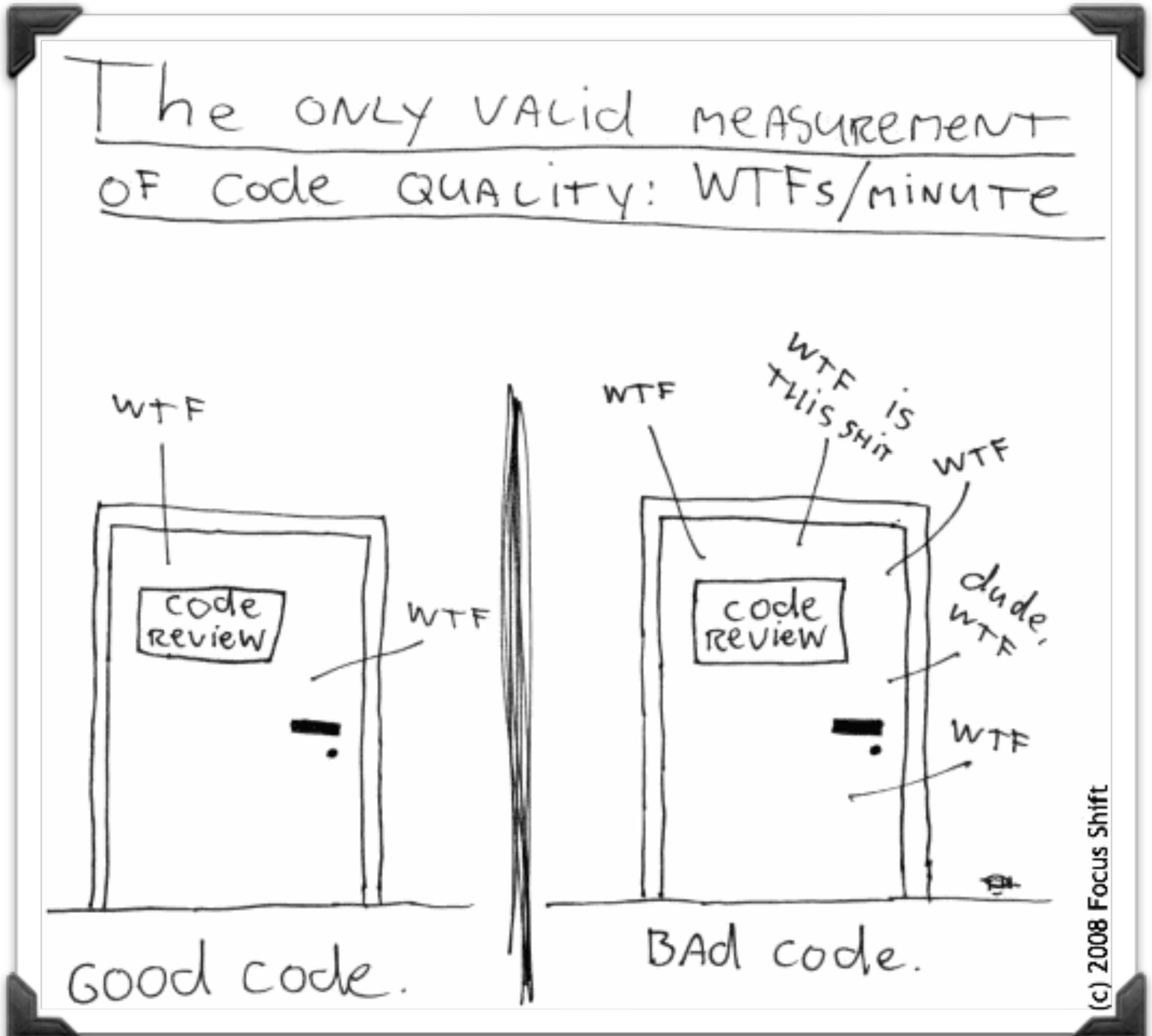
# Refactoring



# 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)` | `f x` |
| • associates to the left | `(f x) y` | `f x y` |
| • binds most strongly | `(f x) + (g y)` | `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


```haskell
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
```

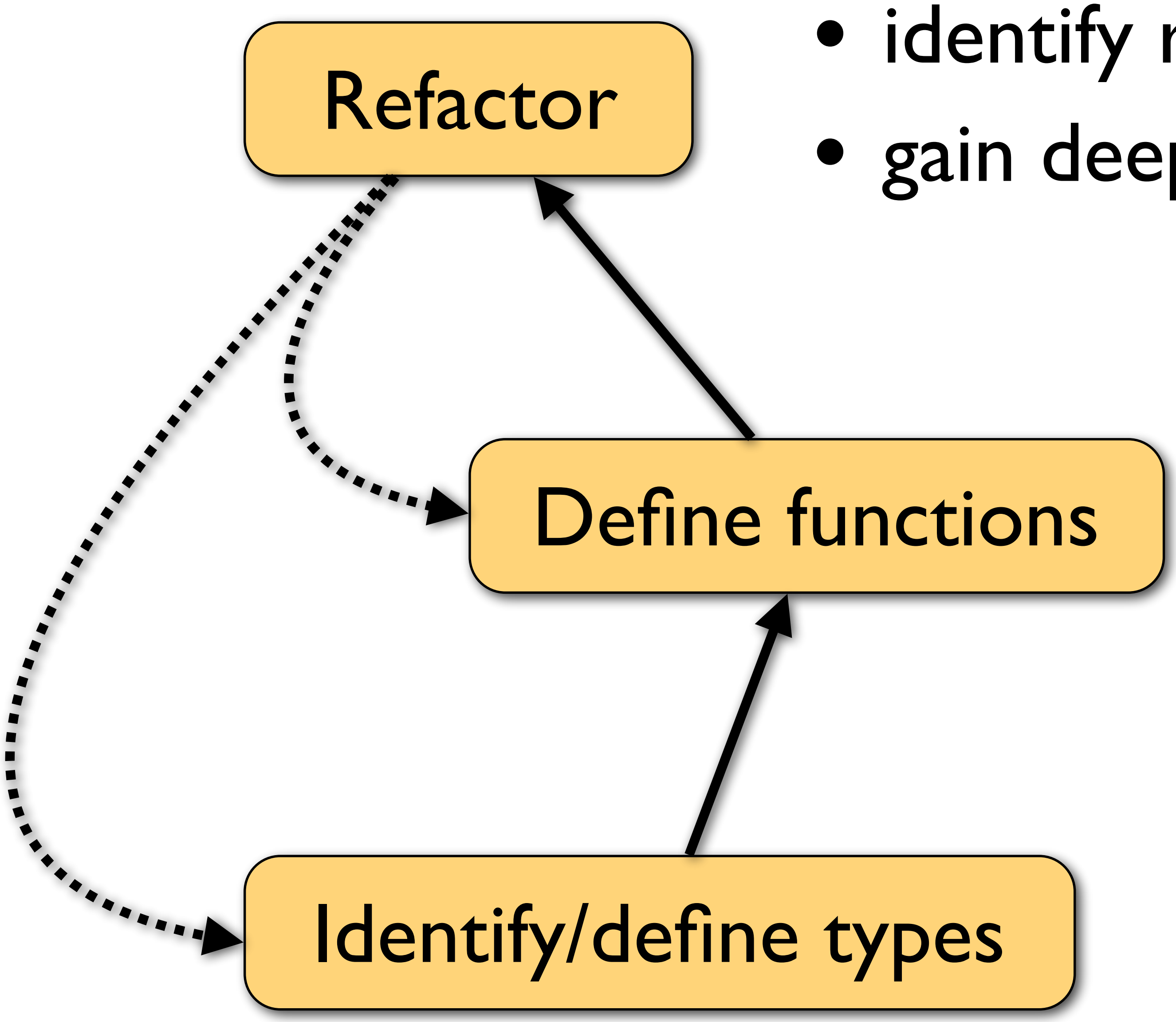
```haskell
elem :: Int -> Tree Int -> Bool
elem _ Leaf            = False
elem x (Node y l r)
        | x == y       = True
        | x <  y       = elem l x
        | otherwise    = elem r x
```

# Outline

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

# Why refactor?

- make code easier to read and maintain
- generalize to new/related problems
- identify reusable components
- gain deeper insights



# 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

```
(\x -> f x)  <==>  f
```

Map fusion

```
map f . map g  <==>  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
```

*Repeated structure:*

- *pattern match*
- *default value if empty*
- *apply function otherwise*

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

```
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 across many data types

# Refactoring data types

```
data Expr = Lit Int
          | Ref Var
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
```

*simplifies writing many functions*

*…especially when we don't need to distinguish these cases*

*Factor out shared structure:*

```
data Expr = Lit Int
          | Ref Var
          | Bin Op Expr Expr

data Op = Add | Sub | Mul
```

*(BinOp.[1-3].hs)*

# Outline

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

# Function composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
f (g x) <==> (f . g) x
```

```
f1 (f2 x (f3 (f4 y)))   →   (f1 . f2 x . f3 . f4) y
```

*Advantages:*

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

# Eta reduction

```
(\x -> f x)  <==>  f
```

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

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

*Eta reduction*

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

*Rewrite as composition*

```
f (g x) <==> (f . g) x
```

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

*To lambda-notation*

```
f x = y <==> f = \x -> y
```

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

*Eta reduction*

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

# Point-free style


Functions are defined:

- without referring to their arguments by name
- only by applying and composing other functions

```
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:

- emphasize functions over data
  - *what does this function do?* vs. *how does it do it?*
- result of refactoring – often leads to insights
- shows off how clever you are :-)

*But . . . it's easy to get carried away – leads to 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

```
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!*

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