JFP **19** (1): 27–45, 2009. © 2008 Cambridge University Press
doi:10.1017/S0956796808007016 First published online 1 October 2008 Printed in the United Kingdom

Commercial uses: Going functional on exotic trades

SIMON FRANKAU
Barclays Capital, 5 The North Colonnade, London E14 4BB, UK
(*e-mail:* `Simon.Frankau@barclayscapital.com`)

DIOMIDIS SPINELLIS
Athens University of Economics and Business, Patision 76, GR 104 34, Athens, Greece
(*e-mail:* `dds@aueb.gr`)

NICK NASSUPHIS and CHRISTOPH BURGARD
Barclays Capital, 5 The North Colonnade, London E14 4BB, UK
(*e-mail:* `{Nick.Nassuphis,Christoph.Burgard}@barclayscapital.com`)

Abstract

The Functional Payout Framework (FPF) is a Haskell application that uses an embedded domain-specific functional language to represent and process exotic financial derivatives. Whereas scripting languages for pricing exotic derivatives are common in banking, FPF uses multiple interpretations to not only price such trades, but also to analyse the scripts to provide lifecycle support and more. This paper discusses FPF in relation to the wider trading workflow and our experiences in using a functional language in such a system as both an implementation language and a domain-specific language.

1 Introduction

This application paper is an experience report describing the use of functional programming technologies in the field of exotic equity trading. Our aim is to present the challenges, design decisions and benefits of applying functional programming in a demanding real-world application domain. The main contributions of this paper are (a) the demonstration of the benefits of functional programming in the field of financial derivatives trading, and (b) the lessons learned in using functional constructs as a domain-specific language (DSL) in a production environment. The paper is not intended to cover in depth the internal implementation details of our system. Most of the techniques used have already been described in detail in the existing literature, far better than we could.

In this paper, we present Functional Payout Framework (FPF) through which we manage various lifecycle elements of exotic equity trades. After explaining what an exotic equity derivative actually is and reviewing the pre-existing workflow we describe through examples our language's goals and design (Section 2). We then outline FPF's implementation (Section 3), how our DSL is used in practice (Section 4), and the lessons we learned while working on this project (Section 5). The paper finishes with a discussion of related work (Section 6) and our concluding remarks.

1.1 What is an exotic option anyway?

Options are financial instruments like stocks and bonds. However, in contrast to, say, stocks, which represent ownership interest in a corporation, options are more abstract. An option is a contract whose value is a function of the value of other instruments. As the prices are derived from those of other instruments, options are also termed derivatives.

The purpose of options are to hedge (reduce risk) or speculate (take risk in search of profit). Some options are repackaged for the retail market – to be sold to individuals as a way of getting a return linked to the performance of the stock markets without risking the money investors originally put in. Other forms let speculators express a view by choosing a function that pays out most in scenarios they believe likely and least in those they believe unlikely.

We will start with the definition of a plain or "vanilla" option. When you buy a vanilla option (by paying an initial premium) you get the right, but not the obligation, to buy (or sell) a fixed amount of an asset at a fixed price. You could buy an option at time t_0 that allows you to buy n dollars-worth of asset A at time t_T, but at the t_0 price. If the price goes up, you earn the difference between the current price and the price at time t_T. If the asset's price goes down, you get nothing. Mathematically, if $S_A(t)$ is the price of asset A at time t, the amount you receive will be

$$n \max(S_A(t_T)/S_A(t_0) - 1, 0)$$

Once we move to mathematical notation, we can make the option definition into a more-or-less arbitrary formula. The value n is called the notional of the trade, and the right-hand side can be viewed as an expression specifying the fraction of notional to be paid out. If the payment expression becomes sufficiently complex, the trade is known as an exotic option. Here are some examples of exotic equity options:

- *Best-of.* Take the performance (defined as percentage change: $S_A(t_T)/S_A(t_0)-1$) of several different stocks. Return the best (or worst) performance.
- *Cliquet.* Observe the performance of an asset every month for a year. Apply a cap and floor to each performance, sum the results up, and cap and floor that result.
- *Napoleon.* Observe the performance of an asset every month for a year, take the worst performing month and add that performance on to a fixed value. Floor the payout at zero and pay the result.

By the standards of today's trading desks, the examples we described are simple exotic options. Realistic trades may have several "legs" of the above style, plus extra features. It is important to bear in mind that there's no real "standard" exotic trade – there is a wide variety, and clients want variations on existing structures as economic conditions and investor sentiment change. Flexibility is a key requirement for product innovation in exotic options trading.

Exotic equity options (that is, exotic options on stocks) are traded in the over the counter (OTC) market. This means they are custom-built to a client's request, rather than made into a standard form and traded through an exchange. To give an idea of the amount of money involved, as of June 2006, the outstanding notional in the OTC equity-linked options market was 5.4 trillion dollars (The Bank for International Settlements, 2007). This figure compares with 6.6 trillion dollars outstanding on exchange-traded (vanilla) equity index options. The notional of a typical exotic option is generally on the order of 10 million dollars. While there are larger asset classes, the equity derivatives market is certainly not small.

1.2 Workflow: the lifecycle of exotic trades

The lifecycle of exotic trades involves many different people in different rôles, including structurers, salespeople, quantitative analysts ("quants"), risk management teams and traders. Structurers create new ideas for trades and variations on existing ones. Their aim is to find novel and effective ways to meet the client's requirements. The structurers then develop and test a prototype implementation. Salespeople support interaction with the customer, and write a term sheet that describes the option using a combination of prose and mathematical notation. The quants provide mathematical modelling expertise.

The structurers or the quants write the payout function, which defines how much money the bank will pay out under given market conditions. They will carefully inspect the implementation to ensure it matches the term sheet. A risk management team then independently checks the implementation of the payout function against the term sheet.

A completely new type of trade triggers a much more rigorous sign-off process than for a carbon copy of an existing structure. As such, the ability to innovate is held back by the pressure to standardize.

After the risk management team signs off on an exotic trade, a trader will then receive the coded payout function and create a final price at which the product is sold. After the sale, the traders will manage the financial risk for the lifetime of the trade by pricing the portfolio under various market conditions and hedging with vanilla instruments to minimize the money made or lost as the markets move. In addition to these calculations, the traders will perform other analyses on the structure of the trade itself, to identify, for example, forthcoming events which must be managed carefully.

Other teams handle the actual payments. Some trades pay money to the client at the end, while others pay throughout the lifetime of the trade. In either case, the system must identify when a cash flow occurs and the correct amount to be paid.

1.3 Downstream operations

A formally described trade can be used in various ways. Each trade defines its input parameters, so it should be possible to automatically derive an interactive input form for them. To manage the trade, one would want to analyse how the trade is likely to perform in the future. For this analysis the trade can be compiled into an efficient representation, suitable for plugging into a Monte Carlo simulation engine. Alternatively, extracting from the trade the conditions expressed in it (for instance, the $/€ rate climbing above 1.5) allows the centralized monitoring of the corresponding market conditions. Finally, for those aspects of a trade that require human analysis, it should be possible to represent the trade in a concise mathematical form that analysts can study and discuss.

1.4 An existing implementation

The generality of exotic trade payouts and the corresponding downstream operations lends them to be expressed and manipulated through a DSL (Mernik *et al*., 2005). In the past this DSL has often been some form of imperative scripting language, although there has been a noticeable move towards using functional languages in a number of banks, which is not surprising given that the domain is readily specified by functions. Our pre-FPF pricing engine took in a directed acyclic graph (DAG) equivalent to the syntax tree of an expression representing the payout function, with sharing (a "payout DAG"). This legacy representation had no higher-order functions, so all parts of the trade were fully expanded.

Individual one-off trades were created by hand-writing these payout DAGs. If a trade had several legs, these legs were implemented by cutting and pasting the nodes. As such, rapidly varying the parameters of the trade was difficult. While hand-crafting the payouts is very flexible, it is manually intensive and error-prone. To avoid the overheads of manual labour, once a common pattern of trades was identified, it was converted into a "template". A template provides a standardized input form that covers the majority of the variations people will want on that style of trade. Special cases can still be encoded in a one-off fashion.

The problem with the templates is the effort required to create them. Quants define a standardized pattern to represent a trade type, and write a validation library that ensures a payout DAG matches this pattern. Risk managers sign off the library, and then the IT department develops a front-end to generate these DAGs. The Excel-based front-end will have plenty of trade-type specific code to deal with the particular trade's input fields, and will have a case-specific graph-generator to emit the payout DAG, written in Excel's scripting language, Visual Basic for Applications (VBA). Each trade is saved in an *ad hoc* serialization format. While it can take just hours to generate pricing instructions for an individual trade, the corresponding template may take months to release. Since new templates are so expensive there can be a tendency to cram a new trade type into an existing structure through creative misuse of existing features, increasing the risk of mistakes. At the end of 2006 Barclays managed more than 35 templates, representing a total population of several thousand trades.

A released template in the trade entry system is not the end of the story. The various downstream systems need to be able to process the new trade type. Each system has a central "case" statement to control processing on a per-trade basis, which must be modified with the release of a new template. Hand-crafted trades that are not yet templated must be manually managed.

Given the pace of the business, systems must adapt quickly to cope with new products. The pressure to add new features, rather than maintain existing code (or step back and look for a different, better approach) can lead to increasingly unreadable and unmaintainable code, causing growing operational risk.

The problems we outlined in the preceding paragraphs were the core motivation behind FPF. By spending effort on creating generic analyses that can be applied to all trade types, FPF aims to consistently minimize the cost of production-quality infrastructure, independent of whether we are constructing single one-off trades or wide-ranging templates.

2 FPF overview

We created FPF through an experimental iterative process rather than a formal design based on concrete absolute goals. Nevertheless FPF is best described by looking at its design goals, its structure and some representative examples.

2.1 Design goals

The core requirement for FPF was to be able to drive multiple back-ends from a single trade description. These back-ends provide different interpretations or analyses of the scripts and trades. The artifacts generated can range from a concrete price through to a user interface schema or barrier risk report.

We designed the payment description DSL with the following goals in mind.

Declarative, compositional construction of payouts. This goal is the driving force behind using a functional approach. A declarative language simplifies the creation of multiple interpretations and static analyses. Compositional construction is a natural mapping of the problem domain and encourages code reuse by composing new trades from existing trade components.

Minimal coupling. Downstream systems should not be allowed to pollute the language with incidental details. While extensions to the scope of the project may require extensions to the language, our yardstick was that one should not be able to look at a language feature and identify it with a concrete, implementation-specific detail of other systems.

Extensible support for the payout DAGs. If we could not support the pre-existing payout back-end, the project would be dead before its first release. If the language lacked extensibility we could end up in the worse situation where it failed after we had gained everyone's support.

Painless embedding in Haskell. Once we decided to use Haskell (as described in Section 3), embedding the language within Haskell using the *language specialization* DSL design pattern (Spinellis, 2001) allowed us to reuse much infrastructure (e.g. parsing and typing) and provided us a flexibility which would not otherwise have been available.

Small set of core primitives. By minimizing the number of core primitives, multiple back-ends can be produced with minimal effort. Given a set of primitive functions,

$+;-;*;/$	::	$Double \to Double \to Double$	Arithmetic operators
$min; max; pow$	::	$Double \to Double \to Double$	Arithmetic operators
$abs; log; exp$	::	$Double \to Double$	Arithmetic operators
$observe$	::	$Asset \to Date \to Double$	Asset price observation
$cond$	::	$Bool \to a \to a \to a$	Conditional
map	::	$(a \to b) \to List\ a \to List\ b$	Map
$zipWith$	::	$(a \to b \to c) \to List\ a \to List\ b \to List\ c$	Zip
$foldl$	::	$(a \to b \to a) \to a \to List\ b \to a$	Left fold
$foldl1$	::	$(a \to a \to a) \to List\ a \to a$	Left fold variant
$foldr$	::	$(a \to b \to b) \to b \to List\ a \to b$	Right fold
$mapAccumL$	::	$(a \to b \to (a, c)) \to a \to List\ b \to (a, List\ c)$	Combined map/fold
$name$	::	$String \to a \to a$	Annotation

Fig. 1. The core FPF primitives.

user-friendly helper functions can be wrapped around them. We considered providing a wider set of primitives with default implementations (in the style of Haskell type classes), but we took the simpler approach to avoid unnecessary complexity.

Attachment of metadata. One of the important goals of FPF is to provide an end-to-end solution, including input forms and documentation. One approach would be to use an external tool, similar to Javadoc (Kramer, 1999) or Haddock (Marlow, 2002). FPF uses this approach to annotate the trade type, but we can also annotate individual sub-expressions using the “name” primitive and then extract the metadata by static analysis.

2.2 *Language overview*

As Haskell embeds FPF, FPF’s syntax is simply a restricted subset of Haskell’s. FPF is thus defined by the primitive functions made available. Figure 1 shows the core FPF primitives and their types.

The types *Double* and *Bool* are not those provided by the Haskell prelude, but are types which play the equivalent roles in FPF. *List* is similarly the FPF representation of a list. We use our own types since our “*Double*” may represent, under different interpretations, everything from an actual floating point number through to a syntax tree or set of annotations. The exact definitions of the types vary with the embedding used.

All type variables must be instances of *FPFVal*, the type class representing all values that can exist in FPF. All the types above and tuples of those values are instances of *FPFVal*. This class allows us to distinguish the values in FPF from those that are not, preventing us from applying FPF primitives to non-FPF values. The class also allows us to uniformly traverse FPF programs and is needed to make tuples (and to a lesser degree, functions) part of the embedded language.

In FPF *let* bindings cannot be recursive – the language does not support general recursion. Instead we provide higher-order functions such as *map* and *fold* to process lists of data. The restriction on the use of recursion simplifies both the analysis of scripts from within Haskell and encourages programmers with little functional programming experience to adopt common functional patterns. In other words, we have created something akin to a combinator library (keeping the constraints, but without the points-free approach – see Section 5.2 for details). The fact that the language is not Turing-complete is not an impediment in practice and in many cases simplifies analysis.

We slowly changed the primitives available, adding new primitives as they were requested. Occasionally the new features subsumed old features, at which point the old primitive could be removed, and a helper function added that implemented the old primitive's behaviour in terms of the new. Through such evolutionary changes we were able to modify the language effectively without requiring huge script rewrites.

2.3 Examples

For examples, we shall use those three payouts described in Section 1.1. The code is shown in Figure 2. As most of the functions are modelled on standard Haskell functions of the same name, we shall only discuss the FPF-specific functions. The *name* primitive attaches an annotation to a value which can be extracted and displayed on input forms, for example, but has no numeric effect.

The primitive *observe* is used to get the value of a particular asset on a particular date. To get the value, it must pull the actual asset prices out of some hidden environment. As the *observe* function is deterministic in any particular market state, this apparently impure behaviour is of no concern to the script writer and is purely an implementation detail.

In contrast with earlier work (Peyton Jones *et al.*, 2000), we only support observing asset values at discrete times. This was a conscious decision to simplify the language, based on known usage patterns. In practice, it has not been a problem; where necessary, a sequence of discrete observations makes a good approximation to continuous observation.

Our examples are for some of the simplest trades we do. However, the compositional nature of the problem domain and language means that the script size and complexity scale up linearly with the complexity of the term sheet.

3 Implementation

Using Haskell to implement the FPF was a non-trivial decision in an environment so strongly weighted towards C++ and VBA. The decision can be split into two parts: choosing a functional language and choosing Haskell.

Choosing to use a functional language to implement a functional payout language has had a number of benefits. As functional programming researchers have a tendency to write their languages using existing functional languages, these languages generally provide very good support for language engineering in general and DSL implementation in particular. Pattern matching, for example, vastly simplifies the writing of a peephole optimizer. Previous experience using functional languages meant the developers were confident that the desired result could be achieved

```
perf :: Date -> Date -> Asset -> Double
perf t1 t2 asset = observe asset t2 / observe asset t1 - 1

bestOf :: (List Asset, Date, Date) -> Double
bestOf (assets', startDate', endDate') = foldl1 max perfs where
   assets    = name "Assets"        assets'
   startDate = name "Starting date" startDate'
   endDate   = name "End date"      endDate'
   perfs     = map (perf startDate endDate) assets

cliquet :: (Asset, Double, Double, Date, List Date) -> Double
cliquet (asset', gf', gc', initDate', dates') = max gf $ min gc val where
   asset    = name "Asset"         asset'
   gc       = name "Global cap"    gc'
   gf       = name "Global floor"  gf'
   initDate = name "Initial date"  initDate'
   dates    = name "Observations"  dates'
   cliquetPerf (prevDate, prevSum) (currDate', lc', lf') =
      (currDate, prevSum + currPerf) where
         currDate = name "Observation date" currDate'
         lc       = name "Local cap"        lc'
         lf       = name "Local floor"      lf'
         currPerf  = max lf $ min lc $ perf prevDate currDate asset
   (_, val) = foldl cliquetPerf (initDate, 0) dates

napoleon :: (Asset, Double, Date, List Date) -> Double
napoleon (asset', coupon', initDate', dates') = max 0 $ worst + coupon where
   asset    = name "Asset"         asset'
   coupon   = name "Fixed coupon"  coupon'
   initDate = name "Initial date"  initDate'
   dates    = name "Observations"  dates'
   napCliquet prevDate currDate = (currDate, perf prevDate currDate asset)
   (_, perfs) = mapAccumL napCliquet initDate dates
   worst = foldl1 min perfs
```

Fig. 2. Example trades.

more quickly with a functional language, and with a gentle learning curve. As we performed the work outside of a traditional IT team, management was willing to take a risk on something that promised a good fit and rapid development process.

We tossed away an initial prototype in Prolog after we realized that the code did not use any of the language's logic programming features. Subsequent prototyping continued with functional languages. At that point it had become clear that our main requirements were for strong typing and a large user population. Our previous experience showed that strong typing reduced the effort required to develop and maintain high quality software, by both identifying many "typo"-style bugs and forcing the developer to plan their algorithms more carefully, reducing the maintenance load later. A large user population generally implies better developed, more stable tools and a bigger pool from which to hire developers, both now and in the future.

This paper's second author chose Haskell during the prototype stage due to past familiarity (Spinellis, 1993) and the existence of a mature open-source implementation – GHC (Hudak *et al.*, 2007). Technically, CAML seemed an equal fit. Erlang was somewhat alien to us, F# seemed too unproven and we did not see any other appropriate candidates. Once we wrote the first prototype in Haskell the path was set. A rewrite in another language would have been far from difficult, but there was no strong case for it.

While the advantages of using Haskell were fairly obvious, the possible pitfalls had to be taken into account. These included:

Integration issues. The FPF has to work as part of a large, complicated, distributed and heterogeneous system. In its way, using Haskell here was an advantage, as it made it easier to argue for a command-line API and text-file I/O. In a wider system suffering from tight coupling, having solid control over our API allowed for a clean design.

Maintenance and support. Maintainability is required on a number of levels. Haskell's high level of abstraction made the addition of new features a lot simpler than it would have been in C++. In the longer term, maintaining the source base requires a supply of Haskell programmers, but if rumours of other Haskell-based banking projects are to be believed, the supply of this skill set is likely to increase with demand. Finally, the Haskell libraries are comprehensive and actively developed, as are the compilers.

Performance. This was the real unknown. Performance on the prototype looked good, but we did not know how it would scale. Other systems implemented in Haskell, like Darcs (Roundy, 2005), have shown that performance can be an issue in real-world workloads. In addition to the runtime performance, we also needed to take into account compilation speed and executable size. We found that compilation speed, while not stellar, was perfectly acceptable. Excellent runtime performance was not vital, since the FPF is not used in any computationally-expensive inner loops. Its performance has certainly been adequate, and our experiences are discussed in more depth in Section 5.5. A surprising weakness was executable size, with our executables coming in at 5 megabytes each. With several executables per release, and several prototype releases running simultaneously, FPF turned out to be unexpectedly disk hungry. Disk space consumption is something of a weakness when one takes into account the need for worldwide replication of the files, but has not been a major problem.

A combination of *make* (Feldman, 1979) and Perl (Wall & Schwartz, 1990) scripts complete the build system infrastructure, including a program-level regression test suite. Relatively mainstream tools were selected here to simplify maintenance.

We have not used any tools like QuickCheck (Claessen & Hughes, 2000), but this is purely a matter of priorities. A requirement was to be able to check that modifications to the FPF do not result in unexpected price changes to trades booked on FPF, so we have test coverage at that level. Unit tests would be a useful extension, but we have continually been working on other features with higher priority. We hope to make increasing use of Haskell Program Coverage (Gill & Runciman, 2007) to check that our existing tests are providing adequate coverage.

The system has gradually evolved behind the scenes. Over FPF's lifetime, back-ends have been implemented as both *shallow embeddings*, where the primitive operators actually perform the operations they represent, and a *deep embedding*, where the primitives simply generate their own abstract syntax tree (AST), which can then be interpreted. These approaches match "Language embedding" and "Embedding a compiler" in (Elliott *et al.*, 2003), a paper which explores many of the issues we also faced.

We started off with a single back-end, a shallow embedding which generated payout DAGs. We added further back-ends, also implemented as shallow embeddings. Each back-end was a separate code base which the scripts needed to be run against. While this approach was simple, it required us to manually keep the types consistent, and required a great deal of recompilation when we created a new script.

To relieve these problems, we migrated to a deep embedding. We used a single set of primitives to convert the scripts to ASTs, which could then be processed by a variety of interpreters and static analyses. We now have a two-step process:

1. The script function is run by the Haskell compiler under the deep embedding, producing an AST which is then serialized. Haskell's type system guarantees the AST is correctly typed.
2. The AST can be read in and processed by any of the back-ends, without recompilation. Code from multiple back-ends can live in the same executable without type clashes.

The refactoring we described quickly paid for itself in time savings and maintainability improvements.

Our approach still requires us to compile our scripts as Haskell code. An alternative we have considered, but not yet tried is based on GHC's "Language" library. This library allows one to directly parse Haskell source to ASTs. The downside is that we would have to provide Haskell's semantics and typing ourselves. Currently, using the library seems a complex approach with few benefits, but it does provide us with an "escape hatch" should the requirements of the project change.

4 Usage at Barclays

The FPF (excluding trade scripts) is two to three effort-years of work, with one full-time and one part-time developer. It consists of 20,000 lines of Haskell. The trade scripts contribute a further 30,000 lines, distributed over more than 400 trade types (including variants), and were written by half a dozen individuals. There are around 20 direct users (traders and structurers), with further users hidden behind the scenes outside of the front-office area.

Currently we use each FPF script in a number of downstream operations.

Generation of pricing instructions. Our original target was the generation of instructions in the legacy "payout DAG" format. Since then we have started to extend the range of target languages. Currently, we can write out instructions in another

$$\max\left(0, \min_{i=1}^{\mathrm{len}(t^O)}\left(\frac{S^{TOP}\left({t^O}_i\right)}{S^{TOP}\left(a_{i-1}\right)} - 1\right) + FC\right)$$

where

$$\begin{aligned} a_0 &= t^{ID} \\ a_i &= {t^O}_i \end{aligned}$$

The parameters to the preceding trade type are as follows:

Variable	**Description**	**Type**
TOP	Top-level input	Tuple of $(S^{TOP}, FC, t^{ID}, t^O)$
S^{TOP}	Asset	Asset
FC	Fixed coupon	Double
t^{ID}	Initial date	Date
t^O	Observations	List of Date

Fig. 3. Recursive equations automatically generated from the "Napoleon" trade.

proprietary functional format, and in C, which we plan to compile and link into our Monte Carlo engine on the fly. FPF can also emit parameters for PDE-solving.

Generating recursive equations in TEX. An alternative FPF target was to create human-readable instructions. Most people involved in finance, while not used to functional programming, are conversant with common mathematical notation. As such, we can present the scripts by transforming them into sets of recursive equations. This transformation is useful as it allows us to extend our target audience of people who can read an FPF script without providing any further training. The transformation can be applied to both the abstract trade script without input data, in order to generate a general description of the trade type, and to a script specialized with concrete assets and dates to produce a term sheet for a particular trade.

The TEX back-end is perhaps best explained with demonstration output. The term sheet equations generated for the "Napoleon" trade described in Section 2.3 are shown in Figure 3.

One could argue that generating a set of human-readable equations is simply another form of pricing instruction generation, albeit one with a very different optimization target: of readability rather than efficiency (and where executability is a distinctly secondary target). In FPF this multitude of back-ends is actually the case, since the very last step of generating LATEX can be replaced with emitting Mathematica (Wolfram, 2003) instructions, which can be both visually compared to the LATEX output, and executed in a Monte Carlo engine, with the price checked against our other Monte Carlo systems as part of our validation process.

Input form generation. The parameters to the scripts must be somehow specified by the user. We have created GUI tools to allow us to enter parameter values which match a given type. To generate an input form for a particular trade script, we just need to extract the type of the function's parameters, and any associated annotations, which are used to create labels on the input form.

Other analyses. We are working on tools that allow us to perform generic analyses across the FPF trades in order to simplify trade management. For example, it is simple to pull out the set of dates upon which observations and payments take place, with some context information, or to generate a list of payments that should have taken place.

We are extending these analyses with tools to help monitor "barriers" or discontinuities in the payouts, giving us, for example, the expected size of the discontinuity, and when and under which market conditions the barrier levels are breached.

5 Lessons learned

From the FPF project we learned a lot regarding the application of functional languages to concrete production systems with all the deadlines and hard requirements that this entails. As always, a number of the lessons came from failed approaches. An interesting lesson we learned from these failures is that the expressiveness of Haskell allowed us to rapidly and efficiently make these mistakes, leaving us plenty of time to develop a useful system.

In the following sections we endeavour to highlight some of the most notable issues.

5.1 Efficient processing of embedded languages

We were initially unaware of the efficiency issues which could occur when we evaluated complex payouts. Consider the following script:

```
f x = let x2 = x  + x
          x3 = x2 + x2
          x4 = x3 + x3
          ...
      in x20
```

A straightforward numerical evaluation would calculate the result in 20 steps. An alternative interpretation generating an AST would also take 20 steps, and internally it would create a DAG with much sharing. The problem comes when one wants to evaluate it. Naïve evaluation processes the tree fully, in exponential time. One can try to improve the result with caching, so that if a sub-expression has already been evaluated we can use the pre-existing result. Unfortunately, the built-in comparison operators are purely structural. In other words, in order to avoid fully traversing a data structure, we must perform a full traversal to compare it to cached items!

We eventually discovered this optimization is a well-known problem. Using CSE, as suggested in (Elliott *et al.*, 2003) would not work for us, as we would have to fully traverse the tree, taking exponential time. Solutions based around memoization with stable names (Peyton Jones *et al.*, 1999) and observable sharing (Claessen & Sands, 1999) are discussed in the literature, but not knowing what to look for, we reinvented the wheel with our own form of “hash consing”. After the fact we refactored the code to be closer to the academic ideal.

5.2 Combinators versus functions

Many domain-specific embedded functional languages seem to get implemented as combinator libraries (Cardelli & Davies, 1997). The points-free approach, while elegant, can make code unreadable, especially if it is written by quantitative analysts moonlighting as functional programmers. Therefore, while we could have defined a polymorphic higher-order combinator library, in FPF we encourage the user to write in a more basic functional style. Examples of this style are shown in Figure 2, and seem far clearer to us than the same code written in a points-free style.

5.3 Scrap-Your-Boilerplate and caching

Scrap-Your-Boilerplate (SYB) (Lämmel & Peyton Jones, 2003) has provided a very convenient mechanism to accelerate the writing of transformations. Using it has encouraged us to embrace the common functional pattern of splitting out the recursive traversal and per-node operations. Sadly, the support for monadic generic transformations is not as strong as for non-monadic transformations (lacking, for example, generic monadic queries), leading to some fairly ugly code as we fake up the query with a *WriterT* monad transformer (Grabmüller, 2006).

We mostly use SYB in combination with caches held inside a *StateT* monad transformer. As with the original SYB implementation, we split our code into the function that recurses over the tree (with caching and short-circuiting) and the function that transforms individual nodes.

5.4 Scope creep and incremental change

Haskell is a remarkably refactorable language. For the current application intermediate ASTs provide a perfect interface between sub-systems, so that processing stages are minimally coupled. Where modifications to data types are necessary, type inference minimizes the textual changes and the tools ensure no cases are missed. The pure functional nature of the language means incremental changes tend to have localized effect. Incremental changes, in tandem with a home-grown regression test system, reduce the risk of behaviour changing unexpectedly while modifying the code base. In this way major changes (almost complete rewrites) have been achieved without major problems.

On the other hand, our system’s flexibility can lead to unnecessary playing. Perfectionists can have a field day quietly rearranging the internals when they should be adding new features. Not that rearranging Haskell code is painless; for example, converting code to a monadic form in order to add caching to our transformations was a mind-numbing operation.

One of our initial unconscious decisions was to blur the lines between the embedding and embedded language. In the prototype, the embedded language was a mechanism for generating a payout DAG using whatever Haskell functions seemed useful. In later versions, while the embedded language became better defined, we found the ability to break out into the embedding language in an emergency useful for prototyping and even limited production use. We used calls into non-FPF Haskell as our training wheels for developing the embedded language; we finally removed these calls with the switch to a deep embedding.

As experienced practitioners would expect, FPF has been subject to distinct scope creep.[1] The original plan was to handle all trades that were either generated with existing templates or by hand. These trade types tended to have their complexity limited by what was maintainable. As FPF has allowed scripts to be written with no regard to how complex the underlying payout DAGs got, the complexity of trades attempted has grown markedly.

As an example, we previously had problems maintaining hand-structured trades with a few hundred nodes in the payout DAG. FPF makes it much easier to handle such trades. However, FPF also makes it just as easy to construct trades with 30,000 node payout DAGs (to the extent that the output of FPF is now stressing downstream systems!). Some such trades are simply scaled up versions of existing trades, but for the most part these represent trades we had no way of expressing before. Early FPF scripts were generally around 30 lines in length, while our larger scripts now have about 200 lines.

The original architecture was not designed with this level of scalability in mind, and it is to the system's credit that changes to cope with this level of scaling can be incorporated without major risk.

5.5 Complexity and performance issues

Moving to Haskell allowed complex algorithms to be written in a fraction of the time required for implementation in an equivalent first-order imperative production language. However, the ease with which complex algorithms can be written can lead to severe performance problems; an algorithm that would otherwise take days to write can be implemented with little thought, including perhaps little thought for the algorithm's complexity. We found GHC's profiler (Sansom & Peyton Jones, 1995) a real boon for fixing such problems.

While the exponential blow-up described in Section 5.1 was the worst example seen by our rather naïve functional programmers, even the polynomial time optimization steps start to become very expensive on graphs containing tens of thousands of nodes after CSE.

The most complex performance problems came from the TEX-generation subsystem, which relied heavily on substitution. Straightforward substitution into sub-expressions led to an exponential blow-up as the amount of sharing was reduced. The solution was not to perform the substitution itself, but instead

[1] In fact, FPF was always expected to have some scope creep beyond its initial requirements – indeed, the need for flexibility was one of the main reasons for choosing Haskell. What was surprising, however, was how much scope creep we have seen (and have been able to accommodate) in practice.

insert a node representing the substitution and track that information during later passes.

Haskell is perhaps infamous for its space leaks. However the only time we have seen space leaks is when they have accompanied algorithms with poor complexity characteristics. Perhaps this is the result of always compiling with optimizations (including strictness analysis) and using machines with plenty of memory. While we have seen very few pure space leaks, our use of caching and hash-consing means that our programs are quite allocator-heavy and started to spend most of their time in garbage collection. Simply increasing the default heap size in the runtime system was a quick way of remedying the corresponding overhead.

The lesson here, perhaps, is that there is no silver bullet. While a functional approach greatly simplified our task, commonplace issues such as algorithmic complexity cannot be set aside. Furthermore, these issues may crop up in forms one does not recognise, so that the developer must relearn previous experiences.

5.6 Haskell for non-developers

While the programmers who worked on the core of the FPF framework had previous exposure to functional languages, those implementing the trade scripts generally did not. The scripters are mostly mathematicians with expertise in continuous maths (specifically stochastic calculus). They have been trained in C++, although by no means are they all expert developers. Their previous experience of writing trade scripts was hand-generating the payout DAGs in Excel.

An initial worry was that they would at best write imperative code in a functional language and at worst flounder completely. The first few scripts were distinctly imperative, but with a bit of practice the payouts were being written in a clean functional style. This transformation is perhaps not surprising to those who teach functional programming, but it was a pleasant surprise to see people quickly picking up the ideas in a production environment.

5.7 Re-inventing the wheel

While the developers were experienced with functional languages, this was the first time they had used such a wide set of Haskell libraries in a commercial setting under commercial pressures. When we needed a complex feature, it was generally a case of reinventing the wheel, badly, followed by a refactoring process where we replaced the implementation by a call to a more-general library function (once we identified the appropriate papers). As simple concrete examples, early on we reinvented some state monad machinery, and we are currently looking at moving to a more structured memoisation framework based on earlier work (Peyton Jones *et al.*, 1999; Claessen & Sands, 1999).

The development pattern we described is difficult to fix, since although the papers are there, it is somewhat difficult to identify *a priori* what is relevant, especially under time constraints, and the low-risk approach of rolling-your-own (hacky and restricted) code takes over. The preferred solution would be developers more experienced in Haskell, but the combined Haskell and financial knowledge is rather rare, and it seems that even a relatively inexperienced Haskell programmer can be more productive than an experienced C++ programmer in domains such as ours.

However, reimplementing the wheel is an excellent way of understanding the tricks involved, and a well-designed framework should not make it difficult to replace the offending code with a well-constructed library, so the overall cost is not great.

5.8 Interfacing with other systems

One of the worries typically raised about adopting a new technology in a large and complex system concerns the interfacing of its components. As we described in Section 3, the adoption of Haskell has allowed us to refrain from tight coupling with existing systems. We also found the same approach effective for encapsulating our serialized data structures. Using a Haskell-like text data structure rather than, say, XML we were able to control the representation fully, considerably reducing the risk that third parties will attempt to process these structures themselves. Instead, we provide the translation tools to present the data in the format they need, and we can continue to write fully generic tools.

The command-line interface we adopted made testing and fault identification into a far more productive process. Should in-process operation be required in the future, a COM interface can be integrated (Finne *et al.*, 1999).

Of course, there are issues remaining to do with versioning and configuration management within a production system. While these issues can be tackled using the standard software engineering approaches, we have yet to consider whether Haskell will provide us with any good tricks to ease these problems.

6 Related work

The interested reader may consult Hull (2005) for a general background on derivatives. Vanilla and exchange-traded derivatives are standardized, and describing the trades becomes an exercise in naming conventions. For exotic trades, however, trade representation is a real issue. Little published work addresses the problem, perhaps because the solutions involved are either kludges the authors do not wish to show off, or thought to be proprietary, cutting-edge software best kept secret. The published approaches in the field of trade representation include the use of frameworks (Eggenschwiler & Gamma, 1992) and the adoption of DSL (van Deursen, 1997).

In the past decade, considerable work has been published on the use and implementation of DSL (Spinellis, 2001; Mernik *et al.*, 2005) and in particular the use of Haskell as a vehicle for creating embedded DSLs (Hudak, 1996; Anand *et al.*, 2001). An early attack on our specific problem involved the development of the DSL RISLA, whose code was subsequently compiled into COBOL (Arnold *et al.*, 1995; van Deursen & Klint, 1998). Later work (Peyton Jones *et al.*, 2000; Peyton Jones & Eber, 2003) used Haskell to allow payouts to be defined in an abstract functional language and to provide multiple interpretations of the products. This work was apparently commercially adopted and marketed (MLFi, 2004; LexiFi, 2007). We considered that product as a COTS alternative to FPF, but in the end the risks in integrating the product and loss of control in development meant that we instead opted for an in-house product designed from the start to integrate with existing systems.

Various methods for addressing the trade representation problem have spread through word-of-mouth. Given the lack of formal literature, it seems preferable to report what we have heard of than to say nothing. As such, the rest of this section is necessarily light on references. It seems that each bank has at least one (often more!) proprietary payout language. These languages are commonly custom imperative scripting languages. In such a language, the price-calculation stages can be tied to the underlying time-steps of the trade. This evaluation approach allows the past (with its fixed values) to be calculated once, the state to be cached and then the future dates to be calculated repeatedly under different scenarios, saving much execution time. However, this approach ties the language tightly to the evaluation mechanism and is not convenient for all trade structures. The alternative of binding trade descriptions to an existing general-purpose language (such as VBA or Python (van Rossum & Drake, 2006)) for the payout evaluation not only allows incredible flexibility in the pricing strategy, but also makes it very difficult to perform any interpretation beyond a simple pricing.

Expressing the payout in a DSL (often a subset of an existing language) allows various analyses to be performed, and lets the end-users work with tools they feel comfortable with. However, this approach involves a delicate trade-off between expressivity and the complexity of the translation to specify the subset that is deemed convertible. A common line of attack here involves writing simple payouts as a set of Excel formulae, checking that they work inside the spreadsheet, and then "shredding" the cells into a document suitable for external evaluation.

Other institutions also use a functional payout language, but the DSL itself is typically implemented in C++ or Java, rather than in a functional language. Even then, they generally perform only a single interpretation; that is used for pricing. There may also be other systems in use with particularly novel and interesting approaches but, alas, they apparently have not been published in the open literature.

7 Conclusion

FPF is now integral to our equity exotics business, both tactically and strategically. Not having a system like FPF would give us unacceptable operational risk in a few years time, but in the shorter term it is already making a big impact. We now write all new trades using FPF. In addition, we have migrated all our one-off trades to FPF, freeing up reserve cash devoted for operational risk. Currently we are also starting to migrate templated trades to FPF.

The project is almost a victim of its own success. While we reduced the development cycle, the opportunities this shortened cycle presents mean that we still have a backlog of new ideas to implement that is far greater than our available developer bandwidth!

It will be interesting to see what the future holds for payout description technology. Generating a trade's contract and having each party in the transaction reconcile it with their internal descriptions is currently a manual and error-prone business. Once financial institutions have automated all processing associated with their internal representation of a trade, it makes sense to standardize on external trade descriptions, to eliminate the risk of ambiguous term sheets or human error during translation. What should such a representation look like? Would the constraints of standardization stifle novelty, or would it reduce risks and provide the framework for increased innovation?

References

Anand, S., Chin, W.-N. & Khoo, S.-C. (2001) Charting patterns on price history. In *ICFP '01: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. New York: ACM, pp. 134–145.

Arnold, B. R. T., van Deursen, A. & Res, M. (1995) An algebraic specification of a language for describing financial products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, Seattle, WA, Wirsing, M. (ed.), IEEE, New York, pp. 6–13.

Cardelli, L. & Davies, R. (1997) Service combinators for web computing. In *USENIX Conference on Domain-Specific Languages*. Berkeley, CA: USENIX Association, pp. 1–9 .

Claessen, K. & Hughes, J. (2000) QuickCheck: A lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Not.* **35**(9), 268–279.

Claessen, K. & Sands, D. (1999) Observable sharing for functional circuit description. In *Proceedings of Asian Computer Science Conference*, Phuket Thailand. Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany.

van Deursen, A. (1997) Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *STJA'97: Smalltalk and Java in Industry and Academia*, Erfurt, Germany, Ilmenau Technical University, pp. 35–39.

Eggenschwiler, T. & Gamma, E. (1992) ET++SwapsManager: Using object technology in the financial engineering domain. In *OOPSLA '92: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*. ACM, New York, pp. 166–177.

Elliott, C., Finne, S. & de Moor, O. (2003) Compiling embedded languages. *J. Funct. Prog.* **13**(2). Updated version of paper by the same name that appeared in SAIG '00 proceedings.

Feldman, S. I. (1979) Make—a program for maintaining computer programs. *Softw. Pract. Exp.* **9**(4), 255–265.

Finne, S., Leijen, D., Meijer, E. & Peyton Jones, S. (1999) Calling Hell from Heaven and Heaven from Hell. In *ICFP '99: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*. Paris, France, ACM Press, New York, pp. 114–125.

Gill, A. & Runciman, C. (2007) Haskell program coverage. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. New York: ACM, pp. 1–12.

Grabmüller, M. (2006) *Monad Transformers Step by Step*. Draft paper. Available online `http://uebb.cs.tu-berlin.de/~magr/pub/Transformers.pdf`.

Hudak, P. (1996) Building domain-specific embedded languages. *ACM Comput. Surv.* **28**(4es), 196.

Hudak, P., Hughes, J., Peyton Jones, S. & Wadler, P. (2007) A history of Haskell: Being lazy with class. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. San Diago, CA, ACM, 12-1–12-55 ACM Press: New York, NY, USA, pp. 12-1–12-55.

Hull, J. C. (2005) *Options, Futures and Other Derivatives.* 6th ed. Upper Saddle River, NJ: Prentice Hall.

Kramer, D. (1999) API documentation from source code comments: A case study of Javadoc. In *SIGDOC '99: Proceedings of the 17th Annual International Conference on Computer Documentation.* New York: ACM, pp. 147–153.

Lämmel, R. & Peyton Jones, S. (2003) Scrap your boilerplate: A practical approach to generic programming. In *TLDI 2003: Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation.* ACM Press, New York.

LexiFi. *LexiFi Platform.* `http://www.lexifi.com/`. Last accessed September 2007.

Marlow, S. (2002) Haddock, a Haskell documentation tool. In *Proceedings of the ACM SIGPLAN Workshop on Haskell.* ACM Press, New York.

Mernik, M., Heering, J. & Sloane, A. M. (2005) When and how to develop domain-specific languages. *ACM Comput. Surv.* **37**(4), 316–344.

MLFi. (2004) *Structuring, Pricing, and Processing Complex Financial Products with MLFi.* Available online `http://industries.bnet.com/whitepaper.aspx?&tags=window&docid=103795`. Last accessed January 2007.

Peyton Jones, S. & Eber, J.-M. (2003) How to write a financial contract. In *The Fun of Programming*, Gibbons, J. & de Moor, O. (eds.), Palgrave Macmillan.

Peyton Jones, S., Eber, J.-M. & Seward, J. (2000) Composing contracts: An adventure in financial engineering (functional pearl). In *ICFP '00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming.* New York: ACM, pp. 280–292.

Peyton Jones, S., Marlow, S. & Elliott, C. (1999) Stretching the storage manager: Weak pointers and stable names in Haskell. In *Implementation of Functional Languages.* Springer Verleg, Berlin, pp. 37–58.

Roundy, D. (2005) Darcs: Distributed version management in Haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell.* ACM Press, New York, pp. 1–4.

Sansom, P. M. & Peyton Jones, S. (1995) Time and space profiling for non-strict, higher-order functional languages. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York: ACM, pp. 355–366.

Spinellis, D. (1993) Implementing Haskell: Language implementation as a tool building exercise. *Struct. Prog.* **14**, 37–48.

Spinellis, D. (2001) Notable design patterns for domain specific languages. *J. Syst. Softw.* **56**(1), 91–99.

The Bank for International Settlements. (2007) *BIS Quarterly Review.* Available online `http://www.bis.org/publ/qtrpdf/r_qa0703.pdf`. Last accessed September 2007.

van Deursen, A. & Klint, P. (1998) Little languages: Little maintenance. *J. Softw. Maintenance* **10**(2), 75–92.

van Rossum, G. & Drake, Jr., F. L. (eds). (2006) *An Introduction to Python.* Network Theory, Bristol, UK.

Wall, L. & Schwartz, R. L. (1990) *Programming Perl.* Sebastopol, CA: O'Reilly and Associates.

Wolfram, S. (2003) *The Mathematica Book.* 5th ed. Wolfram Media. Champaign, IL.