## True Lies and False Truths

On the importance of evidence in programming

Andres Löh NWERC, 2021-03-27



#### isValidEmail :: String -> Bool

Checks if an email address is (syntactically) valid.



```
isValidEmail :: String -> Bool
```

Checks if an email address is (syntactically) valid.

```
someCode = do
txt <- readInput
if isValidEmail txt
then someOtherFunction txt
else ...</pre>
```



#### isValidEmail :: String -> Bool

Checks if an email address is (syntactically) valid.

```
someCode = do
txt <- readInput
if isValidEmail txt
then someOtherFunction txt
else ...</pre>
```

someOtherFunction :: String -> ...
someOtherFunction email =

```
...
<mark>sendEmailTo</mark> email message
```

• • •



#### isValidEmail :: String -> Bool

Checks if an email address is (syntactically) valid.

```
someCode = do
txt <- readInput
if isValidEmail txt
then someOtherFunction txt
else ...</pre>
```

someOtherFunction :: String -> ...
someOtherFunction email =

```
sendEmailTo email message
```

. . .

Did we already validate this?



```
isValidEmail :: String -> Bool-
```

Checks if an email address is (syntactically) valid.

```
someCode = do
txt <- readInput
if isValidEmail txt
then someOtherFunction txt
else ...</pre>
```

someOtherFunction :: String -> ...
someOtherFunction email =

```
sendEmailTo email message
```

. . .

Did we already validate this?

We haven't actually learned anything here ...



## Booleans considered harmful

# data Maybe a = Nothing -- signals failure Just a -- signals success, carries evidence



data Maybe a =
 Nothing -- signals failure
 Just a -- signals success, carries evidence

isValidEmail :: String -> Bool



data	Maybe a	=
	Nothing	signals failure
I	Just a	signals success, carries evidence

isValidEmail :: String -> Maybe Email

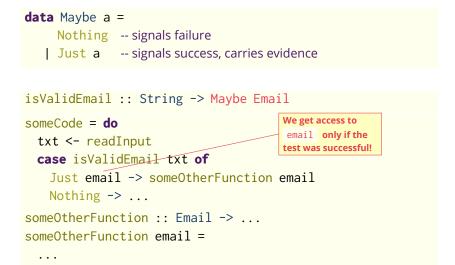


. . .

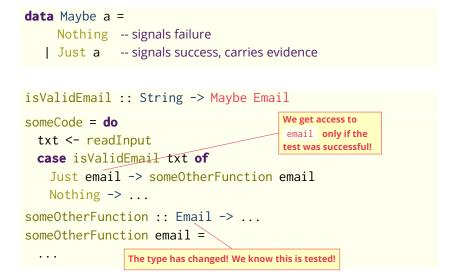
data May	be a =	=
Notl	hing	signals failure
Jus	ta ·	signals success, carries evidence

```
isValidEmail :: String -> Maybe Email
someCode = do
  txt <- readInput
  case isValidEmail txt of
   Just email -> someOtherFunction email
   Nothing -> ...
someOtherFunction :: Email -> ...
someOtherFunction email =
```











#### From Booleans to evidence

#### Before

- ► Test outcome is **Bool**.
- ► To the type system, False and True are interchangeable.
- Easy to forget a test.
- Easy to run a test unnecessarily often.



### From Booleans to evidence

#### Before

- ► Test outcome is Bool .
- ► To the type system, False and True are interchangeable.
- Easy to forget a test.
- Easy to run a test unnecessarily often.

#### After

- Test outcome is Maybe something.
- Successful outcome provides evidence.
- We don't get the evidence if the test fails.
- Functions requiring the evidence can rely on the test having succeeded.



We have used the type <a>Email</a> , but have not defined it ...

- Defining a test that produces a Bool is easy.
- Is providing evidence equally easy?



We have used the type <a>Email</a> , but have not defined it ...

- Defining a test that produces a Bool is easy.
- Is providing evidence equally easy?

We will revisit this question later and first look at other examples.



### Level 2 – Summing elements

```
sumList :: List Int -> Int
sumList list =
    if isEmpty list
    then 0
    else head list + sumList (tail list)
```

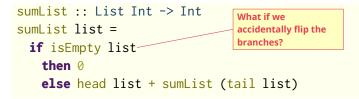


```
sumList :: List Int -> Int
sumList list =
    if isEmpty list
    then 0
    else head list + sumList (tail list)
```

isEmpty :: List a -> Bool -- tests if a list is empty head :: List a -> a -- first element of a non-empty list tail :: List a -> List a -- other elements of a non-empty list



### Level 2 – Summing elements



```
isEmpty :: List a -> Bool -- tests if a list is empty
head :: List a -> a -- first element of a non-empty list
tail :: List a -> List a -- other elements of a non-empty list
```



isEmpty :: List a -> Maybe (a, List a)



#### isEmpty :: List a -> Maybe (a, List a)

Suitable evidence for a non-empty list is exactly the head and the tail!



```
isEmpty :: List a -> Maybe (a, List a)
```

Suitable evidence for a non-empty list is exactly the head and the tail!

```
sumList :: List Int -> Int
sumList list =
    case isEmpty list of
    Nothing    -> 0
    Just (hd, tl) -> hd + sumList tl
```



```
isEmpty :: List a -> Maybe (a, List a)
```

Suitable evidence for a non-empty list is exactly the head and the tail!

```
sumList :: List Int -> Int
sumList list =
    case isEmpty list of
    Nothing -> 0
    Just (hd, tl) -> hd + sumList tl
```

Flipping the cases is no longer possible!

No more potential crashes!



## Secret passage – The definition of lists

```
In Haskell, lists are in fact defined as
data List a =
    Nil
    Cons a (List a)
```



```
In Haskell, lists are in fact defined as
data List a =
    Nil
    Cons a (List a)
sumList :: List Int -> Int
sumList list =
    case list of
    Nil         -> 0
    Cons hd tl -> hd + sumList tl
```



```
In Haskell, lists are in fact defined as
data List a =
    Nil
    Cons a (List a)
sumList :: List Int -> Int
sumList list =
    case list of
    Nil         -> 0
    Cons hd tl -> hd + sumList tl
```

"Sum types" and "pattern matching" are powerful concepts!



```
filter :: (a -> Bool) -> List a -> List a
filter f list =
  case list of
  Nil         -> Nil
   Cons hd tl ->
        if f hd
        then ...
        else ...
```



```
filter :: (a -> Bool) -> List a -> List a
filter f list =
  case list of
  Nil         -> Nil
   Cons hd tl ->
       if f hd
       then ...
       else ...
```

Do we actually want the elements that pass or fail the test?





```
filter :: (a -> Maybe b) -> List a -> List b
filter f list =
  case list of
  Nil         -> Nil
  Cons hd tl ->
      case f hd of
      Just ev -> Cons ev (filter f tl)
      Nothing -> filter f tl
```



```
even :: Int -> Bool
even i = mod i 2 == 0
filterEvens :: List Int -> List Int
filterEvens list = filter even list
```



```
data Even = Twice Int
even :: Int -> Maybe Even
even i =
    case divMod i 2 of
    (j, 0) -> Just (Twice j)
    _ -> Nothing
```



```
data Even = Twice Int
even :: Int -> Maybe Even
even i =
   case divMod i 2 of
    (j, 0) -> Just (Twice j)
    _ -> Nothing
```

Examples:

even 42 = Just (Twice 21)
even 17 = Nothing



```
data Even = Twice Int
even :: Int -> Maybe Even
even i =
   case divMod i 2 of
    (j, 0) -> Just (Twice j)
    _ -> Nothing
```

Examples:

even 42 = Just (Twice 21)
even 17 = Nothing

filterEvens :: List Int -> List Even
filterEvens list = filter even list



```
data Even = Twice Int
even :: Int -> Maybe Even
even i =
   case divMod i 2 of
    (j, 0) -> Just (Twice j)
    _ -> Nothing
```

Examples:

even 42 = Just (Twice 21)
even 17 = Nothing

The type is now much more informative!

```
filterEvens :: List Int -> List Even-
filterEvens list = filter even list
```



```
data Odd = TwicePlusOne Int
parity :: Int -> Either Even Odd
parity i =
   case divMod i 2 of
    (j, 0) -> Left (Twice j)
    (j, 1) -> Right (TwicePlusOne j)
```



```
data Odd = TwicePlusOne Int
parity :: Int -> Either Even Odd
parity i =
    case divMod i 2 of
    (j, 0) -> Left (Twice j)
    (j, 1) -> Right (TwicePlusOne j)
```

#### **data** Either a b =

Left a -- denotes one outcome, carries evidence Right b -- denotes the other outcome, carries evidence



Haskell has: partition :: (a -> Bool) -> (List a, List a)



```
Haskell has:
partition :: (a -> Bool) -> (List a, List a)
```

```
Better would be
partition :: (a -> Either b c) -> (List b, List c)
```



Let's revisit the beginning: isValidEmail :: String -> Maybe Email

What is suitable evidence for having performed a complex validation?



## Cheating is allowed – Lightweight evidence

data Email = MkEmail String

We don't need actual evidence – an abstract type is often enough ...



## Cheating is allowed – Lightweight evidence

#### data Email = MkEmail String

We don't need actual evidence – an abstract type is often enough ...

- Email is isomorphic to String, but a different type.
- We have full control over the interface of Email.
- E.g., most <u>String</u> operations do not make sense on <u>Email</u> at all (and we do not need to introduce them).
- We can make sure that the only way to produce a value of type
   Email is isValidEmail , which isolates the risky code to one function.
- ► No danger of forgetting the test, no danger of duplicating the test.



- Whenever possible, replace Booleans with types that carry evidence!
- Introduce new types for values that have passed tests, even if their internal representation has not changed!



- Whenever possible, replace Booleans with types that carry evidence!
- Introduce new types for values that have passed tests, even if their internal representation has not changed!

### Avoid:

- Functions with implicit assumptions about their inputs.
- Unclear boundaries between untrusted and trusted values.
- Situations where switching cases would not make the type-checker complain!



The idea presented here is old:

- I learned it from Conor McBride around 2005 under the slogan
   Learning by testing.
- Bob Harper wrote a famous blog post titled Boolean Blindness about this topic in 2011.
- Alexis King wrote another influential blog post titled Parse, don't validate in 2019.

But under yet again different names, the idea certainly goes back even further than that.



In dependently typed languages (Agda, Idris, Coq, Lean, ...), we can provide "proper" evidence for far more interesting properties:



In dependently typed languages (Agda, Idris, Coq, Lean, ...), we can provide "proper" evidence for far more interesting properties: (==) :: a -> a -> Bool



In dependently typed languages (Agda, Idris, Coq, Lean, ...), we can provide "proper" evidence for far more interesting properties: (==) :: a -> a -> Bool

Better:

(==) :: (x : a) -> (y : a) -> Either (x = y) (x  $\neq$  y)



# Questions?

## andres@well-typed.com