

# Properties!

And their relation to Types and Tests

# Properties!

And their relation to Types and Tests

Or at least how to write programs that break a bit less.

# Two ways of looking at properties

- The things that define an object.

```
-- | Product numbers are: [[:alnum:]]
-- unicode enc
-- Max length 140
-- Min length 1 (can't be empty)
newtype ProductNumber = ProductNumber { _unProductNumber :: Text}
    deriving (Show,Eq,Ord)
```

- The things that define what an object can do.

```
-- Product numbers uniquely determine a particular product
-- They can be sorted and compared
newtype ProductNumber = ProductNumber { _unProductNumber :: Text}
    deriving (Show,Eq,Ord)
```

- We will mostly be talking about the first.

# The Haskell Toolbox

The skills to pay the bills.

- Type Encoding
- Smart Constructors
- QuickCheck
- Typed Transformation
- LiquidTypes

# The Pattern.

The gist of what is going on...

- At the boundaries, use smart constructors and quick check to make sure types are built correctly.
- Use Type Encoding, Type Transformation, QuickCheck, and immutability to add or change data without having to recondition it.

# Type Level Encoding

Put it where you can find it...

Create types to exactly match some set of properties.

e.g. ...

```
newtype FixedText (lengthMax :: Nat)    -- Max text length
                  (lengthMin :: Nat)    -- Min text length
                  (regex      :: Symbol) -- What characters are allowable
    = FixedText { _unMyText :: Text}
deriving (Show,Ord,Eq)
```

# Correct By Construction!

```
-- Guarantees FixedText will have constrained length
-- and valid characters.
fixedTextFromString :: forall max min regex .
  ( KnownNat    max
  , KnownNat    min
  , KnownSymbol regex) =>
  String ->
  Either FixedTextErrors (FixedText max min regex)

fixedTextFromString str = final
  where
    max'      = fromIntegral $ natVal (Proxy :: Proxy max)
    min'      = fromIntegral $ natVal (Proxy :: Proxy min)
    isTooLittle = length str < min'
    regexStr   = symbolVal (Proxy :: Proxy regex)
    trimmedString = take max' str
    notRegex   = notValidRegex regexStr trimmedString
    final
      | isTooLittle = Left   FixedTextErrorMin
      | notRegex    = Left   (FixedTextErrorRegex regexStr trimmedString)
      | otherwise   = Right . FixedText .   pack $ trimmedString
```

# Examples!

```
-- Just works, example
```

```
exampleFixedText :: Either FixedTextErrors (FixedText 30 1 "[[:alnum:]]")
```

```
exampleFixedText = fixedTextFromString "exampleText1234"
```

```
$> exampleFixedText
```

```
Right (FixedText {_unFixedText = "exampleText1234"})
```

```
-- Cut off too much input.
```

```
exampleOverflowProtection :: Either FixedTextErrors (FixedText 10 1 "[[:alnum:]]")
```

```
exampleOverflowProtection = fixedTextFromString "exampleText1234"
```

```
$> exampleOverflowProtection
```

```
Right (FixedText {_unFixedText = "exampleTex"})
```

```
-- Reject if invalid char
```

```
exampleInvalidChar :: Either FixedTextErrors (FixedText 30 1 "[[:digit:]]")
```

```
exampleInvalidChar = fixedTextFromString "exampleNotAllDigits"
```

```
$> exampleInvalidChar
```

```
Left (FixedTextErrorRegex "[[:digit:]]" "exampleNotAllDigits")
```



# Make Mine a Monoid

```
-- Monoid instance with 0 minimum.
-- No FixedText besides one that has a minimum size of zero
-- should be a Monoid.
instance (KnownNat max, KnownSymbol regex) =>
  Monoid (FixedText (max::Nat) (0::Nat) (regex::Symbol)) where
  mempty = FixedText ""
  mappend s1@(FixedText str1) (FixedText str2) =
    either (const s1)
           id
           (fixedTextFromText (str1 <> str2))
```

# Arbitrary, but really specific

```
-- Arbitrary instance
-- This arbitrary instance takes advantage of
-- the Monoid defined above
instance ( KnownNat      max
          , KnownSymbol  regex ) =>
  Arbitrary (FixedText max 0 regex) where

  arbitrary = let regexStr      = symbolVal (Proxy :: Proxy regex)
               generatedString = Genex.genexPure [regexStr]

               in either (const mempty) id <$>
                 QuickCheck.elements
                   (fixedTextFromString <$>
                     generatedString)
```

# Finally, a Property Test!

```
qcProps :: TestTree
qcProps = testGroup "FixedText properties"
  [ QC.testProperty "((empty <> str) == str)"           leftIdMonoid
  , QC.testProperty "((str <> empty) == str)"           rightIdMonoid
  , QC.testProperty "(strA <> strB) <> strC == strA <> (strB <> strC)" associativityMonoid
  ]

type ExampleFixedText = FixedText 10 0 "[01233456789]{0,3}"
leftIdMonoid :: ExampleFixedText -> Bool
leftIdMonoid str = ((empty <> str) == str)

rightIdMonoid :: ExampleFixedText -> Bool
rightIdMonoid str = ((str <> empty) == str)

associativityMonoid :: ExampleFixedText ->
  ExampleFixedText ->
  ExampleFixedText -> Bool
associativityMonoid strA strB strC = leftAsc == rightAsc
  where
    leftAsc  = (strA <> strB) <> strC
    rightAsc = strA <> (strB <> strC)
```

# And the Results...

Progress: 1/2Tests

Properties

FixedText properties

```
((mempty <> str) == str): OK
+++ OK, passed 100 tests.
((str <> mempty) == str): OK
+++ OK, passed 100 tests.
(strA <> strB) <> strC == strA <> (strB <> strC): OK (0.01s)
+++ OK, passed 100 tests.
```

Victory!

# Our Product Record

```
data Product = Product {
  productNumber      :: ProductNumber,
  productName        :: ProductName,
  version            :: ProductVersion,
  productCustomer    :: Customer,
  productDescription :: TText }
  deriving (Eq,Ord,Show,Generic)

data Customer = Customer {
  customerName      :: CustomerName,
  customerNumber    :: CustomerNumber,
  customerAddress   :: CustomerAddress }
  deriving (Eq,Ord,Show,Generic)

data CustomerAddress = CustomerAddress {
  street :: TText,
  city   :: TText,
  state  :: State}
  deriving (Eq,Ord,Show,Generic)

data State = Oklahoma | Texas | Kansas
  deriving (Eq,Ord,Show,Generic)
```

```
-- | Base fields
newtype ProductNumber = ProductNumber
{ unProductNumber :: TText}
  deriving (Eq,Ord,Show,Generic)

newtype ProductName    = ProductName
{ unProductName       :: TText}
  deriving (Eq,Ord,Show,Generic)

newtype ProductVersion = ProductVersion
{ unProductVersion    :: TText}
  deriving (Eq,Ord,Show,Generic)

newtype CustomerName   = CustomerName
{ unCustomerName      :: TText}
  deriving (Eq,Ord,Show,Generic)

newtype CustomerNumber = CustomerNumber
{ unCustomerNumber    :: TText}
  deriving (Eq,Ord,Show,Generic)

-- 140 characters alphanumeric unicode
type TText = FixedText 140 0 "[[:alnum:]]"
```

# Separate Types Combined

Use the types to construct a document.

```
data ProductDocument = ProductDocument !Product !Customer !CustomerAddress
  deriving(Eq,Ord,Show,Generic)
```

# Product as a Row

```
data ProductRow = ProductRow
{ rowProductNumber  :: ProductNumber,
  rowName            :: ProductName,
  rowVersion         :: ProductVersion,
  rowDescription     :: TText,
  rowCustomerName    :: CustomerName,
  rowCustomerNumber  :: CustomerNumber,
  rowCustomerStreet  :: TText,
  rowCustomerCity    :: TText,
  rowCustomerState   :: State }
deriving(Eq,Ord,Show,Generic)
```

# Isomorphism

```
toProductRow :: Product -> ProductRow
toProductRow Product {..} = ProductRow productNumber productName version productDescription
                                     customerName customerNumber street city state

where
    Customer {..} = productCustomer
    CustomerAddress {..} = customerAddress

fromProductRow :: ProductRow -> Product
fromProductRow (ProductRow {..}) = Product rowProductNumber rowName rowVersion customer rowDescription
where
    customer = Customer rowCustomerName rowCustomerNumber customerAddress
    customerAddress = CustomerAddress rowCustomerStreet rowCustomerCity rowCustomerState
```



# Arbitrary Generation

```
instance Arbitrary ProductVersion where  
  arbitrary = genericArbitrary
```

```
instance Arbitrary ProductNumber where  
  arbitrary = genericArbitrary
```

```
instance Arbitrary ProductName where  
  arbitrary = genericArbitrary
```

```
instance Arbitrary Product where  
  arbitrary = genericArbitrary
```

```
instance Arbitrary CustomerName where  
  arbitrary = genericArbitrary
```

```
...
```

# Most Code is Mutable

```
rowDocumentIsoTest :: Product -> Bool
rowDocumentIsoTest prod = ((== prod) .
                           fromProductRow .
                           toProductRow   ) prod
```

Product Properties

```
round-trip test Product:
+++ OK, passed 100 tests.
```

OK

# Can we do better?

- Proofs not Tests!
- A way to deal with more at the type level.

```
{-@ measure notEmpty @-}
```

```
notEmpty [] = False  
notEmpty _ = True
```

```
{-@ type NotEmptyList a = {xs:[a] | notEmpty xs } @-}  
{-@ headSafe :: NotEmptyList a -> a @-}
```

```
headSafe (x:xs) = x
```

```
tryToUseHeadSafeUnsafely = headSafe []
```