Describing contracts in Haskell

Roman Cheplyaka
Barclays Capital

May 26, 2011

Describing contracts in Haskell

Our clients are investors. What are their needs?

Describing contracts in Haskell

Investor #1:

- is sure that Microsoft stock will go up in a year
- wants to make money on his prediction
- buys Microsoft stock
- sells it in one year, earning (or losing) the difference in prices

Describing contracts in Haskell

Investor #2:

- is sure that Microsoft stock will go up in a year
- wants to make money on his prediction
- concerned about his possible losses
- $\Rightarrow$ does not buy Microsoft stock

Describing contracts in Haskell

What can we offer?

- if the stock goes up, we pay the difference in prices
- otherwise, no payment happens

Mathematically

$$\mathit{payout} = \max(S_1 - S_0, 0)$$

Describing contracts in Haskell

$$payout = \max\left(S_1 - S_0, 0\right)$$

In Haskell:

```
payout :: Double -> Double -> Double
payout s0 s1 = max (s1 - s0) 0
```

Describing contracts in Haskell

```haskell
payout :: Market -> Double
payout market =
     let s0 = observe market "26-05-2011" "MSFT"
         s1 = observe market "26-05-2012" "MSFT"
     in max (s1 - s0) 0
```

Describing contracts in Haskell

Good:

- Unambiguously specifies the contract
- Allows to calculate the payoff when the contract expires

Bad:

- Can't be executed before the contract expires

Describing contracts in Haskell

Things we want to know about the contract:

- Set of observation dates
- Set of underlying securities (eg. stocks)
- Points of discontinuities of the payoff function
- ...

Describing contracts in Haskell

Solution:

- parse the program;
- analyse abstract syntax tree and extract the necessary information

Describing contracts in Haskell

How to represent abstract syntax tree?

Haskell's answer: **Algebraic Data Types**

- combine unions and structs from C
- resemble Backus-Naur form for the grammar

Describing contracts in Haskell

```haskell
data Expr = EAdd Expr Expr
          | ESub Expr Expr
          | EMax Expr Expr
          | EConst Double
          | EAsset String
          | EDate String
          | EObserve Expr Expr
```

Describing contracts in Haskell

Representation of our contract:

```
EMax
    (ESub
        (EObserve (EDate "26-05-2012") (EAsset "MSFT"))
        (EObserve (EDate "26-05-2011") (EAsset "MSFT"))
    (EConst 0)
```

Describing contracts in Haskell

Extract stored values using **pattern matching**

```
listOfDates :: Expr -> [Date]
listOfDates e =
    case e of
        EDate date -> [date]
        EAsset asset -> []
        EConst x -> []

        EAdd e1 e2 -> listOfDates e1 ++ listOfDates e2
        ESub e1 e2 -> listOfDates e1 ++ listOfDates e2
        EMax e1 e2 -> listOfDates e1 ++ listOfDates e2
        ...
```

Describing contracts in Haskell

Parsing is awkward. Can we avoid it?

Describing contracts in Haskell

Parsing is awkward. Can we avoid it?

```haskell
payout market =
    let s0 = observe market "26-05-2011" "MSFT"
        s1 = observe market "26-05-2012" "MSFT"
    in max (s1 - s0) 0
```

```haskell
EMax
    (ESub
        (EObserve (EDate "26-05-2012") (EAsset "MSFT"))
        (EObserve (EDate "26-05-2011") (EAsset "MSFT"))
    (EConst 0)
```

Describing contracts in Haskell

Parsing is awkward. Can we avoid it?

```
payout market =
     let s0 = observe market "26-05-2011" "MSFT"
         s1 = observe market "26-05-2012" "MSFT"
     in max (s1 - s0) 0
```

```
EMax
    (ESub
        (EObserve (EDate "26-05-2012") (EAsset "MSFT"))
        (EObserve (EDate "26-05-2011") (EAsset "MSFT"))
    (EConst 0)
```

Yes! Redefine the functions to **generate** the syntax tree.

Describing contracts in Haskell

Redefine the functions to **generate** the syntax tree.

```
max e1 e2 = EMax e1 e2
e1 + e2 = EAdd e1 e2
e1 - e2 = ESub e1 e2

observe date asset = EObserve date asset
```

Describing contracts in Haskell

We can even overload numeric and string literals!

```
instance Fractional Expr where
    fromRational x = EConst (fromRational x)
```

Investor #3:

- is concerned about possible fluctuations
- wants to average the observations

Describing contracts in Haskell

```haskell
payout =
     let s0 = observe "26-05-2011" "MSFT"

         s1 = observe "26-05-2012" "MSFT"
         s2 = observe "26-06-2012" "MSFT"
         s3 = observe "26-07-2012" "MSFT"
         avg = (s1 + s2 + s3)/3

     in max (avg - s0) 0
```

Describing contracts in Haskell

```
payout =
    let s0 = observe "26-05-2011" "MSFT"

        s1 = observe "26-05-2012" "MSFT"
        s2 = observe "26-06-2012" "MSFT"
        s3 = observe "26-07-2012" "MSFT"
        avg = (s1 + s2 + s3)/3

    in max (avg - s0) 0
```

Good programmers don't write code like this!

Describing contracts in Haskell

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Describing contracts in Haskell

```haskell
foldl :: (a -> b -> a) -> a -> [b] -> a

sum :: [Double] -> Double
sum list = foldl (+) 0 list
```

Describing contracts in Haskell

```haskell
foldl :: (a -> b -> a) -> a -> [b] -> a

sum :: [Double] -> Double
sum list = foldl (+) 0 list

length :: [Double] -> Double
length list = foldl (\acc x -> acc + 1) 0 list
```

Describing contracts in Haskell

```haskell
foldl :: (a -> b -> a) -> a -> [b] -> a

sum :: [Expr] -> Expr
sum list = foldl (+) 0 list

length :: [Expr] -> Expr
length list = foldl (\acc x -> acc + 1) 0 list
```

Describing contracts in Haskell

```
payout =
     let dates = ["26-05-2012",
                  "26-06-2012",
                  "26-07-2012"]
         avg = sum dates / length dates
     in max (avg - 12.0) 0
```

Describing contracts in Haskell

Task: print a mathematical formula that describes the contract

Large ASTs lead to large formulas

Describing contracts in Haskell

Task: print a mathematical formula that describes the contract

Large ASTs lead to large formulas

Solution: make foldl a part of our language!

Describing contracts in Haskell

```haskell
data Expr = ...
          | EFoldl Function2 Expr [Expr]
```

Describing contracts in Haskell

```haskell
data Expr = ...
          | EVar VarId
          | EFoldl Function2 Expr [Expr]

type Function2 = (VarId, VarId, Expr)

type VarId = Int
```

Describing contracts in Haskell

```
foldl f a xs = EFoldl (lambdaToFunction2 f) a xs

lambdaToFunction2 :: (Expr -> Expr -> Expr) -> Function2
lambdaToFunction2 f = ?
```

Describing contracts in Haskell

```
foldl f a xs = EFoldl (lambdaToFunction2 f) a xs

lambdaToFunction2 :: (Expr -> Expr -> Expr) -> Function2
lambdaToFunction2 f =
    (EVar 0, EVar 1, f (EVar 0) (EVar 1))
```

Describing contracts in Haskell

```
foldl f a xs = EFoldl (lambdaToFunction2 f) a xs

lambdaToFunction2 :: (Expr -> Expr -> Expr) -> Function2
lambdaToFunction2 f =
    (EVar 0, EVar 1, f (EVar 0) (EVar 1))
```

(plus extra care to avoid free variable capture)

Describing contracts in Haskell

```
observe (EAsset asset) (EDate date)
```

Can you spot the error?

Describing contracts in Haskell

```
observe (EAsset asset) (EDate date)
```

Can you spot the error?

Correct form:

```
observe (EDate date) (EAsset asset)
```

Can the compiler catch this?

Describing contracts in Haskell

```haskell
newtype Date = Date Expr
newtype Asset = Asset Expr
newtype Number = Number Expr

observe :: Date -> Asset -> Number
observe (Date date) (Asset asset) =
    Number (EObserve date asset)
```

Describing contracts in Haskell

FPF = Functional Payout Framework

Language + Set of tools (backends)

- Generate mathematical formulas
- Extract sets of dates and assets
- Analyse for discontinuities
- Generate C code for Monte-Carlo simulation
- ... and more

Frankau et al. "Going functional on exotic trades"

Describing contracts in Haskell

Using Haskell for a domain-specific language:

- higher-order functions
- no need in parsing
- strong static type system
- type inference
- rich overloading

All for free!

Describing contracts in Haskell

Why work at Barclays Capital?

- real-world usage of functional programming
- work among smart people
- solve interesting problems
- get immediate feedback on your work

Send your CV to **Roman.Cheplyaka@BarclaysCapital.com**