Booleans Considered Harmful!

or

Boolean Blindness Explained for Beginners in Haskell

by Stephen Pimentel

@StephenPiment

Who am I?

What the heck is “boolean blindness”?

15-150 Lecture 9:
Options; Domain-specific Datatypes; Functions as Arguments

Lecture by Dan Licata

February 14, 2012

1.1 Boolean Blindness

Information Poverty

```
data Bool = True | False
```

Information Poverty

```
data () = ()
```

Information Loss

```
myComplicatedFunction :: Something -> Bool
```

Common Anti-pattern

Recomputing information within a branch

List Append

```haskell
appendWithBoolean :: Eq a => [a] -> [a] -> [a]
appendWithBoolean xs ys =
    if xs == []
        then ys
        else head(xs) : appendWithBoolean (tail xs) ys
```

List Append

```haskell
appendWithBoolean :: Eq a => [a] -> [a] -> [a]
appendWithBoolean xs ys =
    if xs == []
        then ys
        else head(xs) : appendWithBoolean (tail xs) ys
```

List Append

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

Principle

Types that fail to capture needed information make you do redundant work later.

What are the problems?

- Partial functions
- Erroneous arguments
- Locality of control

Partial Functions

```
lookupPartial :: Eq a => a -> [(a,b)] -> b
lookupPartial x ((a', b) : ys) =
    if x == a' then b else lookupPartial x ys
```

Partial Functions

```
lookupPartial :: Eq a => a -> [(a,b)] -> b
lookupPartial x ((a', b) : ys) =
    if x == a' then b else lookupPartial x ys
```

Partial Functions

```haskell
contains :: Eq a => a -> [(a,b)] -> Bool
contains  x xs = case xs of
    [] -> False
    (a', b) : c ->
      if x == a'
      then True
      else contains x c

addExtraCreditWithBool :: String -> Float
addExtraCreditWithBool grade =
    if contains grade gradeEquivalence
        then 1.0 + lookupPartial grade gradeEquivalence
        else 0.0 -- default
```

Partial Functions

```haskell
contains :: Eq a => a -> [(a,b)] -> Bool
contains  x xs = case xs of
    [] -> False
    (a', b) : c ->
      if x == a'
      then True
      else contains x c

addExtraCreditWithBool :: String -> Float
addExtraCreditWithBool grade =
    if contains grade gradeEquivalence
        then 1.0 + lookupPartial grade gradeEquivalence
        else 0.0 -- default
```

Partial Functions

```haskell
addExtraCreditWrong :: String -> Float
addExtraCreditWrong grade =
    if contains grade gradeEquivalence
        then 1.0 + lookupPartial grade gradeEquivalence
        else somethingElse $ lookupPartial grade gradeEquivalence
```

Principle

Information-Poor Types Make You “Touch.”

Principle

Information-Poor Types Make You “Touch.”

Null References: The Billion Dollar Mistake

Tony Hoare

Higher-Kinded Types

Avoid partiality

Maybe as a Return Type

```
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup _key []           =  Nothing
lookup  key ((x,y):xys)
    | key == x           =  Just y
    | otherwise          =  lookup key xys
```

Maybe as a Return Type

```
addExtraCredit :: String -> Maybe Float
addExtraCredit grade =
    case lookup grade gradeEquivalence of
        Just x -> Just $ 1.0 + x
        Nothing -> Nothing
```

Maybe as a Return Type

```haskell
addExtraCreditWontCompile :: String -> Maybe Float
addExtraCreditWontCompile grade =
    case lookup grade gradeEquivalence of
        Just x -> Just $ 1.0 + x
        Nothing -> Just $ somethingElse x
```

Principle

Information-poor types make you “touch.”
Richer types let you “see.”

Principle

Information-poor types make you “touch.”
Richer types let you “see.”

Principle

Higher-kinded types let you rewrite partial functions as total ones.

Problem

Erroneous arguments

Problem

```
badDesign :: Bool -> Bool -> Bool -> OMGImLost
```

Example

Input validation with Booleans

Input Validation

```haskell
validate :: Bool -> Bool -> String -> Bool
validate strip printable input =
  let output = if strip
       then filter (not . isSpace) input
       else input
  in if printable
       then all isPrint output
       else True
```

Input Validation

```haskell
validate :: Bool -> Bool -> String -> Bool
validate strip printable input =
  let output = if strip
       then filter (not . isSpace) input
       else input
  in if printable
       then all isPrint output
       else True
```

Problem

Boolean arguments are easy to call out of order.

Solution (Kind Of)

Replace Boolean arguments with algebraic data types.

Algebraic Data Types

```haskell
data ToStrip = Strip | NoStrip

data Printable = CheckPrintable | NoCheckPrintable

validate :: ToStrip -> Printable -> String -> Bool
validate strip printable input =
  let output = case strip of
        Strip -> filter (not . isSpace) input
        NoStrip -> input
  in case printable of
      CheckPrintable -> all isPrint output
      NoCheckPrintable -> True
```

Principle

Locality of control makes it easier to reason about code; locality violation makes it harder.

Principle

Locality of control makes it easier to reason about code; locality violation makes it harder.

Problem

The computation that produces a Boolean value is often distant from the conditional that uses it.

Solution

Move the computation makes a decision to the place that actually needs it.

Solution

Preserve locality.

Principle

Pass functions, not flags.

Input Validation

```
type Transform = String -> String
```

Replace First Flag

```haskell
type Transform = String -> String

validate :: Transform -> Printable -> String -> Bool
validate transform printable input =
  case printable of
      CheckPrintable -> all isPrint $ transform input
      NoCheckPrintable -> True
```

Replace Second Flag

```
type Predicate = String -> Bool

validate :: Transform -> Predicate -> String -> Bool
validate transform predicate input = predicate $ transform input
```

Replace Second Flag

```
type Predicate = String -> Bool

validate :: Transform -> Predicate -> String -> Bool
validate transform predicate input = predicate $ transform input
```

Maybe as Return Type

```haskell
validate :: Transform -> Predicate -> String -> Maybe String
validate transform predicate input =
  let output = transform input in
  if predicate output
    then Just output
    else Nothing
```

Make It Scale

```
validate :: Transform -> [Predicate] -> String -> Maybe String
```

all

```
all :: Foldable t => (a -> Bool) -> t a -> Bool
```

The Right Section of $

```
($x) :: (a -> b) -> b
```

A Scalable Version

```
validate :: Transform -> [Predicate] -> String -> Maybe String
validate transform preds input =
  let output = transform input in
  if all ($output) preds
    then Just output
    else Nothing
```

But There's Still a Conditional

```
validate :: Transform -> [Predicate] -> String -> Maybe String
validate transform preds input =
  let output = transform input in
  if all ($output) preds
    then Just output
    else Nothing
```

Helper Function

```haskell
maybeBool :: Bool -> a -> Maybe a
maybeBool flag x =
  if flag
    then Just x
    else Nothing
```

First Try At Refactoring

```
validate :: Transform -> [Predicate] -> String
validate transform preds input =
  let output = transform input in
  maybeBool (all ($output) preds) output
```

A Better Helper

```haskell
maybePred :: Predicate -> String -> Maybe String
maybePred predicate input =
  if predicate input
    then Just input
    else Nothing
```

Let’s Use a Monad

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Let’s Use a Monad

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Reverse Bind

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
```

Reverse Bind for Maybe

```
(=<<) :: (a -> Maybe b) -> Maybe a -> Maybe b
```

Using a Monad Instance

```
validate :: Transform -> [Predicate] -> String -> Maybe String
validate transform preds input =
  foldr (=<<) (Just $ transform input) $ map maybePred preds
```

Data Flow

```
> validate (filter (not . isSpace)) [(all isPrint)] " f  o  o "
> foldr (=<<)
        (Just $ (filter (not . isSpace)) " f  o  o ")
        (map maybePred [(all isPrint)])
> foldr (=<<)
        (Just $ (filter (not . isSpace)) " f  o  o ")
        [maybePred (all isPrint)]
> foldr (=<<)
        (Just "foo")
        [maybePred (all isPrint)]
> maybePred (all isPrint) =<< Just "foo"
> Just "foo"
```

Pretty Cool?

```haskell
validate :: Transform -> [Predicate] -> String -> Maybe String
validate transform preds input =
  foldr (=<<) (Just $ transform input) $ map maybePred preds
```

Summary

Type signature as API

Summary

Better APIs through richer types

Questions?

@StephenPiment