Design Patterns in Haskell and Derivatives Pricing

Felix Matschke

September 27, 2009

Contents

1 Preamble

Derivatives pricing code should be correct and execute fast. Mainly because of the latter it is often connected to writing it in C++. This choice can be criticised: C++ is hard to write and difficult to debug (compared to other languages), which requires a lot of effort to ensure that the first goal – correctness – is reached. This energy can be better spent.

In the following text which traces a popular book for derivatives pricing in C++ ("C++ Design Patterns and Derivatives Pricing" by Mark S. Joshi), I am trying to show that by choosing Haskell as the programming language the complexity of derivatives pricing code can be vastly decreased - which is certainly a big factor in the production price of the libraries. Haskell has the reputation of being not a big factor off in terms of execution speed. This will not be benchmarked here.

2 Haskell

Haskell is great. Everyone should learn it.

This text will not try to teach Haskell. Haskell looks different on the first look and on the second look. It has a number of concepts and operators that are unusual. The book "Real World Haskell" is a good introduction. Concepts that seem relevant will be briefly introduced, but with that brevity these introductions are not destined to stand alone as any kind of reference.

3 A Simple Monte Carlo Model

3.1 The Theory

Below the commonly known formulas describing stock price evolution and Black-Scholes pricing theory. Given the stock price evolution described by:

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{1}$$

and the final payoff function f, the present value of a derivative is

$$e^{-rT}\mathbb{E}(f(S_t)) \tag{2}$$

if the expectation is calculated under the risk free process

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{3}$$

Following some derivations the price of a vanilla option with terminal payoff f can be written as:

$$e^{-rT}\mathbb{E}(f(S_0 e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}N(0,1)})) \tag{4}$$

So a simple algorithm for a Monte Carlo pricer is to draw n gaussian variables x with a distribution $N(0,1)$ and compute the average of

$$f(S_0 e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}x})$$

3.2 A simple Implementation of a Monte Carlo call option pricer

We need GSL to get the random number generators and Control.Monad for some monadic operations:

```
import Data.List
import GSL.Random.Gen
import GSL.Random.Dist
import Control.Monad
```

This contains our simple Monte Carlo calculator:

```
simpleMC1 :: Double → Double → Double → Double → Double → [Double] → Double
simpleMC1 expiry strike spot vol r sample =
  exp ((−r) ∗ expiry) ∗ sumAll / n
  where
    variance       = vol ∗ vol ∗ expiry
    rootVariance   = sqrt variance
    itoCorr        = (−0.5) ∗ variance
    mSpot          = spot ∗ exp (r ∗ expiry + itoCorr)
    sumAll         = foldl′ (+) 0 $ map sumItem sample
    n              = (fromIntegral $ length sample)
    sumItem gaussian = if payoff > 0 then payoff else 0
      where payoff = (−strike) + mSpot ∗ exp (rootVariance ∗ gaussian)
```

Factoring out the questions for input values significantly shortens the main body:

```
askForInput statement = putStrLn statement ≫ (liftM read $ getLine)
```

Main function doing the input and output:

$$
\begin{aligned}
&\mathit{main} = \mathbf{do} \\
&\quad \mathit{expiry} \quad\quad\;\; \leftarrow \mathit{askForInput}\ \texttt{"Enter Expiry"} \\
&\quad \mathit{strike} \quad\quad\;\; \leftarrow \mathit{askForInput}\ \texttt{"Enter strike"} \\
&\quad \mathit{spot} \quad\quad\quad\; \leftarrow \mathit{askForInput}\ \texttt{"Enter spot"} \\
&\quad \mathit{vol} \quad\quad\quad\;\; \leftarrow \mathit{askForInput}\ \texttt{"Enter vol"} \\
&\quad r \quad\quad\quad\quad\;\; \leftarrow \mathit{askForInput}\ \texttt{"Enter r"} \\
&\quad n \quad\quad\quad\quad\;\; \leftarrow \mathit{askForInput}\ \texttt{"Enter number of paths"} \\
&\quad \mathit{rng} \quad\quad\quad\;\; \leftarrow \mathit{newRNG}\ \mathit{mt19937} \\
&\quad \mathit{randomNums} \leftarrow \mathit{replicateM}\ n\ \$\ \mathit{getGaussian}\ \mathit{rng}\ 1.0 \\
&\quad \mathit{putStrLn}\ \$\ \mathit{show}\ \$\ \mathit{simpleMC1}\ \mathit{expiry}\ \mathit{strike}\ \mathit{spot}\ \mathit{vol}\ r\ \mathit{randomNums}
\end{aligned}
$$

3.3 Concepts introduced

Pure Functions Our pricer function `simpleMC1` is a pure function – no state is generated or read outside the parameters passed on, including the random draws passed over in the parameter `sample`. This has the advantage that the function will never return a different value if given the same parameters - something valuable to know, especially if correctness is a concern.

Left Fold General for and while loops are not commonly used (even if they can be replicated to some extent) - but are mostly replaced by recursion or iterations over lists. The left fold used here (`foldl'`) is the strict version of a left fold of the type

$$\mathit{foldl} :: (a \to b \to a) \to a \to [b] \to a$$

As parameters it takes a function with two parameters (one of type `a`, the second of type `b`, returning a value of type `a`)

$$f :: a \to b \to a$$

and two more values x of type `a` and a list of values y_i – all of type `b` denoted as `[b]` – and computes the value of $f(f(f(x, y_0), y_1), y_2)\ldots$ The strictness of `foldl'` avoids that the lazy evaluation of Haskell (which is often advantageous) makes our function stack up unevaluated nested functions calls until – right at the end of the program – we want to print the value on screen and all calls are evaluated. Haskell uses lazy evaluation where ever it can, except if told otherwise – like here.

map The function `map` is central to the use of the ubiquitous lists in functional languages. It is a function of type

$$\mathit{map} :: (a \to b) \to [a] \to [b]$$

that - given a function f with one parameter (of type `a` and returning a value of type `b`) and a list of values x_i of type `a` returns the new list

$$[f(x_0), f(x_1), \ldots]$$

The main function and the IO monad Without going into details about monads (a concept to be introduced later) - the main function needs to read and write state, otherwise nothing useful can be achieved. The following points are useful to understand the above code:

– Everything in the IO Monad is achieved by chaining "actions" together in the `do` construct.

– In a `do` construct, the return values of pure functions are retrieved with `let`:

```
let result = function a b c
```

– In a `do` construct, values retrieved from actions that are to be used in pure functions are extracted from their monad with `<-`:

```
result <- action
```

The following code:

```
b <- liftM read $ getLine
```

is equivalent to:

```
a <- getLine
let b = read a
```

– Pure functions like `read` (transforming a string it into values of - say - type `Int` or `Double`) need to be "lifted" into the monad with `liftM` if used directly on the output of actions.

– If not using `do`, actions (like `putStrLn`, printing a line on the screen) can be chained to the next action with `>>`.

```
action1 >> action2
```

corresponds to

```
do
  action1
  action2
```

The operator `$` To avoid myriads of parantheses (like (in (lisp ()))) one can use the operator `$`. It makes it possible to write:

```
function a1 $ function2 b $ function3 c
```

instead of

```
function a1 (function2 b (function3 c))
```

Other than making the code (arguably) more readable, this is equivalent.

3.4 Critiquing the approach

Several points of critique come to mind:

- The call payoff is hard coded - a put would need a new function, as would other payoffs like digitals
- Stats like standard error or a convergence tables would need major changes
- Sampling is hard coded - this makes it difficult to integrate other types like antithetic sampling
- The approach does not allow for an efficient termination condition (iterating until a specific standard error or other arises)
- Large samples are not possible and will make the stack overflow (the sample is before being passed on and evaluated in the Monte Carlo function)

Some of these points don't seem too serious: The overall amount of code is small and easy to oversee - one of the major advantages of Haskell. Implementing a different terminal payoff (e.g. for a put) would not need a tremendous amount of code repetition. But it should nevertheless be avoided.

4 Generalising Payoff and Option Data

It would be nice to integrate any types of final payoffs - which should not modify our Monte Carlo routine. A data type that contains the information of a specific payoff and for which this payoff can be evaluated without modifying anything else is the solution to the problem.

4.1 Implementing a Payoff Class

Haskell is not object oriented but still has classes – type classes. If a type is an instance of a type class, a specific set of functions are defined for the type (which makes it actually quite similar to object oriented classes where specific methods are defined for an instance of a specific class.) These will be helpful for defining payoffs. When dealing with different payoffs we want to have a function that tells us, based on the given payoff type and associated data like strike, what the payoff in Dollars would be.

4.2 The Payoff type class

Define a module Payoff defining the Payoff type class, requiring the definition of one function, namely payoff - which takes the terminal spot value and returns the payoff.

```haskell
module Payoff where

class PayoffClass a where
  payoff :: a -> Double -> Double
```

Then we define the data type for vanilla calls and puts:

```haskell
data VanillaOption = Put | Call
  deriving (Show)

data VOPayoff = VOPayoff {
  ptype :: VanillaOption,
  strike :: Double}
  deriving (Show)
```

Now we define the instance of the PayoffClass type class - defining the 2 payoffs of vanilla calls and puts:

```haskell
instance PayoffClass VOPayoff where
  payoff (VOPayoff Call strike) spot = if spot > strike then spot - strike else 0
  payoff (VOPayoff Put strike) spot = if spot < strike then strike - spot else 0
```

4.3 Using the Payoff Class

We need to include our new module as well:

```haskell
import GSL.Random.Gen
import GSL.Random.Dist
import Control.Monad
import Payoff
```

Our simple Monte Carlo calculator changes to:

```haskell
simpleMC2 :: PayoffClass a => a -> Double -> Double -> Double -> Double -> [Double]
  -> Double
simpleMC2 po expiry spot vol r sample =
  exp ((-r) * expiry) * sumAll / n
  where
    variance     = vol * vol * expiry
    rootVariance = sqrt variance
    itoCorr      = (-0.5) * variance
    mSpot        = spot * exp (r * expiry + itoCorr)
    sumAll       = sum $ map sumItem sample
    n            = (fromIntegral $ length sample)
    sumItem gaussian = payoff po randomSpot
      where randomSpot = mSpot * exp (rootVariance * gaussian)
```

```haskell
askForInput statement = putStrLn statement >> (liftM read $ getLine)
```

Main function doing the input and output:

```haskell
main = do
  temp   <- askForInput "Enter option type (1 = Call, other = Put)"
  expiry <- askForInput "Enter Expiry"
  strike <- askForInput "Enter strike"
  spot   <- askForInput "Enter spot"
  vol    <- askForInput "Enter vol"
  r      <- askForInput "Enter r"
  n      <- askForInput "Enter number of paths"
  let payoff = if temp == 1 then VOPayoff Call strike else VOPayoff Put strike
  rng <- newRNG mt19937
  randomNums <- replicateM n $ getGaussian rng 1.0
  putStrLn $ show $ simpleMC2 payoff expiry spot vol r randomNums
```

4.4 Implementing an Option Class

It would be nice to bundle all the information that we have about the option to be priced in one data container - the expiry of the option is still sitting outside. We can easily define a record type data type that will help us over this:

4.5 The Option type class

```haskell
module Option where
import Payoff
```

The Option data type can accommodate any payoff defined in a PayoffClass type class:

```
data Option a = Option {
  expiry :: Double,
  pay :: a
  }
  deriving (Show)
```

One might be tempted to put a type restriction to PayoffClass on the type a - which is not necessary and would force all future functions to have this type restriction as well. As soon as we use the type a in a context where this needs to be a payoff, the type will be inferred there.

4.6 Using the Option Class

We need to include our new module as well:

```
import GSL.Random.Gen
import GSL.Random.Dist
import Control.Monad
import Payoff
import Option
```

Our simple Monte Carlo calculator changes to:

```
simpleMC3 :: PayoffClass a => (Option a) -> Double -> Double -> Double -> [Double]
   -> Double
simpleMC3 op spot vol r sample =
  exp ((-r) * (expiry op)) * sumAll / n
  where
    variance          = vol * vol * (expiry op)
    rootVariance      = sqrt variance
    itoCorr           = (-0.5) * variance
    mSpot             = spot * exp (r * (expiry op) + itoCorr)
    sumAll            = sum $ map sumItem sample
    n                 = (fromIntegral $ length sample)
    sumItem gaussian = payoff (pay op) randomSpot
      where randomSpot = mSpot * exp (rootVariance * gaussian)
```

```
askForInput statement = putStrLn statement >> (liftM read $ getLine)
```

Main function doing the input and output:

```haskell
main = do
  temp   ← askForInput "Enter option type (1 = Call, other = Put)"
  expiry ← askForInput "Enter Expiry"
  strike ← askForInput "Enter strike"
  spot   ← askForInput "Enter spot"
  vol    ← askForInput "Enter vol"
  r      ← askForInput "Enter r"
  n      ← askForInput "Enter number of paths"
  let option = if temp ≡ 1
    then Option expiry (VOPayoff Call strike)
    else Option expiry (VOPayoff Put strike)
  rng    ← newRNG mt19937
  randomNums ← replicateM n $ getGaussian rng 1.0
  putStrLn $ show $ simpleMC3 option spot vol r randomNums
```

4.7 Concepts introduced

Defining new data types There are 2 different ways to define a new data type: `type` and `data`.

```haskell
type Foo1 = (Int, Int)
```

`type` declares a type synonym. This makes it easier to define the type signatures or constraints.

```haskell
data Foo2 = Foo2 Int Int
```

`data` declares a new type via a type constructor - in this case a type that contains two integers. There is a record style syntax for `data`:

```haskell
data Foo3 = Foo3 {one :: Int, two :: Int}
```

which creates the same data constructor function Foo3 of the type

```haskell
Foo3 :: Int → Int → Foo3
```

but also the two functions

```haskell
one :: Foo3 → Int
tow :: Foo3 → Int
```

which do what you would expect them two: extract the two different integers from Foo3. We could define these ourselves for `Foo2`:

```haskell
one (Foo2 a _) = a
two (Foo2 _ a) = a
```

This is eventually what the record syntax does for us.

Type Classes Haskell is a strongly typed language - which helps in the way that errors can often be spotted at compile time vs. runtime. It also makes type inference possible: As seen in the code so far, we don't have to specify types very often but only in ambiguous situations. Sometimes we might choose to write the type signature to help the readability of the code – this is done here for all `SimpleMC` functions.

5 Monadic Random Number Generation

Pure functions in Haskell can only receive data through the parameters and give back results as the result of the function – they are purposefully not made to read or write any other parts of the memory and cannot call functions that do so.

If we would for example want to implement a simple counter function to which we want to pass an increment and get back the new global count we have no choice other than storing the state outside the function and passing it on – and to receive it back. It cannot be done via a global variable – the global variable would be state outside the function that would be accessed.

```haskell
increment (dx, state) = (state + dx, state + dx)
```

If we want to do this a number of times this looks like the following:

```haskell
main = do
  let state0 = 0
  let (c1, state1) = increment (10, state0)
  putStrLn $ show c1
  let (c2, state2) = increment (11, state1)
  putStrLn $ show c2
```

This passing on of the state from one "action" to another is implemented in the Monad typeclass: A monad of type "M a" stands for a chain of actions that results in the type `a` if the monad gets evaluated. Now we need the possibilities to "lift" values into the monad, and to chain actions together:

```haskell
return :: (Monad m) => a -> m a
(>>) :: (Monad m) => m a -> m b -> m b
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

`return` is an action that takes a pure functional value and returns the monad that (if evaluated) gives this value back.

The operator `>>` chains two actions together, dismissing the result of the first. When is this useful? In a parser, the first action might be consuming a required chain of characters in the state (the string of unconsumed characters) before the next actual value to be read.

The operator `>>=` chains two actions together whereby the second action takes a parameter of the same type as the return type of the first action (hidden in the monad).

Pseudo random number generation is a very good example of state that needs to be carried on. One way of passing on state is the following:

It is possible to elegantly simulate state by creating functions that pass the state on to the next - in a chain of "actions". This is implemented in Monads in Haskell - and for random number generation there conveniently is a Monad implementation that wraps a the random number generator from the Gnu Scientific Library.

The following modification of our original code will run with arbitrarily large sample sizes in constant space. Another trick has to be applied to achieve the second goal: the function `sum` is replaced by `foldl' (+) 0` which is the strict version of `foldl`. This ensures that Haskell uses (and garbage collects) the samples that have already been used. We will find out about forcing strictness later.

We need to import the monadic versions of our random number generator:

```haskell
import Data.List
import Control.Monad
import Control.Monad.MC
```

This contains our simple Monte Carlo calculator wrapped in the MC monad:

```haskell
simpleMC1b :: Double -> Double -> Double -> Double -> Double -> Int -> MC Double
simpleMC1b expiry strike spot vol r n = do
  xs <- replicateM n $ normal 0.0 1.0
  let variance         = vol * vol * expiry
      rootVariance     = sqrt variance
      itoCorr          = (-0.5) * variance
      mSpot            = spot * exp (r * expiry + itoCorr)
      sumItem gaussian = if payoff > 0 then payoff else 0
        where payoff = (-strike) + mSpot * exp (rootVariance * gaussian)
      sumAll = foldl' (+) 0 $ map sumItem xs
  return (exp ((-r) * expiry) * sumAll / (fromIntegral n))
```

```haskell
askForInput statement = putStrLn statement >> (liftM read $ getLine)
```

Main function doing the input and output:

```
main = do
  expiry ← askForInput "Enter Expiry"
  strike ← askForInput "Enter strike"
  spot   ← askForInput "Enter spot"
  vol    ← askForInput "Enter vol"
  r      ← askForInput "Enter r"
  n      ← askForInput "Enter number of paths"
  let seed = 0
      val = evalMC (simpleMC1b expiry strike spot vol r n) $ mt19937 seed
  putStrLn $ show val
```

5.1 Parameters

Parameters could be nicely wrapped into a parameters class - implementing it initially for a constant Double:

We need a couple of functions in this class:

```
module Parameter where

class Parameter a where
  integral :: a → Double → Double → Double
  integralSquare :: a → Double → Double → Double
  mean :: a → Double → Double → Double
  mean param t1 t2 = (integral param t1 t2) / (t2 − t1)
  rootMeanSquare :: a → Double → Double → Double
  rootMeanSquare param t1 t2 = (integralSquare param t1 t2) / (t2 − t1)
```

And our instance for a constand Double:

```
instance Parameter Double where
  integral param t1 t2 = param ∗ (t2 − t1)
  integralSquare param t1 t2 = param ∗ param ∗ (t2 − t1)
```

5.2 Using the Parameters Class

We need to include our new module as well:

```
import Data.List
import Control.Monad
import Control.Monad.MC
import Payoff
import Option
import Parameter
```

Our simple Monte Carlo calculator changes to:

```haskell
simpleMC4 :: (PayoffClass a, Parameter b) => (Option a) -> Double -> b -> b -> Int
  -> MC Double
simpleMC4 op spot vol r n = do
  xs <- replicateM n $ normal 0.0 1.0
  let rt                = integral r 0 (expiry op)
      variance          = integralSquare vol 0 (expiry op)
      rootVariance      = sqrt variance
      itoCorr           = (-0.5) * variance
      mSpot             = spot * exp (rt + itoCorr)
      sumItem gaussian  = payoff (pay op) $ mSpot * exp (rootVariance * gaussian)
      sumAll = foldl' (+) 0 $ map sumItem xs
  return (exp (-rt) * sumAll / (fromIntegral n))
```

```haskell
askForInput statement = putStrLn statement >> (liftM read $ getLine)
```

Main function doing the input and output:

```haskell
main = do
  temp   <- askForInput "Enter option type (1 = Call, other = Put)"
  expiry <- askForInput "Enter Expiry"
  strike <- askForInput "Enter strike"
  spot   <- askForInput "Enter spot"
  vol    <- (askForInput "Enter vol") :: IO Double
  r      <- askForInput "Enter r"
  n      <- askForInput "Enter number of paths"
  let option = if temp == 1
    then Option expiry (VOPayoff Call strike)
    else Option expiry (VOPayoff Put strike)
  let seed = 0
    val = evalMC (simpleMC4 option spot vol r n) $ mt19937 seed
  putStrLn $ show val
```

5.3 Concepts introduced

Monads *This needs either in depth explanation here or at the start ...*

5.4 Next Steps

So far most of these changes are trivial - no major hoops have to be jumped through to get to a nice encapsulated approach. Most of the sorrows are being well taken care of by Haskell as a language. This is all as one would hope. Some things stand out as wanting improvement:

- It would be nice to have statistics on the calculations
- Reading of the data line by line is not a nice way of constructing the option data to be priced - reading a data file would be nicer.
- Computers come with multiple cores these days and Monte Carlo is an embarrassing parallel technique - this should be parallelised.

6 Parsing an Input File

To keep our code clean and small we should factor out the reading of input parameters. This is done by parsing an input file - or string if done through a pipe.

Haskell comes with a very strong parser (namely Parsec), replacing complicated manipulations with lexers that is necessary in other languages. Below a short parser that will be able to parse the following format:

6.1 The Input File Format

```
Option {
  Expiry: 1.0
  Payoff: Call at 50.0
}

Fixing {
  Spot: 45.0
  Vol: 0.1
  Rate: 0.1
}

Params {
  Paths: 1000000
}
```

The different parts of the input file are positional and the order as well - which can be easily changed.

6.2 Applicative Parsec

We will be using an applicative extension of Parsec, easily produced through the definition of the following module:

```haskell
module ApplicativeParsec
  (
    module Control.Applicative,
    module Text.ParserCombinators.Parsec
  )
    where
import Control.Applicative
import Control.Monad (MonadPlus (..), ap)
import Text.ParserCombinators.Parsec hiding (many, optional, (<|>))
instance Applicative (GenParser s a) where
    pure = return
    (<*>) = ap
instance Alternative (GenParser s a) where
    empty = mzero
    (<|>) = mplus
```

6.3 Implementing the Parser

The definition of the applicative instance for the parser allows to parse the return data types in a concise and readable way - once one gets used to it.

Imagining the data type foo that takes two integers to be defined:

```haskell
data Foo = Foo {first :: String, second :: String}
```

The constructor is a function of the following type:

```haskell
Foo :: String -> String -> Foo
```

Each individual parser give back the parsed value - for example a string:

```haskell
parseString :: CharParser st String
parseString = string "foo"
```

We would now like to parse some text, retain two string values and return these in the data type Foo:

```
first: foo
second: notfoo
```

Without the applicative parser this would look like this:

```
parseIt1 = do
  spaces >> string "first:" >> spaces
  first <- many1 letter
  spaces >> string "second:" >> spaces
  second <- many1 letter
  return (Foo first second)
```

This can parse our sample input successfully:

```
parse parseIt1 "from string" "first: foo \n  second: notfoo"
```

With ApplicativeParsec we can chain the different parts of the parser with the operator `<*>` and lift the constructor into it with `<$>`. The chained parsers will be given as an argument chain to the lifted function:

```
testParse_a = Foo <$> firstparser <*> secondparser
```

We might want to intersperse the parsers that return values with our syntax. To chain 2 parsers but only give back the result of the first (or second) as an argument, we use the operators `<*` and `*>`. Our parser becomes:

```
parseIt2 = Foo <$>
  (spaces >> string "first:" >> spaces >> many1 letter) <*>
  (spaces >> string "second:" >> spaces >> many1 letter)
```

This halves the lines required to define the parser without defining intermediate names for the different parts that are parsed.

6.3.1 Defining some Data Types

It would be nice to wrap the different entities we want to parse into data types. We already did that for Option and Payoff. Now here come Fixing and InputParams:

Fixing

```
module Fixing where
```

Simple data type for our two fixings.

```
data Fixing = Fixing {
  spot :: Double,
  vol :: Double,
  rate :: Double
  }
  deriving (Show)
```

InputParams

```
module InputParams where
```

Simple data type for our numerical parameters.

```
data Params = Params {
  numPaths :: Int
  }
  deriving (Show)
```

6.3.2 The actual Parser

```
module Parse where
import Numeric
import Payoff
import Option
import Fixing
import InputParams
import ApplicativeParsec
```

Parse all parts of the file:

```
optionFile = returnVals <$> opt <*> fixings <*> numParams <* (spaces >> eof) where
  returnVals a b c = (a, b, c)
```

Parse the product first:

```
opt = Option <$>
  (spaces ≫ string "Option" ≫ spaces ≫ string "{" ≫ p_expiry) <*>
  p_payoff <*
  (spaces ≫ string "}" ≫ eol)
p_expiry = (spaces ≫ (string "Expiry:") ≫ spaces) *> (p_number <?> "d1")
p_payoff = option <$>
  (spaces ≫ (string "Payoff:") ≫ spaces ≫ (string "Call" <|> string "Put")) <*>
  (spaces1 ≫ (string "at") ≫ spaces1 ≫ (p_number <?> "d1")) where
    option str strike | str ≡ "Call" = VOPayoff Call strike
    option str strike | str ≡ "Put" = VOPayoff Put strike
```

Parse our 2 fixings:

```
fixings = Fixing <$>
  (spaces ≫ (string "Fixing") ≫ spaces ≫ (string "{") ≫ p_spot) <*>
  (spaces ≫ p_vol) <*>
  (spaces ≫ p_rate) <*
  (spaces ≫ string "}" ≫ eol)
p_spot = (spaces ≫ (string "Spot:") ≫ spaces) *> (p_number <?> "d1")
p_vol = (spaces ≫ (string "Vol:") ≫ spaces) *> (p_number <?> "d1")
p_rate = (spaces ≫ (string "Rate:") ≫ spaces) *> (p_number <?> "d1")
```

Parse our parameters:

```
numParams = Params <$>
  (spaces ≫ (string "Params") ≫ spaces ≫ (string "{") ≫ p_paths) <*
  (spaces ≫ string "}" ≫ eol)
p_paths = (spaces ≫ (string "Paths:") ≫ spaces) *> (p_integer <?> "d1")
```

Some general parsers:

```
p_number :: CharParser () Double
p_number = do
  s ← getInput
  case readSigned readFloat s of
    [(n, s')] → n <$ setInput s'
    _ → empty
p_integer :: CharParser () Int
p_integer = do
  s ← getInput
  case readSigned readDec s of
    [(n, s')] → n <$ setInput s'
    _ → empty
eol = char '\n'
spaces1 = many1 space
```

6.4 Using the Parser

Plenty of new modules to include:

```haskell
import System
import Data.List
import Control.Monad
import Control.Monad.MC
import ApplicativeParsec

import Payoff
import Option
import Parameter
import InputParams
import Fixing
import Parse
```

Our simple Monte Carlo calculator:

```haskell
simpleMC4b :: (PayoffClass a) => (Option a) -> Fixing -> Params -> MC Double
simpleMC4b opt fixing params = do
  xs <- replicateM (numPaths params) $ normal 0.0 1.0
  let rt              = integral (rate fixing) 0 (expiry opt)
      variance        = integralSquare (vol fixing) 0 (expiry opt)
      rootVariance    = sqrt variance
      itoCorr         = (-0.5) * variance
      mSpot           = (spot fixing) * exp (rt + itoCorr)
      sumItem gaussian = payoff (pay opt) $ mSpot * exp (rootVariance * gaussian)
      sumAll = foldl' (+) 0 $ map sumItem xs
  return (exp (-rt) * sumAll / (fromIntegral $ numPaths params))
```

Main function doing the input and output is now significantly shorter and nicer:

```haskell
main = do
  args <- getArgs
  res <- parseFromFile optionFile (head args)
  let val = case res of
    Right (opt, fixing, params) ->
      show $ evalMC (simpleMC4b opt fixing params) $ mt19937 0
    Left _ -> "Could not parse file."
  putStrLn val
```

7 Gathering Statistics

7.1 The Statistics Type Class

Defining a simple type class for gathering stats, together with one instance to get the mean:

```haskell
module Stats where

class Stats a where
  zero :: a
  dumpOne :: a -> Double -> a
  getRes :: a -> [[Double]]

data Mean = Mean {sum :: Double, nPaths :: Int} deriving Show

instance Stats Mean where
  zero = Mean 0 0
  dumpOne (Mean s n) res = newSum `seq` newN `seq` Mean newSum newN where
    newSum = s + res
    newN   = n + 1
  getRes (Mean s n) = [[s / (fromIntegral n)]]
```

This type class can now be used by our simple Monte Carlo funtion:

More modules:

```haskell
import System
import Data.List
import Control.Monad
import Control.Monad.MC
import ApplicativeParsec

import Payoff
import Option
import Parameter
import InputParams
import Fixing
import Parse
import Stats
```

Our simple Monte Carlo calculator:

```haskell
simpleMC5 :: (PayoffClass a, Stats c) => (Option a) -> Fixing -> Params -> MC c
simpleMC5 opt fixing params = do
  xs <- replicateM (numPaths params) $ normal 0.0 1.0
  let rt              = integral (rate fixing) 0 (expiry opt)
      variance        = integralSquare (vol fixing) 0 (expiry opt)
      rootVariance    = sqrt variance
      itoCorr         = (-0.5) * variance
      mSpot           = (spot fixing) * exp (rt + itoCorr)
      sumItem gaussian = exp (-rt) * (payoff (pay opt) $ mSpot * exp (rootVariance * gaussian))
      sumAll          = foldl' dumpOne zero $ map sumItem xs
  return sumAll
```

Main function doing the input and output is now significantly shorter and nicer:

```haskell
main = do
  args <- getArgs
  res <- parseFromFile optionFile (head args)
  let val = case res of
      Right (opt, fixing, params) ->
        show $ getRes $ evalMC ((simpleMC5 opt fixing params) :: MC Mean) $ mt19937 0
      Left _ -> "Could not parse file."
  putStrLn val
```

7.2 Convergence Table

The convergence table implementation looks very similar:

Defining the convergence table on any stats type:

```haskell
module ConvTable where

import Stats

data ConvTable a = ConvTable {
  count :: Int,
  nextLimit :: Int,
  currStat :: a,
  resList :: [[Double]]}

instance (Stats a) => (Stats (ConvTable a)) where
  zero = (ConvTable 0 1 zero [])
  dumpOne (ConvTable c l curr xs) s =
    nc `seq` ncurr `seq` nl `seq` nxs `seq` (ConvTable nc nl ncurr nxs) where
    nc = c + 1
    ncurr = dumpOne curr s
    (nl, nxs) = if nc == l then (l * 2, (head $ getRes ncurr) : xs)
      else (l, xs)
  getRes (ConvTable c l curr xs) =
    if c == l then xs else (head $ getRes curr) : xs
```

Only the main function needs modification to use this - since the return type changed:

The modules:

```haskell
import System
import Data.List
import Control.Monad
import Control.Monad.MC
import ApplicativeParsec
import Payoff
import Option
import Parameter
import InputParams
import Fixing
import Parse
import Stats
import ConvTable
```

Our simple Monte Carlo calculator:

```haskell
simpleMC5 :: (PayoffClass a, Stats c) => (Option a) -> Fixing -> Params -> MC c
simpleMC5 opt fixing params = do
  xs <- replicateM (numPaths params) $ normal 0.0 1.0
  let rt                = integral (rate fixing) 0 (expiry opt)
      variance          = integralSquare (vol fixing) 0 (expiry opt)
      rootVariance      = sqrt variance
      itoCorr           = (-0.5) * variance
      mSpot             = (spot fixing) * exp (rt + itoCorr)
      sumItem gaussian  = exp (-rt) * (payoff (pay opt) $ mSpot * exp (rootVariance * gaussian))
      sumAll            = foldl' dumpOne zero $ map sumItem xs
  return sumAll
```

Main function doing the input and output is now significantly shorter and nicer:

```haskell
main = do
  args <- getArgs
  res <- parseFromFile optionFile (head args)
  let val = case res of
    Right (opt, fixing, params) ->
      show $ getRes $
        evalMC ((simpleMC5 opt fixing params) :: MC (ConvTable Mean)) $ mt19937 0
    Left _ -> "Could not parse file."
  putStrLn val
```

7.3 Concepts introduced

Backticks To make a function act like an operator one can use the backtick syntax:

```
fu a b = a + b
let c = fu 1 2
let d = 1 `fu` 2
```

The expressions for `c` and `d` are equivalent.

Forcing evaluation with seq Haskell as a lazy language does not evaluate expressions until they are needed. This is often advantageous, but can lead to space leaks – the unevaluated expressions are piling up and use a lot of space. This can be easily found by running the programs with `+RTS -sstderr` which then prints statistics on the runtime. If the garbage collector uses a high percentage, then there are probably inefficiencies there. The expressions

```
a `seq` b
```

makes sure that the expression `a` is fully evaluated, before returning the result of expression `b`. So if we want to return a specific type with the type constructor:

```
let d = Foo a b
```

then we can enforce that no unevaluated bits are left with the following code:

```
let d = a `seq` b `seq` Foo a b
```

Did we not do this already with `foldl'`? We kind of did, except that `foldl'` only evaluates to the head normal form, not the fully evaluated expression. Removing the `` `seq` `` out of the code above makes it perform very poorly, with over 70% of the time spent on garbage collection.

Why does unevaluated code create a space leak and why does this have such an impact on performance? The unevalutated code is big and makes the garbage collector reserve a lot of memory. For this the runtime needs to spend a lot of memory – and time.