Haskell at Barclays: Exotic tools for exotic trades

Tim Williams | 5 December 2013

BARCLAYS

CODE MESH LONDON 2013

Introduction

Exotic equity derivative contracts come in a variety of structures and clients are continually requesting new ones. In order to remain competitive and meet regulatory requirements, Barclays needs to:

- bring new products to market rapidly and efficiently
- manage the resulting highly heterogeneous trade population

This talk summarises *Going functional on exotic trades*, by Frankau, Spinellis, Nassuphis and Burgard [1] and gives an update on the project and some of the techniques we use.

Options

An equity option is a derivative contract giving the owner the right, but not the obligation, to **buy** (call) or **sell** (put) an underlying stock asset at the specified strike price, on or before a specified date.

- For a strike price equal to the initial stock price (at the money):
  - a **call** pays the price difference if the stock goes **up**, or zero otherwise
  - a **put** pays the price difference if the stock goes **down**, or zero otherwise
- Options are popular with investors due to their minimal downside, leverage and hedging potential.

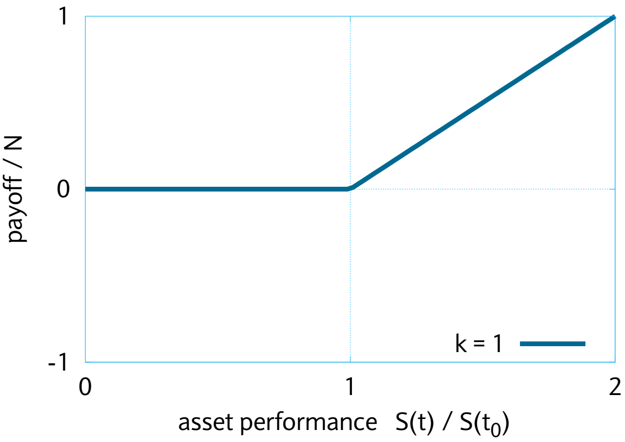
Vanilla Options

$$P_{call} = N\text{max}(S(t_T)/S(t_0) - k, 0) \tag{1}$$

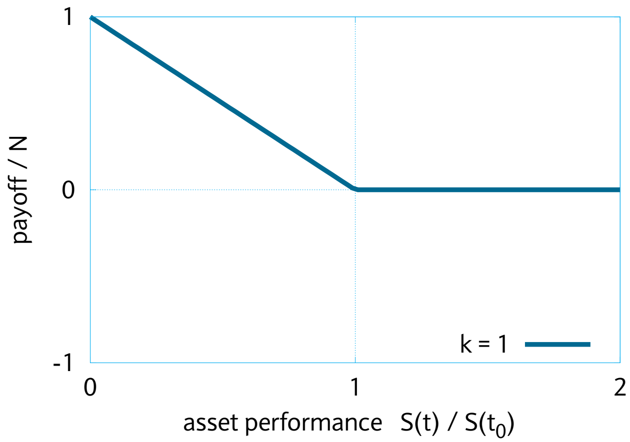
$$P_{put} = N\text{max}(k - S(t_T)/S(t_0), 0) \tag{2}$$

where P is the payoff, N is the notional, k is the strike and $S(t)$ is the price of the underlying at time t.

Long call



Long put



Exotics

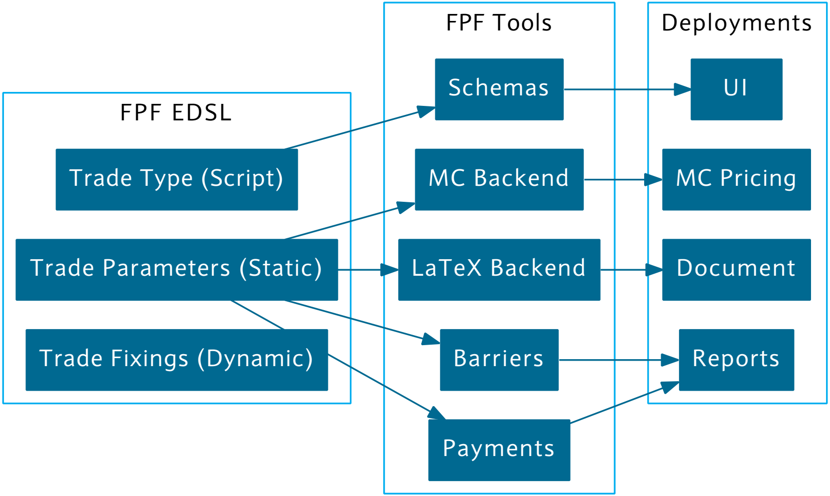
- **Baskets**
  - an option on a portfolio of underlyings
- **Compound options**
  - Options on other options, e.g. a call on a call
- **Time dependent options**
  - Forward start options–option that start at some time in the future
  - Chooser options–buyer or seller may choose when to early redeem
- **Path dependent options**
  - barrier options–payout locked-in when underlying hits trigger
  - lookback options–payout based on highest or lowest price during the lookback period
  - Asian options–payout derived from average value of underlying over a specified window
  - Autocallables–will early redeem if a particular barrier condition is met

Trade Lifecycle

- **Sales** interact with the customers
- **Structurers** create new products, often on customer request
- **Quants** provide mathematical models and formal description of trades (payout functions)
- **Risk management** validate and sign-off the payout functions
- **Traders** derive the final price, manage the trade over its lifetime and analyse upcoming events
- **Payments systems** handle payment events throughout the lifetime of the trade

The Functional Payout Framework

- A standardized representation for describing payoffs
- A common suite of tools for trades which use this representation
  - UI for providing trade parameters
  - mathematical document descriptions
  - pricing and risk management
  - barrier analysis
  - payments and other lifecycle events
- A Haskell EDSL for authoring trade types
  - purely functional and declarative
  - strong static typing
  - produces abstract syntax–allowing multiple interpretations
  - composition of payoffs is just function composition!

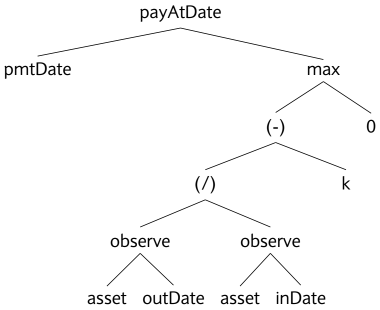

FPF EDSL
Trade Type (Script)
Trade Parameters (Static)
Trade Fixings (Dynamic)
FPF Tools
Schemas
MC Backend
LaTeX Backend
Barriers
Payments
Deployments
UI
MC Pricing
Document
Reports

An FPF payoff contract is represented by a function whose domain is the observed asset values and whose codomain is a set of payments on different dates:

$$\{(\text{Asset}, \text{Date}, \text{Double})\} \rightarrow \{\text{Payment}\} \tag{3}$$

Example: a call option

```
-- TRADETYPE: callDemo_v1
-- TAG: DEV
-- DESC: A long call.
callDemo_v1
  ( name ”Asset”    -> asset
  , name ”Strike”   -> k
  , name ”In date”  -> inDate
  , name ”Out date” -> outDate
  , name ”Pmt date” -> pmtDate
  )
  = payAtDate pmtDate (max 0 (st / s0 - k))
  where
    st = observe asset outDate
    s0 = observe asset inDate
```



Trade parameters (FPF String)

callDemo_v1 (“BARX”, 1-Dec-2013, 1-Dec-2014, 3-Dec-2008)

Trade fixings

[(“BARX”, Close, 1-Dec-2013, 280.1)]

Example: a Cliquet

```
cliquetDemo_v2
  ( name ”Asset”          -> asset
  , name ”Global floor”  -> gf
  , name ”Global cap”    -> gc
  , name ”Local floor”   -> lf
  , name ”Local cap”     -> lc
  , name ”Initial date”  -> inDate
  , name ”Dates”         -> dates
  , name ”Payment date”  -> payDate
  )
  = max gf $ min gc $ sum perfs
  where
    cliquet d d’ = (d’, max lf $ min lc $ perf d d’ asset)
    (_, perfs) = mapAccumL cliquet inDate dates
```

CliquetDemo_v2 Documentation

$$\text{pay}\left(t^{PD}, \min\left(GC, \max\left(GF, \sum_{i=1}^{\text{len}(t^D)} \min\left(LC, \max\left(LF, \frac{S^{TOP}\left({t^D}_i\right)}{S^{TOP}\left(a_{i-1}\right)}\right)\right)\right)\right)\right)$$

where

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

The parameters to this trade type are as follows:

Variable	Description	Type
TOP	Top-level input	Tuple of $(S^{TOP}, GF, GC, LF, LC, t^{ID}, t^D, t^{PD})$
S^{TOP}	Asset	Asset
GF	Global floor	Double
GC	Global cap	Double
LF	Local floor	Double
LC	Local cap	Double
t^{ID}	Initial date	Date
t^D	Dates	List of Date
t^{PD}	Payment date	Date

EDSLs: Deep Embedding

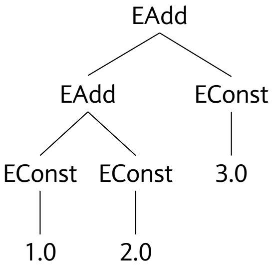
- A deeply embedded DSL yields an abstract-syntax-tree (AST) upon evaluation
- We can then analyse the AST and extract the necessary information

```
data Exp
  = EVar VarId
  | EConst Double
  | EAsset Name
  | EDate Date
  | EObserve Exp Exp
  | EPayAtDate Exp Exp
  | EAdd Exp Exp
  ...
  deriving (Eq, Ord, Show)
```

Overloading Literals

```
instance Num Exp where
  (+) = EAdd
  fromInteger = EConst . fromInteger

instance Fractional Exp where
  fromRational = EConst . fromRational
```

```
λ> 1 + 2 + 3 :: Exp
EAdd (EAdd (EConst 1.0) (EConst 2.0)) (EConst 3.0)
```



Functions

- Function/lambda syntax cannot be overloaded in Haskell;
- but we can reify them:

```
f (x, y) = x + y
```

```
λ> f (EVar ”x”, EVar ”y”)
EAdd (EVar ”x”) (EVar ”y”)
```

Lists

- Lists in FPF have two main uses:
  - contractual data of varying length, e.g. a basket of assets
  - control flow, e.g. stepping forward through a list of observation dates
- FPF has Map, Foldl, Foldr and MapAccumL primitives

```
data Exp = ...
         | EFoldl Fun2 Exp [Exp]

type Fun2 = (VarId, VarId, Exp)
```

```haskell
foldl f a xs = EFoldl (lambdaToFun2 f) a xs

lambdaToFun2 :: (Exp -> Exp -> Exp) -> Fun2
lambdaToFun2 f =
  (EVar 0, EVar 1, f (EVar 0) (EVar 1))
```

Note that we must take care to avoid name capture!

Types

- prove that certain classes of errors do no exist
- offer a form of machine-checked documentation to guide the user

We can use type parameters to constrain the types of terms that can be constructed. For example, using a phantom type:

```
newtype E t = E Exp

payAtDate :: E Date -> E Double -> E Payment
...
```

Datatype Generic Programming

A form of abstraction that allows defining a single function over a class of datatypes.

- generic functions depend only on the structure or *shape* of the datatype
- useful for large complex data-types, where traversal code often dominates
- for recursion schemes, we can capture the pattern as a standalone combinator

Scrap-Your-Boilerplate (SYB)

Generic programming frameworks differ in the mechanism used to access the underlying structure of a datatype.

In our first foray into generic programming, we tried SYB [4], an extremely powerful generics framework, but we were not entirely satisfied:

- performance was significantly worse than non-generic traversal code
- all datatypes needed `Data` and `Typeable` instances
- we lost type safety in some areas, for example traversals accept any datatype with a `Data` instance

Fixed points of Functors

An idea from category theory[3] which gives:

- data-type generic functions
- compositional data

```
-- | the least fixpoint of functor f
newtype Fix f = Fix { unFix :: f (Fix f) }
```

A functor `f` is a data-type of kind `* -> *` together with an `fmap` function.

$$Fix\ f \cong f(f(f(f(f \ldots \text{etc} \quad (4)$$

BARCLAYS

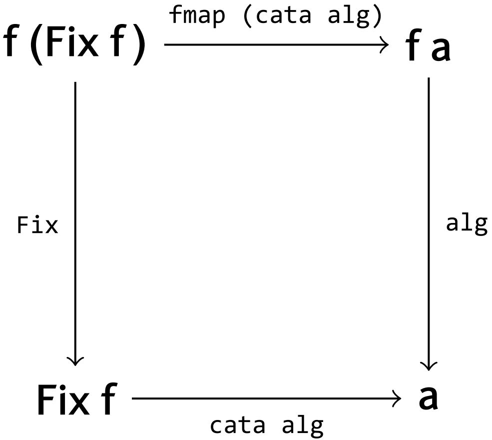
Catamorphisms

A *catamorphism* (cata meaning “downwards”) is a generalisation of the concept of a fold.

- models the fundamental pattern of (internal) *iteration*
- a catamorphism will traverse bottom-up, however top-down or a combination is possible using a function codomain
- category theory shows us how to define it data-type generically for a functor fixed-point

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```

Catamorphism



Example pattern functor

```haskell
data ExpF r                          type Exp = Fix ExpF
  = EVar VarId
  | EConst Double
  | EAsset Name
  | EDate Date
  | EObserve r r
  | EPayAtDate r r
  | EAdd r r
  | EMax r r
  ...
  deriving ( Show, Eq, Ord
           , Functor, Foldable, Traversable
           )
```

Example catamorphisms

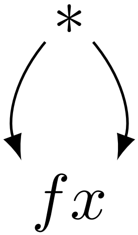
```haskell
-- | collect up all the observation dates
obsDates :: Exp -> Set Date
obsDates = cata alg where
  alg :: ExpF (Set Date) -> Set Date
  alg (EDate i) = S.singleton i
  alg e = fold e

-- | substitute variables using the supplied environment
substitute :: Map VarId (ExpF Exp) -> Exp -> Exp
substitute env = cata alg where
  alg :: ExpF Exp -> Exp
  alg (EVar i) | Just e <- M.lookup i env = Fix e
  alg e = Fix e
```

Recovering Sharing

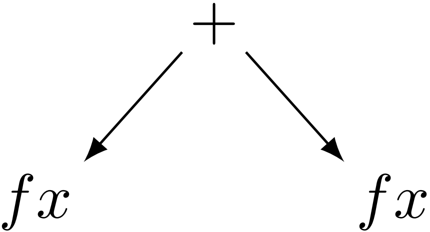
The following Haskell expression:

```haskell
let y = f x in y * y
```

is represented internally as a graph:

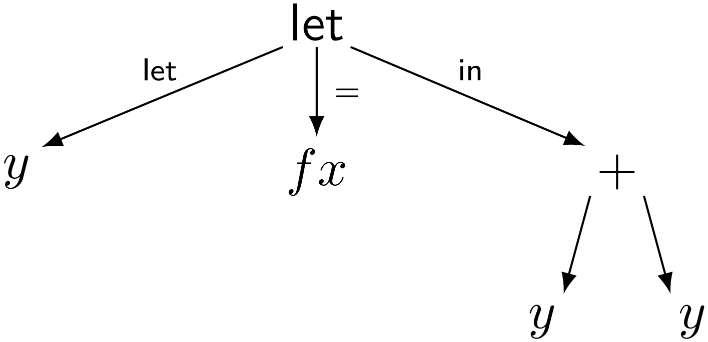


However, when evaluating the expression, we get:



If we were to evaluate this AST, $f\,x$ would be evaluated twice!

Sharing can be captured explicitly in a tree representation by using "`let`" forms:



Two complementary forms of sharing

- **Implicit sharing**–common sub-expression elimination
  - an optimisation
  - non-trivial to preserve evaluation semantics in the presence of side-effects
  - FPF relies upon implicit sharing for compilation of lists
- **Explicit sharing**–sharing explicitly declared by users
  - Naïve use of let-forms in EDSLs leads to code explosion
  - *observable sharing* via GHC's internal unsafe operations can be used to recover the graph structure
  - FPF does not (currently) support explicit sharing, in order to avoid the complexity of working with let-forms or graphs

Stable names

“Stable names” in Haskell are intended for fast $O(1)$ equality and hashing under IO, but can be used to recover explicit sharing in the source code.

For example, using Andy Gill’s `Data.Reify`[2]:

```
f :: Exp -> Exp
f x = let y = x + x in y + y
```

```
λ> reifyGraph $ f (Fix $ EVar ”x”)
let [(1,EAdd 2 2),(2,EAdd 3 3),(3,EVar ”x”)] in 1
```

Hash-consing

- a space optimisation
- at the time of construction, we hold a hash-map of previously constructed expressions and look them up.
  - if a previous instance exists, we return it, tagged with a unique id;
  - otherwise, we add to the hash-map the new expression with a new generated unique id.
- the uniques enable fast $O(1)$ comparisons and hash calculations requiring only a single level of depth.
- unlike pointer equality, the uniques represent structural equality, even if the same expression is constructed with a different constructor invocation

```haskell
-- | Hash-consing for any functor f
data HCF f r = HCF (f r) !Unique

type HC f = Fix (HCF f)
type HCExp = HC ExpF

type HCMap = HashMap (ExpF HCExp) HCExp
type HCM a = State (HCMap, Int) a

runHCM :: HCM a -> a
runHCM m = evalState m (HM.empty, 0)
```

- use mkHC and unHC in place of Fix and unFix respectively

```
mkHC :: ExpF HCExp -> HCM HCExp
mkHC e = do
  v <- lookup e
  case v of
    Just e’ -> return e’
    Nothing -> do
      u <- newUnique
      let e’ = Fix $ HCF e u
      insert e e’
      return e’

unHC :: Functor f => HC f -> f (HC f)
unHC (unFix -> HCF e _) = e
```

```haskell
-- uniques used for fast O(1) equality tests on HCExp
instance Eq (HCF f r) where
  (HCF _ u) == (HCF _ u’) = u == u’

-- uniques used for fast hashing (to first depth level only)
instance Hashable (ExpF HCExp) where
  hashWithSalt s (EConst c)
    = 1 ‘hashWithSalt‘ s ‘hashWithSalt‘ c
  hashWithSalt s (EAdd (Fix (HCF _ u)) (Fix (HCF _ u’)))
    = 2 ‘hashWithSalt‘ s ‘hashWithSalt‘ (u, u’)
  ...
```

- in this example, the separately constructed expressions are represented as one instance, with unique 1.

```
e1 = do
  x <- mkHC $ EVar ”x”
  y <- mkHC $ EVar ”x”
  mkHC $ EAdd x y
```

```
λ> runHCM e1
Fix (HCF (EAdd (Fix (HCF (EVar ”x”) 1))
               (Fix (HCF (EVar ”x”) 1))) 2)
```

- traversals must be monadic, but customised recursion combinators can at least handle the HC annotation unwrapping for us:

```
cataM :: (Monad m, Traversable f) =>
         (f a -> m a) -> HC f -> m a
cataM algM = algM <=< mapM (cataM algM) . unHC

substitute :: M.Map VarId (ExpF HCExp) ->
              HCExp ->
              HCM HCExp
substitute env = cataM alg where
  alg :: ExpF HCExp -> HCM s HCExp
  alg (EVar i) | Just e <- M.lookup i env = mkHC e
  alg e = mkHC e
```

unsafePerformIO

- we may take the view that hash-consing, an optimisation, is not state that we wish to make explicit and that it can be made essentially pure from the outside
- for better or worse, FPF takes the `unsafePerformIO` with `IORef` approach to Hash-consing. This is not to avoid monad traversals, but to avoid sequencing each and every hash-cons (`mkHC`)
- it is not without uglyness–we need a function of type `IO ()` to clear the dictionary

Memoization

- memoization, or caching, lets us trade space for time where necessary
- since we restrict recursion to a library of standard combinators, we can define memoizing variants that can easily be swapped in
- the simplest (pure) memoize function requires some kind of `Enumerable` context

```
memoize :: Enumerable k => (k -> v) -> k -> v
```

A monadic codomain allows us to use e.g. an underlying State monad:

```
memoize :: Memo k v m => (k -> m v) -> k -> m v
memoize f x = lookup x >>= (‘maybe‘ return)
  (f x >>= \r -> insert x r >> return r)

memoFix :: Memo k v m =>
           ((k -> m v) -> k -> m v) -> k -> m v
memoFix f = let mf = memoize (f mf) in mf

class Monad m => Memo k v m | m -> k, m -> v where
  lookup  :: k -> m (Maybe v)
  insert  :: k -> v -> m ()
```

The following runs the memoized computations using a HashMap (`Memo` instance required):

```haskell
type MemoMT k v m a = StateT (HashMap k v) m a
type MemoM k v a = MemoMT k v Identity a

runMemoT :: Monad m => MemoMT k v m a -> m a
runMemoT m = evalStateT m HM.empty

runMemo :: MemoM k v a -> a
runMemo = runIdentity . runMemoT
```

For example, we can use `memoFix` to build a memoizing catamorphism over our Hash-consed types:

```
memoCata :: (Traversable f, Hashable (HC f)) =>
            (f a -> a) -> HC f -> a
memoCata alg x = runMemo $
  memoFix (\rec -> fmap alg . mapM rec . unHC) x

memoCataM :: (Monad m, Traversable f, Hashable (HC f)) =>
             (f a -> m a) -> HC f -> m a
memoCataM algM x = runMemoT $
  memoFix (\rec -> lift . algM <=< mapM rec . unHC) x
```

WARNING: this will result in a slowdown if your AST has no common sub-trees!

The Future of FPF

- FPF Lucid
  - a new front-end standalone DSL
  - more restrictive and easier to use
  - central notion of time
  - control constructs based around schedules
  - Damas-Hindley-Milner type inference with constraints and polymorphic extensible records
- New Monte Carlo backend
  - designed from scratch for massive parallelism
  - GPU capable
- New PDE backend
  - Generic solver

BARCLAYS

References

[1] S. Frankau, D. Spinellis, N. Nassuphis and C. Burgard, “Going functional on exotic trades”, 2009

[2] A. Gill, “Type-Safe Observable Sharing in Haskell”, 2009

[3] E. Meijer et al, “Functional Programming with Bananas , Lenses , Envelopes and Barbed Wire”, 1991.

[4] R. Lammel and S. Peyton Jones, “Scrap your boilerplate with class : extensible generic functions”, 2004.