A Flex Engine In Less Than 101 Lines

Dr. Ingo Fahrner
Thomas Fehnl
Quantitative Research, LBBW - 8976/H, 70173 Stuttgart,
e-mail: ingo.fahrner@lbbw.de

June 5, 2008

Abstract

We present a framework for building a flexible payoff language and linking it to the pricing kernel. The code needed for this task is in fact very short, shows good performance and is included in this presentation.

Keywords: Flexible payoff description language. Exotic products.

1 Introduction

Most traditional front office systems have parameterized payoffs and instruments. This probably stems from the fact that 10 years ago we had long product circles and only a limited number of different products. Today we face a totally different situation: We have a huge and expanding product range. The structuring forces of a modern investment bank need a fast, flexible and easy-to-use tool in order to develop new ideas and connect them within in short time to the front-office system without relying on long IT implementation processes.

On the other hand most major investment banks have discovered that almost all pricing algorithms allow to price fairly general payoffs. Those payoffs are then described by some sort of scripting language which is parsed and generically linked to the front-office system via an interpreter. The implementation of a parser/interpreter and connecting it to the pricing kernel is considered to be rather involved. Indeed, designing a well-defined general purpose programming language and implementing the parser are non-trivial tasks, see e. g. the recent talks Caps (2008) or Odetti and Shukla (2008).

However, in the present paper we show that it is in fact quite simple to realize a flexible payoff language using a programming language as a scripting language. There are a couple of advantages over a parser:

1. The translation can be done using the standard compiler, so no parser or interpreter has to be implemented.
2. The syntax is well-defined and well-known to structurers, so they don't need to be educated.
3. Error messages are generated in an understandable and known format.
4. Moreover, since the code is compiled, we end up having excellent performance.

We will use the Microsoft C# compiler in the .NET 2.0 framework which allows for in-memory compilation and easy linking of the compiled script to the pricing kernel. This facilitates the task considerably but our method does not rely in principle on these features. We would like to note that in Strahl (2002) a similar method is applied to non-finance applications.

2 General Payoffs

Most numerical pricing algorithms work as follows: In a specific node we generate some state variables (either by random as in a Monte-Carlo simulation or in a deterministic way as in PDEs or trees) from which we can calculate the values of all observables (stock values, interest rates, volatilities, fx rates and so on). By plugging in the observables to the payoff we get the cashflows of the product in this state.

We will demonstrate this observation by implementing a toy example for a flex engine. Assume that we have one observable (called R) only. We define the following interface:

```
public interface IPayoff
{
    void SetR(double rate);
    double Cashflow(int position, ref double history);
}
```

After chosen the state variables and transforming them to the value r of R at that node, we call *SetR(r)*. The method *Cashflow* then yields the cashflow in period *position* and given *history*.

The "concrete" implementation of the *IPayoff* interface is the class *Payoff* and is divided in three variables *preCode*, *code* and *postCode*:

```
static string preCode = @"
  using FlexLibrary;

  public class Payoff : IPayoff
  {
    public double R;

    public void SetR(double rate)
    {
        R = rate;
    }

    public double Cashflow(int position, ref double history)
    {
        double cashflow;
    ";

static string code = @"
        if (position == 0)
            cashflow = 4 * R + 0.01;
        else
            cashflow = 4 * R + history;
        history = cashflow;
        ";
static string postCode = @"
        return cashflow;
     }
}
";
```

Note that we have a variable R for our observable and that *SetR(double)* really sets this variable. The method *Cashflow* is split in a very special way: The payoff description is in the variable *code* whereas *preCode* and *postCode* are just the header and the return statement. The last two blocks are the same over all (single observable) payoff functions.

3 The Flex Engine

Combining the findings of the last section with in memory compilation yields our flex engine: Assume we do have a method *Fly(string someCode)* which compiles *someCode* on the fly into memory then we can determine the main part of the payoff function (the variable code) at runtime and do something like:

```
static void Main(string[] args)
{
    while ((code = Console.ReadLine()) != string.Empty)
    {
       IPayoff payoff = FlexEngine.Fly(preCode + code + postCode);
       if (payoff != null)
       {
           double history = 0.0;
           for (int k = 0; k < 5; k++)
           {
               payoff.SetR(0.04 + k / 100.0);
               Console.Write("{0}\t", payoff.Cashflow(k, ref history));
           }
       }
       Console.WriteLine();
    }
}
```

This program asks for the payoff code and compiles it. As an example we set successively R to $4\%, 5\%, \ldots, 8\%$ and evaluate the payoff. Here a more sophisticated Monte-Carlo engine could be implemented. The program stops when the code is empty.

Obviously the description of the payoff must return the cashflow which can be calculated using any C# code and referencing to the observable R and a history. Within the payoff we can set the history. So e. g. we can code

```
cashflow = 100;
```

for a fixed payment of 100, or

```
if (position < 2)
    cashflow = 2 * R + 0.0010;
else
    cashflow = R + 0.0005;
```

for a (leveraged) float leg with a varying spread when R is interpreted as a Libor. For an example with history see the predefined code variable above.

4 In Memory Compilation

The part left so far is the FlexEngine.Fly method.

```
public class FlexEngine
  {
      public static IPayoff Fly(string source)
      {
          CodeDomProvider compiler = new CSharpCodeProvider();
          CompilerParameters parameters = new CompilerParameters();
          parameters.ReferencedAssemblies.Add("FlexLibrary.dll");
          parameters.GenerateInMemory = true;
          CompilerResults results = compiler.CompileAssemblyFromSource(parameters, source);
          if (results.Errors.Count == 0)
```

```
                return (IPayoff)Activator.CreateInstance(
                    results.CompiledAssembly.GetType("Payoff"));
            else
            {
                foreach (CompilerError error in results.Errors)
                    Console.WriteLine(string.Format(
                        "Error in line {0} >>> {1}", error.Line - 15, error.ErrorText));
                return null;
            }
        }
    }
```

This method is also quite simple: We get a compiler, set the reference to the library containing the *IPayoff* interface and compile the code in memory. If there are no errors we create an instance of the *Payoff* class. In case of errors we use the compiler to generate error messages. Since the *preCode* has 15 lines we subtract 15 in the *error.Line*.

5 The Complete Code

The code is divided in two parts: A library called FlexLibrary and a main program.

5.1 FlexLibrary

```
using System;
using System.CodeDom.Compiler;
using Microsoft.CSharp;

namespace FlexLibrary
{
    public interface IPayoff
    {
        void SetR(double rate);
        double Cashflow(int position, ref double history);
    }

    public class FlexEngine
    {
        public static IPayoff Fly(string source)
        {
            CodeDomProvider compiler = new CSharpCodeProvider();
            CompilerParameters parameters = new CompilerParameters();
            parameters.ReferencedAssemblies.Add("FlexLibrary.dll");
            parameters.GenerateInMemory = true;
            CompilerResults results = compiler.CompileAssemblyFromSource(parameters, source);
            if (results.Errors.Count == 0)
                return (IPayoff)Activator.CreateInstance(
                    results.CompiledAssembly.GetType("Payoff"));
            else
            {
                foreach (CompilerError error in results.Errors)
                    Console.WriteLine(string.Format("Error in line {0} >>> {1}",
                        error.Line - 15, error.ErrorText));
                return null;
            }
        }
```

```
    }
}
```

5.2 Main program

```
using System;
using FlexLibrary;

namespace FlexPaper
{
    class Program
    {
        static string preCode = @"
                    using FlexLibrary;

                    public class Payoff : IPayoff
                    {
                        public double R;

                        public void SetR(double rate)
                        {
                            R = rate;
                        }
                        public double Cashflow(int position, ref double history)
                        {
                            double cashflow;
                        ";
        static string code = @"
                            if (position == 0)
                                cashflow = 4 * R + 0.01;
                            else
                                cashflow = 4 * R + history;
                            history = cashflow;
                        ";
        static string postCode = @"
                            return cashflow;
                        }
                    }
                    ";

        static void Main(string[] args)
        {
            while ((code = Console.ReadLine()) != string.Empty)
            {
                IPayoff payoff = FlexEngine.Fly(preCode + code + postCode);
                if (payoff != null)
                {
                    double history = 0.0;
                    for (int k = 0; k < 5; k++)
                    {
                        payoff.SetR(0.04 + k / 100.0);
                        Console.Write("{0}\t", payoff.Cashflow(k, ref history));
                    }
                }
                Console.WriteLine();
```

```
            }
        }
    }
}
```

6 Implications for the Future

Due to the speed of new developments in the markets we believe it is essential to have some sort of flex engine even for the smaller banks and more and more importance will be given to them. On the other hand we need easy and fast implementations such that we don't waste time on debugging parsers or on integrating new products.

For example, the LBBW inhouse eXotic Front Office Risk Management system (XForm) is build on the idea presented in this paper. Combined with an object model for products which can represent virtually every product in the market new product ideas can be integrated within near-no time by structuring and sales. Thus the quant team can concentrate on the essential properties of models and markets.

References

Caps, O. (2008) Using Compiler Engineering Algorithms for Building Payoff Languages. *Presentation at Frankfurt MathFinance Conference* (http://conference.mathfinance.de/2008/papers/caps/slides.pdf).

Odetti, A. and Shukla, S. (2008) High Performance Computing Techniques in Finance. *Presentation at Frankfurt MathFinance Conference* (http://conference.mathfinance.de/2008/papers/shukla/slides.pdf).

Strahl, R. (2002) Dynamically executing code in .Net. *Web Presentation.* Available at http://www.west-wind.com/presentations/dynamicCode/dynamicCode.htm.