True Lies and False Truths

On the importance of evidence in programming

Andres Löh

NWERC, 2021-03-27

Well-Typed
The Haskell Consultants

Level 1 – A simple test

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

Checks if an email address is (syntactically) valid.

Level 1 – A simple test

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

Checks if an email address is (syntactically) valid.

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

Level 1 – A simple test

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

Checks if an email address is (syntactically) valid.

```haskell
someCode = do
  txt <- readInput
  if isValidEmail txt
    then someOtherFunction txt
    else ...
```

```haskell
someOtherFunction :: String -> ...
someOtherFunction email =
  ...
  sendEmailTo email message
  ...
```

Level 1 – A simple test

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

Checks if an email address is (syntactically) valid.

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

```
someOtherFunction :: String -> ...
someOtherFunction email =
  ...
  sendEmailTo email message
  ...
```

Did we already validate this?

Well-Typed

Level 1 – A simple test

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

We haven't actually learned anything here …

Checks if an email address is (syntactically) valid.

```haskell
someCode = do
  txt <- readInput
  if isValidEmail txt
    then someOtherFunction txt
    else ...
```

```haskell
someOtherFunction :: String -> ...
someOtherFunction email =
  ...
  sendEmailTo email message
  ...
```

Did we already validate this?

Booleans considered harmful

Provide evidence!

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

Provide evidence!

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

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

Provide evidence!

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

```
isValidEmail :: String -> Maybe Email
```

Provide evidence!

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

```haskell
isValidEmail :: String -> Maybe Email
```

```haskell
someCode = do
  txt <- readInput
  case isValidEmail txt of
    Just email -> someOtherFunction email
    Nothing -> ...

someOtherFunction :: Email -> ...
someOtherFunction email =
  ...
```

Provide evidence!

```
data Maybe a =
     Nothing   -- signals failure
   | Just a    -- 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 =
  ...
```

We get access to `email` only if the test was successful!

Provide evidence!

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

```haskell
isValidEmail :: String -> Maybe Email
```

```haskell
someCode = do
  txt <- readInput
  case isValidEmail txt of
    Just email -> someOtherFunction email
    Nothing -> ...

someOtherFunction :: Email -> ...
someOtherFunction email =
  ...
```

We get access to `email` only if the test was successful!

The type has changed! We know this is tested!

Well-Typed

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.

Boss key needed – What is an email?

We have used the type `Email`, but have not defined it . . .

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

Boss key needed – What is an email?

We have used the type `Email` , 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)
```

Level 2 – Summing elements

```
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

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

What if we accidentally flip the branches?

```
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
```

Provide evidence!

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

Provide evidence!

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

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

Provide evidence!

```
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
```

Provide evidence!

```
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!

Well-Typed

Secret passage – The definition of lists

In Haskell, lists are in fact defined as

```
data List a =
     Nil
   | Cons a (List a)
```

Secret passage – The definition of lists

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
```

Secret passage – The definition of lists

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!

Level 3 – Filtering a list

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

Level 3 – Filtering a list

```
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?

Level 3 – Filtering a list

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

Provide evidence!

```
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
```

Time trial – Filtering even elements

```haskell
even :: Int -> Bool
even i = mod i 2 == 0
```

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

Provide evidence!

```haskell
data Even = Twice Int

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

Provide evidence!

```haskell
data Even = Twice Int

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

Examples:

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

Provide evidence!

```haskell
data Even = Twice Int

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

Examples:

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

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

Provide evidence!

```
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
```

Easter egg – Evidence of failure

```
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)
```

Easter egg – Evidence of failure

```haskell
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)
```

```haskell
data Either a b =
     Left  a   -- denotes one outcome, carries evidence
   | Right b   -- denotes the other outcome, carries evidence
```

Partitioning

Haskell has:

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

Partitioning

Haskell has:

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

Better would be

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

Boss – Evidence for complex tests

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

```haskell
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 `String` operations do not make sense on `Email` 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.

How to win

- 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!

How to win

- 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!

Credits

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.

Bonus level – Dependent types

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

Bonus level – Dependent types

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

```
(==) :: a -> a -> Bool
```

Bonus level – Dependent types

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 ≠ y)
```

Well-Typed

Questions?

andres@well-typed.com