Making EDSLs fly

Nottingham 2013-Feb-25

Lennart Augustsson
Standard Chartered Bank
lennart@augustsson.net
Plan

- A simple EDSL
- Shallow embedding
- Deep embedding
- LLVM code generation
A simple EDSL

Valuation of exotic equity trading, taken from the paper *Going functional on exotic trades*, by Frankau, Nassuphis, and Burgard from Barclay Capital.

```haskell
data Date    -- type of dates
data Asset   -- type of assets
data List a  -- lists

(+),(-),(*),(/) :: EDouble -> EDouble -> EDouble
min, max        :: EDouble -> EDouble -> EDouble
abs             :: EDouble -> EDouble
(<)             :: EDouble -> EDouble -> EBool
observe         :: Asset -> Date -> EDouble
cond            :: EBool -> a -> a -> a
foldl1          :: (a->a->a) -> List a -> a
map             :: (a->b) -> List a -> List b
```
Some examples

- Best performing asset.

```haskell
bestOf :: List Asset -> Date -> Date -> EDouble
bestOf assets startDate endDate =
    foldl1 max (map (perf startDate endDate) assets)

perf :: Date -> Date -> Asset -> EDouble
perf t1 t2 asset =
    (observe asset t2) / (observe asset t1) - 1
```
Some examples

☐ cliquet.

cliquet ::
  (Asset, EDouble, EDouble, EDate, List Date) -> EDouble
cliquest (asset, floor, cap, initDate, dates) =
  max floor $ min cap val
where cliquetPerf (prevDate, prevSum) currDate =
  (currDate, prevSum + currPerf)
where currPerf = perf prevDate currDate asset
(_, val) = foldl cliquetPerf (initDate, 0) dates
Shallow embedding

- Shallow embedding means that each of the EDSL functions is implemented directly by a Haskell function that does the job.

- Advantages: easy, reasonably efficient

- Disadvantage: less flexible
Date and Asset

- The date type is simply days from some start.
  data Date = Date { unDate :: Word32 }

- The asset type is used to get the value of an asset at a specific date. Let’s make it a list for simplicity.
  data Asset = Asset [(Date, Double)]
Shallow embedding

- The types can simply be the Haskell types, and most operations are existing Haskell functions.

```haskell
type EDouble = Double
type EBool = Bool
type List a = [a]

-- +, -, *, /, min, max, abs, <, foldl1, map
-- from Haskell
observe :: Asset -> Date -> EDouble
observe (Asset dvs) d = lookupObs d dvs

cond :: EBool -> a -> a -> a
cond c t e = if c then t else e

lookupObs :: Date -> [(Date, Double)] -> Double
lookupObs d dvs = fromMaybe 0 $ lookup d dvs
```
Shallow embedding

☐ That's it! We're done.

☐ Boring

☐ Also, given this representation there is no way to do anything but compute a Double.

☐ Other things we might want
  • getUsedAssets :: EDouble -> [Asset]
  • plot :: EDouble -> PDF
Deep embedding

- Deep embedding means that each of the EDSL functions builds a syntactic representation.

- Advantages: flexible, can be very efficient

- Disadvantage: more work, can be very inefficient

- Flexibility means we can do other things than execute the EDSL, e.g., pretty print, analyze, compile, ...
Deep embedding

- First, “untyped” expressions (could be typed using, e.g., GADTs):

```haskell
data Expr
  -- Functions
  -- Constants
  | EDouble Double | EBool Bool | EAsset Asset | EDate Date
```

- This type will never be exposed to users.
Deep embedding

Instead, a phantom typed version will be exposed.

```haskell
data E a = E Expr

(+) :: E Double -> E Double -> E Double
(<) :: E Double -> E Double -> E Double
observe :: E Asset -> E Date -> E Double
cond :: E Bool -> a -> a -> a

type EDouble = E Double
type EBool = E Bool
```
Deep embedding

- If we use `Expr` directly we would have a dynamically typed EDSL.

- By using the `E` type we get static type checking by embedding the EDSL type system in Haskell's type system.
Deep embedding

Many functions can use overloading.

```haskell
instance Num EDouble where
  (+) = binOp Add
  (-) = binOp Sub
  (*) = binOp Mul
  abs x = cond (x < 0) (-x) x
  fromInteger = E . EDouble . fromInteger

instance Fractional EDouble where
  (/) = binOp Div
  fromRational = E . EDouble . fromRational

binOp :: (Expr->Expr->Expr) -> E a -> E b -> E c
binOp op (E x) (E y) = E (op x y)
```
Deep embedding

And some regular functions.

```haskell
import Prelude hiding ((<))
...

observe :: E Asset -> E Date -> EDouble
observe = binOp Observe

infix 4 <
(<) :: E Double -> E Double -> E Bool
(<) = binOp Less

cond :: E Bool -> E a -> E a -> E a
cond (E c) (E t) (E e) = E (Cond c t e)
```
Deep embedding

Converting to and from the embedding.

```haskell
class Value a where
  lift :: a -> E a
  down :: E a -> a

instance Value Double where
  lift = E . EDouble
  down (E (EDouble x)) = x

instance Value Date where
  lift = E . EDate
  down (E (EDate x)) = x

instance Value Bool where
  lift = E . EBool
  down (E (EBool x)) = x

instance Value Asset where
  lift = E . EAsset
  down (E (EAsset x)) = x
```
Deep embedding

- For example

```ghci
ghci> 1 + 2 :: EDouble
E (Add (EDouble 1.0) (EDouble 2.0))

ghci> cond (1 < 2) (abs 1.1) (2 * 3.2) :: EDouble
E (Cond (Less (EDouble 1) (EDouble 2)) (Abs (EDouble 1.1)) (Mul (EDouble 2.0) (EDouble 3.2)))

ghci> 1 + (1 < 2) :: EDouble

<interactive>:0:6:
  Couldn't match expected type `Double' with actual type `Bool'
  Expected type: EDouble
    Actual type: E Bool
  In the second argument of `(+)`, namely `(1 < 2)'
  In the expression: 1 + (1 < 2) :: EDouble
```
Neritic (between shallow and deep) embedding

- What about the list type?

- Deep
  
  ```
  type List a = E [a]
  ```

- Shallow
  
  ```
  type List a = [a]
  ```
Neritic embedding

- An example, expanding the list type.
- It's a matter of staging, i.e., when we want the list to exist.

- The list exists at “compile time”
  
  \[
  \text{bestOf} \colon [E \text{ Asset}] \to E \text{ Date} \to E \text{ Date} \to \text{EDouble}
  \]

- The list exists at “run time”

  \[
  \text{bestOf} \colon E [\text{Asset}] \to E \text{ Date} \to E \text{ Date} \to \text{EDouble}
  \]
Deep embedding

Expressions.

data Expr
  -- Functions
  = Add Expr Expr | Sub Expr Expr
  | Mul Expr Expr | Div Expr Expr
  | Log Expr | Exp Expr
  | Less Expr Expr
  | Cond Expr Expr Expr
  | Observe Expr Expr
  | EFoldl1 (Expr->Expr->Expr) Expr
  | EMap (Expr->Expr) Expr
  -- Constants
  | EDouble Double | EBool Bool
  | EAsset Asset | EDate Date
  | EList [Expr]
An Expr evaluator

eval :: Expr -> Expr

eval (Add e1 e2) = case (eval e1, eval e2) of
  (EDouble x1, EDouble x2) -> EDouble (x1 + x2)

eval (Sub e1 e2) = case (eval e1, eval e2) of
  (EDouble x1, EDouble x2) -> EDouble (x1 - x2)

 eval (Mul e1 e2) = case (eval e1, eval e2) of
  (EDouble x1, EDouble x2) -> EDouble (x1 * x2)

eval (Div e1 e2) = case (eval e1, eval e2) of
  (EDouble x1, EDouble x2) -> EDouble (x1 / x2)

 eval (Less e1 e2) = case (eval e1, eval e2) of
  (EDouble x1, EDouble x2) -> EBool (x1 < x2)

 eval (Abs e) = case eval e of EDouble x -> EDouble (abs x)

 eval (Cond c t e) = case eval c of
  EBool b -> if b then eval t else eval e

 eval (Observe e1 e2) =
  case (eval e1, eval e2) of
    (EAsset (Asset dvs), EDate d) -> EDouble (lookupObs d dvs)

 eval e = e -- constants
An E evaluator

- Note that the `eval` function can fail in many places.

- It never fails on a well-typed `Expr`.

- We only expose functions that create well-typed `Expr`, so evaluation will not fail.

```haskell
evalE :: (Value a) => E a -> a
evalE (E e) = down (E (eval e))
```
A test

☐ A simple test

\[d_1, d_2 :: \text{Date}\]
\[d_1 = \text{Date 100}\]
\[d_2 = \text{Date 200}\]

\[a_1, a_2 :: \text{Asset}\]
\[a_1 = \text{Asset } [(d_1, 5), (d_2, 5)]\]
\[a_2 = \text{Asset } [(d_1, 4), (d_2, 6)]\]

\[t :: \text{Double}\]
\[t = \text{evalE } \$ \text{ bestOf [lift } a_1, \text{ lift } a_2\text{]} \text{ (lift } d_1\text{)} \text{ (lift } d_2\text{)}\]

☐ Testing, testing

\[\text{ghci}\gt t\]
\[0.5\]
What is LLVM?

- Low Level Virtual Machine
- Assembly code for a VM
- Batch and JIT for many architectures
- x86, x86_64, ARM, SPARC, ...
- Accessible from many languages
  - C++, C, Haskell, OCaml
Using the LLVMJIT:

- Build instruction sequence
- Call JIT
- Returns a pointer to the code

LLVM normally uses C calling conventions.
(There is GHC backend for LLVM.)
A simple LLVM example

A simple test of LLVM: \( fcn \ x \ y = (x+x) \ast y \)

```haskell
import LLVM.Core
import LLVM.ExecutionEngine

-- \( \ x \ y \rightarrow (x+x) \ast y \)
mkFcn ::
    CodeGenModule (Function (Double -> Double -> IO Double))
mkFcn = createFunction InternalLinkage $
    \ x \ y -> do  x2 <-> add x x
                 tmp <-> mul x2 y
                ret tmp

main :: IO ()
main = do
    initializeNativeTarget
    fcnIO <-> simpleFunction mkFcn
    let fcn :: Double -> Double -> Double
        fcn = unsafePurify fcnIO
    print $ fcn 2 3
```
A simple LLVM example

- A simple test of LLVM: \( fcn \ x \ y = (x+x) \times y \)

- The x86_64 assembly code (register calling convention)

```assembly
__fun1:
  addsd %xmm0, %xmm0
  mulsd %xmm1, %xmm0
  ret
```
Efficient EDSLs

- Idea: Instead of interpreting the code we will translate the EDSL code into code in some other language. The latter code will subsequently be run.

- Example: translate to C, run later.

- Example: translate to LLVM, JIT, run

- Two-level language, i.e., two execution times. Cf. macros, C++ templates.
Code generation for Expr

- We will translate (type correct) Expr to LLVM machine code.

- The translation will be type driven, i.e., one code generation function for each type:

  genDouble :: Expr \rightarrow \text{Gen Double}
  genBool   :: Expr \rightarrow \text{Gen Bool}
  genDate   :: Expr \rightarrow \text{Gen GDate}
Code generation Double

Double is pretty easy, except for Cond and Observe.

gendouble :: Expr -> Gen Double
gendouble (Add e1 e2) = genOpDD fadd e1 e2
gendouble (Sub e1 e2) = genOpDD fsub e1 e2
gendouble (Mul e1 e2) = genOpDD fmul e1 e2
gendouble (Div e1 e2) = genOpDD fdiv e1 e2
gendouble (Cond e1 e2 e3) =
    genCond (genBool e1) (gendouble e2) (gendouble e3)
gendouble r (Observe e1 e2) = do
    x1 <- genAsset r e1
    x2 <- genDate r e2
    return $ valueOf d

gendouble r (EDouble d) = do
    return $ valueOf d
Helper function genOpDD

- Generate code for an arithmetic operation.

```haskell
genOpDD :: (Value Double -> Value Double -> Gen a) -> Expr -> Expr -> Gen a
genOpDD op e1 e2 = do
  x1 <- genDouble e1
  x2 <- genDouble e2
  op x1 x2
```

(The type is scarier than the function.)
Conditionals

- Conditionals are a bit difficult in LLVM, because it uses SSA form. Basically, you need three new basic blocks.

```haskell
genCond :: (IsFirstClass a) =>
    Gen Bool -> Gen a -> Gen a
    genCond g1 g2 g3 = do
        tb <- newBasicBlock
        eb <- newBasicBlock
        jb <- newBasicBlock
        x1 <- g1
        condBr x1 tb eb
        defineBasicBlock tb
        x2 <- g2
        tb' <- getCurrentBasicBlock
        br jb
        defineBasicBlock eb
        x3 <- g3
        eb' <- getCurrentBasicBlock
        br jb
        defineBasicBlock jb
        phi [(x2, tb'), (x3, eb')]
```
Code generation Bool

- `Bool is also easy.`

```
genBool :: Expr -> Gen Bool
genBool (Less e1 e2) = genOpDD (fcmp FPOLT) e1 e2
genBool (Cond e1 e2 e3) =
    genCond (genBool e1) (genBool e2) (genBool e3)
genBool (EBool b) = do
    return $ valueOf b
```
Date is even easier easy.

Use `Word32` for the LLVM Date type.

```haskell
type GDate = Word32

genDate :: Expr -> Gen GDate
genDate (Cond e1 e2 e3) =
    genCond (genBool e1) (genDate e2) (genDate e3)
genDate (EDate d) = do
    return $ valueOf $ unDate d
```
Code generation Asset

What about Asset?

type GAsset = ???

genAsset :: Expr -> Gen GAsset
genAsset (Cond e1 e2 e3) =
    genCond (genBool e1) (genDate e2) (genDate e3)
genAsset (EAsset d) = do
    ???

The Asset type again

☐ This is what we have:

```haskell
data Asset = Asset [(Date, Double)]
```

☐ We need to look up a run-time date and return the corresponding value.
Filling the `???

- First, let's modify `genAsset`

  ```haskell
  genAsset :: Asset -> Value GDate -> Gen Double
  ...
  genAsset (EAsset (Asset dvs)) d = genLookup d dvs
      where genLookup d [] = return $ valueOf 0
          genLookup d ((Date d', v) : dvs) =
              genCond (cmp CmpEQ d (valueOf d'))
              (return $ valueOf v)
              (genLookup d dvs)
  ```

- And, `genDouble`

  ```haskell
  genDouble :: Expr -> Gen Double
  ...
  genDouble (Observe e1 e2) = do
      x2 <- genDate e2
      genAsset e1 x2
  ```
The big picture

- We want to evaluate for bestOf:
  
  \[ \text{bestOf :: [E Asset] -> E Date -> E Date -> E Double} \]

- The first argument is “static”, so what we really want code for is:
  
  \[ E \text{ Date -> E Date -> E Double} \]

- So we need this:

  \[ \text{genDateDateDouble ::} \]
  
  \[ (E \text{ Date -> E Date -> E Double}) -> \]
  
  \[ \text{CodeGenModule (Function (GDate -> GDate -> IO Double))} \]
Embedded functions

- How do we generate code for a function?
  \[ \text{E Date} \rightarrow \text{E Date} \rightarrow \text{E Double} \]

- We know how to handle \text{E Double}, since that just \text{Expr}, but not functions.

- The standard trick is to augment the abstract syntax with variables. So

```haskell
data Expr = ...
  | Var String
```
Embedded functions

☐ This means that all code generation functions need a new case. How do we handle variables? We need an environment to look them up.

```haskell
genDouble :: GenEnv -> Expr -> Gen Double
...
  genDouble r (Var x) = do
    return $ glookup x r
```

☐ The other functions are extended the same way.
Now we can deal with code generation for functions by inventing some variables, and building an environment.

genDateDateDouble ::
    (E Date -> E Date -> E Double) ->
    CodeGenModule (Function (GDate -> GDate -> IO Double))
genDateDateDouble fcn =
    createFunction InternalLinkage $ \
        \ x y -> do
        let E efcn = fcn (E (Var "x")) (E (Var "y"))
            env = [ ("x", x), ("y", y) ]
        r <- genDouble env efcn
    ret r
Putting it all together

☐ A test.

\[ \text{a1, a2, a3 :: Asset} \]
\[ \text{d1, d2 :: Date} \]

\[
\text{main = do}
\]
\[
\text{initializeNativeTarget}
\]
\[
\text{let bestOfA = bestOf [a1, a2, a3]}
\]
\[
\text{bestOfAIO <-}
\]
\[
\text{simpleFunction $ genDateDateDouble bestOfA}
\]
\[
\text{let bestOfA' :: GDate -> GDate -> Double}
\]
\[
\text{bestOfA' = unsafePurify bestOfAIO}
\]
\[
\text{print $ bestOfA' (unDate d1) (unDate d2)}
\]
A simple benchmark

- One benchmark (top level loop in C for LLVM).
- Call bestOf 10,000,000 times, with a list of 3 assets.

- Shallow embedding: 3.7 s
- Deep embedding: 15.6 s
- LLVM: 0.3 s

Monday, 25 February 13
Conclusions

- Haskell is very good to make typed EDSLs
- Shallow embedding is easier
- Deep embedding is much more flexible
- Deep embedding can be made more efficient
Questions
Let’s go crazy with EDSLs
In the olden days
Programming in 1974

10 LET I  = 1
20 LET S  = 0
30 LET S  = S + 1/I
40 LET I  = I + 1
50 IF I <> 1000000000 THEN 30
60 PRINT "ALMOST INFINITY IS ";S
70 END
Programming in 1974

10 LET I = 1
20 LET S = 0
30 LET S = S + 1/I
40 LET I = I + 1
50 IF I <> 1000000000 THEN 30
60 PRINT "ALMOST INFINITY IS ";S
70 END
import Language.BASIC

main :: IO ()
main = runBASIC $ do

10 LET I := 1
20 LET S := 0
30 LET S := S + 1/I
40 LET I := I + 1
50 IF I <> 1000000000 THEN 30
60 PRINT "ALMOST INFINITY IS ";S
70 END
Programming in 1974

```
pxor  %xmm0, %xmm0
movl  $1, %eax
movsd LCPI0_0(%rip), %xmm1
LBB0_1:
cvtsi2sd %eax, %xmm2
movaps  %xmm1, %xmm3
divsd %xmm2, %xmm3
incl  %eax
cmpl  $1000000000, %eax
addsd %xmm3, %xmm0
jne   LBB0_1
```