Purely functional imperative programming in Haskell

Simon Meier

September 20, 2011
BMPI AG, Zurich

Outline

1. Pure functions and purely functional programming
2. Imperative programming in Haskell
3. Outlook and conclusion

Pure functions

A function is pure iff

1. it always yields the same value when applied to the same arguments and
2. calling it has no observable side-effect.

Examples of impure functions (in C)

```
// violates 1 and 2: Linux system call to get the system time
int clock_gettime(clockid_t clk_id, struct timespec *tp);

// violates only 2: filling a memory area with a fixed byte
void *memset(void *s, int c, size_t n);
```

Pure functions are context-independent and side-effect-free.

- thread-safe by default
- favor testing/reasoning about them
- provide a solid foundation for compositionality

How do you write pure functions?

Pure functions cannot use global/shared mutable state!

One approach to pure functions: purely functional programming

- disallow mutable state (no destructive updates)
- effectivity depends on programming language features: garbage collection, first-class functions, pattern matching, ...
- data structures must use copy-on-write, but may share parts
- new optimization opportunities: for example, rewrite rules $\texttt{map f} \circ \texttt{map g} = \texttt{map} \, (\texttt{f} \circ \texttt{g})$
- does not exclude an imperative programming style

Haskell: an overview

Language features

- purely functional: first-class function, algebraic datatypes, pattern matching
- lazy by default
- type system ensures purity of functions
- type inference obviates need for explicit type annotations
- parametric polymorphism + type classes

```
sort :: Ord a => [a] -> [a]
```

Language infrastructure

- standards: Haskell 98, Haskell 2010
- interpreted (GHCi) and compiled to native code (GHC)
- supported platforms: Windows, Mac, Linux
- open source: strong user/research community
- very good support for concurrency in GHC
- standard distribution: http://hackage.haskell.org/platform/

Imperative programming in Haskell

Explanation in three steps:

1. Haskell syntax + running example
2. Modeling side-effectful computations purely
3. Using monads to abstract over sequencing

Running example: evaluating arithmetic expressions

```haskell
data Op   = Plus | Minus | Times | Divide

data Expr = Lit Integer
          | Bin Op Expr Expr

--  8 - 2 * 5
expr1 :: Expr
expr1 = Bin Minus (Lit 8) (Bin Times (Lit 2) (Lit 5))


evalOp :: Op -> (Integer -> Integer -> Integer)
evalOp Plus   = (+)
evalOp Minus  = (-)
evalOp Times  = (*)
evalOp Divide = div


eval :: Expr -> Integer
eval (Lit i)        = i
eval (Bin op e1 e2) = (evalOp op) (eval e1) (eval e2)
```

Modeling "side-effectful" computations purely

Idea: represent computation result jointly with "side-effects"

- Computations that could raise exceptions of type 'String'

```
data Error a = Exception String
             | Result a

computation :: Error result
```

- Stateful computations

```
computation :: state -> (result, state)
```

- Environment dependent computations

```
computation :: env -> result
```

- Non-deterministic computations (could return multiple results)

```
computation :: [result]
```

Using ‘Error a’ to handle division by zero

```haskell
data Error a = Exception String
             | Result a

evalOpE :: Op -> Integer -> Integer -> Error Integer
evalOpE Divide x 0 = Exception (show x ++ " / 0")
evalOpE op     x y = Result (evalOp op x y)

evalE :: Expr -> Error Integer
evalE (Lit i)        = Result i
evalE (Bin op e1 e2) =
  case evalE e1 of
    Exception msg -> Exception msg
    Result x1     ->
      case evalE e2 of
        Exception msg -> Exception msg
        Result x2     -> evalOpE op x1 x2
```

This works, but it is ugly: let's get rid of the boilerplate.

Abstracting over "sequencing"

```
data Error a = Exception String | Result a
```

A combinator for sequencing computations with String exceptions:

```
(>>=) :: Error a -> (a -> Error b) -> Error b
Exception msg >>= _ = Exception msg
Result x      >>= f = f x
```

The resulting code reads already quite a bit better

```
evalE (Bin op e1 e2) = evalE e1 >>= (\x1 ->
                        evalE e2 >>= (\x2 ->
                         evalOpE op x1 x2
                       ))
```

than the code we had before.

```
evalE (Bin op e1 e2) =
  case evalE e1 of
    Exception msg -> Exception msg
    Result x1     ->
      case evalE e2 of
        Exception msg -> Exception msg
        Result x2     -> evalOpE op x1 x2
```

Syntactic sugar for sequencing

Using Haskell's do-notation further simplifies our code:

```
evalE :: Expr -> Error Integer
evalE (Lit i)          = return i
evalE (Bin op e1 e2) = do x1 <- evalE e1
                          x2 <- evalE e2
                          evalOpE op x1 x2
```

It's just syntactic sugar for

```
evalE :: Expr -> Error Integer
evalE (Lit i)          = return i
evalE (Bin op e1 e2) = evalE e1 >>= (\x1 ->
                        evalE e2 >>= (\x2 ->
                         evalOpE op x1 x2
                       ))
```

Overloading of `return` and `>>=` is used to share do-notation.

Injection of pure values + sequencing = monad

Haskell's type-classes are used to abstract over all monadic types.

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

- do-notation available for all types in the 'Monad' type-class

All monads must satisfy certain laws to ensure they behave as expected. They are satisfied by the instance below.

```
instance Monad Error where
  return x           = Result x

  Exception e >>= _ = Exception e
  Result x    >>= f = f x
```

For reference: our definition of `Error a`

```
data Error a = Exception String | Result a
```

The 'Error' monad

Most monads also support some special operations other than 'return' to construct a monadic value.

```
throw :: String -> Error a
throw exc = Exception exc

catch :: Error a -> (String-> Error a) -> Error a
catch (Exception msg) handler = handler msg
catch (Result x)      _       = Result x
```

Now our code looks almost as usual :-)

```
evalOpE :: Op -> Integer -> Integer -> Error Integer
evalOpE Divide x 0 = throw (show x ++ " / 0")
evalOpE op     x y = return (evalOp op x y)
```

For real applications: parametrise over type of exceptions.

```
data Error e a = Exception e
               | Result a
```

Outlook: more monads

- the 'State' monad

```
runState :: State s a -> (s -> (a, s))
put      :: s -> State s ()
get      :: State s s
```

- monad transformers allow to stack different effects

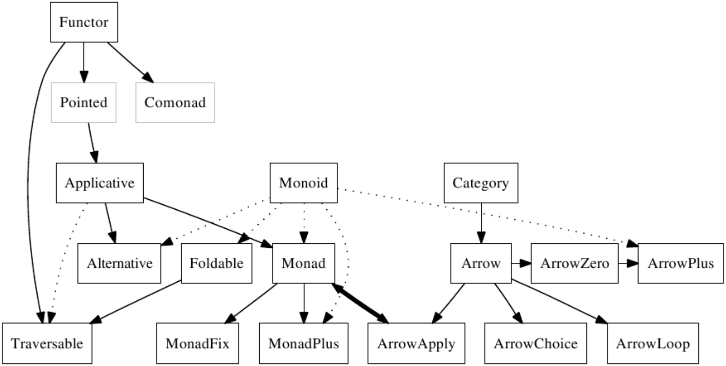
```
type Parser a = StateT String [] a

runParser :: Parser a -> (String -> [(a,String)])
```

- many types are monads:
    - monads abstract that they can be "sequenced"
    - do-notation supports all of them
    - Haskell has a programmable semi-colon ;-)

Outlook: the mathematics behind programming

Haskell let's you abstract over and productively use many more concepts from algebra/category theory.



http://www.haskell.org/haskellwiki/Typeclassopedia by Brent Yorgey

- capture common programming patterns formally
- arrows denote inclusion of source interface in target interface
- the less-expressive an interface, the fewer possible mistakes!

Conclusions

- pure functions are a very valuable commodity!
  Haskell features the largest library of them: `http://hackage.haskell.org`
- purely functional programming in Haskell is expressive and effective
  (as well as horizon expanding)

  Mutable state by default: a sub-optimal design decision?
- monads allow purely functional imperative programming
- many mathematical structures occur in programming;
  Haskell's abstraction facilities allow to exploit them productively

Resources on Haskell

- website: `http://www.haskell.org`
- free online books:
  Real World Haskell (website), Learn You a Haskell (website)
- blog aggregate: `http://planet.haskell.org/`
- mailing lists: beginners@haskell.org, haskell-cafe@haskell.org
- IRC: #haskell, #haskell-in-depth (access infos)

Thank you

Questions?

Slides and source code of examples:
https://github.com/meiersi/talks

Monad laws

1. lifting function application

   ```
   return a >>= k            ==  k a
   ```

2. `return` is right-identity of `>>=`

   ```
   m >>= return              ==  m
   ```

3. "associativity" of `>>=`

   ```
   m >>= (\x -> k x >>= h)   ==  (m >>= k) >>= h
   ```

   a simpler to understand version:

   ```
   (m1 >=> m2) >=> m3  ==  m1 >=> (m2 >=> m3)
   ```

   where

   ```
   (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
   f >=> g = \x -> f x >>= g‘
   ```