Functional Trading

Submitted April 2025, in partial fulfilment of the conditions for the award of the degree BSc Computer Science

Supervised by:
Professor Graham Hutton
graham.hutton@nottingham.ac.uk

Student ID: 20447593
School of Computer Science
University of Nottingham

I hereby declare that this is my own work, except as indicated in the text: Y. Y.

Date: 17/04/2025

Abstract

This dissertation explores the application of functional programming paradigms, particularly Haskell, to model and evaluate financial contracts. Inspired by the work of Peyton Jones et al., the project develops an embedded Domain-Specific Language (eDSL) that decomposes financial contracts into composable combinators. These combinators enable the representation of diverse financial instruments, including bonds, options, and exotic derivatives, while ensuring type safety, modularity, and maintainability.

The implementation focuses on three core components: the eDSL for contract representation, a compiler (valuation engine) that translates contracts into value processes, and a pricing model based on binomial lattices. Key contributions include extensions and modifications to the original combinators, such as support for boolean observables and arithmetic operations, as well as optimisation strategies and validation mechanisms to handle invalid contracts. The project also reflects on Haskell's advantages and limitations in this domain.

Future work could expand the eDSL to support more complex instruments, integrate alternative pricing models, or explore parallelism for performance improvements. This research underscores the potential of functional programming in finance, offering a foundation for further exploration in both academic and practical settings.

Acknowledgment

I would like to express my gratitude to my supervisor, Professor Graham Hutton, for his guidance, support, and insightful feedback throughout this project. His expertise in functional programming and encouragement during challenging moments have been invaluable.

Contents

1 Introduction

In finance, where even small errors can lead to significant financial losses, functional programming paradigms - such as type safety, pure functions, and composability - provide crucial advantages for building reliable and maintainable software systems. These benefits make functional programming an attractive choice for the finance sector, with companies like *Jane Street* and *Barclays* relying heavily on it. The ability to write code that is easy to reason about, predict, and test is essential in algorithmic trading, where large volumes of data must be processed in real-time.

This project is inspired by a curiosity to explore the application of functional programming in a real-life domain and the work of Peyton Jones et al., who demonstrated the benefits of using a **domain-specific language** embedded in Haskell (**eDSL**) to model financial contracts [20]. Their work, which we will refer to as *Composing Contracts (CC)* or *the library*, decomposes financial contracts into small building blocks called **combinators**.

In programming, a combinator is a self-contained function that combines simpler elements (in this case, contracts) to build more complex functionality. These combinators can be composed to express different contracts, making it possible to build applications that work with any contract expressed in the language without needing to write custom code for each new contract type.

While *CC* presents an abstract description of combinators along with formal evaluation semantics for contract pricing, this project seeks to:

- Propose enhancements and extensions to the original library
- Provide a concrete implementation in Haskell
- Explore optimisation strategies for efficient contract evaluation
- Develop validation mechanisms for contract safety
- Provide a user-friendly GUI for interaction with the system
- Assess the suitability of a functional language like Haskell in the domain of financial modelling

The ultimate goals are both a learning experience and complementing the existing research by carrying out the steps above.

The effectiveness of our approach will be assessed by applying it to an expanded range of financial contracts, comprehensive testing strategy consisting of property-based tests and unit tests, and benchmarking of different components.

Note: While this paper assumes the reader has no previous knowledge of trading and finance, a basic understanding of Haskell is presumed.

2 Why and What Do Traders Trade?

Most papers I have read introduce finance concepts on the fly, but this often leaves many questions unanswered. For this reason, I prefer to provide some background before diving into code semantics and implementation.

Every second, even down to milliseconds or microseconds, a wide variety of things called **assets** are traded - **commodities** - like gold and oil; **financial instruments** - such as bonds, stocks or currencies; and even **services** - such as insurances, software or transportation. Some assets may appear "untradable" at first, yet it often just takes a bit of creativity to turn them into marketable products. The list of categories and sub-categories is endless and ever-growing. Naturally, no trader or trading firm could possibly trade everything. Trading firms specialise in trading specific sets of instruments. However, to expand their business, the most obvious approach is to broaden the range of instruments they trade. This expansion requires their systems and methods for describing instruments—and for determining their worth, or how much specific trades could earn—to be easily extendable [11, 20].

To legally trade an instrument, traders enter a **contract**. This contract outlines what each party is receiving or giving and the obligations of each side. For example, when you take out a loan from a bank, you enter a contract in which the bank agrees to provide you with a certain amount of money, and you agree to return this money in monthly instalments along with an agreed-upon interest rate. Here, the bank's obligation is to lend you the money on a specific date, while your obligation is to repay the loan periodically on agreed dates [25].

A very important property of a contract is **when it is acquired**. This acquisition might be tied to a specific date or a certain event that triggers the acquisition. For example, some contracts become valid only when a particular stock price threshold is met. The acquisition date is crucial because it determines the starting point for obligations, valuation, and execution. However, just as the **acquisition** of a contract marks its beginning, it is equally important to consider its end.

In financial terminology, a contract is said to have **expired, matured, or reached its horizon** interchangeably. This means that the obligations defined in the contract have been fulfilled or rendered void, marking the end of its lifecycle. The full lifecycle of a contract—from acquisition to expiration—plays a fundamental role in determining its value, as pricing models depend on **when a contract starts and ends**.

In the paper by Peyton Jones et al.—*Composing Contracts*, a simple set of combinators written in Haskell is proposed that claims to describe a variety of contracts.

Bonds The paper starts with an example of a **bond**. While CC offered a brief description of bonds, I believe if a bit more is explained, a lot could be learned about the motives of traders for trading any other assets in general. A bond operates on a similar principle to a loan: someone who needs money enters an agreement to borrow it from someone else. You might ask, "How can you trade a loan?" This is where bonds differ. In a loan, the borrower repays the amount borrowed periodically until the full sum is returned. In contrast, a bond involves repaying the entire amount at once on a specific date, called the **bond maturity date**. Before the maturity date, there are two common practices: the borrower either pays periodic fees called **coupons** or returns a substantially larger sum at the end [4, 20].

For instance, consider a **zero-coupon discount bond** (ZCDB). Here, you receive a lump sum, say £1,000, and in 10 years, you repay £1,300. Why would someone choose this structure? Imagine you are a startup in desperate need of money now but anticipate little or no profit for the next two years. However, you are optimistic that in four years, you will generate significant revenue. You sell the bond to raise immediate funds, and the investor expects repayment in the future [2].

On the other side, if you have money to invest, you might buy this bond. You lend £1,000 to the startup, expecting to receive £1,300 in four years. While this seems ideal, what happens if the startup goes bankrupt and cannot repay you? This introduces **risk**. Bonds are often traded to mitigate such risks. For instance, if you foresee potential trouble with the startup, you could sell the bond after one year. Due to inflation and other factors, the bond's value may have increased—say to £1,050. You earn a profit while offloading the risk to another trader. Of course, your prediction might be incorrect, and the startup thrives. In that case, the trader who bought the bond from you will profit more than you did [4, 25].

As this example illustrates, numerous variables affect decisions to buy or sell contracts. Ultimately, trading is about making the best possible predictions about the future and using the fastest systems to execute decisions efficiently.

Options, Futures, and Other Derivatives [13] As if bonds are not complex enough on their own, they, as well as many other assets, could serve as the foundation for another layer of traded instruments called **derivatives**. A derivative is simply a financial contract whose value is derived from (i.e., depends on) an **underlying asset**. This asset can be not only bonds but anything really - stocks, cryptocurrencies, natural gas, grains, or even weather metrics — so long as its price or measurable quantity is well-defined. Derivatives are commonly used for both **hedging** - reducing risk by offsetting a potential loss in one position with gains in another, and **speculation** - aiming to profit from future price movements [2, 4, 20].

Among the most widespread derivatives are **options** and **futures**.

Options are contracts granting their holder the right, but *not the obligation*, to buy or sell some underlying asset at a specified *strike price* on (or before) a given date. A *call option* gives the holder the right to buy the underlying asset. If the asset's market price soars above the strike price, the call holder can choose to exercise the option and profit from buying at below-market cost. Conversely, if the market price stays below the strike, the holder may let the option expire worthless, limiting the potential loss to the upfront option premium. A *put option* is similar but grants the right to *sell* the underlying asset at a predetermined price. Put options become especially valuable when the asset price declines, as the holder can sell at a higher, preset strike price [4, 25].

Futures impose an *obligation* for a party to buy or sell an asset at a predetermined price on a set future date [11]. This obligation is mutual: one party agrees to buy, and the counterparty agrees to sell. If you enter a futures contract to buy 1,000 barrels of oil next January at $80 per barrel, you cannot opt-out if the price shoots up to $100. However, if the price falls to $60, the other side still owes you the barrels at $80. This fixed obligation is what distinguishes futures from options.

Other derivatives Beyond these traditional instruments lies a realm of increasingly exotic derivatives most commonly traded *over-the-counter (OTC)*. OTC means that these instruments are traded directly between two parties without exchange supervision, typically involving institutional investors, banks, and specialised trading firms. Unlike exchange-traded derivatives, OTC contracts offer greater *customisability* to meet specific risk management.

Examples include **weather derivatives** that allow companies to hedge against unfavourable weather conditions affecting their operations, and **barrier options** where the payoff of the contract depends on whether an underlying asset price reaches a predetermined price level. Derivative trading predominantly serves risk management purposes -

farmers protect against price collapses, airlines hedge fuel costs, and multinational corporations mitigate currency exposure.

From Theory to Implementation I hope this overview gives you a better perception of the diversity of tradable instruments and the importance of using composable and extendable building blocks for the trading software we create. Markets are complex, with many factors influencing prices and risks, so traders rely on fast, adaptive systems to stay ahead. Whether managing risk or seeking profit, the key is making smart decisions quickly. Although there are many types of instruments—bonds, stocks, equities, etc.—the end goal is to calculate a single number: **how much you can earn or lose** [11, 20].

3 Prior Work

Now that we have touched on the essence of trading, we turn to how Haskell can be used to model these activities effectively. The following discussion presents the changes and extensions to the original combinators of Peyton Jones et al. [20]. While modifications have been made, the implementation remains rooted in their work; thus, the introduction to the main combinators - **Contracts** and **Observables** - together with the evaluation semantics, appropriately belongs in the Prior Work section.

Research revealed that a few years after their original paper, they published - *How to Write a Financial Contract* [15], which introduces new combinators, replacing some older ones and expanding the range of contracts that could be modelled using the eDSL. While I discovered the second paper after developing most of the ideas independently, I still acknowledge their influence on my final implementation and approach to extending the combinators.

It should be noted that the only example borrowed from the original paper is the Zero-Coupon Discount Bond (ZCDB), though it is presented and explained in a unique way here. This example serves as an effective illustration of the idea of trading from the previous section and creates a valuable connection to the following concepts. All other examples are original and designed to provide alternative presentations that clarify the purpose of each combinator.

3.1 Contracts

To start the introduction to the basic combinators, consider the example of a **Zero-Coupon Discount Bond (ZCDB)** that costs £1,000 today and pays £1,300 to the holder four years later. If today is the 1st of January 2025, then four years from now would be the 1st of January 2029. We can break this contract into two parts: paying £1,000 today and acquiring the ZCDB that pays £1,300 on the specified future date. Using the *CC* library, we can write this contract as:

```
bondExample :: Contract
bondExample = pay (date* "01-01-2025") 1000 GBP `and`
              zcdb (date* "01-01-2029") 1300 GBP
```

*`date` is a function that takes a string and converts it to the type expected by the language.

Here, we use three combinators: `pay`, `and`, and `zcdb`. Even though we have not yet formally defined them, the structure of `bondExample` is intuitive because it closely resembles plain English. The `pay` and `zcdb` combinators are not primitives—they are composed of simpler combinators. However, `and` is a **primitive combinator**, which we use to combine two contracts into one:

```
and :: Contract → Contract → Contract
```

Before defining `pay` and `zcdb`, we need to break them down further. Both involve specifying a **date**, an **amount**, and a **currency**. A practical way to represent one unit of a currency is to define a combinator that takes a `Currency` and returns a `Contract`:

```
one :: Currency → Contract
```

However, contracts often involve more than a single unit of currency, so we need a way to scale the amount. For this, we introduce another combinator:

```
scaleK :: Double → Contract → Contract
```

Finally, we need to specify the date on which the contract will be acquired. This is modelled using:

```
acquireOn :: Date → Contract
```

Now, we have all the building blocks we need to describe `zcdb`:

```
zcdb :: Date → Double → Currency → Contract
zcdb t amount cur = acquireOn t (scaleK amount (one cur))
```

Similarly, we define the `pay` combinator to represent the act of giving money. For this, we introduce the `give` combinator:

```
give :: Contract → Contract
```

Using `give` and the combinators we just defined, the `pay` combinator is implemented as:

```
pay :: Date → Double → Currency → Contract
pay t amount cur = acquireOn t (give (scaleK amount (one cur)))
```

The library user can freely define new combinators to improve readability or abstract common patterns. For example, "giving" a contract on a specific date might be a frequently used behaviour. To simplify this, we could introduce the `giveOnDate` combinator:

```
giveOnDate :: Date → Contract → Contract
giveOnDate t c = acquireOn t (give c)
```

Using this abstraction, `pay` can be redefined as:

```
pay :: Date → Double → Currency → Contract
pay t amount cur = giveOnDate t (scaleK amount (one cur))
```

These combinators claim to describe a wide variety of contracts, and Jones asserts that no existing combinator would need modification to accommodate a contract that couldn't otherwise be described—only additions to the system would be required.

In the following subsections, we will introduce the last three foundational combinators together with the second abstraction, which represents uncertainty and dynamic behaviours in financial contracts.

3.2 Numeric Observables

Not all contracts are straightforward, and many depend on **uncertain events in the future**—events that traders can only speculate about. As previously mentioned, numerous variables influence decisions to buy or sell contracts. For instance, the payout of some popular contracts depends on stock price movements or even physical measurements like rainfall or grain yields.

Consider a contract to buy *Disney stock for £100 on 1st April 2025*. This type of contract, known as a **future**, differs from the *Zero-Coupon Discount Bond* in that no cash exchange occurs today. Instead, both the acquisition and the payment happen at an agreed date in the future. Using our library, the skeleton of this contract might look like this:

```
dis100Apr :: Contract
dis100Apr = acquireOn (date "01-04-2025")
                      ( ? `and` give (scaleK 100 (one GBP)))
```

However, how do we represent the acquisition of Disney stock? One might naively think we could use the `one` combinator and write something like `one "DIS"`. Yet, `one` is defined to take a currency, not a stock. Should we then create a new combinator, such as:

```
oneS :: Stock → Contract
```

And what about commodities like gold? Should we define:

```
oneKg :: kgOfGold → Contract
```

This approach quickly becomes unsustainable, especially if we need to model varying units like grams or kilograms.

To come up with an alternative, let's return to the essence of trading. At its core, trading is about profit and loss. Traders care about how much they can earn or lose from a trade, not the physical stock or commodity itself. This perspective leads us to a more elegant solution: scale a unit of currency by the price of the stock or commodity rather than representing them as a contract.

However, while we may know today's stock price, that price fluctuates - it might rise or fall unpredictably by the agreed future date. Predicting such movements is the essence of trading and risk management. Financial institutions rely on sophisticated and often proprietary models — called *stochastic processes* — to forecast how prices might change in the future. To represent such uncertainties, *CC* proposes using an additional combinator called **observable**, which would allow us to model real-world uncertainty and incorporate it into our contracts [4]. While the observable itself does not "carry" its value, it would signal to the evaluation engine to ask the model to provide its corresponding stochastic process.

Now, we can define the stock price as an observable using the stock unique `ticker symbol`[1] and the `scale` combinator which takes an observable and a contract that will be scaled:

[1] A stock ticker symbol is a unique code of letters or numbers that represents a publicly traded company's stock on an exchange.

```
StockPrice :: Stock → Obs Double
data Stock = DIS | TSLA | NVDA

scale :: Obs Double → Contract → Contract
```

In fact, we cheated a bit in our definition of `zcdb` in the previous subsection by using `scaleK`, which is not a primitive combinator but one derived from `scale` and the observable `konst :: Double → Obs Double`. `Konst` is an observable whose value remains constant at all times.

With this new concept and the `scale` combinator, the definition of the Disney stock future could be refined as:

```
dis100Apr :: Contract
dis100Apr = acquireOn (date "01-04-2025")
                (scale (stockPrice DIS) (one GBP)
                `and` give (scale (konst 100) (one GBP)))
```

3.3 Options

To introduce the last few combinators, let's look at another widely traded instrument —**options**. The two most common types of options are **European** and **American** options [4].

- European options allow the holder to *choose* whether to acquire an underlying contract **on** a specific date for a given strike price.
- American options provide the same choice, but it can be exercised anytime **on or before** a specific date.

To model *choices* and the alternative of acquiring *nothing*, two additional combinators would come handy:

```
-- Choice between two contracts
or :: Contract → Contract → Contract
-- A contract with no obligations, or acquiring nothing
none :: Contract
```

With these combinators, we can define options in a way similar to futures. As introduced in the finance section, options are further categorised into **call** and **put** options. Let's consider how we can construct a *European call option for Disney stock*:

```
dis100AprCall :: Contract
dis100AprCall = acquireOn (date "01-04-2025")
                    (scale (stockPrice DIS) (one GBP)
                    `and` give (scale (Konst 100) (one GBP)))
                `or`
                acquireOn (date "01-04-2025")
                    none
```

Since options are among the most widely traded instruments, it makes sense to abstract this recurring pattern into a reusable function:

```
european :: Date → Contract → Contract
european t underlying = acquireOn t underlying
                        `or` acquireOn t none
```

For American options, the only change required is swapping `acquireOn` with `acquireOnBefore`, which, as the name suggests, allows the contract to be acquired at any time on or before the specified date.

If you are familiar with the *CC* paper, you might notice that the definition here differs from the one presented there. One could argue that encapsulating `none` and the underlying contract separately within `acquireOn` is unnecessary. However, once we introduce the evaluation process, it will become clear that the alternative representation of European options does not capture the correct behaviour.

With these combinators in place, we can now accurately represent both European and American options using our eDSL. This wraps up the combinators that formed my initial eDSL version that I developed in the autumn semester.

3.4 Boolean Observables and Complex Derivatives

So far, we have provided examples of commonly traded assets such as bonds, futures, and options. However, one of the goals of this project is to demonstrate that the eDSL can be extended to model *any type* of contract, even the most exotic ones.

The examples we have seen primarily depended on *Date-based* events, where the right to acquire a contract is tied to a specific date, using combinators like `acquireOn`. However, many exotic contracts also depend on discrete events or other binary conditions that may or may not occur. For instance, a contract could depend on a *threshold event* in the underlying asset (e.g., "*If the stock price drops below 50, then acquire another futures contract*") or *corporate actions* such as mergers, acquisitions, or defaults.

To illustrate the importance of boolean observables, let's consider a simplified version of a *barrier option*. As discussed in the trading introduction, a barrier option is a type of derivative whose payoff depends on whether the underlying asset's price crosses a predetermined barrier level during the option's lifetime. Barrier options fall into two main categories: *knock-in* and *knock-out*. Each of these can be further classified into *up-and-in/down-and-in* and *up-and-out/down-and-out* options, depending on the direction in which the barrier is crossed.

To keep the focus on core technical ideas rather than going deeply into financial theory, we will only examine an *up-and-in barrier option*. This type of option becomes active only if the underlying asset's price rises above a specified threshold during the contract period. If this condition is never met, the option remains inactive and expires worthless.

To model such contracts, we would need a boolean observable that indicates whether the price of a stock has crossed a certain upper threshold:

```
stockAboveThreshold :: Double → Stock → Obs Bool
```

We also introduce a new combinator that takes a boolean observable and acquires the underlying contract if and when the observable becomes true:

```
acquireWhen :: Obs Bool → Contract → Contract
```

Using these building blocks, we can specify a simple *up-and-in* contract that pays a fixed amount once the stock price surpasses a certain threshold. In real-world scenarios,

the underlying contract is often a *forward* or *futures* contract rather than a fixed cash payment, but we use the latter here for simplicity:

```
knockInAAPL :: Contract
knockInAAPL = acquireWhen
            (stockAboveThreshold 150.0 AAPL)
            (scale (konst 100) (one USD))
```

This contract is inactive until Apple's stock price goes above $150, at which point it immediately pays $100 to the contract holder. This example demonstrates how boolean observables allow us to model contracts that depend on discrete market events, rather than purely time-based triggers.

The remaining task is to implement the function `stockAboveThreshold`. This function requires *arithmetic on observables*, a concept that depends on understanding certain implementation details, which will be discussed in the next section.

The `acquireWhen` combinator (originally named `when` in Peyton Jones *et al.*'s "How to Write a Financial Contract") is one of three conditionals introduced in their extended work. Interestingly, I had independently begun implementing boolean observables and the `acquireWhen` combinator before encountering their follow-up work, which is a proof of its natural role in contract modelling.

The paper also includes `cond` (conditional branching) and `until` (contract termination), both of which work with boolean observables, but I believe these lie beyond our present focus. While including additional combinators would make the system more flexible, it would shift focus away from how the system works under the hood - a priority for computer science readers. We use only `acquireWhen` because it cleanly demonstrates how boolean observables interact with contracts in a modular way.

3.5 Combinators Wrap Up

With this, we wrap up our overview of the core combinators we will use to describe contracts. While these don't cover every traded contract, they provide a strong foundation for expressing a wide variety of instruments. An overview of all combinators I've implemented can be found in tables 1 and 2.

The primary differences between my implementation and the original *CC* library lie in the naming and structure of certain combinators:

In the original library:

- The `none` combinator is referred to as `zero`.
- The `acquireOn` combinator is implemented as a combination of two original combinators, `truncate` and `get`.
- Similarly, the `acquireOnBefore` combinator combines `anytime` and `get`.

I chose to remove the `get` combinator from my implementation, as I could not find a case that requires it to be a separate combinator. Furthermore, the `get` combinator is absent in the second paper from Peyton Jones at el.

The second paper also replaces `truncate` and `get` (which are equivalent to `acquireOn`) with `acquireWhen`. The motivation behind that is that we can make a date into a boolean observable instead of having a combinator that explicitly takes a date. However, in my

implementation, I have decided to keep the `acquireOn` combinator as it makes reasoning about the validity and behaviour of contracts easier and provides a separation of concerns. Also, contracts depending on a boolean event are a lot rarer.

You might also notice that in the obs there are additional observables that we haven't yet discussed. This is because they will be introduced in the following section when we explain how we can apply arithmetic to observables and why it is useful.

`Contract`	**Description**
`None`	Contract with no obligations
`One Currency`	Contract for one unit of a currency
`Give Contract`	Opposite position of a contract
`And Contract Contract`	Adds the value of two contracts together
`Or Contract Contract`	Takes the contract that will yield the maximum value
`AcquireOn Date Contract`	Sets the acquisition date of a contract where the discounting should begin
`AcquireOnBefore Date Contract`	Shows opportunity to acquire a contract anytime before a certain date
`AcquireWhen (Obs Bool) Contract`	Acquires the underlying contract when the Boolean observable becomes True
`Scale (Obs Double) Contract`	Scales a contract by a numeric observable

Table 1: Overview of Contract Combinators

`Observable`	**Description**
`Konst :: Double → Obs Double`	Observable whose value remains unchanged at any point in time.
`StockPrice :: Stock → Obs Double`	Observable that represents the price of a stock at different points in time.
`LiftD :: UnaryOp → Obs Double → Obs Double`	Applies a unary operator to an `Obs Double`, producing another `Obs Double`.
`Lift2D :: BinaryOp → Obs Double → Obs Double → Obs Double`	Applies a binary operator to two `Obs Double` values, producing another `Obs Double`.
`Lift2B :: CompareOp → Obs Double → Obs Double → Obs Bool`	Compares two `Obs Double` values, resulting in an `Obs Bool`.
`MaxObs :: Obs Double → Obs Double → Obs Double`	Returns an `Obs Double` that represents the combined maximum of the two input `Obs Double` values.

Table 2: Overview of all observables (Obs).

3.6 Evaluation

We return to the fundamental question that traders constantly ask: *What is it worth?* In our embedded Domain-Specific Language (eDSL), `Contracts` and `Observables` are the two primary types, much like integers or floats in other programming languages. However, to make sense of these types and the operations performed on them, we need rules—a formal grammar. Fortunately, the authors of the *CC library* provide evaluation semantics, which serve as the basis for understanding and computing the value of contracts within our eDSL.

The value of a contract changes over time, reflecting its evolution. This is captured through a *value process*, a concept rooted in financial mathematics [22, 25]. In our eDSL, both contracts and observables are modelled as processes that evolve over time.

Applying the evaluation semantics in our language requires a compiler that translates contracts and observables to the intermediate language of value processes. We define the semantics of our compiler using the function:

$$\mathcal{E}_k[] : \texttt{Contract} \rightarrow \texttt{Value Process}$$

where k represents the base currency of the contract.

Before diving into the semantics, it's crucial to introduce a property inherent to contracts: their *acquisition date* or which we will refer to also as *horizon*. By default, contracts have an acquisition date of infinity unless explicitly set using combinators like `acquireOn` or `acquireOnBefore`. The acquisition dates for different combinators are defined in Figure 1.

$$\begin{aligned} H(\text{none}) &= \infty \\ H(\text{one } k) &= \infty \\ H(\text{c1 `and` c2}) &= \max(H(\text{c1}), H(\text{c2})) \\ H(\text{c1 `or` c2}) &= \max(H(\text{c1}), H(\text{c2})) \\ H(\text{acquireOn } t\,\text{c}) &= \min(t, H(\text{c})) \\ H(\text{acquireOnBefore c}) &= H(\text{c}) \\ H(\text{scale } o\,\text{c}) &= H(\text{c}) \end{aligned}$$

Figure 1: Aquisition date or horizon of different contracts

Now, let's explore the evaluation semantics which are summarised in Table 3 based on their horizon. We notice how many of the combinators can be mapped to basic mathematical operations:

E1: `give c` – Negates the value process of contract `c`.

E2: `c1 ‘and‘ c2` – Acts like addition.

E3: `c1 ‘or‘ c2` – This requires understanding why a trader would choose one contract over the other. The answer is unsurprising: the one that makes the most profit. Hence, we need a function that takes the maximum of the two contracts.

E4: `scale o c` – This combinator takes an observable `o` and scales the value process of contract `c` by the value process of `o`. Observables have distinct semantics, but the overall effect is multiplication.

E5: `none` – The value is simply 0 at any point in time.

E6: `one currency` – Represents the value of a contract in currency k_2 converted to the base currency k.

The remaining semantics, E7, E8 and E9, will be discussed in the following section.

By now, you may wonder: How are value processes represented in Haskell? How do we add or multiply them? Are these operations uniform across all cases? These questions lead us directly into the next section, where we will explore the representation and manipulation of value processes in detail.

$(E1)$ $\mathcal{E}_k[\text{give } c]$	$= -\mathcal{E}_k[c]$	
$(E2)$ $\mathcal{E}_k[c_1 \text{ `and` } c_2]$	$= \begin{cases} \mathcal{E}_k[c_1] + \mathcal{E}_k[c_2] \\ \mathcal{E}_k[c_1] \\ \mathcal{E}_k[c_2] \end{cases}$	on $\{t \mid t \leq H(c_1) \wedge t \leq H(c_2)\}$ on $\{t \mid t \leq H(c_1) \wedge t > H(c_2)\}$ on $\{t \mid t > H(c_1) \wedge t \leq H(c_2)\}$
$(E3)$ $\mathcal{E}_k[c_1 \text{ `or` } c_2]$	$= \begin{cases} \max(\mathcal{E}_k[c_1], \mathcal{E}_k[c_2]) \\ \mathcal{E}_k[c_1] \\ \mathcal{E}_k[c_2] \end{cases}$	on $\{t \mid t \leq H(c_1) \wedge t \leq H(c_2)\}$ on $\{t \mid t \leq H(c_1) \wedge t > H(c_2)\}$ on $\{t \mid t > H(c_1) \wedge t \leq H(c_2)\}$
$(E4)$ $\mathcal{E}_k[\text{scale } o\, c]$	$= V[o] \cdot \mathcal{E}_k[c]$	
$(E5)$ $\mathcal{E}_k[\text{none}]$	$= K_0$	
$(E6)$ $\mathcal{E}_k[\text{one k2}]$	$= \text{exch}_k(k2)$	
$(E7)$ $\mathcal{E}_k[\text{acquireOn } T\, c]$	$\text{disc}_k^{H(c)}(\mathcal{E}_k[c](H(c))$	on $\{t \mid t \leq T\}$ if $H(c) \neq \infty$
$(E8)$ $\mathcal{E}_k[\text{acquireOnBefore c}]$	$\text{snell}_k^{H(c)}(\mathcal{E}_k[c])$	if $H(c) \neq \infty$
$(E9)$ $\mathcal{E}_k[\text{acquireWhen o c}]$	$\text{disc}_k(V[o], \mathcal{E}_k[c](H(c))$	if $H(c) \neq \infty$

Table 3: Overview of Evaluation Semantics

3.7 More Related Work

Over the years, *CC* has sparked the interest of many developers. I would like to acknowledge the existing implementations I encountered during my research.

A blog post demonstrating a Scala implementation greatly enhanced my understanding of how the combinators—especially observables—work [3].

Additionally, I came across an implementation of a Monte Carlo simulation evaluation model. Although I have not yet discussed pricing models in detail, it is worth acknowledging this contribution in the Prior Work section [1]. This stands out as the most complex and realistic implementation I have encountered, and it has truly inspired me to consider more advanced evaluation strategies in the future.

Moreover, the recent experience report by Dijkstra et al. from Standard Chartered Bank provides a compelling example of large-scale functional programming in practice [6]. Their use of Haskell and their in-house Mu language to run pricing workflows in the cloud and support real-time analytics shows that typed functional programming can work well even in large, production-critical financial systems.

4 Design and Implementation

Peyton Jones et al.'s work outlines an eDSL for financial contracts along with evaluation semantics, but it does not provide a concrete implementation. My contribution is entirely in designing and implementing the combinators, validation, the language's compiler and different optimisations. Figure 2 illustrates the process of interpreting a contract expressed in our DSL into a *Value Process*. I decided to modularise the system into 3 distinct modules - One for the language itself, one for the compiler or evaluation engine and a separate one for the pricing model. While initially the compiler was only interpreting contracts it now also includes validation, optimisation and caching. I will discuss each component separately in this section.

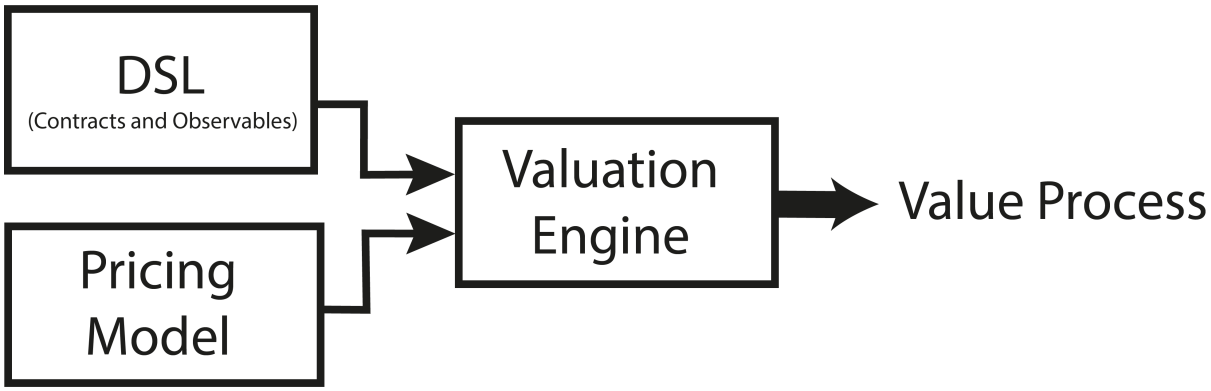


Figure 2: System Architecture

4.1 eDSL

The most crucial design decision was making the eDSL **deeply** embedded rather than using **shallow** embedding. In functional programming, a deeply embedded domain-specific language is one whose constructs are first-class citizens in the host language [8]. Unlike a shallow embedding, where each combinator is essentially a Haskell function that could directly run its semantics, we represent *contracts* and *observables* as **data**. In this way, a contract on an observable itself does not hold any information about the stochastic process that determines its value. All process-related details are delegated to the pricing model and executed by the compiler.

4.1.1 Contracts

The DSL's core type, `Contract`, is declared as a separate abstract syntax tree (AST) through explicit data constructors:

```
data Contract
    = None
    | One  Currency
    | Give Contract
    | And  Contract Contract
    | Or   Contract Contract
    | AcquireOn       Date Contract
```

```
  | AcquireOnBefore Date Contract
  | AcquireWhen     (Obs Bool) Contract
  | Scale           (Obs Double) Contract
 deriving (Show, Eq, Ord)
```

The main advantages of this approach are:

- **Separation of Concerns:** By keeping contracts purely as data, all pricing and evaluation logic resides in the compiler or runtime model. This makes the system more modular and maintainable.
- **Flexible Manipulation:** Since contracts are just data structures, they can be easily pattern-matched, transformed, optimised, or rewritten before evaluation. This is especially useful for implementing algebraic simplifications and optimisations.
- **Multiple Interpretations:** The same contract can be interpreted in various ways—for example, it can be evaluated for pricing, rendered as a diagram, or analysed for validation—without changing the underlying representation.
- **Support for Different Valuation Models:** As we will later see, there is not just a single way a contract could be evaluated. Traders may use different pricing models depending on their assumptions. A deep embedding enables the same contract definition to be evaluated under multiple models (e.g., Black-Scholes, binomial lattices, Monte Carlo) without modifying the contract itself.
- **Extensibility:** New contract forms or observables can be added by simply introducing new data constructors and defining their corresponding evaluation rules. This makes the DSL easy to extend and adapt to new financial products.

4.1.2 Observables

Observables are implemented slightly differently using Generalised Algebraic Data Type (GADT):

```
data Obs a where
    Konst       :: Double → Obs Double
    StockPrice  :: Stock → Obs Double
    ...
```

The main advantage of using GADTs is that they offer greater **type safety** and **expressiveness**. Each constructor can specify precisely which type of `Obs a` it returns, allowing the DSL to support observables that are not limited to numeric types. [5]

This type precision enables the compiler to catch potential type errors at *compile time* rather than at runtime. Furthermore, GADTs support **type-aware pattern matching**, making it possible to write functions that handle different observable types appropriately based on their constructors, while maintaining strong type guarantees. This makes the language more robust and maintainable, especially in the context of financial modelling, where correctness is critical.

4.1.3 Observable Arithmetic

Something I was interested in exploring with this project is modelling exotic contracts without necessarily introducing new combinators. Observable arithmetic looked like an intriguing candidate for this job.

While the first paper introduces two combinators for performing observable arithmetic, it does not elaborate on why and how they are used. The second paper [15] does in fact explain them but I had already developed my own reasoning and implementation before finding out about its existence.

Observables appear in two contexts in our eDSL - when scaling a contract and when acquiring a contract based on a boolean event. Let's consider the first use.

Numeric observable arithmetic While scaling a contract by a single observable is useful, real-world contracts frequently demand combinations of observables or more intricate arithmetic.

The example we would consider is the exotic*production-based grain derivative*, which is a risk management contract whose payoff depends on the actual grain yield in a given region. It typically pays out if harvests fall below a specified threshold (e.g., due to drought), compensating the holder for low production. Conversely, if yields exceed certain levels, the contract may require the holder to pay the counterparty. This structure helps hedge against the uncertainty of agricultural output.

Suppose "GrainGrowers Ltd." wants a derivative that pays off if actual grain production in a particular region falls below a trigger level—e.g. 100,000 tonnes. Concretely, we define an observable capturing the actual grain yield in tonnes:

```
grainYield :: Obs Double
-- Represents the predicted grain production in tonnes.
```

If production is below 100,000 tonnes, the shortfall is (100000 - grainYield). However, if grainYield exceeds 100,000, the shortfall is zero. It would have been very convenient if the user of our eDSL could define the shortfall as:

```
shortfall :: Obs Double
shortfall = maxObs (Konst 0) (Konst 100000 - grainYield)
```

To perform observable arithmetic, we would need the help of two more observables and make `Obs` into an instance of the `Num` class:

```
LiftD   :: UnaryOp → Obs Double → Obs Double
Lift2D  :: BinaryOp → Obs Double → Obs Double → Obs Double

instance Num (Obs Double) where
  fromInteger = Konst . fromInteger
  o1 + o2 = Lift2D BAdd o1 o2
  o1 * o2 = Lift2D BMul o1 o2
  ...etc...
```

The reason LiftD and Lift2D take UnaryOp and BinaryOp respectively and not (Double → Double) and (Double → Double → Double) is that if the GADT constructors for

Observables carry function fields, GHC cannot automatically derive Show, Eq, or Ord for them. Haskell does not provide a built-in way to compare or show arbitrary functions. Therefore, instead of using (Double → Double) directly, I have defined ADTs or enumerations for each unary and binary numeric operator that is supported.

```
data UnaryOp
  = UNegate
  | UAbs
  | USignum
  deriving (Eq, Show, Ord)

data BinaryOp
  ...
```

What making Obs into an instance of the Num class does is just, making the syntax easier for the user. This way, the user doesn't have to write Lift2D BDiv.... Instead, the contract above would automatically be translated to:

```
shortfall :: Obs Double
shortfall = maxObs (Konst 0) (Lift2D BSub (Konst 100000) grainYield)
```

The last piece of the puzzle is `MaxObs`. The reason why we have not simply defined a lifted max operator similar to the other binary operators is because of the behaviour we want. The expected behaviour from max is to return the bigger of the two things we are comparing. For example, if we apply max on lists it would return the larger of the two entire lists based on some global ordering (such as lexicographic), which is not the behaviour we want. However, we want to get a combined list where we compare each element of the first list point-wise with the corresponding element of the second list. In the end, we construct a new list with the "max" elements from both lists. That's why we need to define our custom max observable:

```
MaxObs :: Obs Double → Obs Double → Obs Double
```

Finally, going back to the derivative, suppose each missing tonne pays 3 GBP. We create a contract that multiplies a standard payoff (one GBP) by "shortfall × 3":

```
shortfallContract :: Contract
shortfallContract = scale (shortfall * Konst 3) (one USD)
```

When someone acquires this contract at harvest, the eDSL samples shortfall—the actual difference (100000 - grainYield)—and multiplies all subsequent payoffs in (one USD) by that sampled value. If grainYield is above 100,000, the factor is zero, meaning no payout.

Boolean observable arithmetic Let's revisit the example of a *barrier option* that depends on Apple's stock crossing a certain threshold:

```
knockInAAPL :: Contract
knockInAAPL = acquireWhen
            (stockAboveThreshold 150.0 AAPL)
            (scale (konst 100) (one USD))
```

Rather than requiring our model to directly calculate when Apple's stock would go above or below 150, it would be nicer to construct this condition from more primitive observables. Ideally, we would express this as:

```
stockAboveThreshold :: Double → Stock → Obs Double
stockAboveThreshold threshold stock =
                    ((stockPrice stock) %>= (konst threshold))
```

To perform the comparison, we need to lift the ordinary comparison operators to work with observables. However, Haskell's type system doesn't allow us to directly overload the standard comparison operators for our observable types. To address this while maintaining readable code, we can define a family of specialised relational operators for observables:

```
(%<); (%<=); (%=); (%>=); (%>) :: Ord a => Obs a → Obs a → Obs Bool
(%<) = Lift2B (<)
(%<=) = Lift2B (<=)
...
```

These operators are implemented using a common lifting function:

```
Lift2B :: CompareOp → Obs Double → Obs Double → Obs Bool
```

This lifting function transforms standard binary comparison operations into operations that work on observables. With these operators defined, we can now write boolean conditions that compare observables in a natural and readable way.

For our *up-and-in* option example, we can now express the threshold condition directly:

```
knockInAAPL = acquireWhen
            (stockPrice AAPL %>= konst 150.0)
            (scale (konst 1000) (one USD))
```

This approach provides greater flexibility since we can compose arbitrary boolean expressions from simpler observables and comparison operations, rather than requiring specialised observables for each possible condition.

4.2 Pricing Model

One critical component we have yet to discuss is the *Pricing Model*. Traders rely on various models to price contracts, including binomial models [23], Monte Carlo simulations [2], and partial differential equations such as the famous Black-Scholes formula [7].

For this project, I chose the **binomial pricing model** because it clearly shows how a contract's value changes over time, which fits well with my goal of modelling value processes in Haskell. Its recursive structure works naturally with functional programming, making it easier to implement in my DSL compared to more complex methods like Monte Carlo simulations [1].

The Black-Scholes formula, while fast, does not support American options or show how a contract's value evolves over time, which is important for my project. Monte Carlo simulations are flexible but too complicated and slow for what I need [4].

The binomial model is simpler and more practical, making it the best choice for this work. The way it works is as follows [23]:

1. Divide the total time period between the **model start date** and the **contract expiry date** into discrete time steps (e.g., days, weeks, months, etc.).
2. At each step, the value being modelled can take one of two actions:
   (a) Increase by a specific factor.
   (b) Decrease by a specific factor.

Note: For simplicity, equal probability is assumed for up and down movement.

The factors for increase and decrease are determined by the *volatility* of the underlying asset being modelled - whether it is a stock price, interest rate, or other observable driving the contract. Volatility is a stochastic property that can be estimated using historical price movements—such as the price movement of a stock over the past month or the average movement over several months—to predict how much the price might vary in the future. While I have studied this concept in detail and developed it in my code, I will omit a deeper explanation here as it is less relevant to the broader discussion.

The resulting structure after applying the model is a **tree-like lattice** (Figure 4) that captures these up-and-down movements over time. Different variants of the binomial model apply distinct formulas for these movements. For my implementation, I chose the *Ho-Lee (HL)* [10] model for interest rates, which adds or subtracts a fixed amount at each time step. For exchange rates and stock prices, I selected the *Cox-Ross-Rubinstein (CRR)* [4] model, which accounts for geometric behaviour using percentage-based changes.

4.3 Value Process

Now that we have introduced the concept of a *value process* and the *pricing model*, we can start considering how to represent it in Haskell. The value process reflects the structure of the Pricing Model—in our case, a lattice. To represent this, I use an array of doubles, `ValSlice`, which holds the possible values at a single time step and `PR`, which is the entire evolution of values over time as an array of `ValSlice`.

```
type ValSlice = [Double]
type PR = [ValSlice]
```

Since our value semantics imply that we will apply different mathematical operations on value processes, such as addition, multiplication, and negation, it is useful to define `PR` as an instance of the `Num` class, `instance Num PR`. For this purpose, `PR` should be defined using the `newtype` keyword with a dummy constructor [14]:

```
newtype PR = PR ([ValSlice])
```

This implementation is sufficient for modelling contracts that depend only on numeric properties - the price of a stock, interest rates, etc. However, as we demonstrated earlier some contracts involve boolean events like whether a company defaults on a loan. It would be useful to have the ability to predict whether and when this occurs using the current process representation, but instead of a process of `Doubles`, we would need to model a process of `Booleans`. To achieve this we would parameterise `PR` and `ValSlice` with a type variable:

```
newtype PR a = PR ([ValSlice a])
type ValSlice a = [a]
```

To simplify defining the `Num` instance for `PR`, I also implement two helper functions, `lift` and `lift2`, for applying single-variable and two-variable mathematical functions, respectively, across all elements of a value process.

```
lift :: (a → b) → PR a → PR b
lift f (VP xss) = VP [[f x | x ← xs] | xs ← xss]

lift2 :: (a → b → c) → PR a → PR b → PR c
lift2 f (VP xss) (VP yss) = VP [[f x y | (x, y) ← zip xs ys]
                                       | (xs, ys) ← zip xss yss]
```

The `lift` function maps a unary operation (e.g., negation or scaling) over every value in each time slice of the value process, while `lift2` applies a binary operation (e.g., addition or multiplication) element-wise between two value processes.

Note that because `lift2` uses Haskell's standard `zip` function, when one process is shorter than the other, the resulting process will be truncated to the length of the shorter one. While this behaviour is desirable in some cases, it becomes problematic when combining payouts of contracts with different expiry dates using the `and` combinator. According to the evaluation semantics, the compound contract should take the *later expiry date* of the two contracts. To support this behaviour, we implement an additional helper function:

```
lift2Preserve :: (a → a → a) → PR a → PR a → PR a
lift2Preserve f (PR pr1) (PR pr2) = PR $ combine pr1 pr2
    where
        combine xss [] = xss
        combine [] yss = yss
        combine (xs:xss) (ys:yss) =
            [f x y | (x, y) ← zip xs ys] : combine xss yss
```

Using these functions, we can now define `PR` as an instance of the `Num` class:

```
instance Num a => Num (PR a) where
    (+) = lift2Preserve (+)
    (-) = lift2 (-)
    (*) = lift2 (*)
    negate = lift negate
    ...
```

4.4 Compiler (Valuation Engine)

Now that we have defined all building blocks and semantics, we can move on to the core of the eDSL compiler.

Before diving into the full implementation, let's consider the intuition behind it. Looking back at Figure 2, we observe that the compiler is defined by the evaluation semantics and takes two parameters: a *Contract* and a *Pricing Model*.

Another key observation is that all combinators, apart from `None` and `One`, take a contract as one of their parameters (refer to Table 1). This suggests that the evaluation function is naturally defined in a recursive manner, with `None` and `One Currency` serving as base cases. The way we evaluate all other contracts is guided by the evaluation semantics outlined in Table 3 and the acquisition date property (Figure 1).

```
evalC :: Model → Contract → PR Double
evalC model None = constPr model 0
evalC model (One cur) = exchange model cur
evalC model (Give c) = negate (evalC model c)
evalC model (And c1 c2) = (evalC model c1) + (evalC model c2)
evalC model (Or c1 c2) = maxValToday (evalC model c1) (evalC model c2)
evalC model (Scale obs c) = (evalDO model obs) * (evalC model c)
evalC model (AcquireOn d c) = discDate model d (evalC model c)
evalC model (AcquireOnBefore d c) = snell model d (evalC model c)
evalC model (AcquireWhen obs c) =
                discObs model (evalBO model obs) (evalC model c)
```

The compilation of observables to value processes has a similar pattern:

```
evalDO :: Model → Obs Double → PR Double
evalDO model (Konst k) = constPr model k
evalDO model (StockPrice stk) = stockModel model stk
evalDO model (LiftD op o) = lift (unaryOpMap op) (evalDO model o)
...

evalBO :: Model → Obs Bool →  PR Bool
evalBO model (Lift2B op o1 o2) = lift2 (compareOpMap op) po1 po2
```

The above implementation represents the compiler's core logic—three concise functions that perform pattern matching and leave most of the valuation logic hidden behind the pricing model. While this simplified version illustrates the key concepts, my final implementation of the compiler is a bit more complex. After adding validation and optimisations the size and complexity of the module increased.

Haskell's lazy evaluation plays a significant role in the design of the compiler. In the evaluation semantics, we have specified that the acquisition date for `None` and `One Currency` is effectively set to infinity. Representing this requires an infinite data structure, and lazy evaluation enables computation on demand [14]. This means that values in a `ValueProcess` that are never used or modified will remain uncomputed, saving unnecessary work.

However, this design introduces a challenge when computing the maximum of two processes—if both are infinite and we attempt to use the built-in `max` function, it would result in an infinite computation. To address this, we implement a `maxValToday` function which only compares the value with the most significance in financial terms—today's value.

For evaluating basic combinators such as `give`, `and`, `or`, and `scale`, as well as lifting observables, we use simple arithmetic operations (-, +, max, *, etc.). In contrast, more complex combinators like `One cur`, `AcquireOn d c`, `AcquireOnBefore d c`, `AcquireWhen o c`, and likely observables such as `StockPrice stk` require specific logic

from the pricing model. For instance, the model must predict exchange rates between a given base currency and the currency specified in `One cur` for each time step within the contract horizon. We will now examine the implementation of these model-dependent combinators in greater detail.

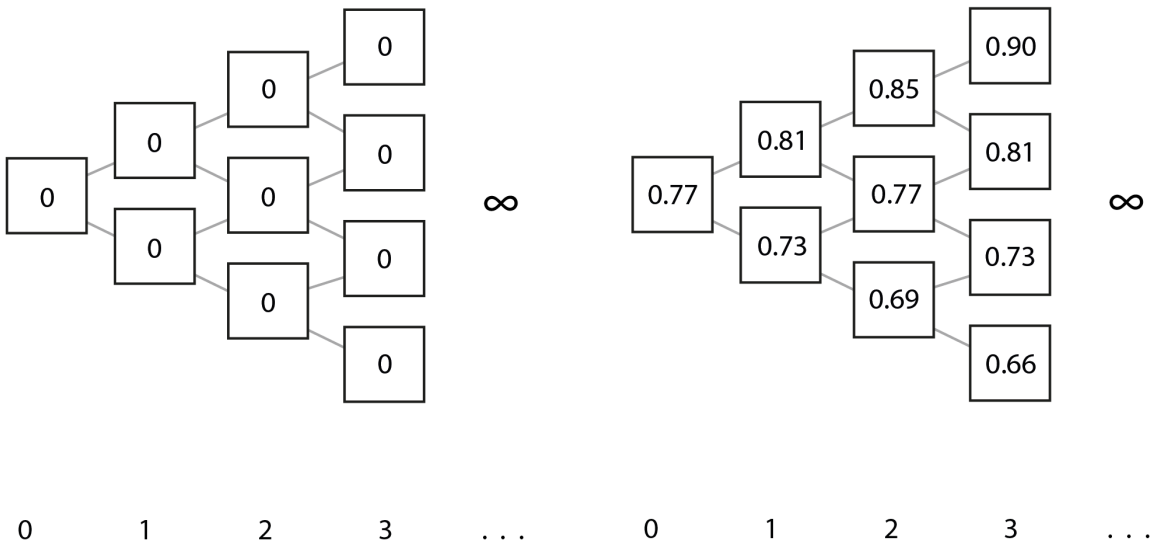
4.5 I Will Buy One

The simplest way to understand the valuation of processes is to visualise them. As mentioned earlier, the `None` contract represents acquiring nothing, so its value remains zero at all times. We use the model defined `ConstPr` function, which generates an infinite lattice with zero value at every node (Figure 3a).

Similarly, if the base currency for our pricer (compiler) is GBP, and we acquire `one GBP`, the exchange rate between identical currencies is a constant, so we use `ConstPr` again. However, the situation becomes more interesting if the contract involves a different currency, such as USD.

Anyone who has exchanged money between currencies more than once knows that exchange rates fluctuate daily. This variability introduces uncertainty in the value of the contract. Predicting future exchange rates is a perfect use case for applying financial models. [23]

For example, suppose the current exchange rate between GBP and USD is 0.77, and the volatility is 0.1819. These values can be used in a model like the Cox-Ross-Rubinstein (CRR) model to predict future exchange rates. The CRR model predicts that at each time step, the exchange rate will either increase by a factor $u = e^{\sigma\sqrt{\Delta t}}$, or decrease by a factor $d = e^{-\sigma\sqrt{\Delta t}}$, where σ is the volatility and Δt is the time step [4]. Explaining the relationship between volatility and time step deserves a section on its own but for the sake of simplicity, we won't explain how this is handled by the model here. By iteratively applying these formulas, we can generate a tree of potential future exchange rates, as depicted in Figure 3b.



(a) Value Process of `none` (b) USD to GBP exchange rate VP

Figure 3

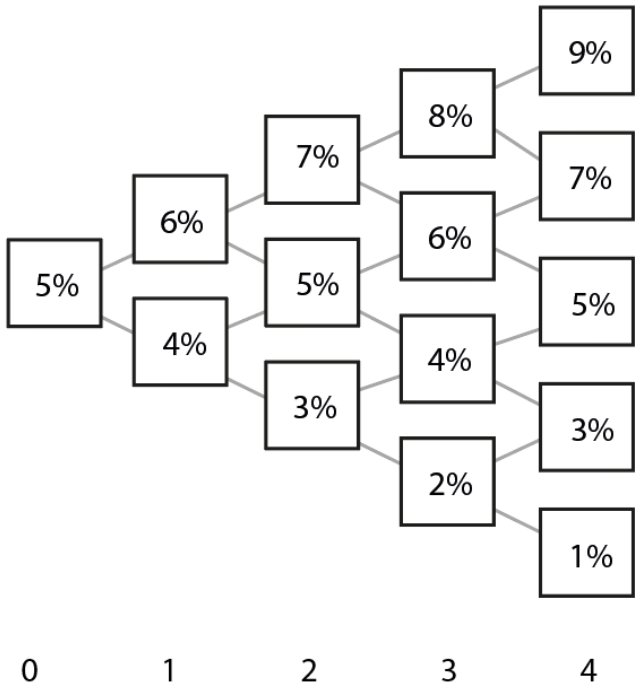


Figure 4: Interest rate evolution

4.6 Discounting

We cannot just carry on working with infinite processes, so in this subsection, we introduce a way of truncating a process.

Let us revisit the zero-coupon discount bond (ZCDB) example, which costs £1,000 today and pays £1,300 to the holder four years later. Does this mean the holder simply earns £400 in four years?

Not exactly. Imagine you have two options: buy the bond or keep your money in a bank account. If you choose to keep the money in your account, you could earn a guaranteed interest rate every year for four years. You would expect to earn at least as much as the bond offers—plus some extra compensation for the risk of holding the bond. Thus, the amount you earn from buying the bond is not as simple as subtracting £1,000 from £1,300. Instead, you must account for how far in the future you will receive the £1,300 and what interest rates were during that time.

Since interest rates are not constant (unless you have signed a long-term contract with a bank), we use models to predict future interest rates. To demonstrate a different model this time, we use the Ho-Lee model, which takes the same inputs as a CCR model - initial value and volatility but uses simple addition rather than multiplication to predict the values at the next time step.

For example, if the interest rate is 5% this year and the volatility is 1%, then next year's rate could be either 6% (up) or 4% (down). Applying the model recursively, in two years, the possible rates become 7%, 5%, or 3%, as shown in the binomial tree (Figure 4). The biggest flaw of this model is that values could go negative which is not a realistic assumption. [10]

After we've generated an evolution of interest rates we can now estimate the present value of £1,300 paid in the future using the process of discounting. First, we need to

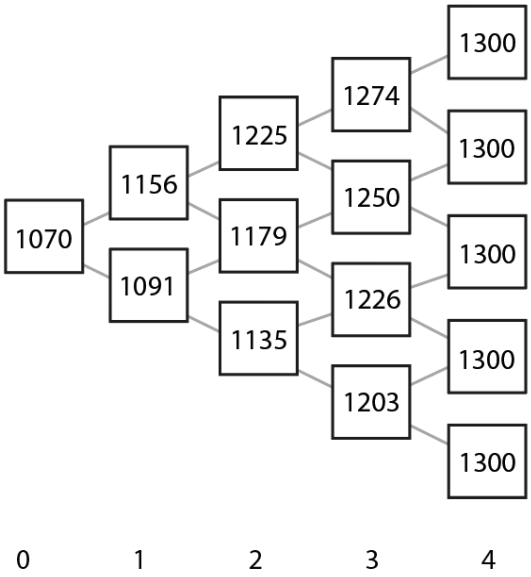


Figure 5: Discounted value process of `zcdb'`

generate the value process for the contract we will be discounting.

```
zcdb' :: Contract
zcdb' = acquireOn (date "01-01-2029") (scale (konst 1300) (one GBP))
```

From the previous section, we know that the value process of `one GBP` has 1 in each node of the lattice (assuming the base currency is GBP). Next, we need to scale this by the value process of the `konst` observable 1,300 which results in a lattice with 1,300 at each node. Lastly, we apply the process of discounting backwards through the tree.

Starting at the terminal nodes, we calculate the value of each node one step earlier using the formula:

$$V_{t,i} = \frac{V_{t+1,j} + V_{t+1,j+1}}{2(1 + R_{t,i}\Delta t)},$$

where $V_{t,i}$ is the value at node i at time t, $R_{t,i}$ is the interest rate at node i at time t, $V_{t+1,j}$ and $V_{t+1,j+1}$ are the values of the next time step (up and down paths), and Δt is the time step - one year in this case. [4]

We repeat the process of discounting through the tree until we reach the root node (Figure 5). According to the model the predicted value today of `zcdb'` is £1,070. The only step we need to take before we get the final payoff is to subtract the Value Process of `pay (date "01-01-2025") 1000 GBP` from the Value Process of `zcdb'` which gives us £70 as a result at the root node.

Implementation Implementing the discounting logic in Haskell turned out to be an engaging challenge. Since discounting operates in reverse—starting from the final layer of the value process and working backward—I needed to prepend each newly computed layer to build the final result. However, prepending introduces a complication: it forces the entire structure to be traversed repeatedly, which can lead to inefficiencies, especially in a lazily evaluated language like Haskell.

To address this, I developed three different implementations of the discount function, each with its own approach to handling the backward traversal and layer accumulation. I've documented these alternatives in the appendix. Choosing the most efficient solution would require benchmarking, as Haskell's lazy evaluation model makes it non-trivial to reason about or measure the time complexity directly. I encountered a similar situation when implementing the `snell` function, which follows the same backward-recursive pattern.

4.7 Options Revised

Now that we are equipped with the understanding of the evaluation of contracts, we can go back to the representation of options. In *CC* European options are defined as:

```
european :: Date → Contract → Contract
european t underlying = acquireOn t (underlying `or` none)
```

As we are evaluating each part of the contract recursively, the order in which each value process is computed is:

1. `underlying`
2. `none`
3. `underlying `or` none`
4. `acquireOn t (underlying `or` none)`

This means that when we are comparing the `underlying` and `none` the underlying would not have been discounted yet to the beginning of the value process. Therefore, a prediction about the value of the underlying would not have been made yet. Without an acquisition date set, what we are actually comparing is the value of the underlying if acquired today versus acquiring nothing today.

In order to fix this, we can move the acquisition of the underlying one level deeper into the recursion so that it is discounted at the right time:

```
european t underlying = (acquireOn t underlying) `or` none
```

Yet, depending on the overall system implementation, this might still cause problems. If acquiring nothing indeed turns out to be more profitable, the evaluation engine will return the value process for none, which is infinite. While a system might be designed to catch such behaviour, if it is not, it would be safest to define European options as

```
european t underlying = (acquireOn t underlying) `or` (acquireOn t none)
```

The same concept would apply to American options.

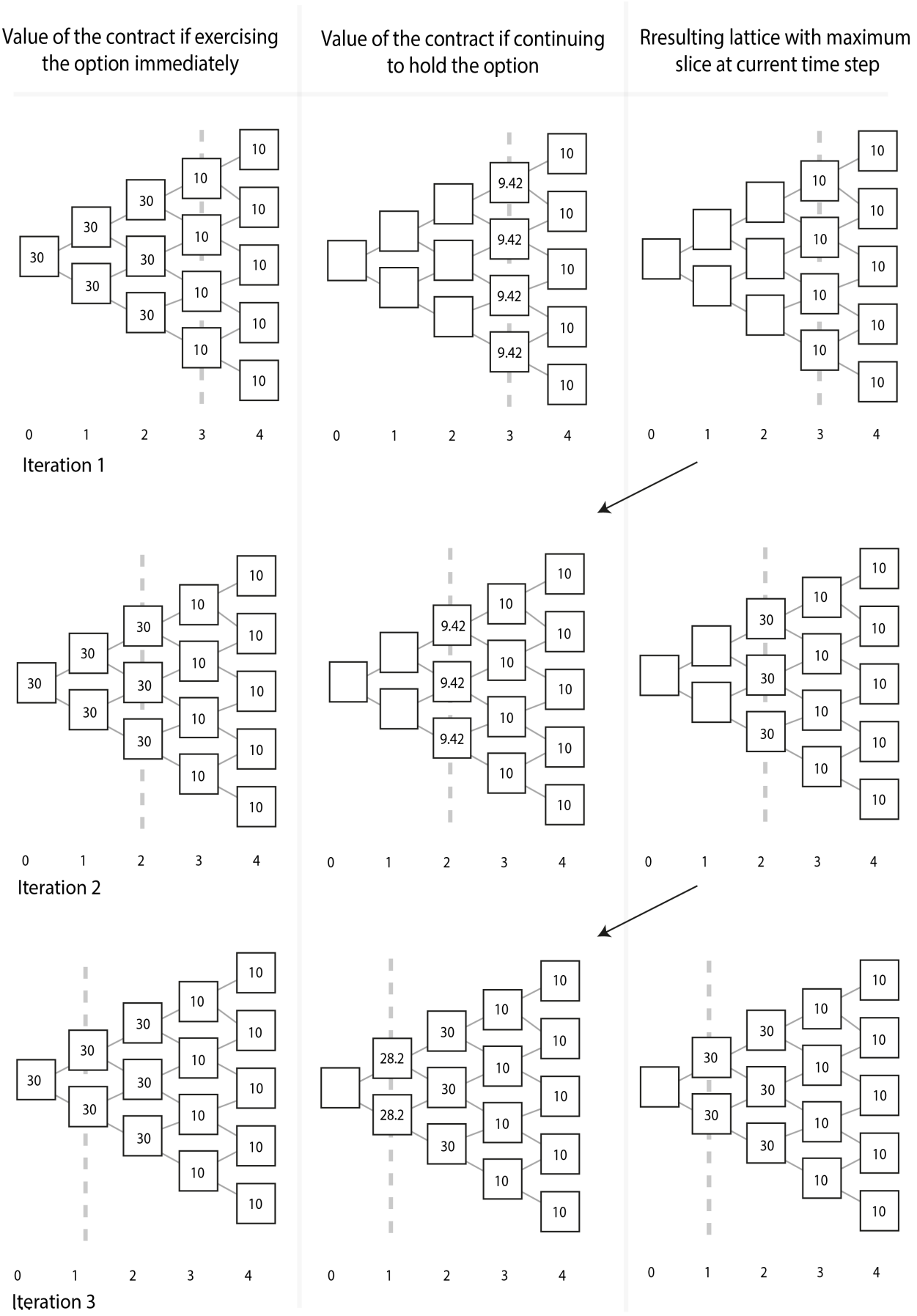


Figure 6: Step by step snell envelope for pricing American RACE Call option

4.8 Snell Envelope

Options are among the most widely traded financial instruments. As previously mentioned, there are two main types: **European** and **American**. **European options** allow the holder to decide whether or not to acquire an underlying contract on a specific date for a given strike price. In contrast, **American options** provide the flexibility to exercise this right at *any time on or before* the expiration date.

For an American option expiring in one month, the challenge lies in deciding the **optimal date to exercise** the option in order to maximize earnings. This is where the *Snell envelope* comes into play. The Snell envelope provides a systematic method for determining the **optimal stopping time** (i.e., when to exercise the option) by recursively calculating the maximum value achievable at each possible exercise date. At its core, it ensures that the decision to exercise the option at any given time is the best possible choice based on future potential gains [18].

Figure 6 breaks down the logic of the Snell envelope into steps. The lattice in the first column remains constant and represents the **possible value of the contract at each time step** if the option is exercised immediately at that step. Essentially, it shows the predicted underlying value minus the strike price for each time step. No discounting has been applied to this lattice, as it represents the **immediate exercise value** without a holding period.

In the second column, the lattice for the first iteration displays two values–the value of the contract at maturity (time step 4) and the value of the contract discounted back one step from maturity (time step 3).

The latter represents the contract's value if the option is held for one more step instead of being exercised immediately. For time step 3, we then take the maximum value from the corresponding slices of the lattices in columns one and two, adding it to our resulting lattice. This process allows us to determine the optimal value at each time step, considering whether it's better to exercise the option at that point or to continue holding it.

In subsequent iterations, we move one step back in time and repeat this process. The lattice in the second column now contains information about the best exercise time identified so far. This ensures that, at every time step, the lattice reflects the *highest attainable value for exercising the option at that moment* while continuously updating the optimal value discovered up to that point.

If we think about the Snell envelope from a practical perspective, we can understand that our current methods for predicting price movements using the *Cox-Ross-Rubinstein (CRR)* or *Ho-Lee (HL)* models present limitations. Specifically, the Snell envelope cannot identify a more optimal exercise date compared to standard discounting methods. Why is this the case? The fundamental reason lies in the linearity characteristics of these models. They do not account for sudden price spikes or crashes along the path.

To address this limitation, I implemented two artificial stock models:

1. Ferrari's stock (RACE) - designed to pay dividends and decrease in value at predetermined points

2. Apple (AAPL) - programmed to exhibit price jumps

Using the GUI, which, as discussed later, is part of the implementation of this project, you can directly compare the results between European and American options with identical parameters. To reproduce the results shown in Figure 6, follow these steps:

1. Select `"American Stock Call"`
2. Input an expiration date **5 months** from the current date
3. Set the **strike price** to **370**
4. Select `RACE` as the **underlying stock**

Then, use the same input for an "European Stock Call". Which one is more profitable?

5 Valuation Engine Optimisation

In a real-world trading system, contracts would rarely be written directly in our DSL by humans. Instead, a component of the system would read market data about different instruments directly, and rather than calculating the potential earnings of a single trade in isolation, it would consider, for example, executing additional trades to hedge against adverse market movements. Thus, the final contract being evaluated is typically a combination of a primary trade and various hedging strategies, assembled programmatically. While this program could theoretically be designed to create the most efficient contracts, mistakes are possible and there may be opportunities for optimisation.

In the previous section, we gave examples of value processes involving fewer than 5 time steps, resulting in relatively shallow trees. However, in practice, these trees can become very deep as more precise calculations are required. For example, a typical binomial lattice might use 100 to 500 steps for options with a 1-year maturity, but for high-frequency trading or more complex models, they might go up to 1,000 or more time steps to achieve higher accuracy. Consequently, for such deep lattices redundant computations could consume significant processing time and thereby comes the need for optimisation. This section explores how contracts can be rewritten or modified to speed up evaluation and how memoisation can reduce redundant work.

5.1 Optimisation Layer

It turns out that there is more than one way we could represent the same contract in our DSL. Building on an idea suggested in CC, I have utilised the evaluation semantics and properties defined in Section 3.5 to formally prove mathematical equivalences between different contracts. These equalities form the basis for an optimisation layer that simplifies and re-organises contracts to reduce the number of computations performed by the evaluation engine.

To illustrate the core idea behind this optimisation, consider a couple of examples. First, observe that applying the `give` operator twice is equivalent to doing nothing at all:

```
give (give c) = c
```

This equivalence follows directly from the algebraic property that negation is its own inverse: $-(-x) = x$. As a result, applying the `give` operator twice cancels its effect. We could prove that this equation is valid for some arbitrary currency k:

$$
\begin{aligned}
&\mathcal{E}_k[\![\text{give (give c)}]\!] \\
&= -(\mathcal{E}_k[\![\text{give c}]\!]) && \text{by (E1)} \\
&= -(-(\mathcal{E}_k[\![\text{c}]\!])) && \text{by (E1)} \\
&= \mathcal{E}_k[\![\text{c}]\!] && \text{by double negation}
\end{aligned}
$$

A slightly more intricate example involves the interaction between the `scale` and `or` combinators:

```
(scale o c1) `or` (scale o c2) = scale o (c1 `or` c2)
```

This equivalence shows that scaling before taking the maximum (using `or`) is equivalent to taking the maximum of two contracts and then scaling the result. The reasoning is based on the distributivity of multiplication over the maximum function, provided that the scaling factor is non-negative:

$$a \cdot \max(x, y) = \max(a \cdot x, a \cdot y) \quad \text{for } a \geq 0.$$

The proof proceeds as follows:

$$
\begin{aligned}
&\mathcal{E}_k[\![\text{(scale o c1) `or` (scale o c2)}]\!] \\
&= \max(\mathcal{E}_k[\![\text{scale o c1}]\!], \mathcal{E}_k[\![\text{scale o c2}]\!]) && \text{by (E3)} \\
&= \max(V[o] \cdot \mathcal{E}_k[\![c1]\!], V[o] \cdot \mathcal{E}_k[\![c2]\!]) && \text{by (E4)} \\
&= V[o] \cdot \max(\mathcal{E}_k[\![c1]\!], \mathcal{E}_k[\![c2]\!]) && \text{by multiplication distributivity} \\
&= V[o] \cdot \mathcal{E}_k[\![c1 \text{`}or\text{`} c2]\!] && \text{by reverse (E3)} \\
&= \mathcal{E}_k[\![\text{scale o (c1 `or` c2)}]\!] && \text{by reverse (E4)}
\end{aligned}
$$

where

o is numeric

$o \geq 0$

The reason why we prefer the second version is that otherwise we would have to construct two lattices representing the observable we are going to scale by. Since, at each time step, the number of nodes in the next layer grows by one to get the total number of nodes in a lattice, we can use the formula: N=(n+1)(n+2)/2. If each contract requires 1000 time steps that would mean creating and multiplying 501501 nodes twice. Instead, if the scaling is factored out, its corresponding lattice is computed only once.

The proofs for other equivalences follow a similar pattern, so these two examples sufficiently illustrate the core idea. The next step is implementing these equivalences in code. I initially developed a straightforward approach that recursively traverses each contract, using Haskell's pattern matching to detect patterns that could be simplified. This followed a top-down approach, where patterns in parent contracts are transformed first, and then nested contracts are checked recursively for non-optimal structures.

However, testing revealed that repeatedly passing contracts through the optimisation function produced further simplifications, indicating the algorithm was not reaching a fixed point. This occurred because top-down recursion can miss optimisation opportunities created by later transformations. To illustrate this limitation, consider one more

equivalence: `(none and c) = c`. The proof is trivial and will be omitted. Now let's look at this example:

```
give (none `and` give (one GBP))
```

When this contract passes through the optimisation layer with a top-down approach:

1. First, the pattern `give c1` is recognised, where `c1 = (none and give (one GBP))`. Since `give c1` is already in its canonical form, it remains unchanged.
2. Then we recursively optimise `c1`, recognising the pattern `(none `and` c2)`, where `c2 = give (one GBP)`. The transformation ignores the left branch of `and`, leaving just `c2` and optimising it recursively.
3. The third and fourth recursive passes handle `give c3` and `one GBP`, both already in their simplest forms.

Note that at each recursive pass, the nested contracts are not visible to the pattern matching at higher levels and that's why I'm using `c1, c2, c3` to represent them.

The result after one complete pass is:

```
give (give (one GBP))
```

This is clearly suboptimal, as it contains a pattern that matches our very first equivalence rule (`give (give c) = c`). The problem is that our optimisation revealed a new pattern that could be simplified, but our top-down approach doesn't revisit parent nodes after optimising children.

A bottom-up approach proves more effective because it ensures that all subcontracts are fully optimised before their containing contracts are checked. This guarantees that when a pattern is checked at a higher level, all potential simplifications in its subcomponents have already been applied. In the example above, a bottom-up approach would first optimise the innermost expressions, ensuring that when we reach the top level, we are working with the fully simplified `one GBP` rather than an expression that could be further reduced. This approach achieves true fixed-point optimisation.

Some additional optimisations which are implemented include:

- Identity scaling: `scale 1 c = c`
- Nested scaling:
  `scale konst k1 (scale konst k2 c) = scale (konst k1 * k2) c`
- Zero comparison simplification: `c `or` none = c`
- AcquireOn simplification:
  `acquireOn d (acquireOn d' c) = acquireOn d c` if $d == d'$
- Scaling distribution over and:
  `(scale obs1 c1) `and` (scale obs2 c2) =`
  `scale obs1 (c1 `and` c2)`
  if $obs1 = obs2$

While these optimisations might seem trivial, they can significantly reduce the computational overhead when dealing with complex, composed contracts. Moreover, the formal semantic foundation ensures that these transformations preserve the contract's meaning while improving its evaluation efficiency. A valuable next step would be to benchmark the optimisation layer to quantify its impact—this has been added to my list of future work.

5.2 Caching

In the context of our eDSL for financial contracts, caching plays a crucial role in improving the performance of contract evaluation. Haskell, being a purely functional language, does not inherently recognise duplicate subcontracts or computations within our domain-specific language. This limitation can lead to redundant evaluations of identical subcontracts, especially in cases where contracts are deeply nested or repetitive.

For instance, consider the following example:

Here, the same sub-contract, `sharedSubcontract`, is repeated multiple times within the structure. Without caching, the evaluation of this contract would redundantly compute the value of `sharedSubcontract` for each occurrence, leading to significant inefficiencies.

5.2.1 Memoisation Approach

Our current approach to caching employs a memoisation technique using a hash map (implemented as a strict `Data.Map`). Each contract or observable is assigned a unique key, represented by the `Key` type, which includes constructors for contracts (`KContract`), double-valued observables (`KObsDouble`), and boolean observables (`KObsBool`). This design allows the same cache to handle both contracts and observables. The evaluation function first checks the cache for a precomputed result before proceeding with the computation. If the result is not found, the computation is performed, and the result is stored in the cache for future use.

```
data Key
  = KContract Contract
  | KObsDouble (Obs Double)
  | KObsBool   (Obs Bool)
  deriving (Eq, Ord, Show)

data Val
  = VDouble (PR Double)
  | VBool   (PR Bool)
  deriving (Show)

type Cache = Map.Map Key Val
data EvalState = EvalState {cache :: Cache}
type EvalM a = StateT EvalState (Either String) a

evalC :: Model → Contract → Date → EvalM (PR Double)
evalC model contract earliestAcDate = do
  st ← get
  case Map.lookup (KContract contract) (cache st) of
    Just (VDouble cachedPR) -> return cachedPR
    Just (VBool _)          -> throwError "Contract was stored with VBool"
    Nothing → do
      -- Not in cache
      result ← evalC' model contract earliestAcDate
      -- Insert the result
```

```
let newMap = Map.insert   (KContract contract)
                          (VDouble result)
                          (cache st)
put st  cache = newMap
return result
```

The `EvalM` monad transformer is used to manage the memoisation cache state throughout the recursive evaluation calls without explicitly passing it as a parameter. `EvalM` also encapsulates the error propagation in the eval function which makes error handling much more graceful. The monadic structure cleanly separates evaluation logic from caching mechanics and error handling. While `evalC` is specifically designed for contracts, `evalDO` and `evalBO`, which handle the evaluation and caching of double-valued and boolean observables, respectively, use the same approach.

Performance improvement Table 4 presents benchmarking results comparing the performance of the evaluation engine with and without caching for different categories of contracts. I used **Criterion** to perform performance analysis and divided the benchmarking examples into 5 distinct categories.

The results are averaged across multiple test runs to ensure statistical significance. The percentage difference is calculated as:

$$\text{Percentage Difference} = \frac{1}{n} \sum \left(\frac{\text{Without Cache} - \text{With Cache}}{\text{Without Cache}} \times 100 \right)$$

Where positive values indicate performance improvement (time reduction) with caching, and negative values represent performance degradation.

Contract Category	Difference
Simple Contracts Without Repetition	-1%
Complex Contracts Without Repetition	-76%
Simple Contracts With Repetition	2%
Complex Contracts With Repetition	116%
Random Generated Contracts	4%

Table 4: Benchmarking results for contract evaluation with and without caching.

The results demonstrate significant performance variability across different contract types. For simple contracts without repetition, caching introduces minimal overhead (-1%), while complex contracts without repetition show substantial performance degradation (-76%), likely due to cache management overhead exceeding benefits. Conversely, contracts with repetitive elements benefit from caching, with complex repetitive contracts showing dramatic improvement (116%). Random generated contracts show modest gains (4%), suggesting that caching effectiveness depends heavily on contract structure.

These findings indicate that adaptive caching strategies would be more effective than a one-size-fits-all approach. A potential improvement could be to **selectively enable caching** based on contract characteristics. However, below I suggest an even better strategy.

Important to note that I've created artificial test examples to push the boundaries of contract complexity within our DSL. These test cases demonstrate what's possible but

may not represent typical real-world usage. For truly meaningful benchmarks, we would need actual market data and real *portfolios*. In trading, a portfolio refers to a group of financial assets held by an individual or institution. In practice, these contracts often share common sub-structures (such as repeated payoff conditions or market observables), making cross-contract caching potentially beneficial. However, using real market data is beyond this project's scope.

Limitations Some other limitations of this approach include:

- **Memory Overhead:** Maintaining the cache introduces significant memory costs, especially for complex contracts with numerous unique subcontracts. This overhead scales with contract complexity and can impact performance. However, in the world of high-frequency trading, memory is rarely an issue. Since speed matters most in this context, I focused on testing performance rather than memory usage.
- **Single-Contract Caching:** The system currently clears the cache after evaluating each contract, preventing the reuse of repeating structures across multiple contracts. This limitation becomes particularly relevant when evaluating a full *portfolio* of contracts.
- **Structural Equality Inefficiency:** The current implementation relies on structural equality for cache keys, which becomes inefficient for large or deeply nested contract structures. [9]
- **Observable Sharing vs. Haskell's Purity:** This limitation of memo functions is pointed out in *CC* [20]. Further research revealed that in pure functional languages like Haskell, textually identical expressions aren't necessarily pointer-identical [9]. Compiler optimisations can inline or reorder code, losing apparent source code sharing. Relying on pointer equality to detect previously computed subexpressions is inherently brittle, as GHC transformations may cause identical sub-contracts to go unrecognized as such. This effectively relegates shared sub-contract discovery to runtime, rather than compile time.

5.2.2 Alternative Graph-Based Approach

An alternative to memoisation approach is a graph-based representation that explicitly preserves sharing in the contract structure. The paper *"Functional Programming with Structured Graphs" by B. Oliveira at el.* presents a model that extends conventional algebraic data types with explicit definitions and manipulations of cycles and sharing [19].

Implementing this idea would involve altering the base AST representation of the contracts and observables data type to a **directed acyclic graph (DAG)** where each unique subcontract exists exactly once. Each contract constructor becomes a node in this graph, with references to its subcomponents represented as edges. The evaluation traverses this graph, computing each node's value exactly once and storing the result with the node. This approach guarantees that shared subcontracts are evaluated only once, regardless of how many times they appear in the larger contract structure. Unlike memoisation, which relies on detecting structural equality during evaluation, the graph approach enforces sharing during the construction phase of the contract itself.

This technique requires additions to the code to maintain the node table and create smart constructors that ensure sharing. It's more complicated to build than simple

caching but provides better guarantees about performance and avoids the problems of cache methods that rely on comparing structures or checking pointer identity. Though I haven't implemented and tested this approach yet, it appears to be a promising way to improve the caching system.

6 Contract Validation

A critical aspect of the DSL that deserves greater emphasis than found in CC is the handling of invalid contracts. It is worthwhile discussing this since, as mentioned before, contracts will often be constructed algorithmically, which increases the likelihood of errors. Consequently, this was one of the areas where I invested significant effort in my implementation. All the invalid contracts discussed below are caused by ill-defined acquisition condition - lack of acquisition condition, expired contracts, expired nested contracts and acquisition conditions that could never be reached. I have identified that for some of the invalid contract structures, the most straightforward tool for detecting such contracts is a validation function. Different strategies are also discussed below.

6.1 Contracts With Infinite Horizon

For a start, an obvious invalid contract would be one without a maturity or expiry condition (e.g., a date or boolean event). More specifically, contracts lacking `aquireOn`, `acquireOnBefore`, or `acquireWhen`, which would later be referred to for shortness as the "acquisition combinators", carry no meaningful semantics and cannot be properly evaluated. While one might argue that such contracts should be acquired immediately, this approach goes against the defined evaluation semantics, which say that the maturity date of such contracts is infinity.

Prevention. One prevention strategy would be to use a validation function that recursively traverses a contract and takes a boolean parameter, signifying whether an acquisition combinator has been found. When any of the "base combinators" - `One :: Currency → Contract` or `None :: Contract` - are reached, we check if the boolean flag has been triggered and return failure or success accordingly.

Alternatively, if we simply need to check for the presence of an acquisition combinator the validation function could return success early, instead of setting a boolean flag as true and carrying on with other checks. In this case failure is assumed if a base combinator is reached. This is the approach I have implemented.

6.2 Expired Contracts

Second, another issue arises when a contract has already expired by the time we want to evaluate it - e.g. a contract that expires on 01-01-2024 but today is 01-01-2025. Since our typical goal is to determine a contract's present value, an already-expired contract logically cannot carry any value today — it cannot be acquired retroactively. This becomes more complex with compound contracts, such as:

```
c = acquireOn (date "01-01-2024") c1
    `and` acquireOn (date "01-01-2026") c2
```

In this example, we must decide whether the entire compound contract is invalid or the contract should be evaluated regardless and the second branch should be considered worthless. This situation is very similar to the third type of potentially invalid contracts.

Prevention. To prevent these, the validation function is extended to take the model start date as input parameter and once `acquireOnBefore` or `acquireOn` is encountered we check if the date the combinator carries is after the model start date. In case it is before an error is thrown, otherwise it either continues traversing the rest of the contract or returns success right away. As to the case above where only one branch of a compound contract is expired, in my implementation I regard the whole contract as expired.

For `acquireWhen` checking gets trickier, in fact it might not be required, and it will be examined later in this section.

Note that here, we only check for top-level expired contracts, and below, we discuss handling nested ones.

6.3 Expired Nested Contracts

The third pattern occurs when a contract contains nested contracts with earlier maturity dates than their parent. The handling of such cases is more controversial than the previous two. Consider:

```
c = acquireOn (date "01-09-2025")
      ( And (scale (konst 50) (one GBP))
            (acquireOn (date "01-07-2025")
               (give (scale (konst 100) (one GBP))))
      )
```

The top-level acquisition date of the nested contract ('01-07-2025') is valid from today's perspective, but not the maturity date of its parent contract ('01-09-2025'). This raises a fundamental question: should the entire contract be invalidated, or should expired components merely be considered worthless? I have gathered arguments for both approaches.

Prevention 1: Flagging the Entire Contract as Invalid

Advantages and Use Cases. This approach is definitely the safer option of the two and provides several advantages. First, it signals to users that their contract is ill-formed or logically inconsistent. Second, it exposes logical mistakes immediately rather than allowing them to propagate. If a parent contract attempts to acquire an already-expired underlying, this typically indicates a conceptual error in the contract design. Failing early forces attention to these mistakes and requires the resolution of date conflicts.

From a debugging and maintenance perspective, this approach is more reliable. Silently treating expired components as no-ops can allow errors to pass unnoticed, with users potentially assuming their contract is valid when critical subcomponents are actually worthless. For instance, in our example above, if the "01-07-2025" date were meant to be "01-10-2025," treating the expired component as valueless would create a significant miscalculation—the system would think the compound contract is profitable by ignoring the 100 GBP payment obligation. At scale, such errors could lead to substantial financial losses in minimal time. If, however, the earlier date was not a mistake but the contract was flagged as invalid, there would be missed potential earnings.

Implementation. Experimenting with how to prevent or catch invalid contracts has been an utterly interesting aspect of this project. I explored various methods to prevent or detect invalid contracts. One approach involved using GADTs to check dates at compile time, but this proved impractical since contracts are typically constructed at runtime.

I also considered wrapping contracts in a structure like this:

```
data ContractWrapper = ContractWrapper
    -- Acquisition date is either Inf or UTCTime
    { acqDate   :: AcqDate
    , contract  :: Contract
    }
```

The idea was to use this wrapper together with a safe constructor for the `AcquireOn` and `AcquireOnBefore` combinators that checks the acquisition date of child contracts. However, this approach compromises readability.

My final implementation uses a validation function that recursively traverses the contract, taking both the contract and model start date as parameters. When encountering an acquisition combinator, it compares the date parameter with the new acquisition date. If the new date precedes the old date, an error is thrown:

```
validateC :: Contract → Date → Either Error ()
...
validateC (AcquireOn d2 c) d1
   | d1 > d2 =
        Left "Error: Attempting to acquire contract
             with an earlier expiry date at a later date."
   | otherwise =
        c' <- validateC c d2
...
```

Initially, I attempted to integrate validation within the evaluation function, believing that this would reduce redundant contract traversals—once for validation and again for evaluation. However, I soon realised that this approach introduces other inefficiencies. Consider a contract of the form:

```
acquireOn x (c1 `and` c2)
```

Here, c1 is a complex but valid contract, and c2 is invalid. If validation were only performed during evaluation, the system might spend significant computational resources processing c1, only to fail later when evaluating c2. While Haskell's lazy evaluation mitigates some of this inefficiency by only evaluating what is strictly needed, I preferred a design that validates contracts in advance, ensuring a clear separation of concerns.

At one point, I experimented with combining the optimisation layer and the validation once more to minimise the number of times a contract is recursively traversed. However, I discarded this idea as it would have a negative impact on code maintainability. It would make errors harder to detect, and, in some cases, optimisations might even transform an invalid contract into a form where the invalidity goes unnoticed. Therefore my final decision was to have a stand alone validation function.

Prevention 2: Treating the Expired Part as Worthless

Advantages and Use Cases. There is a subtle hint in CC that these should be regarded as valid contracts, and the expired nested part should be ignored. This is implied from the valuation rules for the `or` combinator, which specify that both branches are compared only if neither has expired. Otherwise, the non-expired branch is selected regardless of potential profitability. Similar logic applies to the `and` combinator.

When a date or event in the contract has passed, that portion of the contract transforms to zero value—carrying neither obligations nor payoffs once beyond its horizon. This aligns with the idea that contracts evolve over time, and certain rights or obligations naturally become void once their maturity date passes.

This approach offers practical advantages in certain contexts. From a real-world perspective, defaulting to a zero-valued underlying rather than triggering errors is helpful in portfolio-level or algorithmic contexts where date misalignments are common. When constructing large portfolios with multiple layers of optionality, minor date misalignments might occur, and this approach allows valuation to continue rather than fail entirely.

Furthermore, this approach elegantly handles knock-out or barrier scenarios, where certain exotic contracts become effectively empty once specific conditions are met. After trigger conditions are satisfied, treating the contract as zero-valued is conceptually simpler than raising exceptions. This method also accommodates situations with partial or uncertain knowledge of dates, where final schedules may not be known far in advance.

Implementation. For this version, the validation function no longer needs to verify the acquisition date of every nested contract. Instead, it would check only for the first two types of invalid contracts.

The check for expired nested dates is moved to the evaluation function, but rather than throwing an error, it evaluates the expired nested contract to a constant process of zeros, representing contracts without obligations or payoffs. In order to facilitate the check, a date parameter, representing the earliest date the nested contract could be acquired, is passed to the evaluation function. Same as with the validation function, this parameter is initially set to the model start date. Alternatively, the check could also be done in the optimisation layer, and the matured nested portion could replaced with the non-combinator, which similarly represents obligation-free and payoff-free contracts.

An interesting case arises when considering the `acquireOnBefore` combinator, which allows the acquisition of an underlying contract at any time before a given date. Consider the following example:

```
c = acquireOnBefore (date "01-09-2025")
      ( And (scale (konst 50) (one GBP))
            (acquireOn (date "01-07-2025")
               (give (scale (konst 100) (one GBP))))
      )
```

Here, the contract allows acquisition before the 1st of September 2025. The question arises: how should the expired components be handled? Should the expired underlying contract be treated as an empty process, or should it still be evaluated? Since the `acquireOnBefore` combinator provides flexibility in when the contract can be acquired, there remains a possibility that the contract was exercised before its maturity. In this case, it may be appropriate to evaluate it while disregarding any obligations beyond its

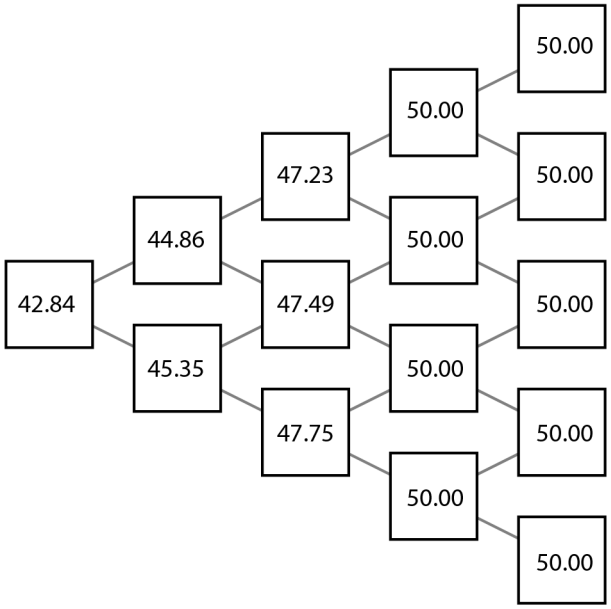


Figure 7: American with nested expired part

maturity. If this approach were followed and we assume today's date is 01-05-2025, the resulting lattice for this contract `acquireOnBefore` combinator would be (Figure 5). The snell envelope algorithm would judge that the best time to exercise the right to acquire the obligations is just after the nested contract (which carries losses) has expired.

While this scenario is highly uncommon in traditional financial contracts, and someone trained in finance would unlikely ask themselves such question, as a developer I inevitably find myself thinking of such edge cases since often I don't know if this is a possible real life scenario and whether it is worth reasoning about. However, as we said, the financial sector is ever-growing, and every day, new contracts emerge, so it might not be long until such edge cases become more common. That's why I believe addressing such edge cases is essential for building systems that can handle all types of financial contracts.

Verdict

After careful consideration, my conclusion is that the first approach — invalidating contracts with expired nested components — provides the safer option for most practical applications. This approach prevents subtle logic errors from propagating through the system, forces date conflicts to be resolved explicitly and aligns with the principle that contracts should maintain temporal consistency.

Moreover, since this DSL is embedded in a functional language known for strong type safety, it makes sense to apply the same principles here. Catching invalid contracts early through validation aligns with functional programming's core philosophy and gives users immediate feedback about contract problems.

However, exploring both directions has proven valuable from both theoretical and implementation perspectives. The second method works well for complex portfolios or

exotic contracts where some parts expiring early is actually intended. It raises interesting questions about American-style options and how time works in contract models. Since I found second approach more technically challenging to implement, it is the one I retained in my final implementation.

For production systems, I recommend configurable validation that defaults to the stricter first approach but allows for opt-in behaviour of the second approach when explicitly requested and documented.

6.4 Boolean Observables Validation

The case for the `acquireWhen` combinator differs significantly from time-based acquisition. While `acquireOn` and `acquireOnBefore` rely on deterministic dates, `acquireWhen` depends on boolean observables (e.g., `stockPrice > 100`), which introduces uncertainty about whether the condition will ever be satisfied.

A contract with an `acquireWhen` condition that never evaluates to `True` is not inherently invalid. This behaviour is fundamental to many financial instruments:

- **Knock-in options**: The contract only activates if the observable meets a threshold (e.g., "acquire when stock price > 100"). If the threshold is never reached, the contract remains dormant.
- **Knock-out options**: The contract terminates if the observable triggers, but persists otherwise.

In such cases, returning a zero-value process (via `constPr 0`) is semantically correct-the contract carries no obligations or payoffs if its condition remains unsatisfied. However, this approach requires careful consideration of termination conditions for observables that may never trigger.

Consider an observable such as `stockPrice > 100`. If the underlying asset's price never crosses this threshold during the contract's lifetime, we must establish appropriate stopping criteria. The current implementation uses a fixed iteration limit (500 steps) as a fallback termination condition, but this solution may not be optimal for all cases. Further experimentation with real-world contract structures and market data is needed to determine better approaches like, for example time-based expiration windows.

7 Testing and Evaluation

Given the criticality of such a library, the need for rigorous testing and evaluation goes without saying.

My testing approach evolved throughout the development process. While I considered starting with a test-driven approach, I decided that this would slow down the process and initially I placed more emphasis on *rapid development* and *prototyping*. My initial testing approach—if it could be called one—consisted of manually executing individual examples. Soon, this evolved into a helper function that ran all examples automatically. However, these examples functioned more as demonstrations of the system behaviour than actual tests, since I verified correctness merely through observation of outputs.

Once I had a basic working implementation of all system components and was satisfied with the structure, I implemented a more structured testing strategy, combining property-based testing with custom unit tests using the **QuickCheck library**.

7.1 Property Based Testing

Property-based testing generates numerous contracts of increasing complexity based on rules defined in an `Arbitrary instance` of the Contract and Observable data classes. For each test, a specific property is verified—for example, whether the optimisation layer preserves the semantic meaning of a contract. This specific test evaluates both the original contract and its optimised version, compares their results and verifies that they produce identical values. Thanks to QuickCheck's ability to generate hundreds, even thousands of random contracts, this single property test helped me catch that not only did my optimisation layer alter the meaning of a contract, but also many edge cases that caused the system to crash.

Without it, manually constructing such test cases would have taken weeks, and many edge cases would likely have remained undetected. QuickCheck proved to be extremely beneficial for ensuring that the library can handle any arbitrary contract.

Another crucial property tested was whether the optimisation layer reaches a fixed point. The evolution of this approach is detailed in the optimisation layer section. The property tests also verify the semantic correctness of core operations, ensuring that contract combinators like `Give`, `And`, and `Or` behave according to their mathematical specifications.

7.2 Unit Tests

While property-based testing provides broad coverage and randomly generated scenarios, unit tests enable targeted verification of specific functionality.

One important set of unit tests focused on validating the structure of the produced lattice during contract evaluation. Since the evaluation engine generates a binomial lattice to represent the value of a contract over time, it was essential to ensure that the lattice's length matched the expected number of steps based on the contract's expiry date and the step size of the model (e.g., day or month). Tests were written to verify that the lattice length was consistent with the number of time steps between the start date and the expiry date, divided by the step size.

Additionally, the test suite includes specific scenarios for option contracts, particularly checking whether the system correctly distinguishes between profitable and unprofitable cases. For instance, tests were implemented to ensure that European or American option contracts behave as expected when the underlying asset's value is above or below the strike price.

Further tests verified the correct evaluation of expired components, ensuring that expired branches contribute no value to the overall contract. Unit tests also validate the system's response to invalid contracts, such as those involving unsupported currencies or stocks, and whether they were correctly rejected by the evaluation engine.

7.3 Testing Coverage

Test coverage was measured using **HPC (Haskell Program Coverage)**. The current test suite achieves approximately 89% expression coverage, 85% alternative coverage, and 45% top-level definition coverage across the core modules.

The relatively low coverage of top-level definitions is expected, as testing does not directly execute type class instances such as `Eq, Show, Num,` etc. Similarly, the incomplete

alternative and expression coverage reflects branches added for theoretical completeness but unreachable in the current implementation.

If these parts of the code were to be ignored from the coverage report, the system would achieve nearly **100% test coverage**.

7.4 Conclusion

Developing a thorough testing strategy was a gradual but highly rewarding process. Since rigorous testing was introduced in the later stages of the project, I first had to take a step back to ensure the full correctness of all functions. As the project progressed, test automation became especially valuable when adding new features or modifying existing code. With each change, I could run the tests to quickly identify and fix any bugs that might have been introduced elsewhere in the system.

Although test coverage looks substantial, there is always room for improvement, particularly in adding more unit tests to ensure the evaluation engine performs correctly across all conceivable contract types.

8 Codebase Architecture and GUI

The system is implemented as a modular Haskell application with a web-based GUI client. The whole codebase is around ~2000 lines of code. The architecture separates concerns between core library components, main executable and user interface components. This design enables independent development and testing of components while maintaining system cohesion. The key components of the project are as follows:

8.1 DSL Library

The core modules of the project are stored in a library folder. This includes:

- `ContractsDSL.hs` - definitions for language combinators
- `ModelUtil.hs` - logic for the binomial pricing model
- `ValuationEngine.hs` - compiler with validation and optimisation components

Since we have already provided comprehensive explanations of each component's functionality in previous sections, we will proceed to the next piece rather than reiterating their details here. The full pipeline of evaluating a contract is illustrated at Figure 8.

8.2 Service Layer Components

The service layer bridges core logic with external interfaces through three principal modules:

- `Main.hs` - Runs the REST API server using Haskell's `Servant` framework
- `ContractVisualiser.hs` - Generates SVG representations of value processes
- `ContractParser.hs` - Implements a combinator parser for textual contract notation using `Parsec`

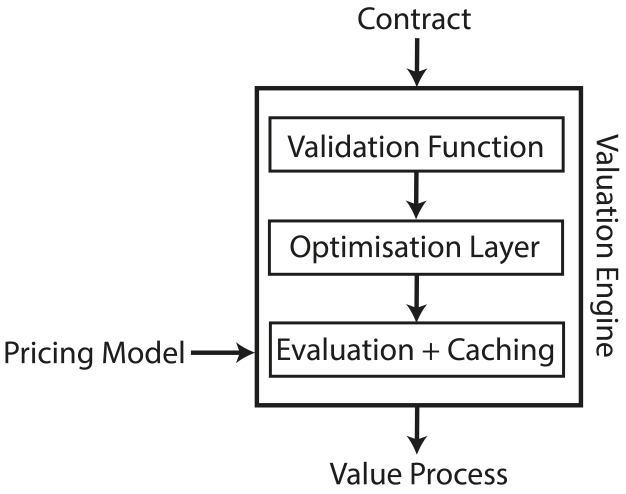


Figure 8: Full Contract Valuation Pipeline

The service layer has the following flow - incoming requests are first parsed into the DSL's abstract syntax tree via `ContractParser.hs`, then evaluated through the valuation engine, and lastly, the resulting value process is rendered visually by `ContractVisualiser.hs`. In fact, some of the graphs in this paper have been generated using the Contract Visualiser.

This layer also includes `ContractDefinitions.hs`, which provides template functions for common financial instruments, including European and American options, barrier options, and zero-coupon bonds. These predefined contracts are used in the GUI for testing and for benchmarking purposes.

8.3 GUI Client

Initially, I considered implementing the GUI client in Haskell. However, given the steep learning curve and limited functionality of Haskell's front-end libraries (unlike server-side libraries like Servant), I quickly reconsidered this approach. Instead, I developed a lightweight client-side interface using HTML, JavaScript and CSS, which provided greater flexibility and significantly reduced development time.

The client interface has very simple functionality (Figure 9). It provides a dropdown menu where users can select from the predefined contract structures from `ContractDefinitions.hs` we discussed above, or compose a contract from scratch by selecting Custom Contract. Upon selection, the page dynamically renders (via JavaScript) input fields corresponding to the parameters required by the selected contract type. When the user clicks the Evaluate button, the client sends a request to the server through the REST API. The server responds with a message containing the definition of the contract that has been just evaluated and an SVG visualisation of the generated value process lattice graph.

8.4 Testing and Benchmarking

The project includes a comprehensive test suite (`TestSuite.hs`) to ensure the correctness of the evaluation logic, optimisation layer and parser. Additionally, benchmarking scripts

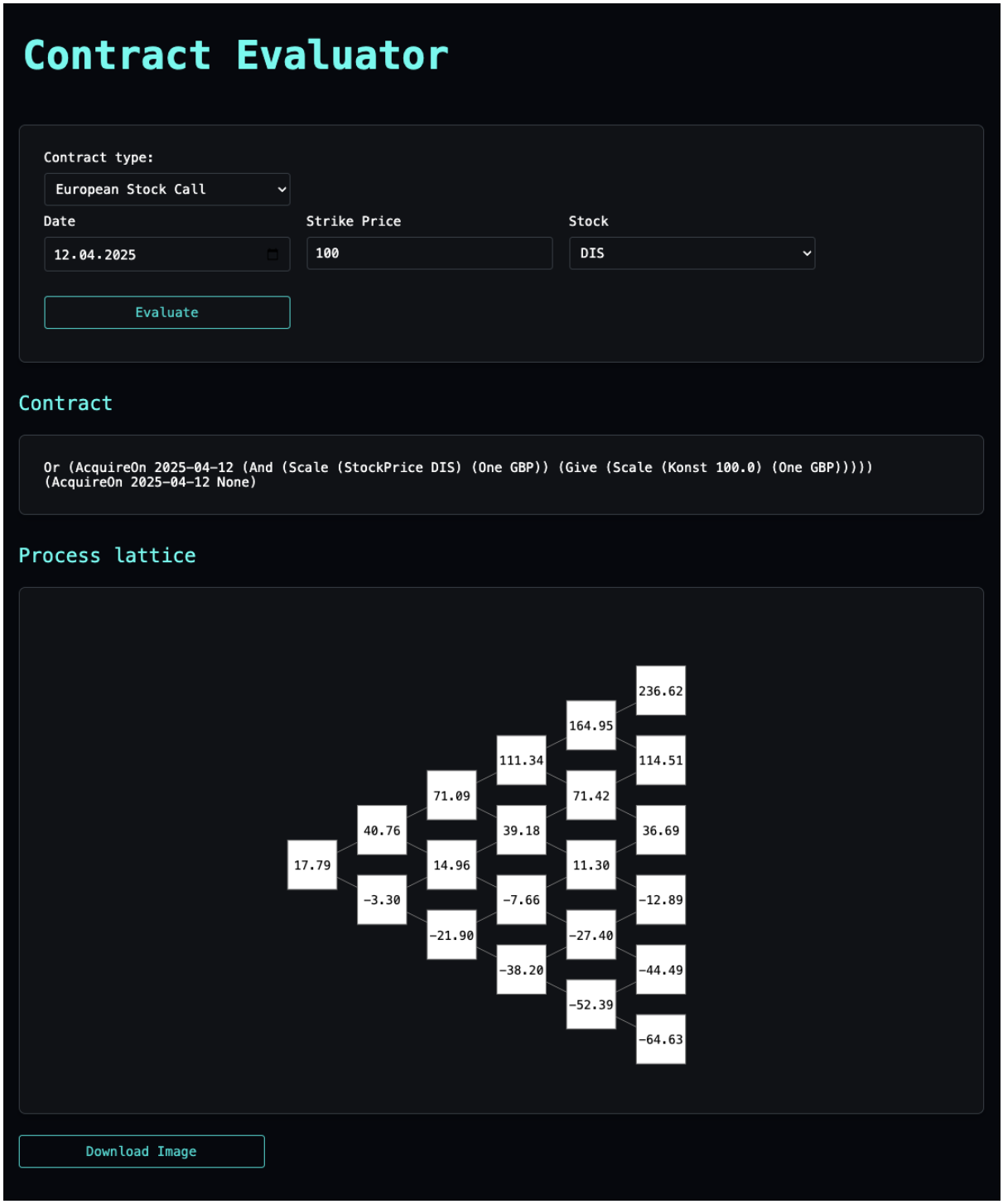


Figure 9: Client web interface (browser: Google Chrome)

(`CachingBench.hs`, `OptimisationBench.hs`) are provided to measure the performance of key components, such as the valuation engine.

8.5 Cabal Integration

The project is configured using Cabal, which facilitates building, testing, and benchmarking. The Cabal configuration ensures that all components, including the library, tests, and benchmarks, can be executed seamlessly.

9 Reflection

Overall, to me, my implementation of the eDSL effectively translates the theoretical concepts outlined by Peyton Jones et al. into a functional system. Representing contracts in a declarative style and structuring them as combinators has proven to be an intuitive and interpretable approach. By implementing a lattice evaluation model, I have demonstrated that these combinators offer not only clarity to users but also enable clean and maintainable code for evaluating contracts.

I am particularly happy with my choice of the binomial pricing model because I thoroughly enjoyed reasoning about how to implement it in Haskell. This choice also shows why Haskell works well for financial math: the tree structure of the binomial model fits naturally with recursive data types. The model also allowed me to harness some of the language's most profound features, like lazy evaluation, type classes and higher-order functions. [14]

Beyond recreating the original combinators and ideas in *CC*, I also added support for more contract types. Boolean observables allowed for contracts that trigger based on conditions (like knock-in options), and arithmetic operations on observables supported exotic financial instruments such as grain yield derivatives. Additions were also made for handling of invalid contracts, plus optimisation and caching mechanisms to improve performance. Benchmarking confirmed that the caching layer significantly reduced evaluation time for complex contracts.

Developing a web-based interface was initially a bonus goal and I am very happy that I implemented it in the end. It proved to be a valuable addition that provides an expressive yet simple way to compose and visualise contracts. It is a satisfying conclusion to the project's development.

Looking back, the project exceeded the initial goals and objectives set in the interim report. Not only that, but every feature and functionality was thoroughly tested, which strengthens my research and implementation even further. I believe this work successfully demonstrates the potential of Haskell for efficient and human-friendly financial modelling.

9.1 Pros and Cons of Haskell

Before starting my dissertation, I was determined to use Haskell. While I knew some benefits and drawbacks of pure functional languages, choosing Haskell was never in question—my focus was on finding the right application domain. Given my internship at a trading company and a full-time offer in trading after graduation, I was naturally drawn to exploring how my favourite programming language could be applied in finance.

My research revealed that several well-established financial companies already use functional programming languages in their technology stack: Barclays uses Scala, Jane

Street uses OCaml [17] and Standard Chartered Bank utilises a dialect of Haskell called Mu [6]. This confirmed the strong potential for applying functional languages to finance. Therefore, I set on the task of proving Haskell's suitability for financial modelling, domain-specific languages, and systems requiring high reliability.

Although my initial research pointed to Haskell's benefits, my understanding deepened significantly throughout this project. Here is my assessment of the pros and cons of using Haskell.

Benefits I tried synthesising the benefits of Haskell into a few distinct categories to highlight its strengths and how they align with the needs of this project.

- **Immutability and Referential Transparency:** In pure functional languages, functions are referentially transparent [24], meaning their output depends solely on their input and has no side effects. This property simplifies reasoning about code, as functions behave predictably and consistently. For example, in the context of my project, this property ensures the evaluation of a contract will always yield the same result for the same input, which is crucial for this domain.

- **Type Safety:** Haskell's strong type system catches errors at compile time rather than runtime, which is crucial when modelling financial contracts where mistakes can be costly. In this project, type safety proved invaluable when implementing Observables using Generalized Algebraic Data Types (GADTs). For example, when a function expects an `Obs Double`, the compiler guarantees it won't accidentally receive an `Obs Bool`, eliminating an entire class of logical errors.

- **Extensibility with Type Classes:** Haskell's type classes provide a powerful mechanism for extensibility and abstraction [12]. For this project, they allow defining generic operations over observables, contracts, and processes. By making observables and processes instances of the `Num class`, I could express complex mathematical operations using natural syntax, making the code both concise and readable. Furthermore, creating an Arbitrary instance for contracts facilitated property-based testing.

- **Mathematical Alignment:** Haskell's paradigms closely align with mathematics, which makes it a natural fit for financial modelling. This characteristic was especially helpful for enforcing and testing the mathematical properties of contracts expressed in the DSL. It also made the implementation of the optimisation layer very natural since it is based on the mathematical properties of our contracts.

- **Higher-Order Functions and Composability:** Haskell's support for higher-order functions and composability is a key strength in building modular and reusable code [12]. In this project, higher-order functions are used extensively in the parsing logic, such as with `map` to transform lists of currencies or stocks into parsers, and combinators like choice and buildExpressionParser to construct complex parsers from simpler ones. Composability ensures that small, well-defined functions can be combined to build complex financial models, making the codebase more maintainable, expressive, and easier to reason about. This aligns perfectly with the domain of financial contracts, where modularity and reusability are essential for handling diverse and evolving requirements.

- **Lazy Evaluation:** Haskell's lazy evaluation allows computations to be deferred until their results are actually needed [14], which can lead to significant performance improvements in certain scenarios. In the context of this project, lazy evaluation ensures that only the relevant parts of a financial contract are evaluated, avoiding unnecessary computations for parts of the contract that do not affect the final result. For example, when evaluating contracts with deeply nested structures or conditional branches (e.g., Or or AcquireWhen), only the active branch is computed.
- **Concurrency and Parallelism:** Even though I did not implement this in my project, my research highlighted Haskell's strength in this area [16]. Since data is immutable and there is an absence of side effects, there is no risk of race conditions or shared state issues, which simplifies the implementation of parallel algorithms. This is particularly beneficial in computationally intensive tasks, such as evaluating large financial contract lattices.

Limitations Despite its benefits, Haskell also has its limitations, which can pose challenges in certain scenarios.

- **Verbosity and Explicit Parameter Passing:** In Haskell, everything must be explicitly passed as a parameter, as there is no implicit state or global variables [12]. While this ensures clarity and avoids hidden dependencies, it can lead to verbose code, especially in systems requiring extensive parameterisation. For example, in this project, the evaluation engine requires the model, contract, and other parameters to be explicitly passed, which can make the code more cumbersome to write and read.
- **Performance Overheads:** While Haskell's lazy evaluation model can improve efficiency in some cases, it can also introduce performance overheads due to deferred computations and increased memory usage [21]. In computationally intensive applications, such as financial contract evaluation, careful optimisation is required to avoid performance bottlenecks.
- **Limited Ecosystem and Libraries:** Compared to imperative languages like Python or Java, the ecosystem of Haskell is smaller, with fewer libraries and tools available. This can make it harder to find pre-built solutions for specific problems or require a steep learning curve. This was the case when developing the GUI for the system.
- **Interfacing with External Systems:** Pure functional languages can be less convenient for tasks involving extensive interaction with external systems, such as databases, file systems, or GUIs [12]. The need to encapsulate side effects in monads like IO can make the code more complex and harder to follow.

Conclusion Understanding the trade-offs of using Haskell was crucial for the designing of my system and offered a great learning experience in the underlying principles of functional programming languages. While I was initially attracted to Haskell just because I found it really fun to code in this language, my appreciation for the language's strengths and limitations developed significantly throughout this project. Working through these challenges improved my ability to think recursively, write elegant code and grew my curiosity in exploring the capabilities of Haskell and other functional languages.

9.2 Project Management

The development of this project naturally divided into two stages - Stage 1, in the first semester, focused on research, rapid development and prototyping, and Stage 2, in the second semester, was dedicated to refinement, extension and optimisation. In Stage 1, October was dedicated to deciding on a project topic, thoroughly researching this area, and submitting my final proposal. November was then spent implementing my version of the combinators described in *CC*, together with the pricing model and evaluation engine, writing my interim report, and establishing clear goals for the second semester.

My work during the second semester progressed in three distinct phases, which emerged organically as the project developed:

1. Refining the system and completing the optimisation layer, caching and validation by early March.

2. Adding support for observable arithmetic and boolean observables; Finalising the dissertation draft by the end of March.

3. Developing the GUI, writing the reflection section, finalising the report, and creating a demo video by the project deadline (17th April).

These milestones were further broken down into smaller steps which I tracked in a personal diary, along with unpredicted tasks that emerged during development. Throughout both stages, I met with my supervisor approximately every two weeks, though not on a strictly scheduled basis, to discuss progress and refine the next steps.

One of the major challenges of this project was striking the right balance between developing new features and implementation depth, especially during Milestone 2. Since I thoroughly enjoyed researching and reasoning about new functionality I would often become absorbed in adding novelty. After guidance from my supervisor, I shifted focus to strengthening my existing code and documentation before adding more features, which greatly improved my time management during that milestone.

Looking back, I believe I found the right balance between research, expanding contract variety, implementing optimisation techniques, and conducting rigorous testing.

10 Future Work

Despite the strengths of my implementation, I recognise that there is still room for improvement. Importantly, this is fundamentally a research project, and while it aims to demonstrate the potential of a financial DSL in real-life scenarios, it is far from a state where it could be used in practice.

10.1 Next Steps in the Current Scope

To strengthen the research foundations and further demonstrate this DSL's suitability and advantages, several extra steps could be taken.

Extending the DSL A straightforward continuation of this work would involve extending the DSL itself to support more complex financial instruments, such as exotic options, credit derivatives, or multi-asset contracts. This would involve introducing new

combinators and observables to the language, as demonstrated with the extension for barrier options.

Refining the System In parallel with extending the DSL, efforts could focus on expanding the existing features of the remaining components and refining the system further. This could include developing more comprehensive property-based tests and unit tests to ensure stability, as well as additional benchmarking tests to better evaluate performance more precisely in a variety of cases. Formalising and documenting the proofs of the equivalences in the optimisation layer would also strengthen the theoretical foundation of the project. The optimisation layer itself could be extended to transform more equivalences. Similarly, experimenting with the alternative caching approach described earlier could yield performance improvements. The system could also be enhanced by implementing alternative binomial models beyond the Ho-Lee and Cox-Ross-Rubinstein models currently used.

Concurrency and Parallelism As mentioned in the previous section Haskell's immutability and absence of side effects make it particularly well-suited for parallel computation. This could be utilised to evaluate large financial contract lattices or multiple contracts simultaneously, significantly improving performance. Distributed evaluation across multiple nodes or machines could also be explored for large-scale trading.

User Experience Enhanced error handling and debugging tools could also be introduced to provide more detailed feedback on invalid contracts or evaluation failures, making the system more user-friendly.

10.2 Expanding the Scope

While the improvements outlined above would enhance the system within its current boundaries, there are numerous opportunities to expand the scope of the project. If we set a more ambitious goal of using the foundations we established to apply them to real-world trading or financial education a whole new avenue of possibilities opens up.

New Tools and Components Many different components could be developed based on the DSL. For example, mechanisms could be introduced to translate real-world market data into the language, enabling risk management and valuation in dynamic market conditions. Integrating external data sources, such as live stock prices or interest rates, would make the system more practical.

Alternative Evaluation Models While I considered implementing alternative pricing models, such as Monte Carlo simulations or the Black-Scholes formula, I ultimately decided against it. The balance between introducing new features for the sake of experimentation and strengthening the core research focus would have tipped toward the former, which I wanted to avoid. However, implementing such models in the future could significantly enhance the system's capabilities, particularly for evaluating more complex financial instruments.

Black-Scholes formula is specifically designed for European options and is proven to be more accurate [13]. It requires specific inputs derived from models that analyse market

data—a completely different direction that could be explored. This direction would focus on harnessing Haskell's strengths in processing large volumes of data efficiently.

Educational Applications Finally, there is potential for adapting the system for educational purposes. It could serve as a tool for teaching financial modelling and contract evaluation in academic settings, providing students with a hands-on understanding of how financial instruments can be modelled and evaluated using functional programming.

11 Laws, Social, Ethical and Professional Issues

While this project demonstrates the potential of functional programming for financial modelling, it is crucial to emphasise that the system is designed purely for academic research purposes and should not be used to make real-world financial decisions. This research prototype is not certified for accuracy, completeness, or regulatory compliance. Users and developers should consult licensed financial and legal experts before applying these concepts in practice.

References

[1] Joakim Ahnfelt-Rønne and Michael Flænø Werk. *Pricing Composable Contracts on the GP-GPU*. 2011.

[2] P. Boyle, M. Broadie, and P. Glasserman. *Monte Carlo Methods for Security Pricing. Journal of Economic Dynamics and Control*, 21:1267–1321, 1997.

[3] Shahbaz Chaudhary. *Adventures in Financial and Software Engineering*.

[4] J. C. Cox, S. A. Ross, and M. Rubinstein. *Option Pricing: A Simplified Approach. Journal of Financial Economics*, 7:229–263, 1979.

[5] Anton Dergunov. *Generalized Algebraic Data Types in Haskell. The Monad. Reader Issue 22*, page 5, 2013.

[6] Atze Dijkstra, José Pedro Magalhães, and Pierre Néron. *Functional Programming in Financial Markets (Experience Report). Proceedings of the ACM on Programming Languages*, 8(ICFP):234–248, 2024.

[7] Daniel J Duffy. *Finite Difference Methods in Financial Engineering: A Partial Differential Equation Approach*. John Wiley & Sons, 2013.

[8] Jeremy Gibbons and Nicolas Wu. *Folding Domain-specific Languages: Deep and Shallow Embeddings (Functional Pearl)*. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347, 2014.

[9] Andy Gill. *Type-Safe Observable Sharing in Haskell*. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 117–128, 2009.

[10] T. Ho and S. Lee. *Term Structure Movements and Pricing Interest Rate Contingent Claims. Journal of Finance*, 41:1011–1028, 1986.

[11] P. Hudak. *Building Domain-Specific Embedded Languages. ACM Computing Surveys*, 28, 1996.

[12] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. *A history of Haskell: Being Lazy with Class*. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007.

[13] John C Hull and Sankarshan Basu. *Options, Futures, and Other Derivatives*. Pearson Education India, 2016.

[14] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2 edition, 2016.

[15] SL Peyton Jones and Jean-Marc Eber. *How To Write a Financial Contract*. 2003.

[16] Simon Marlow, Simon Peyton Jones, and Satnam Singh. *Runtime Support for Multicore Haskell*. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 65–78, 2009.

[17] Yaron Minsky and Stephen Weeks. *Caml Trading–Experiences with Functional Programming on Wall Street. Journal of Functional Programming*, 18(4):553–564, 2008.

[18] Sabrina Mulinacci and Maurizio Pratelli. *Functional Convergence of Snell Envelopes: Applications to American Options Approximations. Finance and Stochastics*, 2(3):311–327, 1998.

[19] Bruno CdS Oliveira and William R Cook. *Functional Programming with Structured Graphs.* In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 77–88, 2012.

[20] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. *Composing Contracts: An Adventure in Financial Engineering (Functional Pearl). SIGPLAN Not.*, 35(9):280–292, September 2000.

[21] Simon L Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Nall International Series in Computer Science).* Prentice-Hall, Inc., 1987.

[22] D. Revuz and M. Yor. *Continuous Martingales and Brownian Motion.* Springer, 1991.

[23] Steven Shreve. *Stochastic Calculus for Finance I: The Binomial Asset Pricing Model.* Springer Science & Business Media, 2005.

[24] Harald Søndergaard and Peter Sestoft. *Referential Transparency, Definiteness and Unfoldability. Acta Informatica*, 27:505–517, 1990.

[25] P. Willmot, J. Dewyne, and S. Howison. *Option Pricing: Mathematical Models and Computation.* Oxford Financial Press, 1993.