Functional Programming

4. The Functional Paradigm

Frank C Langbein
frank@langbein.org

Version 1.4.0

Monoids - Functional Design Patterns

- A **monoid** is a pair of a binary operator `(@@)` and a value `u` where the operator has the value as identity and is associative

```
u @@ x = x
x @@ u = x
(x @@ y) @@ z = x @@ (y @@ z)
```

- So we can define

```
@@_concat :: m -> m -> m
@@_concat u ys      = ys
@@_concat (x:xs) ys = x @@ (xs @@_concat ys)
```

  - This should look very familiar, e.g. (`(++)`, `[]`)

    ```
    (++) :: m -> m -> m
    [] ++ ys      = ys
    (x:xs) ++ ys = x : (xs ++ ys)
    ```

- More: `((+),0)`, `((*),1)`, `((||),False)`, `((&&),True)`, `((>>),done)`

Monoid Typeclass

- Define a general pattern of Monoids

```
class MyMonoid m where
    my_mempty  :: m
    my_mappend :: m -> m -> m
    my_mconcat :: [m] -> m
    my_mconcat = foldr my_mappend my_mempty
```

- Define list append Monoid

```
instance MyMonoid [a] where
    my_mempty = []
    my_mappend = (++)
```

  - Then `my_append [1,2,3] [4,5,6]` works like `(++)`

- Note, `Monoid` is already defined in the `prelude`, hence the use of `My`/`my_`

Typeclasses

- **Type** declaration: defines how a particular type is created
- **Typeclass**: defines how a set of types are consumed / used in computations
  - Generalise over a set of types to define and execute a standard set of features for those types
  - A type which is part of a typeclass implements the typeclass behaviour
  - It is not an OO class, but some kind of interface definition
- Note, each type can only have one instance of each typeclass
  - To achieve global uniqueness of instances (by design in Haskell)

Bool typeclass Example

- E.g. `Eq` typeclass for `Bool` (already defined, of course)

  ```
  class Eq a where
      (==) :: a -> a -> Bool
      (/=) :: a -> a -> Bool

  data Bool = False | True
  instance Eq Bool
      (==) True  True  = True
      (==) False False = True
      (==) _     _     = False
  ```

  - Actually defined via `deriving`, which creates the matching patterns

    ```
    data Bool = False | True deriving (Eq)
    ```

    - Deriving is done by the compiler (allowed for specific classes in the `Prelude` only)
    - Extensions to write your own mechanisms exist

Functor

- **Functor**: apply a function to elements of a data structure
  - Inbuilt class with definition

    ```
    class Functor f where
        fmap :: (a->b) -> f a -> f b
    ```

  - Take a function `a->b`, a structure of `a` and create the same structure of `b` (applying the function)
- E.g. `fmap (\x -> x > 3) [1..6]`

- This generalises `map`

Applicative

- **Applicative**: monoidal functors
  - Same as functor, but the function is also embedded in a structure

    ```
    class Functor f => Applicative f where
          pure :: a -> f a
          (<*>) :: f (a->b) -> f a -> f b
    ```

  - Take a structure of functions `a->b`, a structure of `a` and create the same structure of `b` (by applying the functions)
  - Using `pure` that creates a structure of `a` from `a`
- E.g. how to apply a list of functions?
  - `[(*0),(+10),(^2)] <*> [1,2,3]`
- The list type constructor `[]` is an applicative

  ```
  instance Applicative [] where
      pure x = [x]
      fs <*> xs = [f x | f <- fs, x <-xs]
  ```

Monads

- A monad is a specialisation of an applicative (which is a special functor)
  - A Monad allows to run actions depending on the outcomes of earlier actions

```
do
    text <- getLine;
    if null text
      then putStrLn "You did not enter anything"
      else putStrLn "You entered " ++ text
```

- Recall `(>>=)` and `return`

```
return v >>= \x -> m          = m[x:=v]
m >>= \x -> return x          = m
(m >>= \x -> n) >>= \y -> o  = m >>=  \x -> (n >>= \y -> o)
(m >>= f) >>= g               = m >>= (\x -> (f x >>= g)    )
```

- Anything that is a monad is an applicative, is a functor
  - E.g. `[1..3] >>= return . (+1)`
  - is `[(+1)] <*> [1..3]`
  - is `fmap (+1) [1..3]`

Monad Operations

```
class Applicative m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
```

- `return` takes a value and returns it inside the structure (like `pure`)
- >> (sequencing operator) sequences two actions while discarding any resulting value of the first action
- >>= is the bind operator (the special part of monads)
  - Take a structure of a and a function `a -> m b` creating a structure of b from a and return a structure of b

- Monads change the structure involved via a function

Concat, Join and Bind

- Functors:

```
andOne x = [x,1]
andOne 10              -- [10,1]
fmap andOne [4,5,6]    -- [[4,1],[5,1],[6,1]]
```

  - What if we only wanted a list, not a list of lists?
  - I.e. > `concat $ fmap andOne [4,5,6]`

```
concat :: Foldable t => t [a] -> [a]  --- or just concat : [[a]] -> [a]
```

- Monads generalises concat

```
import Control.Monad (join) -- defines: join :: Monad m => m (m a) -> m a
```

  - Now bind (like >>=) is

```
bind :: Monad m => m a -> (a -> m b) -> m b
bind a b = join (fmap b a)
```

  - E.g. `bind [1..3] andOne` or `[1..3] >>= addOne`

Maybe

```haskell
data MyMaybe a  =  MyNothing | MyJust a  deriving Show

safeLog :: (Floating a, Ord a) => a -> MyMaybe a
safeLog x | x > 0 = MyJust (log x)
          | otherwise = MyNothing

instance Functor MyMaybe where
  fmap f MyNothing = MyNothing
  fmap f (MyJust a) = MyJust (f a)
  -- fmap log (MyJust 10) or fmap log MyNothing

instance Applicative MyMaybe where
  pure x = MyJust x
  (<*>) MyNothing _ = MyNothing
  (<*>) _ MyNothing = MyNothing
  (<*>) (MyJust f) (MyJust x) = MyJust (f x)
  -- (MyJust log) <*> (MyJust 10) or (MyJust log) <*> MyNothing

instance Monad MyMaybe where
  MyNothing >>= f  = MyNothing
  (MyJust x) >>= f = f x
  return           = MyJust
  -- (MyJust 10) >>= safeLog or (MyJust (-10)) >>= safeLog
```

Programming Paradigms

- **Imperative**: Use statements, e.g. assignment statements
    - Explicitly change the state / the memory of the computer
- **Logic**: Use terms of mathematical logic (mainly first-order logic)
    - Reasoning with relations

- **Functional**: Evaluate mathematical functions
    - Define functions without side-effects / no state
- **Object-Oriented**: associate behaviour with data-structures (objects), belonging to classes, structured into a hierarchy
    - Change the state of objects, communicate between objects
- Most programming languages use multiple paradigms, but have a preference

---

The Imperative Paradigm

- Computers have re-usable memory that can change state
    - Imperative languages are explicitly based on von Neumann-Zuse computer architectures
- Statements affect the state of the machine
    - Mathematically this means a sequence of values is associated with a variable/state x
    - `[x(1),...,x(t),...]` where `t` is a discrete time variable
    - Becomes increasingly complicated as the state-changes
- Imperative languages can relatively easily be translated into efficient machine-code
    - They are considered to be highly efficient
- Many people find the imperative paradigm quite natural
- Such languages do have functional features (e.g. arithmetic expression, etc.)

The Logic Paradigm

- Logical paradigm focuses on predicate logic

    - Basic concept is a relation
- Useful for problems where it is no obvious what the functions should be
    - Search and construct valid relations
    - To answer questions
- Mathematical logic plays key role in computation, since Boolean logic is basis of the design of logic circuits
    - Note, first-order logic is incomplete
- Of course, we can create a functional definition of a logic evaluation procedure

---

The Functional Paradigm

- Emphasis on evaluating mathematical functions
    - Combined into expressions
    - Based on lambda calculus
- While common in other languages, pure functional languages have no side-effects
    - Makes proofing correctness simple
    - E.g. `f(x) * f(x) == f(x)^2` (not true if `f` has side-effects)
- Often functional programs are less efficient compared to imperative programs
    - Recursion can be expensive (but does not have to be)
    - Absence of side-effects requires more memory (hard to determine which memory can be reused)

The Object-Oriented Paradigm

- OO associates data with behaviour to create objects

    - Methods change the state of objects (not a procedure operating on data)
    - Objects have a class indicating their behaviour
    - Often classes have a hierarchy to group behaviour and enable inheritance
    - Behaviour is encapsulated to allow only legitimate enquiries

- Based on **closures**

    - An association of behaviour (i.e. some code) with data
    - Data can only be accessed by passing appropriate arguments

- Often implies modelling the world according to objects

    - Ignores how data is processed and leads to inefficient code

---

The Object-Oriented Paradigm

- Focus is on the data, not functions

    - `f(x,y,z)` can be thought of as `(x,y,z).f ()`
    - But side-effects are allowed in OO

- Message-passing sub-paradigm

    - `send OBJECT MESSAGE` instead of `OBJECT.MESSAGE()`
    - Useful for communication in networks
    - Single dispatch: not so useful to send messages to multiple objects

- Distributed function definition sub-paradigm

    - Functions are defined in multiple places
    - Objects have methods with the same names
    - Overloading