A DSL for Financial Contracts

BSc Thesis Defence

Kasper Schiller, Department of Computer Science, 27-06-2023

KØBENHAVNS UNIVERSITET

Agenda

- **Introduction**
- **Design and Implementation**
- **Management of Contracts**
- **Pricing Contracts**
- **Demonstration, Testing and Future Work**

Introduction

The financial derivatives market has revolved in the past decades, and the aggregate value of assets linked to these instruments is several times larger than the world GDP [Hull, 2018].

New instruments are created every day and thus there is a need of a system that can express, manage and value new and unforeseen contracts as the market evolves.

Solution: Create a DSL, which can express contracts. Supply with functions that allow for management, and employ methods from continuous time finance for pricing.

Closely Related Work

Peyton, Eber and Seward have designed and implemented a DSL for exactly this purpose [Peyton et al., 2000].

SimCorp, a global FinTech company, have specifically created a module called XpressInstruments to minimise time to market for new, unforeseen derivatives [XpressInstruments User Manual, 2008].

Design: Time Representation

In [Peyton et al., 2000] and SimCorps solution, time is represented using absolute dates:

Acquire(2023-06-28, Scale(Underlying("AAPL", 2023-06-28)), One USD)

In our DSL, we represent such a contract using relative dates instead. Assume that todays date is *2023-06-27*. Then the contract above can be expressed as

Acquire(1, Scale(Underlying("AAPL", 0)), One USD)

Notice! The Acquire constructor shoots the time elements of the contract to be acquired. The absolute time choice was made primarily due to easing the process of pricing contracts.

Design: Constructors

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

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

Listing: Primitives for the Obs data type

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

Listing: Primitives for the Contract data type

Design: Logic Principles

Basic reasoning about contracts. When expressing and managing contracts, we assume that they hold. When pricing them, we **make sure** that they do hold.

1. $\mathit{Give}(\mathit{Or}(c_1, c_2)) \neq \mathit{Or}(\mathit{Give}(c_1), \mathit{Give}(c_2))$
2. $\mathit{Acquire}(t_1, \mathit{Acquire}(t_2, \ldots, \mathit{Acquire}(t_n, c))) = \mathit{Acquire}(t_1 + t_2 + \ldots + t_n, c)$
3. $\mathit{Give}(\mathit{Give}(c)) = c.$
4. $\mathit{Scale}(o_1, \mathit{Scale}(o_2, \ldots, \mathit{Scale}(o_n, c))) = \mathit{Scale}(o_1 \cdot o_2 \cdot \ldots \cdot o_n, c)$
5. $\mathit{Scale}(\mathit{Value}\ 0.0, c) = \mathit{Scale}(\mathit{Value}\ o, \mathit{All}[])$

… and **many** more.

Implementation: F#

We have chosen the DSL to be embedded in F# for several reasons, among

- functional languages have a declarative style: helps non-programmers validate contracts
- type-safety
- .NET libraries for unit testing

Expressing Contracts: A first example

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

A simple example of defining probably the most simple financial instrument there is: a bond which pays an amount of a currency at maturity date.

Expressing Contracts: Derivatives

In the DSL we can express derivatives using its payoff function or using constructors directly

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

Listing: European Call option defined from its payoff, $max(0, S_T - K)$.

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

Listing: European Call defined from its definition, $Or(0, S_T - K)$.

Expressing Contracts: Multiple flows

The All constructor can be utilized to define a contract which contain multiple flows, such as a coupon bond

```
let cb (T : int) (face : float) (rate : float) (yearlyFreq : float)
    (ccy : Currency) : Contract =
    let dates = int(yearlyFreq * 365.) // amount of coupon rate dates
    let rateDates = [dates .. dates .. T]
                    |> List.ofSeq // the actual coupon rate dates
    let couponFlows : Contract List = // a list of coupon flows
        List.map (fun x -> flow x (face * rate) ccy) rateDates
    let faceFlow = zcb T face ccy // the face flow at maturity
    All(faceFlow :: couponFlows) // the coupon bond contract
```

Listing: A coupon bond instrument.

```
let cb1 = 365 100.0 0.02 0.5 USD
> All
  [Acquire (182, Scale (Value 2.0, One USD));
   Acquire (364, Scale (Value 2.0, One USD));
   Acquire (365, Scale (Value 100.0, One USD))]
```

Listing: A concrete example of a bond with 1 year maturity, 100 USD in face value, 2% coupon rate and semi-annual coupon payments.

Management of Contracts

We want to be able to get information on contracts and manage them according to the information that we have.

Management of Contracts

Information Functions
maturity :: *Contract* $\rightarrow$ int *maturity(c)* returns the maturity date of c.
flows :: *Contract* $\rightarrow (int \times flow)\ List$ flows(c) returns a list containing the the time and value (if certain) of the incoming flows from the contract and whether they are certain or uncertain.
causal :: *Contract* $\rightarrow (int \times flow)\ List$ causal(c) returns the flows of c which are causal, at which time they are executed and at which time in the future the flows are dependent on. A causal flow is a flow which we do not know the value of as it is executed.
underlyings :: *Contract* $\rightarrow (S \times int)\ List$ underlyings(c) returns the name and time of the underlyings that c is dependent on.

Listing: Functions for getting information on contracts

Management Functions
simplify :: $((string \times int) \rightarrow \mathbb{R}) \rightarrow$ *Contract* $\rightarrow$ *Contract* *simplify E c* simplifies the contract c given the environment E.
advance :: $((string \times int) \rightarrow \mathbb{R}) \rightarrow$ Contract $\rightarrow int \rightarrow$ Contract *advance E c d* advances the contract c with d days given the environment E.
choose :: (Contract $\rightarrow \mathbb{R} + \bot) \rightarrow$ Contract $\rightarrow$ Contract *choose f c* makes a choice on the $Or(c_1, c_2)$ constructor based on the expected payoff of c_1 and c_2 according to f. If f returns $\bot$ for either c_1 or c_2, it returns $Or(c_1, c_2)$.

Listing: Functions for managing contracts

Management of Contracts

The *flows* function

$$
\begin{aligned}
ccy &:= USD \mid EUR \mid GBP \mid DKK \mid \dots \\
k &:= Value(\mathbb{R}) \mid Underlying\,(\mathcal{S}, int) \mid k_1 \times k_2 \mid k_1 + k_2 \mid k_1 - k_2 \\
&\mid max(k_1, k_2) \\
c &:= One(ccy) \mid Scale(k, c) \mid All\,([c_1 \dots c_n])\,,\ n \geq 1 \mid Acquire(int, c) \mid Give(c) \\
&\mid Or(c_1, c_2)
\end{aligned}
$$

Listing: Semantic Notation for constructors

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

Listing: The flows function, denoted as $\mathcal{F}$. The float option s keeps track of uncertainty, and the integer t keeps track of the date of the flow.

Management of Contracts

The *flows* function

$$c_1 = Acquire(10, Scale(Underlying("MSFT", 0), One\ DKK))$$
$$c_2 = Acquire(2, Scale(Value\ 10.0, One\ DKK))$$

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

Listing: An example

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

Listing: The flows function, denoted as $\mathcal{F}$. The float option s keeps track of uncertainty, and the integer t keeps track of the date of the flow.

Management of Contracts

The design philosophy should be clear: every function is recursive and goes through each constructor. They have been implemented in F# according to their semantics.

Information Functions
maturity :: *Contract* $\rightarrow$ int *maturity(c)* returns the maturity date of c.
flows :: *Contract* $\rightarrow (int \times flow)\ List$ flows(c) returns a list containing the the time and value (if certain) of the incoming flows from the contract and whether they are certain or uncertain.
causal :: *Contract* $\rightarrow (int \times flow)\ List$ causal(c) returns the flows of c which are causal, at which time they are executed and at which time in the future the flows are dependent on. A causal flow is a flow which we do not know the value of as it is executed.
underlyings :: *Contract* $\rightarrow (\mathcal{S} \times int)\ List$ underlyings(c) returns the name and time of the underlyings that c is dependent on.

Listing: Functions for getting information on contracts

Management Functions
simplify :: $((string \times int) \rightarrow \mathbb{R}) \rightarrow$ *Contract* $\rightarrow$ *Contract* *simplify E c* simplifies the contract c given the environment E.
advance :: $((string \times int) \rightarrow \mathbb{R}) \rightarrow$ Contract $\rightarrow int \rightarrow$ Contract *advance E c d* advances the contract c with d days given the environment E.
choose :: (Contract $\rightarrow \mathbb{R} + \bot) \rightarrow$ Contract $\rightarrow$ Contract *choose f c* makes a choice on the $Or(c_1, c_2)$ constructor based on the expected payoff of c_1 and c_2 according to f. If f returns $\bot$ for either c_1 or c_2, it returns $Or(c_1, c_2)$.

Listing: Functions for managing contracts

Evaluating Contracts (Pricing)

We want to be able to value any that can be expressed within the language. To do this, we have created recursive evaluation functions that operate on the created data types.

Evaluating Contracts (Pricing)

$$\mathcal{C} : (ccy \to \mathbb{R}) \to ccy \to \mathbb{R}$$

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

Listing: The currency evaluation function.

Evaluating Contracts (Pricing)

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

Listing: The Observable evaluation function.

Evaluating Contracts (Pricing)

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

Listing: The Contract evaluation function.

Evaluating Contracts (Pricing)

Problem: The environment E looks up the price of the underlying. But if the value is in the future we cannot measure it with certainty, so we cannot price the contract...

$$\mathit{Acquire(t,\ Scale(Underlying(\text{"foo"},\ 0),\ One\ cur)},\quad t > 0$$

Evaluating Contracts (Pricing)

Problem: The environment E looks up the price of the underlying. But if the value is in the future we cannot measure it with certainty, so we cannot price the contract...

$$\textit{Acquire(t, Scale(Underlying("foo", 0), One cur)},\quad t > 0$$

Solution: Integrate continuous time finance methods and utilize Monte Carlo methods!

A Quick Introduction to Continuous Time Finance

In the case of a non-dividend paying stock, we assume that its price follows the stochastic process also known as a Geometric Brownian Motion [Björk, 2009, p. 69]

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

The W_t term is a Wiener Process

$$\begin{aligned} W_0 &= 0 \\ W_{t+\Delta t} &= W_t + \mathcal{N}_{t+\Delta t}\sqrt{\Delta t} \end{aligned} \tag{2}$$

where $\mathcal{N}_1, \mathcal{N}_2, \ldots \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1)$ and Δt is a time increment of our choice [Werk et al., 2012].

Pricing Options: A European Call

Recall the payoff of a European Call

$$(S_T - K)^+ \tag{3}$$

We can substitute in (1) and get

$$C(T) = \left(S_0 e^{\left(u - \frac{1}{2}\sigma^2\right)T + \sigma W_T} - K\right)^+ \tag{4}$$

where we have $\Delta t = T$ so $W_T = \sqrt{T}\mathcal{N}$. Now, conduct Monte Carlo simulation and the value of the call is

$$V_{Call}(T) = I(T) \times \frac{1}{N} \sum_{i=1}^{N} C_i(T) \tag{5}$$

Pricing Derivatives: The General Case

In the general, non path-dependent case we have

$$V(T) = I(T) \times \frac{1}{N} \sum_{i=1}^{N} f\left(\begin{bmatrix} S_i^0(T_0) \\ S_i^1(T_1) \\ \vdots \\ S_i^k(T_k) \end{bmatrix} \right) \tag{6}$$

where f is the payoff function, calculated using the evaluation functions, and k denotes a specific stock. This is the essence of Monte Carlo simulation [Glasserman, 2003].

Pricing Derivatives: Implementation

$$\text{simulateContract} :: \text{int} \rightarrow \text{Contract} \rightarrow \mathbb{R}$$

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

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

Listing: Simulating the price of a derivative in the DSL using the Monte Carlo method.

Pricing Derivatives: The European Call

Recall the Black Scholes Formula which calculates the theoretical value of a European call

$$BS(S, \sigma, T, r, K) = S\Phi\left(d_1(S, \sigma, T, r, K)\right) - e^{-rT} K\Phi\left(d_2(S, \sigma, T, r, K)\right) \tag{7}$$

where $d_1 = \frac{log(S/K) + \left(r + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}$, $d_2 = \frac{log(S/K) + \left(r - \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}$ and Φ is the standard cumulative distribution.

Pricing Derivatives: The European Call

Consider a European Call with $T = 10$, $K = 95$, and the underlying stock to be AAPL, where $(S_0, \sigma, r) = (100, 0.05, 0.02)$.

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

Listing: The price of the option according to the MC simulation

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

Listing: The price of the option according to the BS formula

Correct by tolerance 10^{-2} with 100.000 simulations.

Demonstration

Now, let us define a Chooser Option instrument

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

Demonstration

Conceive that we have the following portfolio

```
let c = chooserOption 1 10 "AAPL" 120.0 USD
```

Demonstration

Conceive that we have the following portfolio

```
let c = chooserOption 1 10 "AAPL" 120.0 USD
```

Assume that one day has gone by. We update our environment and utilize the advance function

```
// Environment
let E(s,t) : float =
    match (s,t) with
    | ("AAPL", 0) -> 100.0 // Assume S_0 = 100.0 for the AAPL stock
    | _ -> failwith "price not found"
    // Advance on c
advance E c 1
```

Demonstration

Today we have to make a choice: acquire the put or the call. How to choose?

```
// Advance on c
advance E c 1
> Or // today we have to make a choice
  (Acquire (10, // acquire the call,
    Or (All [], Scale (Sub (Underlying ("AAPL", 0), Value 120.0), One USD))
    ),
   Acquire(10, // or acquire the put
    Or (All [], Scale (Sub (Value 120.0, Underlying ("AAPL", 0)), One USD))
    )
  )
```

Demonstration

Utilize the *choose* and simulateContract function together!

> **choose** :: (Contract $\rightarrow \mathbb{R} + \perp$) $\rightarrow$ Contract $\rightarrow$ Contract
>
> *choose f c* makes a choice on the $Or(c_1, c_2)$ constructor based on the expected payoff of c_1 and c_2 according to f. If f returns $\perp$ for either c_1 or c_2, it returns $Or(c_1, c_2)$.

```
let c1 = advance E c 1
choose (simulateContract 100_000) c1
> Acquire
  (10,
    Or (All [], // we chose the put option
        Scale (Sub (Value 120.0, Underlying ("AAPL", 0)), One USD)))
```

The choice can be validated using the BS formula for a put and call, respectively.

Evaluation of the DSL

Recall the logic principles stated earlier

1. $Give(Or(c_1, c_2)) \neq Or(Give(c_1), Give(c_2))$
2. $Acquire(t_1, Acquire(t_2, \ldots, Acquire(t_n, c))) = Acquire(t_1 + t_2 + \ldots + t_n, c)$
3. $Give(Give(c)) = c$.
4. $Scale(o_1, Scale(o_2, \ldots, Scale(o_n, c))) = Scale(o_1 \cdot o_2 \cdot \ldots \cdot o_n, c)$.

It is easy to show that they hold by going through the semantics of the evaluation functions. This is done in the paper.

Testing

To test the program, we have conducted black box unit testing using the XUnit .NET Framework along with FsUnit for idiomatic support. This is a seperate .NET Project.

A total of 175 test cases were conducted in total for management, evaluation and simulation functions. All passed.

```
Passed!  - Failed:     0, Passed:   175, Skipped:     0, Total:   175,
Duration: 1 m 49 s - Tests.dll (net7.0)
```

Future Work

- **Exchange Rates.** Needs to be non-constant... Redefine evalccy to be on the form $\mathcal{C} : (\mathit{ccy} \to \mathit{int} \to \mathbb{R}) \to \mathit{ccy} \to \mathbb{R}$
- **Interest Rates.** Implemnet $I(t)$ according to models such as the Hull-White or Cox-Ingersoll-Ross model to get a non-constant interest rate.
- **Correlation.** Consider correlation and covariance matrices to implement correlated Wiener processes for pricing.
- **Path-dependent Options and Early Exercising**. In Peytons DSL they can express and price contracts such as american options which are path dependent and have early exercise options. We are missing a constructor for this purpose.
- **Stochastic Volatility Models**. Implement the Heston Model instead of the GBM [Heston, 1993].
- **Optimization**. Quasi-Monte Carlo methods, parallel-programming...
- **Dates**...

Conclusion

We have successfully implemented a DSL which allows us to express, manage and price selected contracts.

There are still certain areas that demand further attention. Notably,

- path-dependent options cannot be expressed
- various aspects of mathematical finance theory is simplified such as constant interest rates, no correlation between stochastic processes, etc.

References

- [Björk, 2009] T. Björk. (2009). *Arbitrage Theory in Continuous Time.* Oxford University Press; 3rd edition.
- [Heston, 1993] S. Heston. (1993) *A closed-form solution for options with stochastic volatility and applications to bond and currency options*. Rev. Fin. Stud. 6, 327-343.
- [Hull, 2018] John C. Hull. (2018). *Options, Futures, and Other Derivatives*. 10th edition. Pearson.
- [Glasserman, 2003] P. Glasserman. (2003). *Monte Carlo Methods in Financial Engineering*. Springer.
- [Peyton et al., 2000] Jones SP, Eber JM, Seward J. (2000). *Composing contracts: an adventure in financial engineering (functional pearl)*. In: Proceedings of the 20th ACM SIGPLAN international conference on functional programming, ACM. pp 280292

References

- [Werk et al., (2012)] Michael Flænø Werk, Joakim Ahnfelt-Rønne, and Ken Friis Larsen. 2012. *An embedded DSL for stochastic processes*: Research article. In Proceedings of the 1st ACM SIGPLAN Workshop on Functional High- Performance Computing (FHPC12). ACM, New York, NY, 93102.
- [XpressInstruments User Manual, 2008] *Simcorp A/S (2008). XpressInstruments - Workflow. User Manual.* Responsible: IMS Data and Pricing. Based on SimCorp Dimension, Version 4.3.