Domain Modeling with Haskell

Data Structures

Oskar Wickström

March 2018

Domain Modeling

- Clear and unambigous naming
- Reify the domain we are working with in our code
- Separate bounded contexts

Haskell

- All the flexibility we need
    - Sum types
    - Product types
    - Type classes
- Powerful type system and compiler
- Mature language and ecosystem

Modeling in Haskell

- To a large extent, our domain model should be reified with data types
- Separation with data types:
    - Separate bounded contexts using different data types
    - Define explicit interfaces and translations between them
- Structure computation as data structures
- Use modules for cohesive units
    - Domain logic, repositories/data access, rendering
    - Capture interfaces and responsibilities using type classes
    - Avoid the `Types.hs` trap
- Leverage all the good abstractions in Haskell

Type-Driven Development

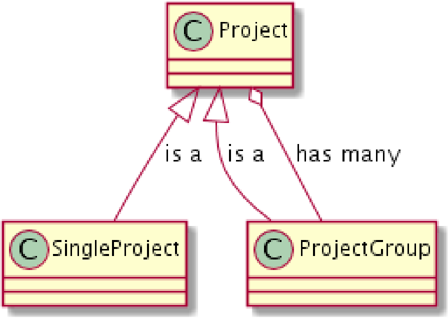
- "Type, define, refine"
- When modifying existing code:
    - Change to data types to model the new behavior
    - Fix all the type errors
    - Test, and possibly refine your model
- Great for changing business requirements
- Focus testing on behaviour

Example: Project Management System

Project Management

- Not terribly exciting, but relatable
- We’ll explore:
    - Data types
    - Some very useful abstractions

Projects



Project

```haskell
data Project
  = SingleProject ProjectId
                  Text
  | ProjectGroup Text
                 [Project]
  deriving (Show, Eq)
```

Budget

```haskell
data Budget = Budget
  { budgetIncome      :: Money
  , budgetExpenditure :: Money
  } deriving (Show, Eq)
```

Transaction

```haskell
data Transaction
  = Sale Money
  | Purchase Money
  deriving (Eq, Show)
```

Reporting

```haskell
data Report = Report
  { budgetProfit :: Money
  , netProfit    :: Money
  , difference   :: Money
  } deriving (Show, Eq)
```

Calculating a Report

```haskell
calculateReport :: Budget -> [Transaction] -> Report
calculateReport budget transactions = Report
  { budgetProfit = budgetProfit'
  , netProfit    = netProfit'
  , difference   = netProfit' - budgetProfit'
  }
 where
  budgetProfit' = budgetIncome budget - budgetExpenditure budget
  netProfit'    = getSum (foldMap asProfit transactions)
  asProfit (Sale     m) = pure m
  asProfit (Purchase m) = pure (negate m)
```

Aggregating Reports

```haskell
calculateProjectReport :: Project -> IO Report
calculateProjectReport project =
  case project of
    SingleProject p _ ->
      calculateReport
      <$> DB.getBudget p
      <*> DB.getTransactions p
    ProjectGroup _ projects ->
      foldMap calculateProjectReport projects
```

Printing Projects

```haskell
asTree :: Project -> Tree String
asTree project =
  case project of
    SingleProject (ProjectId p) name ->
      Node (printf "%s (%d)" name p) []
    ProjectGroup name projects ->
      Node (Text.unpack name) (map asTree projects)

prettyProject :: Project -> String
prettyProject = drawTree . asTree
```

Printing Projects in the REPL

```
*Demo> putStrLn (prettyProject someProject)
Sweden
|
+- Stockholm (1)
|
+- Gothenburg (2)
|
`- Malmö
   |
   +- Malmö City (3)
   |
   `- Limhamn (4)
```

Printing Reports

```haskell
prettyReport :: Report -> String
prettyReport r =
  printf
    "Budget: %.2f, Net: %.2f, difference: %+.2f"
    (unMoney (budgetProfit r))
    (unMoney (netProfit r))
    (unMoney (difference r))
```

Printing Reports in the REPL

```
*Demo> r <- calculateProjectReport someProject
*Demo> putStrLn (prettyReport r)
Budget: -14904.17, Net: 458.03, difference: +15362.20
```

What we’ve used so far

- Basic Haskell data types
- Explicit recursion
- Monoid
- Functor
- Foldable

New Requirements!

A Tree Of Reports

- One big report for the entire project is not enough
- The customer needs them for all individual projects

Parameterizing Project

```haskell
data Project a
  = SingleProject Text
                  a
  | ProjectGroup Text
                 [Project a]
  deriving (Show, Eq, Functor, Foldable, Traversable)
```

Calculating Project Reports with Traversable

```haskell
calculateProjectReports :: Project ProjectId -> IO (Project Report)
calculateProjectReports =
  traverse $ \p ->
    calculateReport
      <$> DB.getBudget p
      <*> DB.getTransactions p
```

Accumulating Reports with Foldable

```
accumulateProjectReport :: Project Report -> Report
accumulateProjectReport = fold
```

Adapting the Pretty Printing

```haskell
asTree :: (a -> String) -> Project a -> Tree String
asTree prettyValue project =
  case project of
    SingleProject name x ->
      Node (printf "%s: %s" name (prettyValue x)) []
    ProjectGroup name projects ->
      Node (Text.unpack name) (map (asTree prettyValue) projects)

prettyProject :: (a -> String) -> Project a -> String
prettyProject prettyValue = drawTree . asTree prettyValue
```

Pretty Printing the Reports

```
*Demo> pr <- calculateProjectReports someProject
*Demo> putStrLn (prettyProject prettyReport pr)
Sweden
|
+- Stockholm: Budget: -2259.99, Net: 391.23, difference: +2651.22
|
+- Gothenburg: Budget: -3204.79, Net: -228.31, difference: +2976.48
|
`- Malmö
   |
   +- Malmö City: Budget: -6958.82, Net: 2811.88, difference: +9770.70
   |
   `- Limhamn: Budget: 5856.93, Net: 1941.43, difference: -3915.50
```

Pretty Printing the Reports (cont.)

```
*Demo> putStrLn (prettyReport (accumulateProjectReport pr))
Budget: -6566.67, Net: 4916.23, difference: +11482.90
```

What we’ve added to our toolbox

- Parameterized Data Type
- Traversable

“No, that’s not what we want.”

Actual Requirements

- The customer wants reporting on *all* levels:
    - project groups
    - single projects
- We need to change our model again

Parameterizing Project Even More

```haskell
data Project g a
  = SingleProject Text
          a
  | ProjectGroup Text
                 g
                 [Project g a]
  deriving (Show, Eq, Functor, Foldable, Traversable)
```

Calculating Project Reports with Traversable

```haskell
calculateProjectReports
  :: Project g ProjectId
  -> IO (Project Report Report)
calculateProjectReports project =
  fst <$> runWriterT (calc project)
  where

    -- ...
```

Calculating Project Reports with Traversable (cont.)

```
calc (SingleProject name p) = do
  report <- liftIO $
    calculateReport
      <$> DB.getBudget p
      <*> DB.getTransactions p
  tell report
  pure (SingleProject name report)
```

Calculating Project Reports with Traversable (cont.)

```
calc (ProjectGroup name _ projects) = do
  (projects', report) <- listen (mapM calc projects)
  pure (ProjectGroup name report projects')
```

Adapting the Pretty Printing

```
asTree
  :: (g -> String)
  -> (a -> String)
  -> Project g a
  -> Tree String

prettyProject
  :: (g -> String)
  -> (a -> String)
  -> Project g a
  -> String
```

Pretty Printing the Reports

```
*Demo> pr <- calculateProjectReports someProject
*Demo> putStrLn (prettyProject prettyReport prettyReport pr)
Sweden: Budget: -9278.10, Net: +4651.81, difference: +13929.91
|
+- Stockholm: Budget: -3313.83, Net: -805.37, difference: +2508.46
|
+- Gothenburg: Budget: -422.48, Net: +1479.00, difference: +1901.48
|
`- Malmö: Budget: -5541.79, Net: +3978.18, difference: +9519.97
   |
   +- Malmö City: Budget: -4069.45, Net: +2185.02, difference: +6254.47
   |
   `- Limhamn: Budget: -1472.34, Net: +1793.16, difference: +3265.50
```

Even More Learnings

- Explicit recursion might still be necessary
- The Writer monad transformer
- There are many ways to leverage Monoid
- Computation as a data structure

Is there more?

Explicit Recursion

- Explicit recursion can, with large data types, be error-prone
- Current `Project` type has a hidden coupling to the reporting module
  - The `g` and `a` parameters are only there for reporting
- Can we decouple `Project` from that concern?

Factoring Out Recursion

```haskell
data ProjectF f
  = SingleProject ProjectId
                  Text
  | ProjectGroup Text
                 [f]
  deriving (Show, Eq, Functor, Foldable, Traversable)

type Project = Mu ProjectF
```

Fix

```
*Project> import Data.Generics.Fixplate.Base
*Project Data.Generics.Fixplate.Base> :t Fix
Fix :: f (Mu f) -> Mu f
```

Project Constructors

```haskell
singleProject :: ProjectId -> Text -> Project
singleProject p = Fix . SingleProject p

projectGroup :: Text -> [Project] -> Project
projectGroup name = Fix . ProjectGroup name
```

Decorating with Attr

```
type ProjectReport = Attr ProjectF Report
```

Bottom-Up Reporting with Fixplate

```haskell
calculateProjectReports :: Project -> IO ProjectReport
calculateProjectReports = synthetiseM calc
  where
    calc (SingleProject p _) =
      calculateReport <$> DB.getBudget p <*> DB.getTransactions p
    calc (ProjectGroup _ reports) = pure (fold reports)
```

Pretty Printing Without Explicit Recursion

```haskell
prettyResult :: Ann ProjectF Report a -> String
prettyResult (Ann report project') =
  case project' of
    SingleProject (ProjectId p) name ->
      printf "%s (%d): %s" name p (prettyReport report)
    ProjectGroup name _ ->
      printf "%s: %s" name (prettyReport report)
```

Fixplate Draws Our Trees!

```
*Demo> pr <- calculateProjectReports someProject
*Demo> drawTreeWith prettyResult pr
 \-- Sweden: Budget: +2191.60, Net: +1238.19, difference: -953.41
      |-- Stockholm (1): Budget: +5092.27, Net: -1472.80, difference: -656 ...
      |-- Gothenburg (2): Budget: -4325.22, Net: +2252.52, difference: +65 ...
      \-- Malmö: Budget: +1424.55, Net: +458.47, difference: -966.08
           |-- Malmö City (3): Budget: -6456.93, Net: +2400.33, difference ...
           \-- Limhamn (4): Budget: +7881.48, Net: -1941.86, difference: - ...
```

Summary

- Use Haskell data types
- Leverage great abstractions
    - Functor
    - Monoid
    - Foldable
    - Traversable
    - and many more
- Maybe check out recursion schemes
- Enjoy evolving and refactoring existing code

Questions?