



Advanced Functional Programming

Week 4 – Functors, Record Syntax, Case Study: A Simple Parser

René Thiemann

Department of Computer Science

Last Week

- generalization of types: higher order type-expressions using partial application
 - type-expressions can be used in function definitions and in type-class definitions
 - kinds are used to “type” type-expressions
 - example

```
class IArray a e where
  bounds :: Ix i => a i e -> (i,i)
```

```
ghci> :k IArray
IArray :: (* -> * -> *) -> * -> Constraint
```

- explicit `forall` can be used for existential quantification
 - implementation can choose how to instantiate type variables of parameter
- using `forall` requires user-specified types
 - automation cannot infer `forall`s automatically
 - type-inference is undecidable with explicit `forall`s

Functor

map

- consider the following Haskell source

```
sqrtInt :: Int -> Double  
sqrtInt x = sqrt (fromIntegral x)
```

```
sqrtList [] = []  
sqrtList (x : xs) = sqrtInt x : sqrtList xs
```

- we clearly see that `sqrtList` just applies `sqrtInt` on every number in a list
- there are many more functions that process the elements in a list pointwise
- abstraction**: program `map` once, and then apply it several times

```
map :: (a -> b) -> [a] -> [b]  
map f [] = []  
map f (x : xs) = f x : map f xs
```

```
sqrtList = map sqrtInt  
upperString = map toUpper  
...
```

mapTree

- now consider trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show
```

```
sqrtTree (Leaf x) = Leaf (sqrtInt x)
```

```
sqrtTree (Node l r) = Node (sqrtTree l) (sqrtTree r)
```

- we see that `sqrtTree` just computes the square-root of every number in the tree
- there are many more functions that might process a tree in the same way, i.e., performing pointwise updates in the tree
- **abstraction**: program `mapTree` once, and then apply it several times

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

```
mapTree f (Leaf x) = Leaf (f x)
```

```
mapTree f (Node l r) = Node (mapTree f l) (mapTree f r)
```

```
sqrtTree = mapTree sqrtInt
```

```
upperTree = mapTree upperString
```

```
...
```

Functor

- clearly, there are strong similarities between `map` and `mapTree`
`map` :: (a -> b) -> [a] -> [b]
`mapTree` :: (a -> b) -> Tree a -> Tree b
- we could also have written further map-functions for other types, e.g.
`mapMaybe` :: (a -> b) -> Maybe a -> Maybe b
- generalize common idea of **structure preserving map-functions**
 - consider some **unary type-constructor** `f` for containers over arbitrary type `a`,
i.e., `f a` stores values of type `a`
 - a map-function `fmap` for `f` takes an arbitrary function of type `a -> b` in order to convert some `f a`-element to an `f b`-element in a way that
 - the structure is not altered (same shape of list, tree, ...)
 - `fmap id = id`
 - `fmap (g . h) = fmap g . fmap h`

in this case we say that `f` is a **functor**
- examples
 - the list type-constructor is a functor, with `map` being the map-function
 - the tree type-constructor is a functor, with `mapTree` being the map-function

Functors in Haskell

- in Haskell it is possible to define a type-class to represent functors

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

- note: higher-order kinds are required: `Functor :: (* -> *) -> Constraint`
- instance declarations are as usual

```
instance Functor [] where  
    fmap = map                -- use existing function
```

```
instance Functor Tree where  
    fmap = mapTree            -- use existing function
```

```
instance Functor Maybe where  
    fmap g Nothing = Nothing    -- define map within functor instance  
    fmap g (Just x) = Just (g x)
```

- observe: instances are type-constructors (`[]`, `Maybe`, ...), not types (`[a]`, `Maybe a`)

Functors in Haskell, Continued

- now it is possible to write one function which applies the square-root operation on **arbitrary** functors

```
fmapSqrt = fmap sqrtInt
```

- type: `fmapSqrt :: Functor f => f Int -> f Double`
- type-substitution: `fmapSqrt` has the following more concrete types

- `fmapSqrt :: [Int] -> [Double]`

`f = []`

- `fmapSqrt :: Tree Int -> Tree Double`

`f = Tree`

- ...

- fact: for several types, there is no explicit named `map`-function such as `mapMaybe`, `mapTree`, but only the `fmap`-instance
- note: there is a `Set.map` function, but no `Functor` instance for `Set`
 - reason: `Set.map` is not structure preserving, i.e., it does not give rise to a functor
- view instances at <https://hackage.haskell.org/package/base/docs/Control-Monad.html#t:Functor>

Example of Using Functors, Syntax: `fmap` = `<$>`

- note that `fmap` is also available as infix `<$>` operator
 - `f $` applies a function `f` to an argument
 - `f <$>` applies a function `f` to values within container argument
- note the similarity of the unsafe and the safe version to compute $\lfloor \frac{x}{y} \rfloor^2$

```
safeDiv :: Int -> Int -> Maybe Int
```

```
safeDiv _ 0 = Nothing
```

```
safeDiv x y = Just (x `div` y)
```

```
unsafeSquareAfterDiv x y = (^2) $ x `div` y
```

```
safeSquareAfterDiv x y = (^2) <$> x `safeDiv` y
```

Functors of Non-Unary Type-Constructors

- consider types

```
data (a,b) = (a,b)
```

```
data (a,b,c) = (a,b,c)
```

```
data Either a b = Left a | Right b
```

- for all of these types, there is also a natural map-function
- there are two approaches in Haskell
- first approach: make a functor instance w.r.t. the last type variable

```
instance Functor (Either a) where
```

```
    fmap f (Left x) = Left x
```

```
    fmap f (Right y) = Right (f y)
```

```
instance Functor ((,) a) where
```

```
    fmap f (x,y) = (x, f y)
```

```
instance Functor ((,,) a b) where
```

```
    fmap f (x,y,z) = (x, y, f z)
```

Bifunctors

- second approach: use a **bifunctor**, map over last two type variables

```
class (forall a. Functor (p a)) => Bifunctor p where
  bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
  first :: (a -> b) -> p a c -> p b c
  second :: (b -> c) -> p a b -> p a c
  first f = bimap f id
  second g = bimap id g
```

```
instance Bifunctor Either where
  bimap f g (Left x) = Left (f x)
  bimap f g (Right y) = Right (g y)
```

```
instance Bifunctor (,) where
  bimap f g (x,y) = (f x, g y)
```

```
instance Bifunctor ((,,) a) where
  bimap f g (x,y,z) = (x, f y, g z)
```

Case Study: Parsing PGM-Graphics

Parsing

- parsing
 - read some structured input format into internal representation
 - or report error
- examples
 - ghc parses Haskell source and converts it into abstract syntax tree; this is one of the first steps of the compilation process
 - browser parses HTML-file from server, and afterwards renders it
- parsers can be automatically generated in Haskell via `deriving Read`
 - however, then the input format will be Haskell expressions
 - in this course, we do not restrict to this approach
- generic idea of a parser for type `ty`
 - take input
 - consume first part of input and try to convert it to element `x :: ty`
 - return `x` and remaining input, or fail if some error occurred
- example in this section: parse PGM raw format for `.pgm` images (portable grey map)

PGM Raw Format

- PGM raw format has following structure
 - first two characters are “P5”
 - then there are three numbers separated by arbitrary amount of white space
 - width
 - height
 - maximal grey value
 - after the last of these numbers, a single white space character appears
 - finally the correct number of grey values of the image are provided as bytes
- example:

```
P5 1366 1036
```

```
255
```

```
here the binary part starts
```



Representation of Output – Datatypes with Record Syntax

- store width, height, max grey value and binary grey values
- `data Greymap = Greymap Int Int Int L.ByteString`
 - is an obvious choice
 - might be confusing: order of `Ints` unclear
 - adding another entry will require to change patterns in function definitions
- data types can also use **record syntax**: more verbose, more flexible

```
data Greymap = Greymap {  
    greyWidth :: Int  
    , greyHeight :: Int  
    , greyMax :: Int  
    , greyData :: L.ByteString  
} deriving Eq
```

```
greyWidth :: Greymap -> Int
```

```
ex1 = Greymap { greyHeight = 10, greyWidth = 5, greyData = ..., ... }
```

```
ex2 = ex1 { greyHeight = 30 } -- update by name
```

Representation of Input

- input mixes ASCII and binary encoding
 - use Haskell `ByteString` as compact representation (uses arrays internally)
 - `ByteStrings` can be read both in binary and in character-based mode
 - sometimes conversion required, e.g., between `String` and `ByteString`

- example code

```
import qualified Data.ByteString.Lazy.Char8 as L8  -- ASCII
import qualified Data.ByteString.Lazy as L         -- binary
```

```
L.readFile :: FilePath -> IO L.ByteString
L.drop :: Int64 -> L.ByteString -> L.ByteString
L.length :: L.ByteString -> Int64
L8.pack :: [Char] -> L.ByteString
L8.isPrefixOf :: L.ByteString -> L.ByteString -> Bool
L8.dropWhile :: (Char -> Bool) -> L.ByteString -> L.ByteString
L8.readInt :: L.ByteString -> Maybe (Int, L.ByteString)
```

- note: `L.ByteString -> Maybe (a, L.ByteString)` is **type of parser for a-values**

An Ad-Hoc Parser for PGM P5 Files

```
parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5 s =
  case matchHeader (L8.pack "P5") s of
    Nothing -> Nothing
    Just s1 ->
      case getNat (dropSpace s1) of
        Nothing -> Nothing
        Just (width, s2) ->
          case getNat (dropSpace s2) of
            Nothing -> Nothing
            Just (height, s3) ->
              case getNat (dropSpace s3) of
                Nothing -> Nothing
                Just (maxGrey, s4)
                  | maxGrey > 255 -> Nothing
                  | otherwise ->
                    case getBytes 1 s4 of
                      Nothing -> Nothing
                      Just (_, s5) ->
                        case getBytes (width * height) s5 of
                          Nothing -> Nothing
                          Just (bitmap, s6) -> Just (Greymap width height maxGrey bitmap, s6)
```

An Ad-Hoc Parser for PGM P5 Files – Auxiliary Functions

```
dropSpace = L8.dropWhile isSpace
```

```
matchHeader :: L.ByteString -> L.ByteString -> Maybe L.ByteString
matchHeader prefix str
  | prefix `L8.isPrefixOf` str = Just (L.drop (L.length prefix) str)
  | otherwise = Nothing
```

```
getNat :: L.ByteString -> Maybe (Int, L.ByteString)
getNat s = case L8.readInt s of
  Nothing -> Nothing
  Just (num,rest)
    | num < 0 -> Nothing
    | otherwise -> Just (fromIntegral num, rest)
```

```
getBytes :: Int -> L.ByteString -> Maybe (L.ByteString, L.ByteString)
getBytes n str = let count           = fromIntegral n
                  both@(prefix,_) = L.splitAt count str
                  in if L.length prefix < count
                     then Nothing
                     else Just both
```

Problems of Ad-Hoc Parser

- problem 1: **repetitive case-analysis** on **Maybe**-values
 - if we got a failure, then fail
 - otherwise, extract the current input and proceed with the next parser
- solution: refactor by abstraction
- problem 2: **direct pattern matching on pairs** (parsed-value, remaining input)
 - if we want to make a more verbose parser, e.g., tracking failure positions, we have to change all occurrences of pairs within parser
- solution: refactor by abstraction and data-hiding

Solving Repetitive Case-Analysis

- general abstract scheme
 - if we got a failure, then fail
 - otherwise, extract the current input and proceed with the next parser
- idea of defining abstract scheme as function
 - first argument is optional current value
 - second argument is function how to proceed
- in Haskell

```
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
Nothing >>? _ = Nothing
```

```
Just v >>? f = f v
```

Solving Repetitive Case-Analysis: Adjusted Parser

```
parseP5_take2 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5_take2 s =
    matchHeader (L8.pack "P5") s      >>?
    skipSpace                          >>?
    getNat                            >>?
    \(width, s1) -> skipSpace s1      >>?
    getNat                            >>?
    \(height, s2) -> skipSpace s2    >>?
    getNat                            >>?
    \(maxGrey, s3) -> getBytes 1 s3  >>?
    (getBytes (width * height) . snd) >>?
    \(bitmap, s4) -> Just (Greymap width height maxGrey bitmap, s4)
```

```
skipSpace :: L.ByteString -> Maybe L.ByteString
skipSpace s = Just (L8.dropWhile isSpace s)
```

Observations

- nested case-analysis is gone
- still two stylistic problems
 - state s ($s1, s2, \dots$) is explicitly passed around
 - pattern matching on pairs is still present
- both problems will be handled by abstract new type for parsers

A Datatype for Parsing

Parser-State and Parser

- **parser-state** stores input and offset

- stored in dedicated datatype

```
data ParseState = ParseState {  
    string :: L.ByteString -- remaining input  
    , offset :: Int64      -- location w.r.t. global input  
} deriving Show
```

- **parser** for elements of type **a** is a function from parser-state to either an error-message or a pair consisting of an **a**-element and a new parser-state

- we encapsulate such a function in a separate type

```
newtype Parse a = Parse {  
    runParse :: ParseState -> Either String (a, ParseState)  
}
```

- using **newtype** at this point: constructor **Parse** is not visible at runtime

Chaining Parsers

- recall definition from previous slide

```
newtype Parse a = Parse {  
    runParse :: ParseState -> Either String (a, ParseState)  
}
```

- design primitive for chaining two parsers for sequential composition

```
(==>) :: Parse a -> (a -> Parse b) -> Parse b  
firstParser ==> secondParser = Parse chainedParser  
  where chainedParser initState =  
    case runParse firstParser initState of  
      Left err -> Left err  
      Right (firstResult, newState) ->  
        runParse (secondParser firstResult) newState
```

- both a `Parse a`-element and also `p1 ==> \ x -> p2 x` never execute the functions

- chaining two parsers without result dependence

```
(==>&) :: Parse a -> Parse b -> Parse b  
p1 ==>& p2 = p1 ==> \_ -> p2
```

Four Basic Parsers

```
newtype Parse a = Parse {  
  runParse :: ParseState -> Either String (a, ParseState) }
```

- the parser that always succeeds and does not alter the state

```
identity :: a -> Parse a  
identity a = Parse (\s -> Right (a, s))
```

- the parser that always fails

```
bail :: String -> Parse a  
bail err = Parse (\s -> Left $  
  "byte offset " ++ show (offset s) ++ ": " ++ err)
```

- the parser that reveals the internal state

```
getState :: Parse ParseState  
getState = Parse (\s -> Right (s, s))
```

- the parser that changes the internal state

```
putState :: ParseState -> Parse ()  
putState s = Parse (\_ -> Right ((), s))
```

Another Primitive: Parsing a Single Byte

```
parseByte :: Parse Word8
```

```
parseByte =
```

```
    getState ==> \state ->
```

```
    case L.uncons (string state) of
```

```
        Nothing ->
```

```
            bail "no more input"
```

```
        Just (byte, remainder) ->
```

```
            putState newState ==>&
```

```
            identity byte
```

```
        where newState = state { string = remainder,  
                                offset = newOffset }
```

```
            newOffset = offset state + 1
```

```
L.uncons :: L.ByteString -> Maybe (Word8, L.ByteString)
```

Switching to Characters: Parsing a Single Char

```
parseByte :: Parse Word8 -- previous slide  
parseChar :: Parse Char  -- do not copy code of previous slide
```

```
w2c :: Word8 -> Char  
w2c = chr . fromIntegral
```

```
parseChar :: Parse Char  
parseChar = w2c <$> parseByte
```

```
-- requires Functor instance of Parse  
instance Functor Parse where  
    fmap f parser = parser ==> \result ->  
        identity (f result)
```

Parsing Multiple Bytes

```
-- look at the first byte, without consuming it
```

```
peekByte :: Parse (Maybe Word8)
```

```
peekByte = (fmap fst . L.uncons . string) <$> getState
```

```
-- parsing multiple bytes
```

```
parseWhile :: (Word8 -> Bool) -> Parse [Word8]
```

```
parseWhile p = (fmap p <$> peekByte) ==> \mp ->  
    if mp == Just True  
    then parseByte ==> \b ->  
        (b:) <$> parseWhile p  
    else identity []
```

```
-- and using conversion from Word8 to other type
```

```
parseWhileWith :: (Word8 -> a) -> (a -> Bool) -> Parse [a]
```

```
parseWhileWith f p = fmap f <$> parseWhile (p . f)
```

Final Parser

```
parseRawPGM :: Parse Greymap
```

```
parseRawPGM =
```

```
    parseWhileWith w2c notWhite ==> \header -> skipSpaces ==>&  
    assert (header == "P5") "invalid raw header" ==>&  
    parseNat ==> \width -> skipSpaces ==>&  
    parseNat ==> \height -> skipSpaces ==>&  
    parseNat ==> \maxGrey ->  
    parseByte ==>&  
    parseBytes (width * height) ==> \bitmap ->  
    identity (Greymap width height maxGrey bitmap)  
where notWhite = (`notElem` " \r\n\t")
```

```
-- clear structure
```

```
-- no handling of explicit states
```

```
-- assert, parseBytes, parseNat, skipSpaces: see next slides
```

Remaining Primitives (1/2)

```
skipSpaces :: Parse ()  
skipSpaces = parseWhileWith w2c isSpace ==>&  
    identity ()
```

```
assert :: Bool -> String -> Parse ()  
assert True _ = identity ()  
assert False err = bail err
```

```
parseNat :: Parse Int  
parseNat = parseWhileWith w2c isDigit ==> \digits ->  
    assert (not (null digits)) "digit expected" ==>&  
    let n = read digits in  
    assert (show n == digits) "integer overflow" ==>&  
    identity n
```

Remaining Primitives (2/2)

```
parseBytes :: Int -> Parse L.ByteString
parseBytes n =
    getState ==> \st ->
    let n' = fromIntegral n
        (h, t) = L.splitAt n' (string st)
        st' = st { offset = offset st + n', string = t }
    in assert (L.length h == n') "end of input" ==>&
        putState st' ==>&
        identity h

-- running a parser
parse :: Parse a -> L.ByteString -> Either String a
parse parser input = fst <$> runParse parser (ParseState input 0)
```

Literature

- Real World Haskell, Chapter 10