Types and Testing in Haskell

Daniel Patterson

Position Development
positiondev.com

https://dbp.io/static/types-testing-haskell-meetup-2014.pdf

Who am I?

- Using Haskell since late 2007, a lot of that web stuff (early stuff like HAppS, Gershom's hvac, etc)
- Current work - Position Development - Haskell web development shop - positiondev.com
- Wrote `hspec-snap` this year.
- Like types and testing and Haskell, so why not talk about types and testing in Haskell.

Introduction, why test?

- "If it type checks, it's correct."
- "Tests are important in languages like Ruby..."
- But – *"[Types] say something about the function but not so much that you can't fit it in your head at one time... [Types] are a little bit weak, precisely so that they can be crisp"* – Simon Peyton Jones (in Coders at Work)

Point of this talk

1. Tests in Haskell are critical for real-world reliable code. It's important to isolate code that is underspecified by types (and this code is often impure) from the parts of your code that encode logic and often have much more descriptive types.
2. Testing is more fun in Haskell, because types let us focus. Testing drives you towards using pure functions, as they are much easier to work with. The type system also sometimes pushes you this way.
3. Types and tests are not two distinct approaches to better software - they're deeply connected.
4. But let's talk about some code...

Quicksort... you may have seen quicksort.

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) =
      quicksort (filter (<= x) xs)
      ++ [x]
      ++ quicksort (filter (>= x) xs)

> quicksort [1,1,2,3]
[1,1,1,2,3]
```

Quicksort... you may have seen quicksort.

```haskell
quicksort' :: [Int] -> [Int]
quicksort' [] = []
quicksort' (x:xs) =
      quicksort' (filter (< x) xs)
      ++ [x]
      ++ quicksort' (filter (> x) xs)

> quicksort' [1,1]
[1]
```

Quicksort... you may have seen quicksort.

```
quicksort'' :: [Int] -> [Int]
quicksort'' [] = []
quicksort'' (x:xs) =
      quicksort'' (filter (<= x) xs)
      ++ quicksort'' (filter (>= x) xs)

> quicksort'' [1,2,3]
[]
```

Quicksort... you may have seen quicksort.

```
quicksort''' :: [Int] -> [Int]
quicksort''' [] = []
quicksort''' (x:xs) =
      quicksort''' (filter (< x) xs)
      ++ quicksort''' (filter (>= x) xs)

> quicksort''' [1,2,3]
[]
```

Web handler example

That may seem like a toy example... But those functions all type checked.

And it's easy to find examples with similar problems in any real code-base.

```
h = do token <- getLastUserToken uid
       case tokenCharge token of
         Nothing -> ...
         Just charge -> do
           toks <- getUserTokensWithCharge uid charge
           case length toks of
             4 ->
               do ref <- refundChargeById charge
                  ...
             _ -> ...
```

Web handler example

Problem: functions with different behavior can have the same type.

```
getUserTokensWithCharge ::
  UserId -> ChargeId -> AppHandler [Token]
getUnfulfilledUserTokensWithCharge ::
  UserId -> ChargeId -> AppHandler [Token]
```

Since tests are examples of how code should behave, this problem isn't, well, a problem. If the test passes, we know it exhibits the demonstrated behavior.

Web handler example

This particular bug was caught by the following test:

```
it "shouldn’t cancel a partly fulfilled sub." (do
  subscriber <- create id
  create id :: HspecApp Issue
  loginAs subscriber (post "/cancel" (params []))
  eval tokenCount >>= shouldEqual 4)
```

As if it was cancelled, the tokens would have been deleted.

That’s a test using `hspec` and my `hspec-snap` package (both on Hackage).

- We create some example data (a subscriber and an issue)
- Then as the subscriber try to cancel the subscription
- Finally, we assert that the tokens are still in the system

Introduction - limit of types

What we are starting to tease out (and **spoiler**: this is the whole point of this talk) is that **plenty of important behavior is not feasible to adequately express in types**.

Let's make that more concrete.

We have functions like the following:

```
id :: a -> a
id = undefined
fst :: (a,b) -> a
fst = undefined
```

Where there is only one definition that type checks:

```
id :: a -> a
id a = a
fst :: (a,b) -> a
fst (a,_) = a
```

Introduction - limit of types

But we don't have to get far to find a function like:

```haskell
not :: Bool -> Bool
not = undefined
```

Which has four (well-typed) implementations, only one of which seems to be the expected (ie, correct) one:

```haskell
not True = True -- not = id
not False = False

not True = False -- not = const False
not False = False

not True = True -- not = const True
not False = True

not True = False
not False = True
```

Types restrict the space of possible implementations

Which isn't to say I don't think types are incredibly useful. Indeed, they are what makes programming in Haskell so special!

And there are a lot of things we can do to make types do more.

Parametricity helps, as a function with this type:

```
f :: [a] -> [a]
```

is harder to write incorrectly than one with:

```
f :: [String] -> [String]
```

In this case, the type variable means that we cannot inspect or synthesize new list elements

Types restrict the possible ways that functions are used

Even more, when we start to think about our entire systems, we are as concerned that we *use* certain pieces correctly as that we write those pieces correctly in the first place.

Writing code from scratch correctly is a lot easier than extending existing systems.

(If that's all you do, you probably don't need to bother with tests)

Types restrict the possible ways that functions are used

For example:

```
addOrder :: CustomerId -> ProductId -> UTCTime -> IO Res
addOrder = undefined
```

Is less bug prone than:

```
addOrder :: Int -> Int -> UTCTime -> IO Bool
addOrder = undefined
```

Even when the body looks like:

```
addOrder :: CustomerId -> ProductId -> UTCTime -> IO Res
addOrder (CustomerId c) (ProductId p) t = undefined
```

Tests rule out invalid points in that space

So we have some ideas of how we can use types to help us write reliable* code (we'll come back to more).

Tests function similarly, but very specifically - they either rule out specific properties of bad implementations, or they restrict to a set that includes good implementations (but might include other things as well). Like types, they are often a crisp explanation of how code should work.

* Which encompasses not only 'correctness' but also, resilience in the case of failure, etc.

TDD in Haskell

That's a lot of talk. Let's make this more practical. Our pattern:

1. **Figure out the types.** Writing code without types is usually not very useful. If you need to do some experimentation to figure out your types, by all means do that first, but then come back here.
2. **Write out the primary functions, leaving all implementations as `undefined`.** If some functions are obvious wrappers of others, put those calls in. Get this to type check (if you can't, go to 1).
3. **Write tests against those functions.** Remember, we aren't trying to be exhaustive, but we are trying to pick representative points in the space. Edge cases are important. Get this to type check (if you can't, go to 1).
4. **Fill in the implementations.** When our code has no `undefined`, we're done.

Tree flattening

As a small example, we'll write code to convert a binary tree to a flat representation.

```haskell
import Test.Hspec
```

1. (Types)

```haskell
data Tree = Node Tree Tree
          | Leaf Int deriving (Show, Eq)
data TFlat = NStart | NEnd
           | NLeaf Int deriving (Show, Eq)
```

2. (Functions)

```haskell
flatten :: Tree -> [TFlat]
flatten = undefined
```

Tree flattening

4. (Tests)

```haskell
flattenTo a b =
  it ("flatten " ++ show a ++ " = " ++ show b)
     (flatten a `shouldBe` b)
main = hspec (do
  Leaf 10 `flattenTo` [NLeaf 10]
  Node (Leaf 1) (Leaf 2) `flattenTo`
    [NStart, NLeaf 1, NLeaf 2, NEnd]
  Node (Node (Leaf 1) (Leaf 2)) (Leaf 3) `flattenTo`
    [ NStart, NStart, NLeaf 1, NLeaf 2, NEnd
            , NLeaf 3, NEnd]
  Node (Leaf 1) (Node (Leaf 2) (Leaf 3)) `flattenTo`
    [ NStart, NLeaf 1
            , NStart, NLeaf 2, NLeaf 3, NEnd
    , NEnd])
```

Tree flattening

```
$ runghc flattening.hs

- flatten Leaf 10 = [NLeaf 10] FAILED [1]
- flatten Node (Leaf 1) (Leaf 2) = [NStart, ... FAILED [2]
- flatten Node (Node (Leaf 1) (Leaf 2)) ... FAILED [3]
- flatten Node (Leaf 1) (Node (Leaf 2) ... FAILED [4]
1) flatten Leaf 10 = [NLeaf 10]
uncaught exception: ErrorCall (Prelude.undefined)
2) flatten Node (Leaf 1) (Leaf 2) = [NStart,...
uncaught exception: ErrorCall (Prelude.undefined)
3) flatten Node (Node (Leaf 1) (Leaf 2)) ...
uncaught exception: ErrorCall (Prelude.undefined)
4) flatten Node (Leaf 1) (Node (Leaf 2) ...
uncaught exception: ErrorCall (Prelude.undefined)
...
4 examples, 4 failures
```

Tree flattening

5. (Implementation)

```
flatten :: Tree -> [TFlat]
flatten (Leaf n) = [NLeaf n]
flatten (Node left right) =  [NStart]
                          ++ flatten left
                          ++ flatten right
                          ++ [NEnd]
```

Tree flattening

```
$ runghc flattening.hs

: - flatten Leaf 10 = [NLeaf 10]
: - flatten Node (Leaf 1) (Leaf 2) = [NStart,NLeaf 1,NLeaf
: - flatten Node (Node (Leaf 1) (Leaf 2)) (Leaf 3) =
    [NStart,NStart,NLeaf 1,NLeaf 2,NEnd,NLeaf 3,NEnd]
: - flatten Node (Leaf 1) (Node (Leaf 2) (Leaf 3)) =
    [NStart,NLeaf 1,NStart,NLeaf 2,NLeaf 3,NEnd,NEnd]
:
: Finished in 0.0042 seconds
: 4 examples, 0 failures
```

Tree flattening

At the base level, **that's all there is to it**. It gets a little more complicated for impure code, and you can get lots of value out of property based testing like `QuickCheck`, but you can start writing tests of your code today as easily as that example!

Summary (of part 1)

1. Testing in Haskell is still important (if you want reliable code that you can continue to improve). But, it works a little differently than in languages with weaker type systems.
2. Types should always be primary. Do what you can reasonably do with them, because they are guaranteed to be consistent, and are (almost always) very lightweight.
3. Test before you write implementations (`undefined` is your friend!). It's all too easy to have tests that are telling you nothing. If the test fails, and then you write the code, and then it passes, you know the test measured something.

Questions

Questions thus far?

Part 2: Where do types and tests meet?

So how much of each do we need, or should we have?

We can (almost) always make code have stronger types.

- In limit, hard to do, and there may be bugs in types!

We can always add more tests.

- In limit, exhaustively express inputs and outputs - will be unmaintainable!

Both extremes seem non-ideal. We want to be somewhere in the middle.

Boundary vs Internal code

- **Boundary** code exists in interfaces with the outside world, which is usually relatively untyped. It necessarily has relatively unexpressive types. For example:

  ```
  parse :: Text -> Either ErrorMessage Value
  parse = undefined
  ```

  Functions like this need to be tested. It's the only way to ensure they are behaving at all sensibly.
- You are already testing this stuff.
- Our goal for these functions is to make them as small as possible.

Boundary vs Internal code

Another example of this type of code are functions in monads that allow you to do lots of things that aren't needed.

```
mkCalendar :: Date -> Handler App App Calendar
```

Which is in a Snap web monad, and may:

- look up GET or POST parameters
- read from the database
- write out response body or headers
- short-circuit the whole handler (ie, 404, sort of)

Since we may not expect it to do most of those things, this function is underspecified by its types.

```
mkCalendar :: Date -> [Day] -> Calendar
```

Is a lot clearer, as we know it can't depend on anything outside of it's arguments (It's also a *lot* easier to test pure functions like this).

Boundary vs Internal code

- **Internal** code, on the other hand, can (almost always) be given more descriptive / restrictive types.
  As an example, say we have a type:

  ```
  data Value = N Int | B Bool | T Text
  ```

  And we want to display it on a web page. This means we could have a function:

  ```
  render :: Value -> Text
  render = undefined
  ```

Boundary vs Internal code

```haskell
render :: Value -> Text
render = undefined
```

- But escaping is tricky..
- better to separate it, at the type level:

  ```haskell
  escape :: Value -> SafeValue
  data SafeValue = ...
  render :: SafeValue -> Text
  ```

- could use phantom types, etc.

How this relates to tests

Our main thesis is that **we want to use tests to cover functionality that is unspecified (or weakly specified) in types**.

This means that boundary code needs to be tested, always, and pretty thoroughly. Internal code, on the other hand, can often leverage the type system and need fewer tests.

In the last example, we still definitely need to test `escape`, as it could be passing the `Text` through unchanged. But maybe `render` is simple enough to no longer need tests, or at least, need fewer.

Types are global, tests are local

Tests assert properties about what's going on within a given function.

- Take a function like:

```
f x = ...
```

- Tests for it assert that `f` was written correctly; they don't say anything about how `f` is used.

Types are global, tests are local

This is a big difference between types and tests. If we type `f` as:

```haskell
f :: Foo -> Bar
```

We assert some about how `f` is written (but often, not a whole lot), but also how it can be used.

- Example: not expose the constructors for `Bar` - restrict what you can do with output.
- Example: only expose certain ways of constructing `Foo` - restrict how input can be made.

Making code easier to test

One complaint is that “real code” is hard to test. And talking about little examples like tree flattening isn’t doing much to combat that.

So here’s a recent, real example:

- Putting upload job for an application into the background.
- Previously, the upload was processed within a single request. This limited the size significantly because of request timeouts.
- Once in the background, a lot is the same, but a new issue is how to inform the user.
- Decided on a simple scheme: was 'processing' until it either was marked done or a certain time period passed (at which point, it was deemed to have failed).

Making code easier to test

Isolated that into a pure function:

```haskell
data UploadState = UploadProcessing
                 | UploadDone
                 | UploadFailed
uploadStatus :: UTCTime -> Upload -> UploadState
uploadStatus now upload = undefined
```

Now we can test this. We generate some uploads (in memory, not in the database), and can write down what the states should be at each point.

Making code easier to test

```haskell
setT :: UTCTime -> NominalDiffTime -> Upload
setT t diff = def { uploadedAt = addUTCTime diff t }


it "should be in progress after being created" $
  uploadStatus now (setT now (-10))
    `shouldBe` UploadProcessing
it "should be failed after 5 hours" (
  uploadStatus now (setT now (- 5 * 3600))
    `shouldBe` UploadFailed)
it "should be finished if uploadFinishedAt /= Nothing" $
    ...
```

Making code easier to test

This isn't a huge example, but lots of code can be split into these small parts.

And that's kind of the point - pure functions are easy to test. **Whenever we have non-trivial logic, we should factor it out into pure functions**.

Another example - snaplet-wordpress

I've been writing code to build Snap frontends to Wordpress sites (https://github.com/dbp/snaplet-wordpress - but it's not done yet).

One part of this is turning JSON that we get from API endpoints into Heist splices that we can use in our applications. We want to, essentially, map a JSON tree onto a tree of nested tags, so that you can write:

```
<wpPosts limit=10 tags="+haskell,-experiment">
  <wpTitle/><br/>
  <wpContent/>
  <hr/>
</wpPost>
```

Another example - snaplet-wordpress

There are many different types of tests for this:

1. Given the proper cached values / http responses, the tags can result in the proper values.
2. The right queries are generated for given splices (so, for example, the example generates the proper query to get 10 posts with the tag `haskell` and without the tag `experiment`).
3. That various utility functions work - like adding in custom JSON tree to nested tag mappings, and transforming the hyphenated names into the `wp`-prefixed tags.

Another example - snaplet-wordpress

1. Given the proper cached values / http responses, the tags can result in the proper values.

We can make this testable by allowing the HTTP requester to be swapped out. It is, essentially, a `Text -> IO Text` function, with a default implementation that actually makes network requests.

Then, we can have it return the same response each time, and test only that the tags are set up properly.

Another example - snaplet-wordpress

3. The right queries are generated for given splices (so, for example, the example generates the proper query to get 10 posts with the tag `haskell` and without the tag `experiment`).

This can use the same requester swap, except in this case, the requester is storing what URLs where asked for, so that we can verify, in the test, that the right thing was queried for.

Another example - snaplet-wordpress

4. Various utility functions work - like adding in custom JSON tree to nested tag mappings, and transforming the hyphenated names into the `wp`-prefixed tags.

These are just pure functions. While the code for the JSON tree merging code is somewhat complicated, the test cases are easy to write.

Harder to test?

Real world code is only hard to test when it isn't easy to isolate parts from one another.

Haskell gives us lots of tools to make it easy to do this sort of isolation.

This usually looks the same as factoring out pure functions, or splitting functions into smaller pieces with more expressive type boundaries.

Which means that testing drives us towards more idiomatically Haskell code.

Questions

Questions?

https://dbp.io/static/types-testing-haskell-meetup-2014.pdf

Links Mentioned:
http://hackage.haskell.org/package/hspec-snap
https://github.com/dbp/snaplet-wordpress