 90, 0)^+
    Scale(Value 500.0, One USD) // same as before, no choice
    Or( // the chooser option is the same as before, no choice yet
        Acquire(9, // call in the chooser
        Or (All [],
            Scale (Sub (Underlying ("AAPL", 0), Value 120.0), One USD))),
            Acquire(9, // put in the chooser
                Or (All [],
                    Scale (Sub (Value 120.0, Underlying ("AAPL", 0)),
                        One USD))))
    ]
```

Listing 13: Making a choice on the first call option of the portfolio from Listing 12

```
let c2 = choose (evalc f I E) c1
simulateContract 1_000_000 c2
> 529.9381028
```

.

Listing 14: The value of the portfolio in Listing 13 according to the simulation

which corresponds with the result we got in the simulation by tolerance 10^{-2}.

8 Future Work

In the this section, we explore some of the directions for future development within the project.

Exchange Rates. As of now, the $evalccy$ function assumes a constant exchange rate and thus the function f which $evalccy$ takes as input does not take into account the time at which to fetch the exchange rate. This approach is unrealistic, and it is acknowledged as an area for further work. The way this problem could be solved is to redefine the $evalccy$ function so that it has the form

$$evalccy :: (ccy \times int \to \mathbb{R}) \to ccy \to \mathbb{R}$$

such that the function takes as input a function which calculates the exchange rate at time of the integer input. This will also require a change in the evaluation of the Acquire constructor so that the value of the currency in relation to USD is shoot in time, as with the stock price.

Interest Rates. The interest rate is also assumed to be constant. This unrealistic assumption be solved by re-implementing the $I(t)$ function according to interest rate models such as the Hull-White or Cox-Ingersoll-Ross model (Shreve, 2004, p. 265-266).

Correlation. When simulating contracts that have multiple underlyings, we do not consider the correlation of the stocks. This could be solved by considering of correlation and covariance matrices (Hull, 2018, p. 505), which can used to implement correlated wiener process for pricing (Björk, 2009, p. 60).

Stochastic Volatility Models. We have chosen to simulate a stock under the condition that it follows a Geometric Brownian Motion. However, this model assumes constant volatility, which is a unrealistic assumption. This problem could be solved by implementing stochastic volatility models such as the Heston Model (Heston, 1993, p. 327-343).

Path-dependent Options. It was concluded in section 5 that as of now we are not able to price path-dependent options in the DSL, such as the Asian option. However, we have implemented and tested the GBM that simulates the whole trajectory of a stock up till time T, so all we need is a new constructor that utilizes this implementation. One way to do this for the asian option case could be to define a function $average$:

$$average :: Obs \to int \to Obs \tag{44}$$

where $average$ calculates the average of the Obs from today to the time of the input integer and returns this value as an Obs. Then we could define the following contract in the DSL

$$\begin{aligned}&Acquire(t,\\&\quad Scale(average(t, Underlying("foo", 0)), One\,ccy)).\end{aligned} \tag{45}$$

These considerations are left for future work.

Optimization. As of now the pricing models depend on Monte Carlo simulation. A common drawback of this method is that it requires a large amount of simulations to obtain accurate results, leading of low computation times pricing multiple contracts. There are various ways to attack this problem, such as variance reduction techniques (Glasserman, 2003, p. 185), sobol sequences, etc.

American Optionality. To be able to price options with early exercise features we could rely on methods such as the binomial model which makes this possible. The paper (Peyton et al., 2000) actually utilizes this method and successfully incorporates pricing of American options in their DSL using a constructor $anytime$.

Validity. We could consider the concept of a contract being valid when expressing it. As of now, we only have one function employing this concept, the $causal$ function, which warns the user if it has an incoming flow that is dependent on an event which occurs later than the flow. We could also warn the user with other validity rules, such as if they have a contract with a flow which is deemed in the past (Elsman, 2010, p. 16).

Dates. The dates in the DSL are expressed with relative time in integers. To allow for absolute time representation we could define a function which maps a string to a date, and then define a data type which allows for arithmetic operations on days, as in (Shreve, 2004, p. 281). This will make it easier to implement the contracts in a GUI and it will allow for floating-representation of dates: something that we do not have as of now but it is critical when valuing contracts that depend on a specific time of the day. In addition, it could allow for consideration of holidays and trading days when pricing and managing contracts.

Testing. The test method used in this thesis have been black box unit testing. To test the program further we could employ test methods such as white box testing, which examines the internal structure of each function tested, in contrast to our test method, which focuses on the external behavior of the program (Sporring, 2020, p. 128).

9 Conclusion

Over the last decades, the financial derivatives market has undergone significant development, highlighting the necessity for comprehensive frameworks to express, manage and price contracts. We have successfully designed and implemented a DSL deeply embedded in $F\#$, which can do exactly this. We have shown that the language is capable of expressing a wide range of contracts, and that any contract which can be expressed in the DSL can be managed and priced using generic logic principles. There are still certain areas that demand further attention for the implementation to be complete, notably path-dependent options and options with early exercise features are currently not supported.

References

[Björk, 2009] T. Björk. (2009). *Arbitrage Theory in Continuous Time.* Oxford University Press; 3rd edition.

[Bulmer, 1979] Bulmer, M. G. (1979). Principles of Statistics. Dover. 3rd edition. ISBN 0-486-63760-3.

[Cox et al., 1979] Cox, J. C., Ross, S. A., and Rubinstein, M. (1979). *Option pricing; a simplified approach*. Journal of Financial Economics, 7:299-263.

[Elsman, 2010] Martin Elsman, Simcorp A/S. (2010). *Functional Programming for Trade Management and Valuation*. Seminar on Functional High Performance Computing in Finance. Link: https://elsman.com/pdf/FPforTradeManagement.pdf. Visited 05-31-2023.

[Filinski, 2023] A. Filinski (2023). *Lecture Notes for Semantics and Types*. University of Copenhagen.

[Fowler et al., 2010] M. Fowler (2010). *Domain Specific Languages.* Addison-Wesley Professional.

[Heston, 1993] S. Heston. (1993) *A closed-form solution for options with stochastic volatility and applications to bond and currency options*. Rev. Fin. Stud. 6, 327-343.

[Hudak, 2012] P. Hudak, *Modular domain specific languages and tools*, Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203), Victoria, BC, Canada, 1998, pp. 134-142, doi: 10.1109/ICSR.1998.685738.

[Hull, 2018] John C. Hull. (2018). *Options, Futures, and Other Derivatives*. 10th edition. Pearson.

[Glasserman, 2003] P. Glasserman. (2003). *Monte Carlo Methods in Financial Engineering*. Springer.

[Minsky et al., (2008)] Minsky, Y., & Weeks, S. (2008). *Caml trading – experiences with functional programming on Wall Street.* Journal of Functional Programming, 18(4), 553-564. doi:10.1017/S095679680800676X

[Peyton et al., 2000] Jones SP, Eber JM, Seward J. (2000). *Composing contracts: an adventure in financial engineering (functional pearl)*. In: Proceedings of the 20th ACM SIGPLAN international conference on functional programming, ACM. pp 280–292

[Shreve, 2004] S. E. Shreve (2004). *Stochastic Calculus for Finance II: Continuous-Time Models.* Springer.

[Sporring, 2020] Jon Sporring. (2008). *Learn to Program with F#.* University of Copenhagen.

[Werk et al., (2012)] Michael Flænø Werk, Joakim Ahnfelt-Rønne, and Ken Friis Larsen. 2012. *An embedded DSL for stochastic processes*: Research article. In Proceedings of the 1st ACM SIGPLAN Workshop on Functional High- Performance Computing (FHPC'12). ACM, New York, NY, 93–102.

[XpressInstruments User Manual, 2008] *Simcorp A/S (2008). XpressInstruments - Workflow. User Manual.* Responsible: IMS Data and Pricing. Based on SimCorp Dimension, Version 4.3.