Refactoring

The only valid measurement of code quality: WTFs/minute

Good code.

Bad code.
Outline

• Good Haskell style
• What is refactoring?
• Strategies for refactoring
• Emphasizing function composition
Good Haskell style

Why it matters:
• layout is significant!
• expunging misconceptions
• we care about elegance

Easy stuff:
• use spaces (layout)
• align patterns and guards

See course web page for links to style guides
Formatting function applications

Function application . . .

• is *just a space*  \( f(x) \)
• associates to the left  \((f \ x) \ y\)
• binds most strongly  \((f \ x) + (g \ y)\)

*Use parentheses only to override this behavior:*

\[ f \ (g \ x) \]
\[ f \ (x + y) \]
Use pattern matching

pop :: [a] -> (a, [a])
pop xs = if not (null xs)
  then (head xs, tail xs)
  else error "empty"

pop :: [a] -> (a, [a])
pop xs = case xs of
  (y:ys) -> (y, ys)
  []    -> error "empty"

pop :: [a] -> (a, [a])
pop (x:xs) = (x, xs)
pop []    = error "empty"
Prefer pattern guards

```
elem :: Int -> Tree Int -> Bool
elem _ Leaf = False
elem x (Node y l r) =
  if x == y then True
  else if x < y then elem l x
  else elem r x
```

```
elem :: Int -> Tree Int -> Bool
elem _ Leaf = False
elem x (Node y l r)
  | x == y = True
  | x < y  = elem l x
  | otherwise = elem r x
```
Outline

• Good Haskell style
• What is refactoring?
• Strategies for refactoring
• Emphasizing function composition
FP workflow

Motivations:
- easier to read and maintain
- reusable components
- gain deeper insights

“obsessive compulsive refactoring disorder”
What is refactoring?

…a disciplined technique for restructuring existing code, altering its internal structure without changing its external behavior

— Martin Fowler

sum xs = if null xs then 0 else head xs + sum (tail xs)

sum [] = 0
sum (x:xs) = x + sum xs

sum = foldr (+) 0
Refactoring relations

Laws that are the formal basis for refactoring

**Eta reduction**

\[(\lambda x \to f x) \iff f\]

**Map fusion**

\[\text{map } f \ . \ \text{map } g \iff \text{map } (f \ . \ g)\]

“Algebra of computer programs”

John Backus, Can Programming be Liberated from the von Neumann style?
ACM Turing Award Lecture, 1978
Outline

• Good Haskell style
• What is refactoring?
• Strategies for refactoring
• Emphasizing function composition
Strategy: systematic generalization

commas :: [String] -> [String]
commas [] = []
commas [x] = [x]
commas (x:xs) = x : "", " : commas xs

... introduce parameters for constants

seps :: String -> [String] -> [String]
seps _ [] = []
seps _ [x] = [x]
seps s (x:xs) = x : s : seps s xs

... then broaden the types

intersperse :: a -> [a] -> [a]
intersperse _ [] = []
intersperse _ [x] = [x]
intersperse s (x:xs) = x : s : intersperse s xs
Strategy: abstract repeated templates

showResult :: Maybe Float -> String
showResult Nothing  = "ERROR"
showResult (Just v) = show v

getCommand :: Maybe Dir -> Command
getCommand Nothing  = Stay
getCommand (Just d) = Move d

addToMaybe :: Int -> Maybe Int -> Int
addToMaybe x Nothing  = x
addToMaybe x (Just y) = x + y

maybe :: b -> (a -> b) -> Maybe a -> b
maybe b _ Nothing  = b
maybe _ f (Just a) = f a

Repeated structure:
• pattern match
• default value if empty
• apply function otherwise

showResult   = maybe "ERROR" show
getCommand   = maybe Stay Move
addToMaybe x = maybe x (x+)
Notes on abstraction

*abstraction*: to separate a concept from its specific instances and make it reusable

Haskell has powerful tools for abstraction:

- **referential transparency**
  shared code can always safely be factored out

- **higher-order functions**
  can capture high-level patterns as functions

- **lazy evaluation**
  supports separation of concerns and definition of new control structures

- **type classes**
  describe common interface for many data types
Refactoring data types

```haskell
data Expr = Lit Int
  | Var Name
  | Add Expr Expr
  | Sub Expr Expr
  | Mul Expr Expr
```

**Factor out shared structure:**

```haskell
data Expr = Lit Int
  | Var Name
  | Bin Op Expr Expr
```

```haskell
data Op = Add | Sub | Mul
```

simplifies writing many functions

... especially when we don't need to distinguish these cases
Outline

• Good Haskell style
• What is refactoring?
• Strategies for refactoring
• Emphasizing function composition
Function composition

\[ f \ (g \ x) \iff (f \ . \ g) \ x \]

\[ f1 \ (f2 \ x \ (f3 \ (f4 \ y))) \rightarrow (f1 \ . \ f2 \ x \ . \ f3 \ . \ f4) \ y \]

Advantages:
- emphasizes functions over results
- reveals opportunities for eta reduction (next slide)

What is the type of the ( . ) operator?

\[ (\ . ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \]
Eta reduction

$$\lambda x \rightarrow f\ x \iff f$$

```haskell
data Base = A | C | G | T deriving Show
type DNA = [Base]

showDNA :: DNA -> String
showDNA bs = concat (map (\b -> show b) bs)
```

**Eta reduction**

Rewrite as composition

$$f\ (g\ x) \iff (f \circ g)\ x$$

To lambda-notation

$$f\ x = y \iff f = \lambda x \rightarrow y$$

**Eta reduction**
Point-free style

*Functions are defined:*

- *without referring to their arguments by name*
- *only by (compositions of) other functions*

```haskell
sum :: [Int] -> Int
sum = foldr (+) 0

showDNA :: DNA -> String
showDNA = concat . map show

topGrades :: [(Name, Grade)] -> [Name]
topGrades = map fst . filter (>(= 0.9) . snd)
```
Point-free tradeoffs

Advantages:
• emphasizes functions (high-level) over data (low level)
• result of refactoring – often leads to insights
• shows off how clever you are :-)

But … it's easy to get carried away – obfuscation

```
flip flip snd . (ap .) . flip flip fst .
((.) .) . flip . (((.) . (,))) .)
```

“pointless style”

```
vs.  (\f g (x,y) -> (f x, g y))
```
Ordering arguments

Note that library functions are always:

- parameters first
- primary data structure last

```haskell
map :: (a -> b) -> [a] -> [b]
foldr :: (a -> b -> b) -> b -> [a] -> b
maybe :: b -> (a -> b) -> Maybe a -> a
```

Supports partial application and composition – you should do it too!

```haskell
showMaybeInts :: [Maybe Int] -> String
showMaybeInts = concat . intersperse ""," . map (maybe "" show)
```