Making EDSLs fly

TechMesh London 2012-Dec-05

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.

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

(+),(-),(*),(/) :: EDouble -> EDouble -> EDouble
min, max        :: EDouble -> EDouble -> EDouble
abs, log, exp   :: 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.

```
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
cliquet (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.

```
newtype 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.

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

-- +, -, *, /, min, max, <, 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

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):

```
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
  -- 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.

```
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

- Many functions can use overloading.

```
instance Num EDouble where
   (+) = binOp Add
   (-) = binOp Sub
   (*) = binOp Mul
   abs x = cond (x < 0) (-x) x
   fromInteger = E . EDouble . fromInteger
   signum x = cond (x < 0) (-1) (cond (0 < x) 1 0)


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

unOp :: (Expr -> Expr) -> E a -> E
unOp op (E x) = E (op x)
```

Deep embedding

- Many functions can use overloading.

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

instance Floating EDouble where
    exp = unOp Exp
    log = unOp Log
```

Deep embedding

- And some regular functions.

```
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.

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

ghci> cond (1 < 2) (exp 1.1) (2 * 3.2) :: EDouble
E (Cond (Less (EDouble 1) (EDouble 2)) (Exp (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"

```
bestOf :: [E Asset] -> E Date -> E Date -> EDouble
```

- The list exists at "run time"

```
bestOf :: E [Asset] -> E Date -> E Date -> 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 (Log e) = case eval e of EDouble x -> EDouble (log x)
eval (Exp e) = case eval e of EDouble x -> EDouble (exp 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.

```
evalE :: (Value a) => E a -> a
evalE (E e) = down $ E $ eval e
```

A test

- A simple test

```
d1, d2 :: Date
d1 = Date 100
d2 = Date 200

a1, a2 :: Asset
a1 = Asset [(d1, 5), (d2, 5)]
a2 = Asset [(d1, 4), (d2, 6)]

t :: Double
t = evalE $ bestOf [lift a1, lift a2] (lift d1) (lift d2)
```

- Testing, testing

```
ghci> t
0.5
```

LLVM

- 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

LLVM, cont

- Using the LLVM JIT:
  - 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)*y`

```
import LLVM.Core
import LLVM.ExecutionEngine

-- \ x y -> (x+x) * 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
```

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 -> Gen Double
genBool   :: Expr -> Gen Bool
genDate   :: Expr -> 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
    ???
genDouble r (EDouble d) = do
    return $ valueOf d
```

Helper function genOpDD

- Generate code for an arithmetic operation.

```
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 big difficult in LLVM, because it uses SSA form. Basically, you need three new basic blocks.

```
genCond :: (IsFirstClass a) =>
           Gen Bool -> Gen a -> 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
```

Code generation Date

- Date is even easier easy.
- Use Word32 for the LLVM Date type.

```
type GDate = Word32

genDate :: Expr -> 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:

```
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

```
genAsset :: Expr -> 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

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

The big picture

- We want to evaluate for bestOf:

```
bestOf :: [E Asset] -> E Date -> E Date -> E Double
```

- The first argument is "static", so what we really want code for is:

```
E Date -> E Date -> E Double
```

- So we need this:

```
genDateDateDouble ::
    (E Date -> E Date -> E Double) ->
    CodeGenModule (Function (GDate -> GDate -> IO Double))
```

Embedded functions

- How do we generate code for a function?

```
E Date -> E Date -> E Double
```

- We know how to handle **`E Double`**, since that just **`Expr`**, but not functions.

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

```
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.

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

- The other functions are extended the same way.

Embedded functions

- 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.

```
a1, a2, a3 :: Asset
d1, d2 :: Date

main = do
    initializeNativeTarget

    let bestOfA = bestOf [a1, a2, a3]
    bestOfAIO <-
        simpleFunction $ genDateDateDouble bestOfA
    let bestOfA' :: Word32 -> Word32 -> Double
        bestOfA' = unsafePurify bestOfAIO
    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

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