A List of Haskell Articles on good design, good testing

William Yao
Matt Parsons
David Luposchainsky
Alexis King
Jasper Van der Jeugt
Tom Ellis
Michael Snoyman
Sandy Maguire
Oskar Wickstrm
Scott Wlaschin
Hillel Wayne

0e49238b23659b50150cbde4f229ccada3c10e7b

The homepage for this document is: https://github.com/oswald2/haskell_articles

Contents

List of Figures

1. Introduction

William Yao created a link collection (see here and here [1]) of Haskell articles about design and testing. I did like this collection very much but wanted to have all of them in one document, with one formatting. So the idea of this document (which is now more like a book) was born.

I included Williams original comments to the articles at the beginning of each chapter. The articles itself were included as unchanged as possible. Sometimes the formatting needed to change, sometimes they referenced each other, sometimes LaTeX needed to be convinced to apply a certain formatting. Anyway, now it is done, so I wish all of you readers a lot of fun reading this document while hopefully learning something new.

Michael Oswald

2. Introduction by William Yao

For a language that's beloved for its ability to guide the structure of programs into being easier to understand, easier to maintain, and easier to get correct, there's not a lot of resources on how to best use the tools that Haskell provides. Lots of terms and buzzwords, not a lot of in-depth practical guidance on best practices. So I've put together a list of community blog posts and articles on the theme of *building more correct programs*.

These are split roughly in two groups: posts about how best to leverage the type system to eliminate errors before they even occur, while striking a balance between type complexity and usability; and posts about using Haskell's best-in-class testing facilities to shore up what can't be typed[1]. Despite the hype around Haskell's type system, it's unlikely in a program of any complexity that you'll be able to completely eliminate the possibility of bugs and errors purely through the type system alone. Using both types *and* tests gives you a better power-to-weight ratio for building maintainable, bug-free programs than either does alone.

Note that I'm explicitly not including articles and resources about basics or setup of the topics in question. Instead of "what is a `Maybe` and why use it," think "what are some typical patterns around using `Maybes` in a real codebase." Instead of "what is property-based testing", think "here are best practices around choosing properties to test." Since there are already plenty of good introductory resources on things like "how do I get started with QuickCheck", we'll focus here instead on how best to use the tools we have.

I'm fully aware that this is not a complete listing even of topics that Haskell programmers know about and regularly make use of[2]. If you feel there are articles missing from here that are clear, easy-to-understand, and go in-depth on how to use correctness-enforcing techniques, please let me know!

Found this useful? Still have questions? Talk to me!

Posts in each section are roughly ordered by difficulty.

1 Or at least, not typed easily

2 Tagged datatypes, for instance. It's easy to explain what they are, but I haven't seen good examples of how people use them in a real codebase.

Part I.

Posts on designing and structuring code

3. Type Safety Back and Forth - Matt Parsons

> ***William Yao:***
> *Essential reading. A very typical design technique of restricting what inputs your functions take to ensure that they can't fail.*
>
> *The most typical instances of using this are `NonEmpty` for lists with at least one element and `Natural` for non-negative integers. Oddly, I don't often see people do the same thing for other structures where this would be useful; for instance, nonempty `Vectors` or nonempty `Text`. Thankfully, it's easy enough to define yourself.*
>
> ```
> data NonEmptyVec a = NonEmptyVec a (Vector a)
>
> -- An additional invariant you might want to enforce is
> -- 'not all whitespace'
> data NonEmptyText = NonEmptyText Char Text
> ```
>
> *Original Article: [2]*

Types are a powerful construct for improving program safety. Haskell has a few notable ways of handling potential failure, the most famous being the venerable `Maybe` type:

```
data Maybe a
    = Nothing
    | Just a
```

We can use Maybe as the result of a function to indicate:

> *Hey, friend! This function might fail. You'll need to handle the `Nothing` case.*

This allows us to write functions like a safe division function:

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide i 0 = Nothing
safeDivide i j = Just (i `div` j)
```

I like to think of this as pushing the responsibility for failure forward. I'm telling the caller of the code that they can provide whatever `Int`s they want, but that some condition might cause them to fail. And the caller of the code has to handle that failure later on.

This is the easiest technique to show and tell, because it's one-size-fits-all. If your function can fail, just slap `Maybe` or `Either` on the result type and you've got safety. I can write a 35 line blog post to show off the technique, and if I were feeling frisky, I could use it as an introduction to `Functor`, `Monad`, and all that jazz.

Instead, I'd like to share another technique. Rather than push the responsibility for failure forward, let's explore pushing it back. This technique is a little harder to show, because it depends on the individual cases you might use.

If pushing responsibility forward means accepting whatever parameters and having the caller of the code handle possibility of failure, then *pushing it back* is going to mean we accept stricter parameters that we can't fail with. Let's consider `safeDivide`, but with a more lax type signature:

```
safeDivide :: String -> String -> Maybe Int
safeDivide iStr jStr = do
    i <- readMay iStr
    j <- readMay jStr
    guard (j /= 0)
    pure (i `div` j)
```

This function takes two strings, and then tries to parse `Int`s out of them. Then, if the `j` parameter isn't `0`, we return the result of division. This function is safe, but we have a much larger space of calls to `safeDivide` that fail and return `Nothing`. We've accepted more parameters, but we've pushed a lot of responsibility forward for handling possible failure.

Let's push the failure back.

```
safeDivide :: Int -> NonZero Int -> Int
safeDivide i (NonZero j) = i `div` j
```

We've required that users provide us a `NonZero Int` rather than any old `Int`. We've pushed back against the callers of our function:

> *No! You must provide a `NonZero Int`. I refuse to work with just any `Int`, because then I might fail, and that's annoying.*

So speaks our valiant little function, standing up for itself!

Let's implement `NonZero`. We'll take advantage of Haskell's `PatternSynonyms` language extension to allow people to pattern match on a "constructor" without exposing a way to unsafely construct values.

```
{-# LANGUAGE PatternSynonyms #-}

module NonZero
  ( NonZero()
  , pattern NonZero
  , unNonZero
  , nonZero
  ) where

newtype NonZero a = UnsafeNonZero a

pattern NonZero a <- UnsafeNonZero a

unNonZero :: NonZero a -> a
unNonZero (UnsafeNonZero a) = a
```

```
nonZero :: (Num a, Eq a) => a -> Maybe (NonZero a)
nonZero 0 = Nothing
nonZero i = Just (UnsafeNonZero i)
```

This module allows us to push the responsibility for type safety backwards onto callers.

As another example, consider head. Here's the unsafe, convenient variety:

```
head :: [a] -> a
head (x:xs) = x
head []     = error "oh no"
```

This code is making a promise that it can't keep. Given the empty list, it will fail at runtime.

Let's push the responsibility for safety forward:

```
headMay :: [a] -> Maybe a
headMay (x:xs) = Just x
headMay []     = Nothing
```

Now, we won't fail at runtime. We've required the caller to handle a `Nothing` case.

Let's try pushing it back now:

```
headOr :: a -> [a] -> a
headOr def (x:xs) = x
headOr def []     = def
```

Now, we're requiring that the *caller* of the function handle possible failure before they ever call this. There's no way to get it wrong. Alternatively, we can use a type for nonempty lists!

```
data NonEmpty a = a :| [a]

safeHead :: NonEmpty a -> a
safeHead (x :| xs) = x
```

This one works just as well. We're requiring that the calling code handle failure ahead of time.

A more complicated example of this technique is the justified-containers library. The library uses the type system to prove that a given key exists in the underlying Map. From that point on, lookups using those keys are total: they are guaranteed to return a value, and they don't return a Maybe.

This works even if you `map` over the `Map` with a function, transforming values. You can also use it to ensure that two maps share related information. It's a powerful feature, beyond just having type safety.

3.1. The Ripple Effect

When some piece of code hands us responsibility, we have two choices:

1. Handle that responsibility.
2. Pass it to someone else!

In my experience, developers will tend to push responsibility in the same direction that the code they call does. So if some function returns a `Maybe`, the developer is going to be inclined to also return a `Maybe` value. If some function requires a `NonEmpty Int`, then the developer is going to be inclined to also require a `NonEmpty Int` be passed in.

This played out in my work codebase. We have a type representing an `Order` with many `Items` in it. Originally, the type looked something like this:

```
data Order = Order  { items :: [Item] }
```

The `Items` contained nearly all of the interesting information in the order, so almost everything that we did with an `Order` would need to return a `Maybe` value to handle the empty list case. This was a lot of work, and a lot of `Maybe` values!

The type is *too permissive*. As it happens, an `Order` may not exist without at least one `Item`. So we can make the type *more restrictive* and have more fun!

We redefined the type to be:

```
data Order = Order { items :: NonEmpty Item }
```

All of the `Maybes` relating to the empty list were purged, and all of the code was pure and free. The failure case (an empty list of orders) was moved to two sites:

1. Decoding JSON
2. Decoding database rows

Decoding JSON happens at the API side of things, when various services `POST` updates to us. Now, we can respond with a `400` error and tell API clients that they've provided invalid data! This prevents our data from going bad.

Decoding database rows is even easier. We use an `INNER JOIN` when retrieving `Orders` and `Items`, which guarantees that each `Order` will have at least one `Item` in the result set. Foreign keys ensure that each `Item`'s `Order` is actually present in the database. This does leave the possibility that an `Order` might be orphaned in the database, but it's mostly safe.

When we push our type safety back, we're encouraged to continue pushing it back. Eventually, we push it all the way back – to the edges of our system! This simplifies all of the code and logic inside of the system. We're taking advantage of types to make our code simpler, safer, and easier to understand.

3.2. Ask Only What You Need

In many senses, designing our code with type safety in mind is about being as strict as possible about your possible inputs. Haskell makes this easier than many other languages, but there's nothing stopping you from writing a function that can take literally any binary value, do whatever effects you want, and return whatever binary value:

```
foobar :: ByteString -> IO ByteString
```

A `ByteString` is a totally unrestricted data type. It can contain any sequence of bytes. Because it can express any value, we have very little guarantees on what it actually contains, and we are very limited in how we can safely handle this.

By restricting our past, we gain freedom in the future.

3. *Type Safety Back and Forth - Matt Parsons*

4. Keep your types small. . . and your bugs smaller - Matt Parsons

Original article: [3]

In my previous article "Type Safety Back and Forth" (see chapter 3), I discussed two different techniques for bringing type safety to programs that may fail. On the one hand, you can push the responsibility forward. This technique uses types like `Either` and `Maybe` to report a problem with the inputs to the function. Here are two example type signatures:

```
safeDivide
    :: Int
    -> Int
    -> Maybe Int

lookup
    :: Ord k
    => k
    -> Map k a
    -> Maybe a
```

If the second parameter to `safeDivide` is `0`, then we return `Nothing`. Likewise, if the given `k` is not present in the `Map`, then we return `Nothing`.

On the other hand, you can push it back. Here are those functions, but with the safety pushed back:

```
safeDivide
    :: Int
    -> NonZero Int
    -> Int

lookupJustified
    :: Key ph k
    -> Map ph k a
    -> a
```

With `safeDivide`, we require the user pass in a `NonZero Int` – a type that guarantees that the underlying value is not `0`. With `lookupJustified`, the `ph` type guarantees that the `Key` is present in the `Map`, so we can pull the resulting value out without requiring a `Maybe`. (Check out the tutorial for `justified-containers`, it is pretty awesome).

4.1. Expansion and Restriction

"Type Safety Back and Forth" uses the metaphor of "pushing" the responsibility in one of two directions:

- forwards: the caller of the function is responsible for handling the possible error output
- backwards: the caller of the function is required to providing correct inputs

However, this metaphor is a bit squishy. We can make it more precise by talking about the "cardinality" of a type – how many values it can contain. The type `Bool` can contain two values – `True` and `False`, so we say it has a cardinality of 2. The type `Word8` can express the numbers from 0 to 255, so we say it has a cardinality of 256.

The type `Maybe a` has a cardinality of $1 + a$. We get a "free" value `Nothing :: Maybe a`. For every value of type `a`, we can wrap it in `Just`. The type `Either e a` has a cardinality of $e + a$. We can wrap all the values of type `e` in `Left`, and then we can wrap all the values of type `a` in `Right`.

The first technique – pushing forward – is "expanding the result type." When we wrap our results in `Maybe`, `Either`, and similar types, we're saying that we can't handle all possible inputs, and so we must have extra outputs to safely deal with this.

Let's consider the second technique. Specifically, here's `NonZero` and `NonEmpty`, two common ways to implement it:

```
newtype NonZero a
    = UnsafeNonZero
    { unNonZero :: a
    }

nonZero :: (Num a, Eq a) => a -> Maybe (NonZero a)
nonZero 0 = Nothing
nonZero i = Just (UnsafeNonZero i)

data NonEmpty a = a :| [a]

nonEmpty :: [a] -> Maybe (NonEmpty a)
nonEmpty []     = Nothing
nonEmpty (x:xs) = x :| xs
```

What is the cardinality of these types?

`NonZero a` represents "the type of values `a` such that the value is not equal to `0`." `NonEmpty a` represents "the type of lists of a that are not empty". In both of these cases, we start with some larger type and remove some potential values. So the type `NonZero a` has the cardinality $a - 1$, and the type `NonEmpty a` has the cardinality $[a] - 1$.

Interestingly enough, `[a]` has an infinite cardinality, so $[a] - 1$ seems somewhat strange – it is also infinite! Math tells us that these are even the same infinity. So it's not the mere cardinality that helps – it is the specific value(s) that we have removed that makes this type safer for certain operations.

These are custom examples of refinement types. Another closely related idea is quotient types. The basic idea here is to restrict the size of our inputs. Slightly more formally,

- Forwards: expand the range
- Backwards: restrict the domain

4.2. Constraints Liberate

Runar Bjarnason has a wonderful talk titled Constraints Liberate, Liberties Constrain. The big idea of the talk, as I see it, is this:

> *When we restrict what we can do, it's easier to understand what we can do.*

I feel there is a deep connection between this idea and Rich Hickey's talk Simple Made Easy. In both cases, we are focusing on simplicity – on cutting away the inessential and striving for more elegant ways to express our problems.

Pushing the safety forward – expanding the range – does not make things simpler. It provides us with more power, more options, and more possibilities. Pushing the safety backwards – restricting the domain – does make things simpler. We can use this technique to take away the power to get it wrong, the options that aren't right, and the possibilities we don't want.

Indeed, if we manage to restrict our types sufficiently, there may be only one implementation possible! The classic example is the `identity` function:

```
identity :: a -> a
identity a = a
```

This is the only implementation of this function that satisfies the type signature (ignoring undefined, of course). In fact, for any function with a sufficiently precise type signature, there is a way to automatically derive the function! Joachim Breitner's justDoIt is a fascinating utility that can solve these implementations for you.

With sufficiently fancy types, the computer can write even more code for you. The programming language Idris can write well-defined functions like `zipWith` and `transpose` for length-indexed lists nearly automatically!

4.3. Restrict the Range

I see this pattern and I am compelled to fill it in: I've talked about restricting the

	Restrict	**Expand**
Range		`:(`
Domain	`:D`	

domain and expanding the range. Expanding the domain seems silly to do – we accept more possible values than we know what to do with. This is clearly not going to make it easier or simpler to implement our programs. However, there are many functions in Haskell's standard library that have a domain that is too large. Consider:

```
take :: Int -> [a] -> [a]
```

`Int`, as a domain, is both too large and too small. It allows us to provide negative numbers: what does it even mean to take `-3` elements from a list? As `Int` is a finite type, and `[a]` is infinite, we are restricted to only using this function with sufficiently small `Int`s. A closer fit would be `take :: Natural -> [a] -> [a]`. Natural allows any non-negative whole number, and perfectly expresses the reasonable domain. Expanding the domain isn't desirable, as we might expect.

`base` has functions with a `range` that is too large, as well. Let's consider:

```
length :: [a] -> Int
```

This has many of the same problems as `take` – a list with too many elements will overflow the `Int`, and we won't get the right answer. Additionally, we have a guarantee that we *forget* – a length for any container must be positive! We can more correctly express this type by restricting the output type:

```
length :: [a] -> Natural
```

4.4. A perfect fit

The more precisely our types describe our program, the fewer ways we have to go wrong. Ideally, we can provide a correct output for every input, and we use a type that tightly describes the properties of possible outputs.

5. Algebraic blindness - David Luposchainsky

William Yao:

*The motivating problem: if I have a `Maybe Int`, I only implicitly know what the two branches **mean**. `Nothing` could mean "something bad happened, abort", or it could mean "I haven't found anything yet, keep going." Conversely, a value of `Just x` could be a useful value, or it could be a subroutine signalling that an error code occurred. The names are **too generic**; the structure only tells us what we have, not what it's for.*

Goes through how we might solve this problem by defining new datatypes that are isomorphic to, say, `Bool`, but with more useful names. Note that the problem that the article talks about with regards to not having typeclass instances for your custom types can (since GHC 8.6) be solved using `DerivingVia`.

Original article: [4]

5.1. Abstract

Algebraic data types make data more flexible, but also prone to a form of generalized Boolean Blindness, making code more difficult to maintain. Luckily, refactoring the issues is completely type-safe.

5.2. Boolean blindness

In programming, there is a common problem known as *Boolean Blindness*, what it "means" to be `True` depends heavily on the context, and cannot be inferred by the value alone. Consider

```
main = withFile True "file.txt" (\handle -> do stuff)
```

The Boolean parameter could distinguish between read-only/write-only, or read-only-/read+write, whether the file should be truncated before opening, and countless other possibilities.

And often there is an even worse issue: we have two booleans, and we do not know whether they describe something in the same problem domain. A boolean used for read-only-vs-write-only looks just the same as one distinguishing red from blue. And then, one day, we open a file in "blue mode", and probably nothing good follows.

The point is: in order to find out what the `True` means, you probably have to read documentation or code elsewhere.

5.3. Haskell to the rescue

Haskell makes it very natural and cheap to define our own data types. The example code above would be expressed as

```
main = withFile ReadMode "file.txt" (\handle -> do stuff)
```

in more idiomatic Haskell. Here, the meaning of the field is obvious, and since a data type `data IOMode = ReadMode | WriteMode` has nothing to do with a data type `data Colours = Red | Blue`, we cannot accidentally pass a `Red` value to our function, despite the fact that they would both have corresponded to `True` in the Boolean-typed example.

5.4. The petting zoo of blindness

Boolean blindness is of course just a name for the most characteristic version of the issue that most standard types share. An `Int` parameter to a server might be a port or a timeout, a `String` could be a host, a route, a log prefix, and so on.

The Haskell solution is to simply wrap things in newtypes, tagging values with phantom types, or introducing type synonyms. (I'm not a fan of the latter, which you can read more about in a previous article.)

5.5. Algebraic blindness

Haskell has a lot more "simple, always available, nicely supported" types than most other languages, for example `Maybe`, `Either`, tuples or lists. These types naturally extend Boolean Blindness.

- `Maybe` adds another distinct value to a type. `Nothing` is sometimes used to denote an error case (this is what many assume by default, implicitly given by its Monad instance), sometimes the "nothing weird happened" case, and sometimes something completely different.

  `Maybe a` is as blind as `a`, plus one value.

- `Either` is similar: sometimes `Left` is an exceptional case, sometimes it's just "the other" case.

- `Either a b` is as blind as `a` plus as blind as `b`, plus one for the fact that `Left` and `Right` do not have intrinsic meaning.

- Pairs have two fields, but how do they relate to each other? Does one maybe tell us about errors in the other? We cannot know.

  `(a,b)` is as blind as `a` times the blindness of `b`.

- Unit is not very blind, since even synonyms of it mostly mean the same thing: we don't really care about the result.

In GHCi's source code, there is a value of type `Maybe Bool` passed around, which has three possible values:

1. `Nothing` means there is no more input to process when GHCi is invoked via `ghc -e`.
2. `Just True` reports success of the last command.
3. `Just False` is an error in the last command, and `ghc -e` should abort with an exit code of 1, while a GHCi session should continue.

It is very hard to justify this over something like

```
data CommandResult
    = NoMoreInput -- ^ Haddock might go here!
    | Success
    | Failure
```

which is just as easy to implement, has four lines of overhead (instead of 0 lines of overheadache), is easily searchable in full-text search even, and gives us type errors when we use it in the wrong context.

5.6. Haskell to the rescue, for real this time

We're lucky, because Haskell has a built-in solution here as well. We can prototype our programs not worrying about the precise meaning of symbols, use `Either () ()` instead of Bool because we might need the flexibility, and do all sorts of atrocities.

The type system allows us to repair our code: just understand what the different values of our blind values mean, and encode this in a new data type. Then pick a random use site, and just put it in there. The compiler will yell at you for a couple of minutes, but it will report every single conflicting site, and since you're introducing something entirely new, there is no way you are producing undetected clashes. I found this type of refactoring to be *one of the most powerful tools Haskell has to offer*, but we rarely speak about it because it seems so normal to us.

5.7. Drawbacks

Introducing new domain types has a drawback: we lose the API of the standard types. The result is that we sometimes have to write boilerplate to get specific parts of the API back, which is unfortunate, and sometimes this makes it not worth bothering with the anti-blindness refactoring.

5.8. Conclusion

When you have lots of faceless data types in your code, consider painting them with their domain meanings. Make them distinct, make them memorable, make them maintainable. And sometimes, when you see a type that looks like

```
data IOMode
    = ReadMode
    | WriteMode
```

```
    | AppendMode
    | ReadWriteMode
```

take a moment to appreciate that the author hasn't used

```
Either Bool Bool
--      ^    ^
--      |    |
--      |    Complex modes: append, read+write
--      |
--      Simple modes: read/write only
```

instead.

6. Parse, don't validate - Alexis King

> ***William Yao:***
> *Once all your functions have their inputs suitably restricted, how to actually go about ingesting real-world data (which will always have issues) and producing the types we need? Enforce it at the "barriers" in your code. That might be at the very start of an HTTP handler, it might be between two modules in your codebase. Do it right and your core logic contains only what it needs to solve the actual problem; no need for cluttering it up with extraneous error handling.*
>
> *Original article: [5]*

Historically, I've struggled to find a concise, simple way to explain what it means to practice type-driven design. Too often, when someone asks me "How did you come up with this approach?" I find I can't give them a satisfying answer. I know it didn't just come to me in a vision – I have an iterative design process that doesn't require plucking the "right" approach out of thin air—yet I haven't been very successful in communicating that process to others.

However, about a month ago, I was reflecting on Twitter about the differences I experienced parsing JSON in statically- and dynamically-typed languages, and finally, I realized what I was looking for. Now I have a single, snappy slogan that encapsulates what type-driven design means to me, and better yet, it's only three words long:

> ***Parse, don't validate.***

6.1. The essence of type-driven design

Alright, I'll confess: unless you already know what type-driven design is, my catchy slogan probably doesn't mean all that much to you. Fortunately, that's what the remainder of this blog post is for. I'm going to explain precisely what I mean in gory detail—but first, we need to practice a little wishful thinking.

6.1.1. The realm of possibility

One of the wonderful things about static type systems is that they can make it possible, and sometimes even easy, to answer questions like "is it possible to write this function?" For an extreme example, consider the following Haskell type signature:

```
foo :: Integer -> Void
```

Is it possible to implement foo? Trivially, the answer is *no*, as `Void` is a type that contains no values, so it's impossible for any function to produce a value of type

`Void`[1] That example is pretty boring, but the question gets much more interesting if we choose a more realistic example:

```
head :: [a] -> a
```

This function returns the first element from a list. Is it possible to implement? It certainly doesn't sound like it does anything very complicated, but if we attempt to implement it, the compiler won't be satisfied:

```
head :: [a] -> a
head (x:_) = x

warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for 'head': Patterns not matched: []
```

This message is helpfully pointing out that our function is partial, which is to say it is not defined for all possible inputs. Specifically, it is not defined when the input is `[]`, the empty list. This makes sense, as it isn't possible to return the first element of a list if the list is empty—there's no element to return! So, remarkably, we learn this function isn't possible to implement, either.

6.2. Turning partial functions total

To someone coming from a dynamically-typed background, this might seem perplexing. If we have a list, we might very well want to get the first element in it. And indeed, the operation of "getting the first element of a list" isn't impossible in Haskell, it just requires a little extra ceremony. There are two different ways to fix the head function, and we'll start with the simplest one.

6.2.1. Managing expectations

As established, head is partial because there is no element to return if the list is empty: we've made a promise we cannot possibly fulfill. Fortunately, there's an easy solution to that dilemma: we can weaken our promise. Since we cannot guarantee the caller an element of the list, we'll have to practice a little expectation management: we'll do our best return an element if we can, but we reserve the right to return nothing at all. In Haskell, we express this possibility using the Maybe type:

```
head :: [a] -> Maybe a
```

This buys us the freedom we need to implement head—it allows us to return Nothing when we discover we can't produce a value of type a after all:

1 Technically, in Haskell, this ignores "bottoms", constructions that can inhabit *any* value. These aren't "real" values (unlike `null` in some other languages) – they're things like infinite loops or computations that raise exceptions – and in idiomatic Haskell, we usually try to avoid them, so reasoning that pretends they don't exist still has value. But don't take my word for it – I'll let Danielsson et al. convince you that Fast and Loose Reasoning is Morally Correct.

```haskell
head :: [a] -> Maybe a
head (x:_) = Just x
head []    = Nothing
```

Problem solved, right? For the moment, yes... but this solution has a hidden cost.

Returning `Maybe` is undoubtably convenient when we're implementing head. However, it becomes significantly less convenient when we want to actually use it! Since head always has the potential to return Nothing, the burden falls upon its callers to handle that possibility, and sometimes that passing of the buck can be incredibly frustrating. To see why, consider the following code:

```haskell
getConfigurationDirectories :: IO [FilePath]
getConfigurationDirectories = do
  configDirsString <- getEnv "CONFIG_DIRS"
  let configDirsList = split ',' configDirsString
  when (null configDirsList) $
    throwIO $ userError "CONFIG_DIRS cannot be empty"
  pure configDirsList

main :: IO ()
main = do
  configDirs <- getConfigurationDirectories
  case head configDirs of
    Just cacheDir -> initializeCache cacheDir
    Nothing -> error "should never happen; already checked configDirs is non-empty"
```

When `getConfigurationDirectories` retrieves a list of file paths from the environment, it proactively checks that the list is non-empty. However, when we use head in main to get the first element of the list, the `Maybe FilePath` result still requires us to handle a `Nothing` case that we know will never happen! This is terribly bad for several reasons:

1. First, it's just annoying. We already checked that the list is non-empty, why do we have to clutter our code with another redundant check?
2. Second, it has a potential performance cost. Although the cost of the redundant check is trivial in this particular example, one could imagine a more complex scenario where the redundant checks could add up, such as if they were happening in a tight loop.
3. Finally, and worst of all, this code is a bug waiting to happen! What if `getConfigurationDirectories` were modified to stop checking that the list is empty, intentionally or unintentionally? The programmer might not remember to update main, and suddenly the "impossible" error becomes not only possible, but probable.

The need for this redundant check has essentially forced us to punch a hole in our type system. If we could statically prove the `Nothing` case impossible, then a modification to `getConfigurationDirectories` that stopped checking if the list was empty would invalidate the proof and trigger a compile-time failure. However, as-written, we're forced to rely on a test suite or manual inspection to catch the bug.

6.2.2. Paying it forward

Clearly, our modified version of `head` leaves some things to be desired. Somehow, we'd like it to be smarter: if we already checked that the list was non-empty, `head` should unconditionally return the first element without forcing us to handle the case we know is impossible. How can we do that?

Let's look at the original (partial) type signature for head again:

```
head :: [a] -> a
```

The previous section illustrated that we can turn that partial type signature into a total one by weakening the promise made in the return type. However, since we don't want to do that, there's only one thing left that can be changed: the argument type (in this case, `[a]`). Instead of weakening the return type, we can *strengthen* the argument type, eliminating the possibility of head ever being called on an empty list in the first place.

To do this, we need a type that represents non-empty lists. Fortunately, the existing `NonEmpty` type from `Data.List.NonEmpty` is exactly that. It has the following definition:

```
data NonEmpty a = a :| [a]
```

Note that `NonEmpty a` is really just a tuple of an `a` and an ordinary, possibly-empty `[a]`. This conveniently models a non-empty list by storing the first element of the list separately from the list's tail: even if the `[a]` component is `[]`, the a component must always be present. This makes head completely trivial to implement[2]:

```
head :: NonEmpty a -> a
head (x:|_) = x
```

Unlike before, GHC accepts this definition without complaint – this definition is *total*, not partial. We can update our program to use the new implementation:

```
getConfigurationDirectories :: IO (NonEmpty FilePath)
getConfigurationDirectories = do
  configDirsString <- getEnv "CONFIG_DIRS"
  let configDirsList = split ',' configDirsString
  case nonEmpty configDirsList of
    Just nonEmptyConfigDirsList -> pure nonEmptyConfigDirsList
    Nothing -> throwIO $ userError "CONFIG_DIRS cannot be empty"

main :: IO ()
main = do
  configDirs <- getConfigurationDirectories
  initializeCache (head configDirs)
```

2 in fact, `Data.List.NonEmpty` already provides a `head` function with this type, but just for the sake of illustration, we'll reimplement it ourselves.

Note that the redundant check in `main` is now completely gone! Instead, we perform the check exactly once, in `getConfigurationDirectories`. It constructs a `NonEmpty a` from a `[a]` using the `nonEmpty` function from `Data.List.NonEmpty`, which has the following type:

```
nonEmpty :: [a] -> Maybe (NonEmpty a)
```

The `Maybe` is still there, but this time, we handle the `Nothing` case very early in our program: right in the same place we were already doing the input validation. Once that check has passed, we now have a `NonEmpty FilePath` value, which preserves (in the type system!) the knowledge that the list really is non-empty. Put another way, you can think of a value of type `NonEmpty a` as being like a value of type `[a]`, plus a proof that the list is non-empty.

By strengthening the type of the argument to `head` instead of weakening the type of its result, we've completely eliminated all the problems from the previous section:

- The code has no redundant checks, so there can't be any performance overhead.
- Furthermore, if `getConfigurationDirectories` changes to stop checking that the list is non-empty, its return type must change, too. Consequently, main will fail to typecheck, alerting us to the problem before we even run the program!

What's more, it's trivial to recover the old behavior of `head` from the new one by composing head with `nonEmpty`:

```
head' :: [a] -> Maybe a
head' = fmap head . nonEmpty
```

Note that the inverse is not true: there is no way to obtain the new version of head from the old one. All in all, the second approach is superior on all axes.

6.3. The power of parsing

You may be wondering what the above example has to do with the title of this blog post. After all, we only examined two different ways to validate that a list was non-empty – no parsing in sight. That interpretation isn't wrong, but I'd like to propose another perspective: in my mind, the difference between validation and parsing lies almost entirely in how information is preserved. Consider the following pair of functions:

```
validateNonEmpty :: [a] -> IO ()
validateNonEmpty (_:_) = pure ()
validateNonEmpty [] = throwIO $ userError "list cannot be empty"

parseNonEmpty :: [a] -> IO (NonEmpty a)
parseNonEmpty (x:xs) = pure (x:|xs)
parseNonEmpty [] = throwIO $ userError "list cannot be empty"
```

These two functions are nearly identical: they check if the provided list is empty, and if it is, they abort the program with an error message. The difference lies entirely in the return type: `validateNonEmpty` always returns `()`, the type that contains no information, but `parseNonEmpty` returns `NonEmpty a`, a refinement of the input type that preserves the knowledge gained in the type system. Both of these functions check the same thing, but `parseNonEmpty` gives the caller access to the information it learned, while `validateNonEmpty` just throws it away.

These two functions elegantly illustrate two different perspectives on the role of a static type system: `validateNonEmpty` obeys the typechecker well enough, but only `parseNonEmpty` takes full advantage of it. If you see why `parseNonEmpty` is preferable, you understand what I mean by the mantra "parse, don't validate." Still, perhaps you are skeptical of `parseNonEmpty`'s name. Is it really parsing anything, or is it merely validating its input and returning a result? While the precise definition of what it means to parse or validate something is debatable, I believe `parseNonEmpty` is a bona-fide parser (albeit a particularly simple one).

Consider: what is a parser? Really, a parser is just a function that consumes less-structured input and produces more-structured output. By its very nature, a parser is a partial function – some values in the domain do not correspond to any value in the range – so all parsers must have some notion of failure. Often, the input to a parser is text, but this is by no means a requirement, and `parseNonEmpty` is a perfectly cromulent parser: it parses lists into non-empty lists, signaling failure by terminating the program with an error message.

Under this flexible definition, parsers are an incredibly powerful tool: they allow discharging checks on input up-front, right on the boundary between a program and the outside world, and once those checks have been performed, they never need to be checked again! Haskellers are well-aware of this power, and they use many different types of parsers on a regular basis:

- The aeson library provides a Parser type that can be used to parse JSON data into domain types.
- Likewise, optparse-applicative provides a set of parser combinators for parsing command-line arguments.
- Database libraries like persistent and postgresql-simple have a mechanism for parsing values held in an external data store.
- The servant ecosystem is built around parsing Haskell datatypes from path components, query parameters, HTTP headers, and more.

The common theme between all these libraries is that they sit on the boundary between your Haskell application and the external world. That world doesn't speak in product and sum types, but in streams of bytes, so there's no getting around a need to do some parsing. Doing that parsing up front, before acting on the data, can go a long way toward avoiding many classes of bugs, some of which might even be security vulnerabilities.

One drawback to this approach of parsing everything up front is that it sometimes requires values be parsed long before they are actually used. In a dynamically-typed language, this can make keeping the parsing and processing logic in sync a little tricky without extensive test coverage, much of which can be laborious to maintain.

However, with a static type system, the problem becomes marvelously simple, as demonstrated by the `NonEmpty` example above: if the parsing and processing logic go out of sync, the program will fail to even compile.

6.4. The danger of validation

Hopefully, by this point, you are at least somewhat sold on the idea that parsing is preferable to validation, but you may have lingering doubts. Is validation really so bad if the type system is going to force you to do the necessary checks eventually anyway? Maybe the error reporting will be a little bit worse, but a bit of redundant checking can't hurt, right?

Unfortunately, it isn't so simple. Ad-hoc validation leads to a phenomenon that the language-theoretic security field calls shotgun parsing. In the 2016 paper, The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them, its authors provide the following definition:

> Shotgun parsing is a programming antipattern whereby parsing and input-validating code is mixed with and spread across processing code—throwing a cloud of checks at the input, and hoping, without any systematic justification, that one or another would catch all the "bad" cases.

They go on to explain the problems inherent to such validation techniques:

> Shotgun parsing necessarily deprives the program of the ability to reject invalid input instead of processing it. Late-discovered errors in an input stream will result in some portion of invalid input having been processed, with the consequence that program state is difficult to accurately predict.

In other words, a program that does not parse all of its input up front runs the risk of acting upon a valid portion of the input, discovering a different portion is invalid, and suddenly needing to roll back whatever modifications it already executed in order to maintain consistency. Sometimes this is possible—such as rolling back a transaction in an RDBMS – but in general it may not be.

It may not be immediately apparent what shotgun parsing has to do with validation – after all, if you do all your validation up front, you mitigate the risk of shotgun parsing. The problem is that validation-based approaches make it extremely difficult or impossible to determine if everything was actually validated up front or if some of those so-called "impossible" cases might actually happen. The entire program must assume that raising an exception anywhere is not only possible, it's regularly necessary.

Parsing avoids this problem by stratifying the program into two phases – parsing and execution – where failure due to invalid input can only happen in the first phase. The set of remaining failure modes during execution is minimal by comparison, and they can be handled with the tender care they require.

6.5. Parsing, not validating, in practice

So far, this blog post has been something of a sales pitch. "You, dear reader, ought to be parsing!" it says, and if I've done my job properly, at least some of you are

sold. However, even if you understand the "what" and the "why," you might not feel especially confident about the "how".

My advice: *focus on the datatypes.*

Suppose you are writing a function that accepts a list of tuples representing key-value pairs, and you suddenly realize you aren't sure what to do if the list has duplicate keys. One solution would be to write a function that asserts there aren't any duplicates in the list:

```
checkNoDuplicateKeys :: (MonadError AppError m, Eq k) => [(k, v)] -> m ()
```

However, this check is fragile: it's extremely easy to forget. Because its return value is unused, it can always be omitted, and the code that needs it would still typecheck. A better solution is to choose a data structure that disallows duplicate keys by construction, such as a `Map`. Adjust your function's type signature to accept a `Map` instead of a list of tuples, and implement it as you normally would.

Once you've done that, the call site of your new function will likely fail to typecheck, since it is still being passed a list of tuples. If the caller was given the value via one of its arguments, or if it received it from the result of some other function, you can continue updating the type from list to `Map`, all the way up the call chain. Eventually, you will either reach the location the value is created, or you'll find a place where duplicates actually ought to be allowed. At that point, you can insert a call to a modified version of `checkNoDuplicateKeys`:

```
checkNoDuplicateKeys :: (MonadError AppError m, Eq k) => [(k, v)] -> m (Map k v)
```

Now the check cannot be omitted, since its result is actually necessary for the program to proceed!

This hypothetical scenario highlights two simple ideas:

1. **Use a data structure that makes illegal states unrepresentable**. Model your data using the most precise data structure you reasonably can. If ruling out a particular possibility is too hard using the encoding you are currently using, consider alternate encodings that can express the property you care about more easily. Don't be afraid to refactor.

2. **Push the burden of proof upward as far as possible, but no further**. Get your data into the most precise representation you need as quickly as you can. Ideally, this should happen at the boundary of your system, before any of the data is acted upon[3].

   If one particular code branch eventually requires a more precise representation of a piece of data, parse the data into the more precise representation as soon as the branch is selected. Use sum types judiciously to allow your datatypes to reflect and adapt to control flow.

3 Sometimes it is necessary to perform some kind of authorization before parsing user input to avoid denial of service attacks, but that's okay: authorization should have a relatively small surface area, and it shouldn't cause any significant modifications to the state of your system.

In other words, write functions on the data representation you wish you had, not the data representation you are given. The design process then becomes an exercise in bridging the gap, often by working from both ends until they meet somewhere in the middle. Don't be afraid to iteratively adjust parts of the design as you go, since you may learn something new during the refactoring process!

Here are a handful of additional points of advice, arranged in no particular order:

- **Let your datatypes inform your code, don't let your code control your datatypes**. Avoid the temptation to just stick a `Bool` in a record somewhere because it's needed by the function you're currently writing. Don't be afraid to refactor code to use the right data representation – the type system will ensure you've covered all the places that need changing, and it will likely save you a headache later.

- **Treat functions that return `m ()` with deep suspicion**. Sometimes these are genuinely necessary, as they may perform an imperative effect with no meaningful result, but if the primary purpose of that effect is raising an error, it's likely there's a better way.

- **Don't be afraid to parse data in multiple passes**. Avoiding shotgun parsing just means you shouldn't act on the input data before it's fully parsed, not that you can't use some of the input data to decide how to parse other input data. Plenty of useful parsers are context-sensitive.

- **Avoid denormalized representations of data, *especially* if it's mutable**. Duplicating the same data in multiple places introduces a trivially representable illegal state: the places getting out of sync. Strive for a single source of truth.

  - **Keep denormalized representations of data behind abstraction boundaries**. If denormalization is absolutely necessary, use encapsulation to ensure a small, trusted module holds sole responsibility for keeping the representations in sync.

- **Use abstract datatypes to make validators "look like" parsers**. Sometimes, making an illegal state truly unrepresentable is just plain impractical given the tools Haskell provides, such as ensuring an integer is in a particular range. In that case, use an abstract newtype with a smart constructor to "fake" a parser from a validator.

As always, use your best judgement. It probably isn't worth breaking out singletons and refactoring your entire application just to get rid of a single `error "impossible"` call somewhere – just make sure to treat those situations like the radioactive substance they are, and handle them with the appropriate care. If all else fails, at least leave a comment to document the invariant for whoever needs to modify the code next.

6.6. Recap, reflection, and related reading

That's all, really. Hopefully this blog post proves that taking advantage of the Haskell type system doesn't require a PhD, and it doesn't even require using the latest and

greatest of GHC's shiny new language extensions – though they can certainly sometimes help! Sometimes the biggest obstacle to using Haskell to its fullest is simply being aware what options are available, and unfortunately, one downside of Haskell's small community is a relative dearth of resources that document design patterns and techniques that have become tribal knowledge.

None of the ideas in this blog post are new. In fact, the core idea – "write total functions" – is conceptually quite simple. Despite that, I find it remarkably challenging to communicate actionable, practicable details about the way I write Haskell code. It's easy to spend lots of time talking about abstract concepts – many of which are quite valuable! – without communicating anything useful about process. My hope is that this is a small step in that direction.

Sadly, I don't know very many other resources on this particular topic, but I do know of one: I never hesitate to recommend Matt Parson's fantastic blog post *Type Safety Back and Forth* (see section 3). If you want another accessible perspective on these ideas, including another worked example, I'd highly encourage giving it a read. For a significantly more advanced take on many of these ideas, I can also recommend Matt Noonan's 2018 paper Ghosts of Departed Proofs, which outlines a handful of techniques for capturing more complex invariants in the type system than I have described here.

As a closing note, I want to say that doing the kind of refactoring described in this blog post is not always easy. The examples I've given are simple, but real life is often much less straightforward. Even for those experienced in type-driven design, it can be genuinely difficult to capture certain invariants in the type system, so do not consider it a personal failing if you cannot solve something the way you'd like! Consider the principles in this blog post ideals to strive for, not strict requirements to meet. All that matters is to try.

7. On Ad-hoc Datatypes - Jasper Van der Jeugt

> ***William Yao:***
> *In a similar vein to preventing algebraic blindness, make your code more readable by naming things with datatypes, even ones that only live in a single module. Defining new dataypes is cheap and easy, so do it!*
>
> *Original article: [6]*

In Haskell, it is extremely easy for the programmer to add a quick datatype. It does not have to take more than a few lines. This is useful to add auxiliary, ad-hoc datatypes.

I don't think this is used enough. Most libraries and code I see use "heavier" datatypes: canonical and very well-thought-out datatypes, followed by dozens of instances and related functions. These are of course great to work with, but it doesn't have to be a restriction: adding quick datatypes – without all these instances and auxiliary functions – often makes code easier to read.

The key idea is that, in order to make code as simple as possible, you want to represent your data as simply as possible. However, the definition of "simple" is not the same in every context. Sometimes, looking at your data from another perspective makes specific tasks easier. In those cases, introducing "quick-and-dirty" datatypes often makes code cleaner.

This blogpost is written in literate Haskell so you should be able to just load it up in GHCi and play around with it. You can find the raw `.lhs` file here.

```
import Control.Monad (forM_)
```

Let's look at a quick example. Here, we have a definition of a shopping cart in a fruit store.

```
data Fruit = Banana | Apple | Orange
    deriving (Show, Eq)

type Cart = [(Fruit, Int)]
```

And we have a few functions to inspect it.

```
cartIsHomogeneous :: Cart -> Bool
cartIsHomogeneous []                = True
cartIsHomogeneous ((fruit, _) : xs) = all ((== fruit) . fst) xs

cartTotalItems :: Cart -> Int
cartTotalItems = sum . map snd
```

This is very much like code you typically see in Haskell codebases (of course, with more complex datatypes than this simple example).

The last function we want to add is printing a cart. The exact way we format it depends on what is in the cart. There are four possible scenarios.

1. The cart is empty.
2. There is a single item in the customers cart and we have some sort of simplified checkout.
3. The customer buys three or more of the same fruit (and nothing else). In that case we give out a bonus.
4. None of the above.

This is clearly a good candidate for Haskell's `case` statement and guards. Let's try that.

```
printCart :: Cart -> IO ()
printCart cart = case cart of
    []           -> putStrLn $ "Your cart is empty"
    [(fruit, 1)] -> putStrLn $ "You are buying one " ++ show fruit
    _ | cartIsHomogeneous cart && cartTotalItems cart >= 3 -> do
              putStrLn $
                  show (cartTotalItems cart) ++
                  " " ++ show (fst $ head cart) ++ "s" ++
                  " for you!"
              putStrLn "BONUS GET!"
      | otherwise -> do
          putStrLn $ "Your cart: "
          forM_ cart $ \(fruit, num) ->
              putStrLn $ "- " ++ show num ++ " " ++ show fruit
```

This is not very nice. The business logic is interspersed with the printing code. We could clean it up by adding additional predicates such as `cartIsBonus`, but having too many of these predicates leads to a certain kind of Boolean Blindness.

Instead, it seems much nicer to introduce a temporary type:

```
data CartView
    = EmptyCart
    | SingleCart  Fruit
    | BonusCart   Fruit Int
    | GeneralCart Cart
```

This allows us to decompose our `printCart` into two clean parts: classifying the cart, and printing it.

```
cartView :: Cart -> CartView
cartView []           = EmptyCart
cartView [(fruit, 1)] = SingleCart fruit
cartView cart
    | cartIsHomogeneous cart && cartTotalItems cart >= 3 =
        BonusCart (fst $ head cart) (cartTotalItems cart)
    | otherwise = GeneralCart cart
```

Note that we now have a *single* location where we classify the cart. This is useful if we need this information in multiple places. If we chose to solve the problem by adding additional predicates such has `cartIsBonus` instead, we would still have to watch out that we check these predicates in the *same order* everywhere. Furthermore, if we need to add a case, we can simply add a constructor to this datatype, and the compiler will tell us where we need to update our code[1].

Our printCart becomes very simple now:

```
printCart2 :: Cart -> IO ()
printCart2 cart = case cartView cart of
    EmptyCart          -> putStrLn $ "Your cart is empty"
    SingleCart fruit   -> putStrLn $ "You are buying one " ++ show fruit
    BonusCart  fruit n -> do
        putStrLn $ show n ++ " " ++ show fruit ++ "s for you!"
        putStrLn "BONUS GET!"
    GeneralCart items  -> do
        putStrLn $ "Your cart: "
        forM_ items $ \(fruit, num) ->
            putStrLn $ "- " ++ show num ++ " " ++ show fruit
```

Of course, it goes without saying that ad-hoc datatypes that are only used locally should not be exported from the module – otherwise you end up with a mess again.

1 If you are compiling with `-Wall`, which is what you really, really should be doing.

8. Good design and type safety in Yahtzee - Tom Ellis

William Yao:

A worked example of going from a functional, if difficult-to-maintain piece of code, to a design where all the invariants are type-checked and there's no potential crashes to be found.

One thing that you should take away from this article is just how much of the refactoring is completely mechanical and required no understanding of the code at all. This is the biggest thing that people are talking about when they say that Haskell is "easy to maintain": it's possible to make sweeping changes to your code and improve the design with almost 100% certainty that it will still work without needing to think about what the code is doing at all.

Original article: [7]

Mark Dominus wrote an article asking how to take advantage of Haskell's type safety in a simple dice-rolling simulation function for the game Yahtzee. He added wrapper types so that one cannot mistakenly apply the function to values that merely have the correct type "by accident". He says the result is "unreadable" and "unmaintainable". It certainly doesn't look nice! I'd claim it's not even of much practical safety benefit (although I suppose that depends on what the rest of the program looks like).

Mark says:

> *I don't claim this code is any good; I was just hacking around exploring the problem space. But it does do what I wanted.*

But we can't just expect to sprinkle type safety on a bad design and get something good. Type safety and good design are qualities that evolve symbiotically. By using type safety merely to make things safer for our callers we miss out on a host of benefits. Type safety should be used to guide us in the design of our *implementations*, which our callers never see. In fact I would argue that Mark is trying to add type safety in exactly the wrong way.

If there is an interface whose implementation we don't control then all we can do is to slap a type safe wrapper on it and be done with it. That is of some benefit. On the other hand, if we *do* control the implementation then the type safe wrapper specifies invariants that we can take advantage of inside the implementation. They can help us *write* the implementation.

We should build type safe structures and combinators relevant to our domain and implement our solution in terms of those. Likewise we should look at our solution and factor out repeated patterns into type safe structures and combinators relevant to our domain. This is symbiotic evolution!

Type safety and good design proceed hand-in-hand along the road of implementation evolution. Type safety nudges us in the direction of good design and good design nudges us in the direction of type safety. One of the reasons I prefer Haskell to other languages is that the compiler can enforce the type safety end of this bargain. On the other hand, there's no reason a design in Python, say, can't be guided by "type safety" in the same sense as a design in Haskell. It's just that the "type safety" won't be checked by any tooling and so there's no evolutionary pressure in that direction (Python's recent type annotations notwithstanding).

Let's evolve Mark's program in the direction of good design and see what arises. In this case, as we'll see, I'm not sure there's much to be gained by trying to "add type safety". On the other hand, that there is much to be gained by improving the design *guided by* considerations of type safety. The design improvements would apply equally well to an implementation in Python.

I've shown the stages of program evolution as diffs. Sometimes they're easy to read and sometimes they're difficult. If anyone knows how to get diffs to render nicely in Pandoc (perhaps like GitHub renders them) then please please contact me.

8.1. Original implementation

This is the original implementation, before Mark added wrapper types. Most of the refactorings that I will perform are independent of what the implementation actually does; they are simply small changes that are obviously correct. Some are not obviously correct unless you know what the program does, so let's explain that `allRolls` does the following

- takes `DiceState` (a list of dice values and an `Integer n` of rolls left to perform), and also
- takes `DiceChoice` (a choice of which dice to reroll), and
- returns a list of all the possible rerolls along with an `Integer` of rolls left to perform.

Instead of adding wrapper types let's just try to improve the design and see what happens.

```
type DiceChoice = [ Bool ]
type DiceVals   = [ Integer ]
type DiceState = (DiceVals, Integer)

allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls [] ([], n) = [ ([], n-1) ]
allRolls [] _ = undefined
allRolls (chosen:choices) (v:vs, n) =
    allRolls choices (vs, n-1) >>=
        \(roll,_) -> [ (d:roll,  n-1) | d <- rollList ]
          where rollList = if chosen then [v] else [ 1..6 ]

example =
```

```
let diceChoices = [ False, True, True, False, False ]
    diceVals = [ 6, 4, 4, 3, 1 ]
in mapM_ print $ allRolls diceChoices (diceVals, 2)
```

8.2. Explain the invariant

There's a `undefined` in there. That is always a red flag. In this case undefined occurs when there are dice we could reroll but we haven't specifed whether or not to reroll them: the choice list is too short. Let's explain that in an error message. It communicates better to both the end user of the program and the developer.

We don't need to know what the program does to apply this refactoring but knowing it does give us more confidence that we are doing the right thing.

```
1  allRolls :: DiceChoice -> DiceState -> [ DiceState ]
2  allRolls [] ([], n) = [ ([], n-1) ]
3  allRolls [] _ = undefined
4  allRolls (chosen:choices) (v:vs, n) =
5      allRolls choices (vs, n-1) >>=
6          \(roll, ) -> [ (d:roll,  n-1)
7              | d <- rollList ]
8
9
```

```
01 allRolls :: DiceChoice -> DiceState -> [ DiceState ]
02 allRolls [] ([], n) = [ ([], n-1) ]
03 allRolls []   =
04   error "Invariant violated: choices must be same
05     length as vals"
06 allRolls (chosen:choices) (v:vs, n) =
07     allRolls choices (vs, n-1) >>=
08         \(roll, ) -> [ (d:roll,  n-1)
09           | d <- rollList ]
10
11
```

Figure 8.1.: Explain the invariant

8.3. Avoid catch-all pattern

I prefer to have as few overlapping patterns as possible, even "catch-all" (`_`) patterns. This is a fairly minor change, but let's do it anyway. I think it modestly improves the design.

We don't need to know what the program does to apply this refactoring.

```
1  allRolls :: DiceChoice -> DiceState -> [ DiceState ]
2  allRolls [] ([], n) = [ ([], n-1) ]
3  allRolls [] (_:_, _) =
4    error "Invariant violated: choices must be same
5          length as vals"
6
```

```
1  allRolls :: DiceChoice -> DiceState -> [ DiceState ]
2  allRolls [] ([], n) = [ ([], n-1) ]
3  allRolls [] (_:_, _) =
4    error "Invariant violated: choices must be
5           same length as vals"
6  allRolls (_:_) ([], _) =
7   error "Invariant violated: choices must be same
8          length as vals"
9
```

Figure 8.2.: Avoid catch-all pattern

8.4. Add another invariant check

There's actually a missing pattern (which `-Wall` will pick up). Let's add it. Another modest improvement.

Like with the first invariant check, we don't need to know what the program does to apply this refactoring but knowing it does give us more confidence that we are doing the right thing.

```
allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls [] ([], n) = [ ([], n-1) ]
allRolls [] (_:_, _) =


    error "Invariant violated: choices must be same
          length as vals"
```

```
allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls [] ([], n) = [ ([], n-1) ]
allRolls [] (_:_, _) =
  error "Invariant violated: choices must be
        same length as vals"
allRolls (_:_) ([], _) =
 error "Invariant violated: choices must be same
        length as vals"
```

Figure 8.3.: Add another invariant check

8.5. Add pop function

There are two distinct things that `allRolls` does.

1. It checks the invariant and if the invariant holds extracts relevant inputs.
2. It runs the algorithm on the relevant inputs.

Let's separate the concerns by adding a `pop` function that does 1. Note that the interface between `pop` and `allRolls` has type safety! Once `pop` returns, an invalid state is not possible. `allRolls` does not change, except to pass its argument through `pop`.

We don't need to know what the program does to apply this refactoring.

```
pop :: DiceChoice
    -> DiceVals
    -> Maybe ((Bool, Integer), (DiceChoice, DiceVals))
pop [] [] = Nothing
pop (chosen:choices) (v:vs) = Just ((chosen, v), (choices, vs))
pop (_:_) [] = error "Invariant violated: missing val"
pop [] (_:_) = error "Invariant violated: missing choice"

allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls choices (vs, n) = case pop choices vs of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) ->
    allRolls choices (vs, n-1) >>=
        \(roll,_) -> [ (d:roll,  n-1) | d <- rollList ]
          where rollList = if chosen then [v] else [ 1..6 ]
```

8.6. Indicate that a value is unused

This is the first time we have to apply real thinking to the design process. Weirdly, the $n-1$ argument to the recursive call to `allRolls` is not used in the final result. The only way I can suggest that one discovers this is to think through how the the code actually works. Unlike the above changes, this is not just a simple refactoring.

Let's indicate that the argument is unused by applying an error instead. In a language without lazy evaluation you might like to apply some nonsense value like -999999999 instead, and check that the results of the function call are not nonsense!

When we run this we don't get a crash, which implies that that argument was indeed not used.

```
allRolls choices (vs, n) = case pop choices vs of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) ->
    allRolls choices (vs, n-1) >>=
      \(roll, _) -> [ (d:roll,  n-1)
          | d <- rollList ]
        where
          rollList = if chosen then [v] else [ 1..6 ]
```

```
allRolls choices (vs, n) = case pop choices vs of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) ->
    allRolls choices (vs, error "Didn't expect
      to use") >>=
        \(roll, _) -> [ (d:roll,  n-1)
          | d <- rollList ]
      where
        rollList = if chosen then [v] else [ 1..6 ]
```

Figure 8.4.: Indicate that a value is unused

8.7. Prepare to rearrange arguments

Given the observation above I see no reason to package the `DiceVals` and the `Integer` together. Let's prepare to separate them.

We don't need to know what the program does to apply this refactoring. We just have to observe that the `DiceVals` and the `Integer` are not really used together.

```
type DiceChoice = [ Bool ]
type DiceVals   = [ Integer ]
type DiceState = (DiceVals, Integer)

...

allRolls :: DiceChoice -> DiceState -> [ DiceState ]
```

```
type DiceChoice = [ Bool ]
type DiceVals   = [ Integer ]


...

allRolls :: DiceChoice
         -> (DiceVals, Integer)
         -> [ (DiceVals, Integer) ]
```

Figure 8.5.: Prepare to rearrange arguments

8.8. Rearrange arguments

Now let's do the separation of the arguments. The size of the diff makes the change seem bigger than it is. It is merely passing two arguments instead of a tuple!

We don't need to know what the program does to apply this refactoring.

```
allRolls :: DiceChoice
         -> (DiceVals, Integer)
         -> [ (DiceVals, Integer) ]
allRolls choices (vs, n) = case pop choices vs of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) ->
    allRolls choices (vs, error "Didn't expect to use")
      >>= \(roll, _) -> [ (d:roll,  n-1)
        | d <- rollList ]
        where
          rollList = if chosen then [v] else [ 1..6 ]

example =
  let diceChoices = [ False, True, True, False, False
      diceVals = [ 6, 4, 4, 3, 1 ]
  in mapM_ print $ allRolls diceChoices (diceVals, 2)
```

```
allRolls :: DiceChoice
         -> DiceVals
         -> Integer
         -> [ (DiceVals, Integer) ]
allRolls choices vs n = case pop choices vs of
   Nothing -> [ ([], n-1) ]
   Just ((chosen, v), (choices, vs)) ->
    allRolls choices vs (error "Didn't expect to use")
      >>= \(roll, _) -> [ (d:roll,  n-1)
        | d <- rollList ]
        where
          rollList = if chosen then [v] else [ 1..6 ]

example =
  let diceChoices = [ False, True, True, False, False
      diceVals = [ 6, 4, 4, 3, 1 ]
  in mapM_ print $ allRolls diceChoices diceVals 2
```

Figure 8.6.: Rearrange arguments

8.9. Rearrange arguments further

Once we have separated `DiceVals` from the `Integer` we notice that `DiceChoice` and `DiceVals` seem to naturally belong together. Again the diff makes the change look

bigger than it is. We're just passing `DiceChoice` and `DiceVals` as a tuple rather than two arguments.

We don't need to know what the program does to apply this refactoring.

```
pop :: DiceChoice
    -> DiceVals
    -> Maybe ((Bool, Integer), (DiceChoice, DiceVals))
pop [] [] = Nothing
pop (chosen:choices) (v:vs) = Just ((chosen, v),
  (choices, vs))
pop (_:_) [] = error "Invariant violated: missing val"
pop [] (_:_) = error "Invariant violated: missing
  choice"

allRolls :: DiceChoice
         -> DiceVals
         -> Integer
         -> [ (DiceVals, Integer) ]
allRolls choices vs n = case pop choices vs of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) ->
    allRolls choices vs (error "Didn't expect to use")
      >>= \(roll, _) -> [ (d:roll,  n-1)
        | d <- rollList ]
        where
          rollList = if chosen then [v] else [ 1..6 ]

example =
  let diceChoices = [ False, True, True, False, False
      diceVals = [ 6, 4, 4, 3, 1 ]
  in mapM_ print $ allRolls diceChoices diceVals 2
```

```
pop :: (DiceChoice, DiceVals)

    -> Maybe ((Bool, Integer), (DiceChoice, DiceVals))
pop ([], []) = Nothing
pop (chosen:choices, v:vs) = Just ((chosen, v),
  (choices, vs))
pop (_:_, []) = error "Invariant violated: missing val
pop ([], _:_) = error "Invariant violated: missing cho


allRolls :: (DiceChoice, DiceVals)

         -> Integer
         -> [ (DiceVals, Integer) ]
allRolls (choices, vs) n = case pop (choices, vs) of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) ->
    allRolls (choices, vs) (error "Didn't expect to us
      >>= \(roll, _) -> [ (d:roll,  n-1)
        | d <- rollList ]
        where
          rollList = if chosen then [v] else [ 1..6 ]

example =
  let diceChoices = [ False, True, True, False, False
      diceVals = [ 6, 4, 4, 3, 1 ]
  in mapM_ print $ allRolls (diceChoices, diceVals) 2
```

Figure 8.7.: Rearrange arguments further

8.10. Avoid unpacking tuple

We no longer need to unpack the tuple! We don't need to know what the program does to apply this refactoring.

```
allRolls (choices, vs) n = case pop (choices, vs) of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) ->
    allRolls (choices, vs) (error "Didn't expect to use
      >>=
```

```
allRolls t n = case pop t of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), t) ->
    allRolls t (error "Didn't expect to use") >>=
```

Figure 8.8.: Avoid unpacking tuple

8.11. We don't use the Integer. Make this structural.

Given that we have an unused argument in the recursive call let's see if we can change our design to make this obvious, i.e. make the fact that we don't use it an essential part of the structure of the program, not just a property. In this case it amounts to pairing the rolls with $n-1$ after the bulk of the algorithm (`allRollsNoN`) has finished.

This is the second time we have to actually analyse how our program works rather than just apply a mechanical translation.

8.12. Introduce a type synonym

Given that `DiceChoice` and `DiceVals` seem to belong together let's add a type synonym (`DiceTurn`) for that.

```
allRolls :: (DiceChoice, DiceVals)
         -> Integer
         -> [ (DiceVals, Integer) ]
allRolls t n = case pop t of
  Nothing -> [ ([], n-1) ]

  Just ((chosen, v), t) ->
    allRolls t (error "Didn't expect to use")
      >>= \(roll, _) -> [ (d:roll,  n-1)
        | d <- rollList ]
          where
            rollList = if chosen then [v] else [ 1..6 ]
```

```
allRolls :: (DiceChoice, DiceVals)
         -> Integer
         -> [ (DiceVals, Integer) ]
allRolls t n = [ (vals, n-1) | vals <- allRollsNoN t ]

allRollsNoN :: (DiceChoice, DiceVals) -> [ DiceVals ]
allRollsNoN t = case pop t of
  Nothing -> [ [] ]
  Just ((chosen, v), t) ->
    allRollsNoN t >>=
      \roll -> [ d:roll | d <- rollList ]

        where
          rollList = if chosen then [v] else [ 1..6 ]
```

Figure 8.9.: We don't use the Integer

We don't need to know what the program does to apply this refactoring. We just observe that the pair of things are always used together.

```
pop :: (DiceChoice, DiceVals)
    -> Maybe ((Bool, Integer), (DiceChoice, DiceVals))

pop ([], []) = Nothing
pop (chosen:choices, v:vs) = Just ((chosen, v),
  (choices, vs))
pop (_:_, []) = error "Invariant violated: missing val
pop ([], _:_) = error "Invariant violated: missing
  choice"

allRolls :: (DiceChoice, DiceVals)
         -> Integer
         -> [ (DiceVals, Integer) ]
allRolls t n = [ (vals, n-1) | vals <- allRollsNoN t ]

allRollsNoN :: (DiceChoice, DiceVals) -> [ DiceVals ]
```

```
type DiceTurn  = (DiceChoice, DiceVals)

pop :: DiceTurn
    -> Maybe ((Bool, Integer), DiceTurn)
pop ([], []) = Nothing
pop (chosen:choices, v:vs) = Just ((chosen, v),
  (choices, vs))
pop (_:_, []) = error "Invariant violated: missing val
pop ([], _:_) = error "Invariant violated: missing
  choice"

allRolls :: DiceTurn
         -> Integer
         -> [ (DiceVals, Integer) ]
allRolls t n = [ (vals, n-1) | vals <- allRollsNoN t ]

allRollsNoN :: DiceTurn -> [ DiceVals ]
```

Figure 8.10.: Introduce a type synonym

8.13. Make illegal states unrepresentable

Our invariant is that the number of `DiceChoices` must be the same as the number of `DiceVals`. Semantically, we actually want something stronger: each of the `DiceChoices` corresponds to exactly one of the `DiceVals`. In my experience this is the single most common non-trival failure to structurally enforce program behaviour (and I'm not the only one to see it).

The fix is to put pairs of dice choice and dice vals in the same list! We can entirely remove our invariant check. The invariant is enforced by the type.

Consumers will have to change too but they'll be better off for it! In `example` I just `zipped` the args. It could also be something much better.

8.14. Use uncons

Having done that we see that `pop` is just the standard function "`uncons`". We don't need to know what the program does to apply this refactoring.

8.15. Don't need uncons

Having said that, we don't actually need `uncons`. We can just pattern match directly. We don't need to know what the program does to apply this refactoring.

```haskell
type DiceChoice = [ Bool ]
type DiceVals   = [ Integer ]
type DiceTurn  = (DiceChoice, DiceVals)

pop :: DiceTurn
    -> Maybe ((Bool, Integer), DiceTurn)
pop ([], []) = Nothing
pop (chosen:choices, v:vs) = Just ((chosen, v),
  (choices, vs))
pop (_:_, []) = error "Invariant violated: missing
  val"
pop ([], _:_) = error "Invariant violated: missing
  choice"

allRolls :: DiceTurn
         -> Integer

example =
  let diceChoices = [False, True, True, False, False]
      diceVals = [ 6, 4, 4, 3, 1 ]
  in mapM_ print $ allRolls (diceChoices, diceVals) 2
```

```haskell
type DiceVals   = [ Integer ]
type DiceTurn  = [(Bool, Integer)]

pop :: DiceTurn
    -> Maybe ((Bool, Integer), DiceTurn)
pop [] = Nothing
pop (a:as) = Just (a, as)

allRolls :: DiceTurn
         -> Integer

example =
  let diceChoices = [False, True, True, False, False]
      diceVals = [ 6, 4, 4, 3, 1 ]
  in mapM_ print $ allRolls
     (zip diceChoices diceVals) 2
```

Figure 8.11.: Make illegal states unrepresentable

```haskell
type DiceVals   = [ Integer ]
type DiceTurn  = [(Bool, Integer)]

pop :: DiceTurn
    -> Maybe ((Bool, Integer), DiceTurn)
pop [] = Nothing
pop (a:as) = Just (a, as)

allRolls :: DiceTurn
         -> Integer
         -> [ (DiceVals, Integer) ]
allRolls t n = [ (vals, n-1) |
  vals <- allRollsNoN t ]

allRollsNoN :: DiceTurn -> [ DiceVals ]
allRollsNoN t = case pop t of
```

```haskell
import Data.List (uncons)

type DiceVals   = [ Integer ]
type DiceTurn  = [(Bool, Integer)]

allRolls :: DiceTurn
         -> Integer
         -> [ (DiceVals, Integer) ]
allRolls t n = [ (vals, n-1) |
  vals <- allRollsNoN t ]

allRollsNoN :: DiceTurn -> [ DiceVals ]
allRollsNoN t = case uncons t of
```

Figure 8.12.: Use uncons

```haskell
allRollsNoN :: DiceTurn -> [ DiceVals ]
allRollsNoN t = case uncons t of
  Nothing -> [ [] ]
  Just ((chosen, v), t) ->
```

```haskell
allRollsNoN :: DiceTurn -> [ DiceVals ]
allRollsNoN t = case t of
  [] -> [ [] ]
  (chosen, v):t ->
```

Figure 8.13.: Don't need uncons

8.16. Use do notation

The use of the bind operator (»=) and list comprehension are not particularly clear. Let's rewrite it to use do notation. (In fact I recommend defaulting to do notation over operators unless there's some compelling readability benefit to using the latter.)

We don't need to know what the program does to apply this refactoring.

```
allRollsNoN :: DiceTurn -> [ DiceVals ]
allRollsNoN t = case t of
  [] -> [ [] ]
  (chosen, v):t ->
    allRollsNoN t >>=
      \roll -> [ d:roll | d <- rollList ]
```

```
allRollsNoN :: DiceTurn -> [ DiceVals ]
allRollsNoN t = case t of
  [] -> [ [] ]
  (chosen, v):t -> do
    roll <- allRollsNoN t
    d    <- rollList
    [ d:roll ]
```

Figure 8.14.: Use do notation

8.17. Prepare for `mapM`

We can see from the above that what our program does is takes the head of a list, runs recursively on the tail, does something to the head, and then puts it back on the tail. This is a "map" operation. Specifically in this case we are mapping in a monad so we use `mapM`. In modern Haskell you'd use `traverse`, but I'm going to stick to `mapM` because `traverse` does not read so well. (`traverse` really ought to be called `mapA` but people don't like the idea of that change.)

In this change we just make the function we are mapping take explicit arguments. We'll switch to use `mapM` in the next change. We don't need to know what the program does to apply this refactoring.

```
  (chosen, v):t -> do
    roll <- allRollsNoN t
    d    <- rollList
    [ d:roll ]
      where
        rollList = if chosen then [v] else [ 1..6 ]
```

```
  (chosen, v):t -> do
    roll <- allRollsNoN t
    d    <- rollList (chosen, v)
    [ d:roll ]
      where
        rollList (chosen, v)
          = if chosen then [v] else [ 1..6 ]
```

Figure 8.15.: Prepare for `mapM`

8.18. Use `mapM`

Now we can just use `mapM` directly. We don't need to know what the program does to apply this refactoring but we do need to know the general concept of "mapping" and the specific implementation `mapM`. Be careful! This particular "refactoring" actually reverses the order of effects – the list of dice rolls will come out in a different order.

8.19. Avoid boolean blindness

There's ambiguity in the type `DiceTurn = [(Bool, Integer)]`. Does the `Bool` refer to whether we keep the dice or to whether we reroll them? There's no way for me to

```haskell
allRollsNoN t = case t of
  [] -> [ [] ]
  (chosen, v):t -> do
    roll <- allRollsNoN t
    d    <- rollList (chosen, v)
    [ d:roll ]
      where
        rollList (chosen, v) =
          if chosen then [v] else [ 1..6 ]
```

```haskell
allRollsNoN =
    mapM (\(chosen, v) -> if chosen
      then [v]
      else [ 1..6 ])
```

Figure 8.16.: Use `mapM`

tell without seeing this conditional inside the function:

```haskell
if chosen then [v] else [ 1..6 ]
```

Ah, so the `Bool` refers to whether we keep the dice. Perhaps this is written in the documentation somewhere, but why do I believe that the documentation is kept in line with the implementation? I want a single source of truth!

Let's add a type to avoid "boolean blindness". The type of `allRollsBetter` still does not guarantee that the implementation does the right thing with its argument but it does make any deviation glaringly obvious.

We don't need to know what the program does to apply this refactoring. It requires the uncontroversial `LambdaCase` language extension.

```haskell
allRollsNoN =
  mapM (\(chosen, v) -> if chosen
    then [v]
    else [ 1..6 ])
```

```haskell
allRollsNoN = allRollsBetter . map fromTurn

data DiceChoice = Keep Integer | Reroll

fromTurn :: (Bool, Integer) -> DiceChoice
fromTurn (chosen, v) = if chosen
  then Keep v
  else Reroll

allRollsBetter :: [DiceChoice] -> [ DiceVals ]
allRollsBetter = mapM $ \case
  Reroll -> [ 1..6 ]
  Keep v -> [v]
```

Figure 8.17.: Avoid boolean blindness

8.20. Keep the better version

Let's get rid of all vestiges of the old version. What `allRolls` does could be done more clearly at the call site. At this point I wouldn't add wrapper types for "type safety". The rest of the program might be sufficiently complex that they would help, but they certainly don't add anything in this simple example.

The new version communicates *much* better. We "map" over our `DiceVals` list, that is, apply a function to each element in turn. In this case we're taking advantage of the `Monad` instance for lists, so we use `mapM`. The function we map simply says

- Do we want to `Reroll`? If so, the possible results are `[1..6]`
- Do we want to `Keep v`? If so, the possible results are just `[v]`

```haskell
{-# LANGUAGE LambdaCase #-}

type DiceVals   = [Integer]
data DiceChoice = Keep Integer | Reroll

allRollsBetter :: [DiceChoice] -> [DiceVals]
allRollsBetter = mapM $ \case
  Reroll -> [1..6]
  Keep v -> [v]

example =
  let diceVals = [ Reroll, Keep 4, Keep 4, Reroll, Reroll ]
  in mapM_ print $ allRollsBetter diceVals
```

Look at `allRollsBetter` in comparison to the original `allRolls`!

```haskell
allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls [] ([], n) = [ ([], n-1) ]
allRolls [] _ = undefined
allRolls (chosen:choices) (v:vs, n) =
    allRolls choices (vs, n-1) >>=
        \(roll,_) -> [ (d:roll,  n-1) | d <- rollList ]
          where rollList = if chosen then [v] else [ 1..6 ]
```

How did we end up with something so much clearer? We applied a sequence of transformations to improve the design, almost all of which are applicable in any language. The transformations were partly informed by a notion of "type safety". Specifically, we aimed to model our domain using types and functions that make illegal states unrepresentable.

None of this *requires* a language like Haskell. It would be good design in Python as well. One of Python's weaknesses is that it makes dealing with sum types awkward. We would have had to take a slightly different approach for the `Maybe` returned by `pop` (probably `None` or a tuple), the `DiceChoice` type (probably a pair of classes) and the list monad (probably just a recursive generator function). Ultimately though, the benefit of Haskell is not that it allows us to implement well-typed designs, nor particularly that it forbids us from implementing ill-typed designs. The benefit is that it nudges us away from poorly-typed, poorly-structured designs *and holds our hand as it does so*.

9. Using our brain less in refactoring Yahtzee - Tom Ellis

Original article: [8]

Cameron Gera and Taylor Fausak produced a podcast on an article of mine about good design and typesafety (see section 8) using code from an implementation of the game Yahtzee. The article is about refactoring code to improve design and how that goes hand-in-hand with type safety. Intriguingly, listening to others talk about my article gave me fresh ideas.

At one point in the article we observed that a variable to an argument was unused. Subsequently we removed it. The only justification given that the removal of the argument was valid was that we could convince ourselves that it was unused by looking at the implementation of the function, and that we could insert a run time check.

Hearing Cameron and Taylor talk about the article made me think again. There were only two changes to the code that really relied upon understanding what it does; everything else was mechanical transformation. Both of those changes were to do with the unused argument.

Einstein said "chalk is cheaper than grey matter". Can we avoid using "grey matter" (our brains) to remove the unused argument, instead just relying on "chalk" (mechanical transformations)? The answer is yes! Let's see how to do it.

9.1. The starting point

We start from the "Add pop function" stage of the previous article. We've got a suspicion that, although the argument `n` to `allRolls` *is* used, the argument $n-1$ to the recursive call is not. How can we transform the code to make that clear?

```
type DiceChoice = [ Bool ]
type DiceVals   = [ Integer ]
type DiceState  = (DiceVals, Integer)

pop :: DiceChoice
    -> DiceVals
    -> Maybe ((Bool, Integer), (DiceChoice, DiceVals))
pop [] [] = Nothing
pop (chosen:choices) (v:vs) = Just ((chosen, v), (choices, vs))
pop (_:_) [] = error "Invariant violated: missing val"
pop [] (_:_) = error "Invariant violated: missing choice"

allRolls :: DiceChoice -> DiceState -> [ DiceState ]
```

```haskell
allRolls choices (vs, n) = case pop choices vs of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) ->
    allRolls choices (vs, n-1) >>=
        \(roll,_) -> [ (d:roll,  n-1) | d <- rollList ]
          where rollList = if chosen then [v] else [ 1..6 ]

example =
  let diceChoices = [ False, True, True, False, False ]
      diceVals = [ 6, 4, 4, 3, 1 ]
  in mapM_ print $ allRolls diceChoices (diceVals, 2)
```

9.2. Use do-notation

Let's immediately simplify by using do-notation. In the previous article we left this stage until later but given that the recursive call is currently part of a »= expression let's apply the simplification now.

```haskell
allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls choices (vs, n) = case pop choices vs of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) -> do
    (roll, _) <- allRolls choices (vs, n-1)
    [ (d:roll, n-1) | d <- rollList ]
          where rollList = if chosen then [v] else [ 1..6 ]
```

9.3. Observe that both branches pair a list with n-1

If the argument `n` were not used at all then our job would be much easier. However, it `is` used, so let's try to separate the place it is used from the place where (we believe) it is not used.

Where is it used? We can see that each branch of the `case` statement returns a list of tuples where the second element of each tuple is $n - 1$. Put another way, each branch produces a list and then maps the "pair with $n - 1$" function over it.

I'll write the "pair with n-1" function as `(, n-1)` (using the `TupleSections` extension). The usual alternative would be to write
`x -> (x, n-1)` but in this article I want to keep things compact.

```haskell
allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls choices (vs, n) = case pop choices vs of
  Nothing -> [ ([], n-1) ]
  Just ((chosen, v), (choices, vs)) -> do
    (roll, _) <- allRolls choices (vs, n-1)
    [ (d:roll, n-1) | d <- rollList ]
```

```haskell
allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls choices (vs, n) = case pop choices vs of
  Nothing -> fmap (, n-1) [ [] ]
  Just ((chosen, v), (choices, vs)) -> do
    (roll, _) <- allRolls choices (vs, n-1)
    fmap (, n-1) [ d:roll | d <- rollList ]
```

Figure 9.1.: Observe that both branches pair a list with n-1

9.4. Lift fmap outside `do`

Now `n` is used, we think, just twice, and in each case to map the "pair with $n-1$" function over a list. We've made this duplication obvious but we can't yet remove it. First we have to lift the fmap outside the do. We use the rule that

```
do ...
   fmap f e
can be rewritten to

fmap f $ do ...
            e
```

Why is this rewriting valid? Informally, a `do` block is like a procedure, and this rule says that "applying f and then returning from the procedure" is the same as "returning from the procedure and then applying f". Formally, it can be proved using the monad laws.

This is how the rewriting applies in our case:

```
1  Just ((chosen, v), (choices, vs)) -> do
2    (roll, _) <- allRolls choices (vs, n-1)
3    fmap (, n-1) [ d:roll | d <- rollList ]
4
```

```
1  Just ((chosen, v), (choices, vs)) -> fmap (, n-1) $ d
2    (roll, _) <- allRolls choices (vs, n-1)
3    [ d:roll | d <- rollList ]
4
```

Figure 9.2.: Lift fmap outside `do`

9.5. Combine duplicated functions at top level

Now that both branches of the case statement are `fmap (, n-1)` of something, we can apply `fmap (, n-1)` to the overall case statement instead. Specifically, the rule is that we can rewrite

```
case x of
    Case1 -> f $ body1
    Case2 -> f $ body2
```

to

```
f $ case x of
    Case1 -> body1
    Case2 -> body2
```

which in our code leads to

9.6. Split function body into separate function

We want to carefully separate the parts of the code for which the value of `n` matters from the parts of the code for which the value of n does not matter. To this end we split the body of `allRolls` into a separate function called `allRollsBody`.

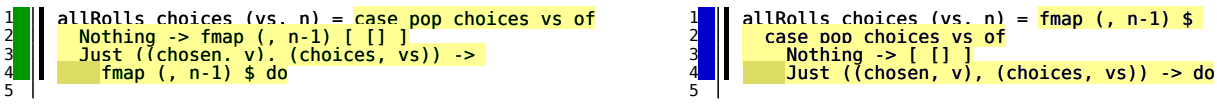

Figure 9.3.: Lift fmap outside `do`

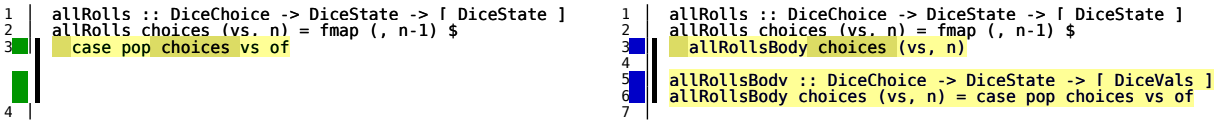

Figure 9.4.: Split function body into separate function

9.7. Substitute definition of `allRolls`

Now we are in the nice situation that, although we are yet to prove it to our satisfaction, the value of `allRollsBody` does not depend on its argument n.

However, we've ended up with a pair of mutually recursive functions! That's somewhat unusual. In order to make `allRollsBody` recurse only on itself we substitute the definition of `allRolls` back into `allRollsBody`. Additionally, that makes `allRolls` not recursive at all.

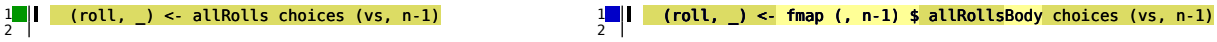

Figure 9.5.: Substitute definition of `allRolls`

9.8. Remove redundant pairing

We're pairing every element of a list with $n-1$ and then immediately removing it. Let's just avoid the pairing in the first place.

9.9. Generalise type of `allRollsBody`

Now the magic happens! Our code currently looks like this.

```
allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls choices (vs, n) = fmap (, n-1) $ allRollsBody choices (vs, n)

allRollsBody :: DiceChoice -> DiceState -> [ DiceVals ]
allRollsBody choices (vs, n) = case pop choices vs of
  Nothing -> [ [] ]
  Just ((chosen, v), (choices, vs)) -> do
    roll <- allRollsBody choices (vs, n-1)
    [ d:roll | d <- rollList ]
          where rollList = if chosen then [v] else [ 1..6 ]
```

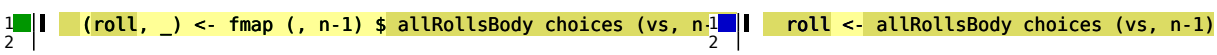

Figure 9.6.: Remove redundant pairing

Previously we had to use our brains to spot that the `Integer` argument to the recursive call was unused. We inserted a run time check to convince ourselves that we were right. Now we have split the original function into two, only one of which contains a recursive call. We can see clearly that the only use for the argument n to `allRollsBody` is to be modified and passed to the recursive call. The value of that argument is never used in any other way. From that observation alone we are probably satisfied that we can remove it.

In fact we can go one step better. We can make a small change to our code so that we do not even have to inspect the implementation to know that n is unused. The compiler will check the property for us! However, the check is demonstrated in a strange way, and if you're not familiar with it then it will look utterly bizarre.

We generalise the type signature so that the function doesn't just work for an `Integer` but rather for *any* type of numeric argument `a`, that is, any type `a` with an instance of the `Num` type class.

```
allRollsBody :: Num a => DiceChoice -> (DiceVals, a) -> [DiceVals]
```

Believe it or not, from this type signature alone, without knowing anything about the implementation, we can conclude that the a argument is not used! How on earth can we conclude that? It's because of the parametricity property enjoyed by Haskell's type system. Basically, the type signature says that the only operations involving type a that `allRollsBody` can use are the ones from the `Num` type class. Looking at them, we see that they give us a way to *make* new as from an `Integer` (`fromInteger`) and ways to combine as to give other as (`+`, `*`, etc.). On the other hand, there is no way that a value of another type can be *created from* an `a`. Therefore, the only way that an argument of type a could be used to affect the result is if the type variable a appears in the type of the result. The result has type `[DiceVals]` so it cannot be affected by the argument of type `a`!

If that seems baffling to you, do not be despondent. Although parametricity is an extremely sophisticated property using it in practice becomes second nature. Haskell programmers use it to great advantage in creating APIs which remain flexible whilst providing strong guarantees via their type signatures.

9.10. Remove unused argument

One way or another, our program transformations have taken us to a place where we feel comfortable removing the *Integer* argument without having to think too hard about the justification. We can inspect the body and see that the argument is not used in a way that can affect the result or we can use parametricity to deduce the same thing. Either way we can remove it.

```
allRolls :: DiceChoice -> DiceState -> [ DiceState ]
allRolls choices (vs, n) = fmap (, n-1) $ allRollsBody choices vs
```

```haskell
allRollsBody :: DiceChoice -> DiceVals -> [ DiceVals ]
allRollsBody choices vs = case pop choices vs of
  Nothing -> [ [] ]
  Just ((chosen, v), (choices, vs)) -> do
    roll <- allRollsBody choices vs
    [ d:roll | d <- rollList ]
          where rollList = if chosen then [v] else [ 1..6 ]
```

9.11. Conclusion

In the earlier article I said that "the only way I can suggest that one discovers [that the argument is unused] is to think through how the the code actually works... this is not just a simple refactoring". I was wrong! There is a small sequence of simple transformations that improve the code whilst at the same time taking us to a place where we easily see that the argument is unused. For the latter either we use a small amount of brainpower to inspect the implementation or we take advantage of parametricity. This is great! We want to save as much brainpower as possible for the really hard problems.

A small amount of Haskell knowledge was required but that is because the code is written in Haskell. Other languages will have their own particular constructs and equivalent transformations, although if they lack `case` statements and expression-valued blocks the transformations might appear a bit more clunky. The only typed-functional-language-specific concept in this article is parametricity. In this small example we were happy to just inspect the body of the function. Parametricity really shines in more complicated codebases where unrelated, opaque, pieces of functionality are being combined.

10. Weakly Typed Haskell - Michael Snoyman

William Yao:
A short example of preventing errors by restricting our function inputs. Uses the streaming library conduit for its example, but should be understandable without knowing too much about it.

Original article: [9]

I was recently doing a minor cleanup of a Haskell codebase. I started off with some code that looked like this:

```
runConduitRes $ sourceFile fp .| someConsumer
```

This code uses Conduit to stream the contents of a file into a consumer function, and ResourceT to ensure that the code is exception safe (the file is closed regardless of exceptions). For various reasons (not relevant to our discussion now), I was trying to reduce usage of `ResourceT` in this bit of code, and so I instead wrote:

```
withBinaryFile fp ReadMode $ \h ->
  runConduit $ sourceHandle h .| someConsumer
```

Instead of using `ResourceT` to ensure exception safety, I used the `with` (or `bracket`) pattern embodied by the `withBinaryFile` function. This transformation worked very nicely, and I was able to apply the change to a number of parts of the code base.

However, I noticed an error message from this application a few days later:

```
/some/long/file/path.txt: hGetBufSome: illegal operation (handle is not
open for reading)
```

I looked through my code base, and sure enough I found that in one of the places I'd done this refactoring, I'd written the following instead:

```
withBinaryFile fp WriteMode $ \h ->
  runConduit $ sourceHandle h.| someConsumer
```

Notice how I used `WriteMode` instead of `ReadMode`. It's a simple mistake, and it's obvious when you look at it. The patch to fix this bug was trivial. But I wasn't satisfied with fixing this bug. I wanted to eliminate it from happening again.

10.1. A strongly typed language?

Lots of people believe that Haskell is a strongly typed language. Strong typing means that you catch lots of classes of bugs with the type system itself. (Static typing means that the type checking occurs at compile time instead of runtime.) I disagree: Haskell is `not` a strongly typed language. In fact, my claim is broader:

> *There's no such thing as a strongly typed language*

Instead, you can write your code in strongly typed or weakly typed style. Some language features make it easy to make your programs more strongly typed. For example, Haskell supports:

- Cheap newtype wrappers
- Sum types
- Phantom type arguments
- GADTs

All of these features allow you to more easily add type safety to your code. But here's the rub:

> *You have to add the type safety yourself*

If you want to write a program in Haskell that passes string data everywhere and puts everything in `IO`, you're still writing Haskell, but you're throwing away the potential for getting extra protections from the compiler.

My `withBinaryFile` usage is a small-scale example of this. The `sourceFile` function I'd been using previously looks roughly like:

```
sourceFile :: FilePath -> Source (ResourceT IO) ByteString
```

This says that if you give this function a `FilePath`, it will give you back a stream of bytes, and that it requires `ResourceT` to be present (to register the cleanup function in the case of an exception). Importantly: there's no way you could accidentally try to send data into this file. The type (`Source`) prevents it. If you did something like:

```
runConduitRes $ someProducer .| sourceFile "output.txt"
```

The compiler would complain about the types mismatching, which is exactly what you want! Now, by contrast, let's look at the types of `withBinaryFile` and `sourceHandle`:

```
withBinaryFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
sourceHandle :: Handle -> Source IO ByteString
```

The type signature of `withBinaryFile` uses the bracket pattern, meaning that you provide it with a function to run while the file is open, and it will ensure that it closes the file. But notice something about the type of that inner function: `Handle -> IO a`. It tells you absolutely nothing about whether the file is opened for reading and writing!

The question is: how do we protect ourselves from the kinds of bugs this weak typing allows?

10.2. Quarantining weak typing

Let's capture the basic concept of what I was trying to do in my program with a helper function:

```
withSourceFile :: FilePath -> (Source IO ByteString -> IO a) -> IO a
withSourceFile fp inner =
  withBinaryFile fp ReadMode $ \handle ->
    inner $ sourceHandle handle
```

This function has all of the same weak typing problems in its body as what I wrote before. However, let's look at the use site of this function:

```
withSourceFile fp $ \src -> runConduit $ src .| someConsumer
```

I definitely can't accidentally pass `WriteMode` instead, and if I try to do something like:

```
withSourceFile fp $ \src -> runConduit $ someProducer .| src
```

I'll get a compile time error. In other words:

> *While my function internally is weakly typed, externally it's strongly typed*

This means that all users of my functions get the typing guarantees I've been hoping to provide. We can't eliminate the possibility of weak typing errors completely, since the systems we're running on are ultimately weakly typed. After all, at the OS level, a file descriptor is just an int, and tells you nothing about whether it's read mode, write mode, or even a pointer to some random address in memory.

Instead, our goal in writing strongly typed programs is to contain as much of the weak typing to helper functions as possible, and expose a strongly typed interface for most of our program. By using `withSourceFile` instead of `withBinaryFile`, I now have just one place in my code I need to check the logic of using `ReadMode` correctly, instead of dozens.

10.3. Discipline and best practices

The takeaway here is that you can *always* shoot yourself in the foot. Languages like Haskell are not some piece of magic that will eliminate bugs. You need to follow through with discipline in using the languages well if you want to get the benefits of features like strong typing.

You can use the some kind of technique in many languages. But if you use a language like Haskell with a plethora of features geared towards easy safety, you'll be much more likely to follow through on it.

11. The Trouble with Typed Errors - Matt Parsons

> ***William Yao:***
> *Or: "Avoiding a monolithic error type"*
> *Good, but not necessary when starting out. While the approach described here is pretty cool, it's also somewhat heavyweight. You'll have to decide for yourself whether it's worth the extra cognitive overhead. Truthfully it can often be quite reasonable to have a single, monolithic sum type for all your application/library errors.*
>
> *Original article: [10]*

You, like me, program in either Haskell, or Scala, or F#, or Elm, or PureScript, and you don't like runtime errors. They're awful and nasty! You have to debug them, and they're not represented in the types. Instead, we like to use `Either` (or something isomorphic) to represent stuff that might fail:

```
data Either l r = Left l | Right r
```

`Either` has a `Monad` instance, so you can short-circuit an `Either l r` computation with an `l` value, or bind it to a function on the `r` value.

So, we take our unsafe, runtime failure functions:

```
head   :: [a] -> a
lookup :: k -> Map k v -> v
parse  :: String -> Integer
```

and we use informative error types to represent their possible failures:

```
data HeadError = ListWasEmpty

head :: [a] -> Either HeadError a

data LookupError = KeyWasNotPresent

lookup :: k -> Map k v -> Either LookupError v

data ParseError
    = UnexpectedChar Char String
    | RanOutOfInput

parse :: String -> Either ParseError Integer
```

Except, we don't really use types like `HeadError` or `LookupError`. There's only one way that `head` or `lookup` could fail. So we just use `Maybe` instead. `Maybe a` is just like using `Either () a` – there's only one possible `Left ()` value, and there's only one possible `Nothing` value. (If you're unconvinced, write `newtype Maybe a = Maybe (Either () a)`, derive all the relevant instances, and try and detect a difference between this `Maybe` and the stock one).

But, `Maybe` isn't great – we've lost information! Suppose we have some computation:

```
foo :: String -> Maybe Integer
foo str = do
    c <- head str
    r <- lookup str strMap
    eitherToMaybe (parse (c : r))
```

Now, we try it on some input, and it gives us `Nothing` back. Which step failed? We actually can't know that! All we can know is that *something* failed.

So, let's try using `Either` to get more information on what failed. Can we just write this?

```
foo :: String -> Either ??? Integer
foo str = do
    c <- head str
    r <- lookup str strMap
    parse (c : r)
```

Unfortunately, this gives us a type error. We can see why by looking at the type of »=:

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

The type variable `m` must be an instance of `Monad`, and the type `m` must be exactly the same for the value on the left and the function on the right. `Either LookupError` and `Either ParseError` are not the same type, and so this does not type check.

Instead, we need some way of accumulating these possible errors. We'll introduce a utility function `mapLeft` that helps us:

```
mapLeft :: (l -> l') -> Either l r -> Either l' r
mapLeft f (Left l) = Left (f l)
mapLeft _ r = r
```

Now, we can combine these error types:

```
foo :: String
    -> Either
        (Either HeadError (Either LookupError ParseError))
        Integer
foo str = do
    c <- mapLeft Left (head str)
    r <- mapLeft (Right . Left) (lookup str strMap)
    mapLeft (Right . Right) (parse (c : r))
```

There! Now we can know exactly how and why the computation failed. Unfortunately, that type is a bit of a monster. It's verbose and all the `mapLeft` boilerplate is annoying.

At this point, most application developers will create a "application error" type, and they'll just shove everything that can go wrong into it.

```
data AllErrorsEver
     = AllParseError ParseError
     | AllLookupError LookupError
     | AllHeadError HeadError
     | AllWhateverError WhateverError
     | FileNotFound FileNotFoundError
     | etc...
```

Now, this slightly cleans up the code:

```
foo :: String -> Either AllErrorsEver Integer
foo str = do
    c <- mapLeft AllHeadError (head str)
    r <- mapLeft AllLookupError (lookup str strMap)
    mapLeft AllParseError (parse (c : r))
```

However, there's a pretty major problem with this code. foo is claiming that it can "throw" all kinds of errors – it's being honest about parse errors, lookup errors, and head errors, but it's also claiming that it will throw if files aren't found, "whatever" happens, and etc. There's no way that a call to foo will result in `FileNotFound`, because foo can't even do `IO`! It's absurd. The type is too large! And I have written about keeping your types small (see 4) and how wonderful it can be for getting rid of bugs.

Suppose we want to handle `foo`'s error. We call the function, and then write a case expression like good Haskellers:

```
case foo "hello world" of
    Right val ->
        pure val
    Left err ->
        case err of
            AllParseError parseError ->
                handleParseError parseError
            AllLookupError lookupErr ->
                handleLookupError
            AllHeadError headErr ->
                handleHeadError
            _ ->
                error "impossible?!?!?!"
```

Unfortunately, this code is brittle to refactoring! We've claimed to handle all errors, but we're really not handling many of them. We currently "know" that these are the only errors that can happen, but there's no compiler guarantee that this is the case. Someone might later modify `foo` to throw another error, and this case expression will break. Any case expression that evaluates any result from `foo` will need to be updated.

The error type is too big, and so we introduce the possibility of mishandling it. There's another problem. Let's suppose we know how to handle a case or two of the error, but we must pass the rest of the error cases upstream:

```
bar :: String -> Either AllErrorsEver Integer
bar str =
    case foo str of
        Right val -> Right val
        Left err ->
            case err of
                AllParseError pe ->
                    Right (handleParseError pe)
                _ ->
                    Left err
```

We know that `AllParseError` has been handled by `bar`, because – just look at it! However, the compiler has no idea. Whenever we inspect the error content of `bar`, we must either a) "handle" an error case that has already been handled, perhaps dubiously, or b) ignore the error, and desperately hope that no underlying code ever ends up throwing the error.

Are we done with the problems on this approach? No! There's no guarantee that I throw the right error!

```
head :: [a] -> Either AllErrorsEver a
head (x:xs) = Right x
head [] = Left (AllLookupError KeyWasNotPresent)
```

This code typechecks, but it's `wrong`, because `LookupError` is only supposed to be thrown by `lookup`! It's obvious in this case, but in larger functions and codebases, it won't be so clear.

11.1. Monolithic error types are bad

So, having a monolithic error type has a ton of problems. I'm going to make a claim here:

> *All error types should have a single constructor*

That is, no sum types for errors. How can we handle this?

Let's maybe see if we can make Either any nicer to use. We'll define a few helpers:

```
type (+) = Either
infixr + 5

l :: l -> Either l r
l = Left

r :: r -> Either l r
r = Right
```

Now, let's refactor that uglier Either code with these new helpers:

```
foo :: String
    -> Either
        (HeadError + LookupError + ParseError)
        Integer
foo str = do
    c <- mapLeft l (head str)
    r <- mapLeft (r . l) (lookup str strMap)
    mapLeft (r . r) (parse (c : r))
```

Well, the syntax is nicer. We can `case` over the nested `Either` in the error branch to eliminate single error cases. It's easier to ensure we don't claim to throw errors we don't – after all, GHC will correctly infer the type of `foo`, and if GHC infers a type variable for any `+`, then we can assume that we're not using that error slot, and can delete it.

Unfortunately, there's still the `mapLeft` boilerplate. And expressions which you'd really want to be equal, aren't.

```
x :: Either (HeadError + LookupError) Int
y :: Either (LookupError + HeadError) Int
```

The values `x` and `y` are isomorphic, but we can't use them in a `do` block because they're not exactly equal. If we add errors, then we must revise all `mapLeft` code, as well as all case expressions that inspect the errors. Fortunately, these are entirely compiler-guided refactors, so the chance of messing them up is small. However, they contribute significant boilerplate, noise, and busywork to our program.

11.2. Boilerplate be gone!

Well, turns out, we can get rid of the order dependence and boilerplate with type classes! The most powerful approach is to use "classy prisms" from the `lens` package. Let's translate our types from concrete values to prismatic ones:

```
-- Concrete:
head :: [a] -> Either HeadError a

-- Prismatic:
head :: AsHeadError err => [a] -> Either err a


-- Concrete:
lookup :: k -> Map k v -> Either LookupError v

-- Prismatic:
lookup
    :: (AsLookupError err)
    => k -> Map k v -> Either err v
```

Now, type class constraints don't care about order – `(Foo a, Bar a) => a` and `(Bar a, Foo a) => a` are exactly the same thing as far as GHC is concerned. The `AsXXX` type classes will automatically provide the `mapLeft` stuff for us, so now our foo function looks a great bit cleaner:

```
foo :: (AsHeadError err, AsLookupError err, AsParseError err)
    => String -> Either err Integer
foo str = do
    c <- head str
    r <- lookup str strMap
    parse (c : r)
```

This appears to be a significant improvement over what we've had before! And, most of the boilerplate with the `AsXXX` classes is taken care of via Template Haskell:

```
makeClassyPrisms ''ParseError
-- this line generates the following:

class AsParseError a where
    _ParseError :: Prism' a ParseError
    _UnexpectedChar :: Prism' a (Char, String)
    _RanOutOfInput :: Prism' a ()

instance AsParseError ParseError where
    -- etc...
```

However, we do have to write our own boilerplate when we eventually want to concretely handle these types. We may end up writing a huge `AppError` that all of these errors get injected into.

There's one major, fatal flaw with this approach. While it composes very nicely, it doesn't decompose at all! There's no way to catch a single case and ensure that it's handled. The machinery that prisms give us don't allow us to separate out a single constraint, so we can't pattern match on a single error.

Once again, our types become ever larger, with all of the problems that entails.

11.3. Generics to the rescue!

What we really want is:

- Order independence
- No boilerplate
- Easy composition
- Easy decomposition

In PureScript or OCaml, you can use open variant types to do this flawlessly. Haskell doesn't have open variants, and the attempts to mock them end up quite clumsy to use in practice.

I'm happy to say that the entire job is handled quite nicely with the amazing `generic-lens` package. I created a gist that demonstrates their usage, but the *magic* comes down to this simple fact: there's an instance of the prismatic `AsType` class for `Either`, which allows you to "pluck" a constraint off. This satisfies all of the things I wanted in my list, and we can consider representing errors mostly solved.

11.4. Mostly?

Well, `ExceptT e IO a` still imposes a significant runtime performance hit, and asynchronous exceptions aren't considered here. A bifunctor IO type like `newtype BIO err a = BIO (IO a)` which carries the type class constraints of the errors it contains is promising, but I haven't been able to write a satisfying interface to this yet.

I also haven't used this technique in a large codebase yet, and I don't know how it scales. And the technique does require you to be comfortable with `lens`, which is a fairly high bar for training new folks on. I'm sure that API improvements could be made to make this style more accessible and remove some of the lens knowledge prerequisites.

12. Type-Directed Code Generation - Sandy Maguire

William Yao:
This post is entirely reasonable to skip, as it requires being familiar with some of GHC's esoteric type extensions. Still, it's a cool introduction to using the more powerful features of the language to make interacting with a complex protocol (gRPC) less error-prone.

Original article: [11]

12.1. Context

At work recently I've been working on a library to get idiomatic gRPC support in our Haskell project. I'm quite proud of how it's come out, and thought it'd make a good topic for a blog post. The approach demonstrates several type-level techniques that in my opinion are under-documented and exceptionally useful in using the type-system to enforce external contracts.

Thankfully the networking side of the library had already been done for me by Awake Security, but the interface feels like a thin-wrapper on top of C bindings. I'm *very, very* grateful that it exists, but I wouldn't expect myself to be able to use it in anger without causing an uncaught type error somewhere along the line. I'm sure I'm probably just using it wrong, but the library's higher-level bindings all seemed to be targeted at Awake's implementation of protobuffers.

We wanted a version that would play nicely with proto-lens, which, at time of writing, has no official support for describing RPC services via protobuffers. If you're not familiar with proto-lens, it generates Haskell modules containing idiomatic types and lenses for protobuffers, and can be used directly in the build chain.

So the task was to add support to proto-lens for generating interfaces to RPC services defined in protobuffers.

My first approach was to generate the dumbest possible thing that could work – the idea was to generate records containing fields of the shape `Request -> IO Response`. Of course, with a network involved there is a non-negligible chance of things going wrong, so this interface should expose some means of dealing with errors. However, the protobuffer spec is agnostic about the actual RPC backend used, and so it wasn't clear how to continue without assuming anything about the particulars behind errors.

More worrisome, however, was that RPCs can be marked as streaming – on the side of the client, server, or both. This means, for example, that a method marked as server-streaming has a different interface on either side of the network:

```
serverSide :: Request -> (Response -> IO ()) -> IO ()
clientSide :: Request -> (IO (Maybe Response) -> IO r) -> IO r
```

This is problematic. Should we generate different records corresponding to which side of the network we're dealing with? An early approach I had was to parameterize the same record based on which side of the network, and use a type family to get the correct signature:

```
{-# LANGUAGE DataKinds #-}

data NetworkSide = Client | Server

data MyService side = MyService
  { runServerStreaming :: ServerStreamingType side Request Response
  }

type family ServerStreamingType (side :: NetworkSide) input output where
  ServerStreamingType Server input output =
      input -> (output -> IO ()) -> IO ()

  ServerStreamingType Client input output =
      forall r. input -> (IO (Maybe output) -> IO r) -> IO r
```

This seems like it would work, but in fact the existence of the `forall` on the client-side is "illegally polymorphic" in GHC's eyes, and it will refuse to compile such a thing. Giving it up would mean we wouldn't be able to return arbitrarily-computed values on the client-side while streaming data from the server. Users of the library might be able to get around it by invoking `IORefs` or something, but it would be ugly and non-idiomatic.

So that, along with wanting to be backend-agnostic, made this approach a no-go. Luckily, my brilliant coworker Judah Jacobson (who is coincidentally also the author of proto-lens), suggested we instead generate metadata for RPC services in proto-lens, and let backend library code figure it out from there.

With all of that context out of the way, we're ready to get into the actual meat of the post. Finally.

12.2. Generating Metadata

According to the spec, a protobuffer service may contain zero or more RPC methods. Each method has a request and response type, either of which might be marked as streaming.

While we could represent this metadata at the term-level, that won't do us any favors in terms of getting type-safe bindings to this stuff. And so, we instead turn to `TypeFamilies`, `DataKinds` and `GHC.TypeLits`.

For reasons that will become clear later, we chose to represent RPC services via types, and methods in those services as symbols (type-level strings). The relevant typeclasses look like this:

```
class Service s where
  type ServiceName     s :: Symbol

class HasMethod s (m :: Symbol) where
```

```
  type MethodInput       s m :: *
  type MethodOutput      s m :: *
  type IsClientStreaming s m :: Bool
  type IsServerStreaming s m :: Bool
```

For example, the instances generated for the RPC service:

```
service MyService {
  rpc BiDiStreaming(stream Request) returns(stream Response);
}
```

would look like this:

```
data MyService = MyService

instance Service MyService where
  type ServiceName    MyService = "myService"

instance HasMethod MyService "biDiStreaming" where
  type MethodInput       MyService "biDiStreaming" = Request
  type MethodOutput      MyService "biDiStreaming" = Response
  type IsClientStreaming MyService "biDiStreaming" = 'True
  type IsServerStreaming MyService "biDiStreaming" = 'True
```

You'll notice that these typeclasses perfectly encode all of the information we had in the protobuffer definition. The idea is that with all of this metadata available to them, specific backends can generate type-safe interfaces to these RPCs. We'll walk through the implementation of the gRPC bindings together.

12.3. The Client Side

The client side of things is relatively easy. We can the HasMethod instance directly:

```
runNonStreamingClient
    :: HasMethod s m
    => s
    -> Proxy m
    -> MethodInput s m
    -> IO (Either GRPCError (MethodOutput s m))
runNonStreamingClient =  -- call the underlying gRPC code

runServerStreamingClient
    :: HasMethod s m
    => s
    -> Proxy m
    -> MethodInput s m
    -> (IO (Either GRPCError (Maybe (MethodOutput s m)) -> IO r)
    -> IO r
runServerStreamingClient =  -- call the underlying gRPC code

-- etc
```

This is a great start! We've got the interface we wanted for the server-streaming code, and our functions are smart enough to require the correct request and response types.

However, there's already some type-unsafety here; namely that nothing stops us from calling `runNonStreamingClient` on a streaming method, or other such silly things.

Thankfully the fix is quite easy – we can use type-level equality to force callers to be attentive to the streaming-ness of the method:

```
runNonStreamingClient
    :: ( HasMethod s m
       , IsClientStreaming s m ~ 'False
       , IsServerStreaming s m ~ 'False
       )
    => s
    -> Proxy m
    -> MethodInput s m
    -> IO (Either GRPCError (MethodOutput s m))

runServerStreamingClient
    :: ( HasMethod s m
       , IsClientStreaming s m ~ 'False
       , IsServerStreaming s m ~ 'True
       )
    => s
    -> Proxy m
    -> MethodInput s m
    -> (IO (Either GRPCError (Maybe (MethodOutput s m)) -> IO r)
    -> IO r

-- et al.
```

Would-be callers attempting to use the wrong function for their method will now be warded off by the type-system, due to the equality constraints being unable to be discharged. Success!

The actual usability of this code leaves much to be desired (it requires being passed a proxy, and the type errors are absolutely *disgusting*), but we'll circle back on improving it later. As it stands, this code is type-safe, and that's good enough for us for the time being.

12.4. The Server Side

12.4.1. Method Discovery

Prepare yourself (but don't panic!): the server side of things is significantly more involved.

In order to run a server, we're going to need to be able to handle any sort of request that can be thrown at us. That means we'll need an arbitrary number of handlers, depending on the service in question. An obvious thought would be to generate a record we could consume that would contain handlers for every method, but there's

no obvious place to generate such a thing. Recall: proto-lens can't, since such a type would be backend-specific, and so our only other strategy down this path would be Template Haskell. Yuck.

Instead, recall that we have an instance of `HasMethod` for every method on `Service s` – maybe we could exploit that information somehow? Unfortunately, without Template Haskell, there's no way to discover typeclass instances.

But that doesn't mean we're stumped. Remember that we control the code generation, and so if the representation we have isn't powerful enough, we can change it. And indeed, the representation we have isn't quite enough. We can go from a `HasMethod s m` to its `Service s`, but not the other way. So let's change that.

We change the Service class slightly:

```
class Service s where
  type ServiceName    s :: Symbol
  type ServiceMethods s :: [Symbol]
```

If we ensure that the `ServiceMethods s` type family always contains an element for every instance of `HasService`, we'll be able to use that info to discover our instances. For example, our previous `MyService` will now get generated thusly:

```
data MyService = MyService

instance Service MyService where
  type ServiceName    MyService = "myService"
  type ServiceMethods MyService = '["biDiStreaming"]

instance HasMethod MyService "biDiStreaming" where
  type MethodInput       MyService "biDiStreaming" = Request
  type MethodOutput      MyService "biDiStreaming" = Response
  type IsClientStreaming MyService "biDiStreaming" = 'True
  type IsServerStreaming MyService "biDiStreaming" = 'True
```

and we would likewise add the `m` for any other `HasMethod MyService m` instances if they existed.

This seems like we can now use `ServiceMethods s` to get a list of methods, and then somehow type-level `map` over them to get the `HasMethod s m` constraints we want.

And we almost can, except that we haven't told the type-system that `ServiceMethods s` relates to `HasService s m` instances in this way. We can add a superclass constraint to `Service` to do this:

```
class HasAllMethods s (ServiceMethods s) => Service s where
  -- as before
```

But what is this `HasAllMethods` thing? It's a specialized type-level `map` which turns our list of methods into a bunch of constraints proving we have `HasMethod s m` for every `m` in that promoted list.

```
class HasAllMethods s (xs :: [Symbol])

instance HasAllMethods s '[]
instance (HasMethod s x, HasAllMethods s xs) => HasAllMethods s (x ': xs)
```

We can think of `xs` here as the list of constraints we want. Obviously if we don't want any constraints (the `'[]` case), we trivially have all of them. The other case is induction: if we have a non-empty list of constraints we're looking for, that's the same as looking for the tail of the list, and having the constraint for the head of it.

Read through these instances a few times; make sure you understand the approach before continuing, because we're going to keep using this technique in scarier and scarier ways.

With this `HasAllMethods` superclass constraint, we can now convince ourselves (and, more importantly, GHC), that we can go from a `Service s` constraint to all of its `HasMethod s m` constraints. Cool!

12.4.2. Typing the Server

We return to thinking about how to actually run a server. As we've discussed, such a function will need to be able to handle every possible method, and, unfortunately, we can't pack them into a convenient data structure.

Our actual implementation of such a thing might take a list of handlers. But recall that each handler has different input and output types, as well as different shapes depending on which bits of it are streaming. We can make this approach work by existentializing away all of the details.

While it works as far as the actual implementation of the underlying gRPC goes, we're left with a great sense of uneasiness. We have no guarantees that we've provided a handler for every method, and the very nature of existentialization means we have absolutely no guarantees that any of these things are the right ype.

Our only recourse is to somehow use our `Service s` constraint to put a prettier facade in front of this ugly-if-necessary implementation detail.

The actual interface we'll eventually provide will, for example, for a service with two methods, look like this:

```
runServer :: HandlerForMethod1 -> HandlerForMethod2 -> IO ()
```

Of course, we can't know a priori how many methods there will be (or what type their handlers should have, for that matter). We'll somehow need to extract this information from `Service s` – which is why we previously spent so much effort on making the methods discoverable.

The technique we'll use is the same one you'll find yourself using again and again when you're programming at the type-level. We'll make a typeclass with an associated type family, and then provide a base case and an induction case.

```
class HasServer s (xs :: [Symbol]) where
  type ServerType s xs :: *
```

We need to make the methods `xs` explicit as parameters in the typeclass, so that we can reduce them. The base case is simple – a server with no more handlers is just an `IO` action:

```
instance HasServer s '[] where
  type ServerType s '[] = IO ()
```

The induction case, however, is much more interesting:

```
instance ( HasMethod s x
         , HasMethodHandler s x
         , HasServer s xs
         ) => HasServer s (x ': xs) where
  type ServerType s (x ': xs) = MethodHandler s x -> ServerType s xs
```

The idea is that as we pull methods `x` off our list of methods to handle, we build a function type that takes a value of the correct type to handle method `x`, which will take another method off the list until we're out of methods to handle. This is exactly a type-level fold over a list.

The only remaining question is "what is this `MethodHandler` thing?" It's going to have to be a type family that will give us back the correct type for the handler under consideration. Such a type will need to dispatch on the streaming variety as well as the request and response, so we'll define it as follows, and go back and fix `HasServer` later.

```
class HasMethodHandler input output (cs :: Bool) (ss :: Bool) where
  type MethodHandler input output cs ss :: *
```

`cs` and `ss` refer to whether we're looking for client-streaming and/or server-streaming types, respectively.

Such a thing could be a type family, but isn't because we'll need its class-ness later in order to actually provide an implementation of all of this stuff. We provide the following instances:

```
-- non-streaming
instance HasMethodHandler input output 'False 'False where
  type MethodHandler input output 'False 'False =
    input -> IO output

-- server-streaming
instance HasMethodHandler input output 'False 'True where
  type MethodHandler input output 'False 'True =
    input -> (output -> IO ()) -> IO ()

-- etc for client and bidi streaming
```

With `MethodHandler` now powerful enough to give us the types we want for handlers, we can go back and fix `HasServer` so it will compile again:

```
instance ( HasMethod s x
         , HasMethodHandler (MethodInput       s x)
                            (MethodOutput      s x)
                            (IsClientStreaming s x)
                            (IsServerStreaming s x)
         , HasServer s xs
         ) => HasServer s (x ': xs) where
  type ServerType s (x ': xs)
      = MethodHandler (MethodInput       s x)
```

```
                    (MethodOutput      s x)
                    (IsClientStreaming s x)
                    (IsServerStreaming s x)
    -> ServerType s xs
```

It's not pretty, but it works! We can convince ourselves of this by asking ghci:

```
ghci> :kind! ServerType MyService (ServiceMethods MyService)

(Request -> (Response -> IO ()) -> IO ()) -> IO () :: *
```

and, if we had other methods defined for `MyService`, they'd show up here with the correct handler type, in the order they were listed in `ServiceMethods MyService`.

12.4.3. Implementing the Server

Our `ServerType` family now expands to a function type which takes a handler value (of the correct type) for every method on our service. That turns out to be more than half the battle – all we need to do now is to provide a value of this type.

The generation of such a value is going to need to proceed in perfect lockstep with the generation of its type, so we add to the definition of `HasServer`:

```
class HasServer s (xs :: [Symbol]) where
  type ServerType s xs :: *
  runServerImpl :: [AnyHandler] -> ServerType s xs
```

What is this `[AnyHandler]` thing, you might ask. It's an explicit accumulator for existentialized handlers we've collected during the fold over `xs`. It'll make sense when we look at the induction case. For now, however, the base case is trivial as always:

```
instance HasServer s '[] where
  type ServerType s '[] = IO ()
  runServerImpl handlers = runGRPCServer handlers
```

where `runGRPCServer` is the underlying server provided by Awake's library. We move to the induction case:

```
instance ( HasMethod s x
         , HasMethodHandler (MethodInput       s x)
                            (MethodOutput      s x)
                            (IsClientStreaming s x)
                            (IsServerStreaming s x)
         , HasServer s xs
         ) => HasServer s (x ': xs) where
  type ServerType s (x ': xs)
      = MethodHandler (MethodInput       s x)
                      (MethodOutput      s x)
                      (IsClientStreaming s x)
                      (IsServerStreaming s x)
    -> ServerType s xs
  runServerImpl handlers f = runServerImpl (existentialize f : handlers)
```

where `existentialize` is a new class method we add to `HasMethodHandler`. We will elide it here because it is just a function `MethodHandler i o cs mm -> AnyHandler` and is not particularly interesting if you're familiar with existentialization.

It's evident here what I meant by `handlers` being an explicit accumulator – our recursion adds the parameters it receives into this list so that it can pass them eventually to the base case.

There's a problem here, however. Reading through this implementation of `runServerImpl`, you and I both know what the right-hand-side means, unfortunately GHC isn't as clever as we are. If you try to compile it right now, GHC will complain about the non-injectivity of `HasServer` as implied by the call to `runServerImpl` (and also about `HasMethodHandler` and `existentialize`, but for the exact same reason.)

The problem is that there's nothing constraining the type variables `s` and `xs` on `runServerImpl`. I always find this error confusing (and I suspect everyone does), because in my mind it's perfectly clear from the `HasServer s xs` in the instance constraint. However, because `SeverType` is a type family without any injectivity declarations, it means we can't learn `s` and `xs` from `ServerType s xs`.

Let's see why. For a very simple example, let's look at the following type family:

```
type family NotInjective a where
  NotInjective Int  = ()
  NotInjective Bool = ()
```

Here we have `NotInjective Int ~ ()` and `NotInjective Bool ~ ()`, which means even if we know `NotInjective a ~ ()` it doesn't mean that we know what `a` is – it could be either `Int` or `Bool`.

This is the exact problem we have with `runServerImpl`: even though we know what type `runServerImpl` has (it must be `ServerType s xs`, so that the type on the left-hand of the equality is the same as on the right), that doesn't mean we know what s and xs are! The solution is to explicitly tell GHC via a type signature or type application:

```
instance ( HasMethod s x
         , HasMethodHandler (MethodInput       s x)
                            (MethodOutput      s x)
                            (IsClientStreaming s x)
                            (IsServerStreaming s x)
         , HasServer s xs
         ) => HasServer s (x ': xs) where
  type ServerType s (x ': xs)
      = MethodHandler (MethodInput       s x)
                      (MethodOutput      s x)
                      (IsClientStreaming s x)
                      (IsServerStreaming s x)
     -> ServerType s xs
  runServerImpl handlers f = runServerImpl @s @xs (existentialize f : handlers)
```

(For those of you playing along at home, you'll need to type-apply the monstrous MethodInput and friends to the existentialize as well.)

And finally, we're done! We can slap a prettier interface in front of this runServerImpl to fill in some of the implementation details for us:

```
runServer
    :: forall s
     . ( Service s
       , HasServer s (ServiceMethods s)
       )
    => s
    -> ServerType s (ServiceMethods s)
runServer _ = runServerImpl @s @(ServiceMethods s) []
```

Sweet and typesafe! Yes!

12.5. Client-side Usability

Sweet and typesafe all of this might be, but the user-friendliness on the client-side leaves a lot to be desired. As promised, we'll address that now.

12.5.1. Removing Proxies

Recall that the `runNonStreamingClient` function and its friends require a `Proxy m` parameter in order to specify the method you want to call. However, `m` has kind `Symbol`, and thankfully we have some new extensions in GHC for turning `Symbols` into values.

We can define a new type, isomorphic to `Proxy`, but which packs the fact that it is a `KnownSymbol` (something we can turn into a `String` at runtime):

```
data WrappedMethod (sym :: Symbol) where
  WrappedMethod :: KnownSymbol sym => WrappedMethod sym
```

We change our `run*Client` friends to take this `WrappedMethod` m instead of the `Proxy m` they used to:

```
runNonStreamingClient
    :: ( HasMethod s m
       , IsClientStreaming s m ~ 'False
       , IsServerStreaming s m ~ 'False
       )
    => s
    -> WrappedMethod m
    -> MethodInput s m
    -> IO (Either GRPCError (MethodOutput s m))
```

and, with this change in place, we're ready for the magic syntax I promised earlier.

```
import GHC.OverloadedLabel

instance ( KnownSymbol sym
         , sym ~ sym'
         ) => IsLabel sym (WrappedMethod sym') where
  fromLabel _ = WrappedMethod
```

This `sym ~ sym'` thing is known as the constraint trick for instances, and is necessary here to convince GHC that this can be the only possible instance of `IsLabel` that will give you back `WrappedMethods`.

Now turning on the `{-# LANGUAGE OverloadedLabels #-}` pragma, we've changed the syntax to call these client functions from the ugly:

```
runBiDiStreamingClient MyService (Proxy @"biDiStreaming")
```

into the much nicer:

```
runBiDiStreamingClient MyService #biDiStreaming
```

12.5.2. Better "Wrong Streaming Variety" Errors

The next step in our journey to delightful usability is remembering that the users of our library are only human, and at some point they are going to call the wrong `run*Client` function on their method with a different variety of streaming semantics.

At the moment, the errors they're going to get when they try that will be a few stanza long, the most informative of which will be something along the lines of `unable to match 'False with 'True`. Yes, it's technically correct, but it's entirely useless.

Instead, we can use the `TypeError` machinery from `GHC.TypeLits` to make these error messages actually helpful to our users. If you aren't familiar with it, if GHC ever encounters a `TypeError` constraint it will die with a error message of your choosing.

We will introduce the following type family:

```
type family RunNonStreamingClient (cs :: Bool) (ss :: Bool) :: Constraint where
  RunNonStreamingClient 'False 'False = ()
  RunNonStreamingClient 'False 'True = TypeError
      ( Text "Called 'runNonStreamingClient' on a server-streaming method."
   :$$: Text "Perhaps you meant 'runServerStreamingClient'."
      )
  RunNonStreamingClient 'True 'False = TypeError
      ( Text "Called 'runNonStreamingClient' on a client-streaming method."
   :$$: Text "Perhaps you meant 'runClientStreamingClient'."
      )
  RunNonStreamingClient 'True 'True = TypeError
      ( Text "Called 'runNonStreamingClient' on a bidi-streaming method."
   :$$: Text "Perhaps you meant 'runBiDiStreamingClient'."
      )
```

The `:$$:` type operator stacks message vertically, while `:<>:` stacks it horizontally.

We can change the constraints on `runNonStreamingClient`:

```
runNonStreamingClient
    :: ( HasMethod s m
       , RunNonStreamingClient (IsClientStreaming s m)
                               (IsServerStreaming s m)
       )
    => s
    -> WrappedMethod m
    -> MethodInput s m
    -> IO (Either GRPCError (MethodOutput s m))
```

and similarly for our other client functions. Reduction of the resulting boilerplate is left as an exercise to the reader.

With all of this work out of the way, we can test it:

```
runNonStreamingClient MyService #biDiStreaming

Main.hs:45:13: error:
    * Called 'runNonStreamingClient' on a bidi-streaming method.
      Perhaps you meant 'runBiDiStreamingClient'.
    * In the expression: runNonStreamingClient MyService #bidi
```

Amazing!

12.5.3. Better "Wrong Method" Errors

The other class of errors we expect our users to make is to attempt to call a method that doesn't exist – either because they made a typo, or are forgetful of which methods exist on the service in question.

As it stands, users are likely to get about six stanzas of error messages, from `No instance for (HasMethod s m)` to `Ambiguous type variable 'm0'`, and other terrible things that leak our implementation details. Our first thought might be to somehow emit a `TypeError` constraint if we don't have a `HasMethod s m` instance, but I'm not convinced such a thing is possible.

But luckily, we can actually do better than any error messages we could produce in that way. Since our service is driven by a value (in our example, the data constructor `MyService`), by the time things go wrong we do have a `Service s` instance in scope. Which means we can look up our `ServiceMethods s` and given some helpful suggestions about what the user probably meant.

The first step is to implement a `ListContains` type family so we can determine if the method we're looking for is actually a real method.

```
type family ListContains (n :: k) (hs :: [k]) :: Bool where
  ListContains n '[]       = 'False
  ListContains n (n ': hs) = 'True
  ListContains n (x ': hs) = ListContains n hs
```

In the base case, we have no list to look through, so our needle is trivially not in the haystack. If the head of the list is the thing we're looking for, then it must be in the list. Otherwise, take off the head of the list and continue looking. Simple really, right?

We can now use this thing to generate an error message in the case that the method we're looking for is not in our list of methods:

```
type family RequireHasMethod s (m :: Symbol) (found :: Bool) :: Constraint where
  RequireHasMethod s m 'False = TypeError
      ( Text "No method "
   :<>: ShowType m
   :<>: Text " available for service '"
   :<>: ShowType s
   :<>: Text "'."
```

```
    :$$: Text "Available methods are: "
    :<>: ShowType (ServiceMethods s)
       )
  RequireHasMethod s m 'True = ()
```

If `found ~ 'False`, then the method `m` we're looking for is not part of the service `s`. We produce a nice error message informing the user about this (using `ShowType` to expand the type variables).

We will provide a type alias to perform this lookup:

```
type HasMethod' s m =
  ( RequireHasMethod s m (ListContains m (ServiceMethods s)
  , HasMethod s m
  )
```

Our new `HasMethod' s m` has the same shape as `HasMethod`, but will expand to our custom type error if we're missing the method under scrutiny.

Replacing all of our old `HasMethod` constraints with `HasMethod'` works fantastically:

```
Main.hs:54:15: error:
    * No method "missing" available for service 'MyService'.
      Available methods are: '["biDiStreaming"]
```

Damn near perfect! That list of methods is kind of ugly, though, so we can write a quick pretty printer for showing promoted lists:

```
type family ShowList (ls :: [k]) :: ErrorMessage where
  ShowList '[]  = Text ""
  ShowList '[x] = ShowType x
  ShowList (x ': xs) = ShowType x :<>: Text ", " :<>: ShowList xs
```

Replacing our final `ShowType` with `ShowList` in `RequireHasMethod` now gives us error messages of the following:

```
Main.hs:54:15: error:
    * No method "missing" available for service 'MyService'.
      Available methods are: "biDiStreaming"
```

Absolutely gorgeous.

12.6. Conclusion

This is where we stop. We've used type-level metadata to generate client- and server-side bindings to an underlying library. Everything we've made is entirely typesafe, and provides gorgeous, helpful error messages if the user does anything wrong. We've found a practical use for many of these seemingly-obscure type-level features, and learned a few things in the process.

In the words of my coworker Renzo Carbonara[1]:

"It is up to us, as people who understand a problem at hand, to try and teach the type system as much as we can about that problem. And when we don't understand the problem, talking to the type system about it will help us understand. Remember, the type system is not magic, it is a logical reasoning tool."

This resounds so strongly in my soul, and maybe it will in yours too. If so, I encourage you to go forth and find uses for these techniques to improve the experience and safety of your own libraries.

1 Whose article "Opaleye's sugar on top" was a strong inspiration on me, and subsequently on this post.

13. The Handle Pattern - Jasper van der Joigt

William Yao:
While there are fancy ways of injecting things like DB access and side effects into your application, such as monad transformers or effect algebras, the dead-simplest way to avoid turning your program into IO-soup is to just pass around records of functions. While there's nothing wrong with just leaving things in IO when bootstrapping a new project, if you find yourself needing to abstract over your concrete effects, try reaching for this before something more complicated.

Original article: [12]

13.1. Introduction

I'd like to talk about a design pattern in Haskell that I've been calling *the Handle pattern*. This is far from novel – I've mentioned this before and the idea is definitely not mine. As far as I know, in fact, it has been around since basically forever[1]. Since it is ridiculously close to what we'd call *common sense*[2], it's often used without giving it any explicit thought.

I first started more consciously using this pattern when I was working together with Simon Meier at Better (aka erudify). Simon did a writeup about this pattern as well. But as I was explaining this idea again at last week's HaskellerZ meetup, I figured it was time to do an update of that article.

The *Handle pattern* allows you write stateful applications that interact with external services in Haskell. It complements pure code (e.g. your business logic) well, and it is somewhat the result of iteratively applying the question:

- Can we make it simpler?
- Can we make it simpler still?
- And can we still make it simpler?

The result is a powerful and simple pattern that does not even require Monads[3] or Monad transformers to be useful. This makes it extremely suitable for beginners

1 Well, `System.IO.Handle` has definitely been around for a while

2 If you're reading this article and you're thinking: *"What does this guy keep going on about? This is all so obvious!"* – Well, that's the point!

3 It does require `IO`, but we don't require thinking about `IO` as a `Monad`. If this sounds weird – think of lists. We work with lists all the time but we just consider them lists of things, we don't constantly call them "List Monad" or "The Free Monoid" for that matter.

trying to build their first medium-sized Haskell application. And that does not mean it is beginners-only: this technique has been applied successfully at several Haskell companies as well.

13.2. Context

In Haskell, we try to capture ideas in beautiful, pure and mathematically sound patterns, for example *Monoids*. But at other times, we can't do that. We might be dealing with some inherently mutable state, or we are simply dealing with external code which doesn't behave nicely.

In those cases, we need another approach. What we're going to describe feels suspiciously similar to Object Oriented Programming:

- Encapsulating and hiding state inside objects
- Providing methods to manipulate this state rather than touching it directly
- Coupling these objects together with methods that modify their state

As you can see, it is not exactly the same as Alan Kay's original definition of OOP[4], but it is far from the horrible incidents that permeate our field such as UML, abstract factory factories and broken subtyping.

Before we dig in to the actual code, let's talk about some disclaimers.

Pretty much any sort of Haskell code can be written in this particular way, but *that doesn't mean that you should*. This method relies heavily on `IO`. Whenever you can write things in a pure way, you should attempt to do that and avoid `IO`. This pattern is only useful when `IO` is required.

Secondly, there are many alternatives to this approach: complex monad transformer stacks, interpreters over free monads, uniqueness types, effect systems... I don't want to claim that this method is better than the others. All of these have advantages and disadvantages, so one must always make a careful trade-off.

13.2.1. The module layout

For this pattern, we've got a very well-defined module layout. I believe this helps with recognition which I think is also one of the reasons we use typeclasses like *Monoid*.

When I'm looking at the documentation of libraries I haven't used yet, the types will sometimes look a bit bewildering. But then I see that there's an `instance Monoid`. That's an "Aha!" moment for me. I *know* what a Monoid is. I *know* how they behave. This allows me to get up to speed with this library much faster!

Using a consistent module layout in a project (and even across projects) provides, I think, very similar benefits to that. It allows new people on the team to learn parts of the codebase they are yet unfamiliar with much faster.

4 And indeed, we will touch on a common way of encoding OOP in Haskell – creating explicit records of functions – but we'll also explain why this isn't always necessary.

13.2.2. A Database Handle

Anyway, let's look at the concrete module layout we are proposing with this pattern. As an example, let's consider a database. The type in which we are encapsulating the state is *always* called `Handle`. That is because we design for qualified import.

We might have something like:

```
module MyApp.Database

data Handle = Handle
    { hPool   :: Pool Postgres.Connection
    , hCache  :: IORef (PSQueue Int Text User)
    , hLogger :: Logger.Handle  -- Another handle!
    , ...
    }
```

The internals of the `Handle` typically consist of static fields and other handles, `MVars`, `IORefs`, `TVars`, `Chans`... With our `Handle` defined, we are able to define functions using it. These are usually straightforward imperative pieces of code and I'll omit them for brevity[5]:

```
module MyApp.Database where

data Handle = ...

createUser :: Handle -> Text -> IO User
createUser = ...

getUserMail :: Handle -> User -> IO [Mail]
getUserMail = ...
```

Some thoughts on this design:

1. We call our functions `createUser` rather than `databaseCreateUser`. Again, we're working with qualified imports so there's no need for "C-style" names.

2. **All functions take the `Handle` as the first argument.** This is very important for consistency, but also for polymorphism and code style.

   With code style, I mean that the `Handle` is often a syntactically simpler expression (e.g. a name) than the argument (which is often a composed expression). Consider:

   ```
   Database.createUser database $ userName <> "@" <> companyDomain
   ```

   Versus:

   ```
   Database.createUser (userName <> "@" <> companyDomain) database
   ```

5 If you want to see a full example, you can refer to this repository that I have been using to teach practical Haskell.

3. Other `Handles` (e.g. `Logger.Handle`) are stored in a field of our `Database.Handle`. You could also remove it there and instead have it as an argument wherever it is needed, for example:

   ```
   createUser :: Handle -> Logger.Handle -> Text -> IO User
   createUser = ...
   ```

   I usually prefer to put it inside the `Handle` since that reduces the amount of arguments required for functions such as `createUser`. However, if the lifetime of a `Logger.Handle` is very short[6], or if you want to reduce the amount of dependencies for `new`, then you could consider doing the above.

4. The datatypes such as `Mail` may be defined in this module may even be specific to this function. I've written about ad-hoc datatypes before (see chapter 7).

13.2.3. Creating a Handle

I mentioned before that an important advantage of using these patterns is that programmers become "familiar" with it. That is also the goal we have in mind when designing our API for the creation of `Handles`.

In addition to always having a type called `Handle`, we'll require the module to always have a type called `Config`. This is where we encode our static configuration parameters – and by static I mean that we shouldn't have any `IORefs` or the like here: this `Config` should be easy to create from pure code.

```
module MyApp.Database where

data Config = Config
    { cPath :: FilePath
    , ...
    }

data Handle = ...
```

We can also offer some way to create a `Config`. This really depends on your application. If you use the configurator library, you might have something like:

```
parseConfig :: Configurator.Config -> IO Config
parseConfig = ...
```

On the other hand, if you use aeson or yaml, you could write:

```
instance Aeson.FromJSON Config where
    parseJSON = ...
```

You could even use a Monoid to support loading configurations from multiple places. But I digress – the important part is that there is a type called `Config`.

Next is a similar pattern: in addition to always having a `Config`, we'll also always provide a function called `new`. The parameters follow a similarly strict pattern:

```
new :: Config          -- 1. Config
    -> Logger.Handle   -- 2. Dependencies
    -> ...             --    (usually other handles)
    -> IO Handle       -- 3. Result
```

Inside the `new` function, we can create some more `IORefs`, file handles, caches... if required and then store them in the `Handle`.

13.2.4. Destroying a Handle

We've talked about creation of a `Handle`, and we mentioned the normal functions operating on a `Handle` (e.g. `createUser`) before. So now let's consider the final stage in the lifetime of `Handle`.

Haskell is a garbage collected language and we can let the runtime system take care of destroying things for us – but that's not always a great idea. Many resources (file handles in particular come to mind as an example) are scarce.

There is quite a strong correlation between scarce resources and things you would naturally use a `Handle` for. That's why I recommend always providing a `close` as well, even if does nothing. This is a form of forward compatibility in our API: if we later decide to add some sort of log files (which will need to be closed), we can do so without individually mailing all our module users that they now need to add a `close` to their code.

```
close :: Handle -> IO ()
close = ...
```

13.2.5. Reasonable safety

When you're given a `new` and `close`, it's often tempting to add an auxiliary function like:

```
withHandle
    :: Config            -- 1. Config
    -> Logger.Handle     -- 2. Dependencies
    -> ...               --    (usually other handles)
    -> (Handle -> IO a)  -- 3. Function to apply
    -> IO a              -- 4. Result, handle is closed automatically
```

I think this is a great idea. In fact, it's sometimes useful to *only* provide the `withHandle` function, and hide `new` and `close` in an internal module.

The only caveat is that the naive implementation of this function:

```
withHandle config dep1 dep2 ... depN f = do
    h <- new config dep1 dep2 ... depN
    x <- f h
    close h
    return x
```

Is **wrong**! In any sort of `withXyz` function, you should always use `bracket` to guard against exceptions. This means the correct implementation is:

```
withHandle config dep1 dep2 ... depN f =
    bracket (new config dep1 dep2 ... depN) close f
```

Well, it's even shorter! In case you want more information on why `bracket` is necessary, this blogpost gives a good in-depth overview. My summary of it as it relates to this article would be:

1. Always use `bracket` to match `new` and `close`
2. You can now use `throwIO` and `killThread` safely

It's important to note that `withXyz` functions do not provide complete safety against things like use-after-close or double-close. There are many interesting approaches to fix these issues but they are *way* beyond the scope of this tutorial – things like Monadic Regions and The Linearity Monad come to mind. For now, we'll rely on `bracket` to catch common issues and on code reviews to catch team members who are not using `bracket`.

13.2.6. Summary of the module layout

If we quickly summarise the module layout, we now have:

```
module MyApp.Database
    ( Config (..)    -- Internals exported
    , parseConfig    -- Or some other way to load a config

    , Handle         -- Internals usually not exported
    , new
    , close
    , withHandle

    , createUser  -- Actual functions on the handle
    , ...
    ) where
```

This is a well-structured, straightforward and easy to learn organisation. Most of the `Handles` in any application should probably look this way. In the next section, we'll see how we can build on top of this to create dynamic, customizable `Handles`.

13.3. Handle polymorphism

It's often important to split between the interface and implementation of a service. There are countless ways to do this in programming languages. For Haskell, there is:

- Higher order functions
- Type classes and type families
- Dictionary passing

- Backpack module system
- Interpreters over concrete ASTs
- ...

The list is endless. And because Haskell on one hand makes it so easy to abstract over things, and on the other hand makes it possible to abstract over pretty much anything, I'll start this section with a disclaimer.

Premature abstraction is a real concern in Haskell (and many other high-level programming languages). It's easy to quickly whiteboard an abstraction or interface and unintentionally end up with completely the wrong thing.

It usually goes like this:

1. You need to implement a bunch of things that look similar
2. You write down a typeclass or another interface-capturing abstraction
3. You start writing the actual implementations
4. One of them doesn't *quite* match the interface so you need to change it two weeks in
5. You add another parameter, or another method, mostly for one specific interface
6. This causes some problems or inconsistencies for interfaces
7. Go back to (4)

What you end up with is a leaky abstraction that is the *product* of all concrete implementations – where what you really wanted is the *greatest common divisor*.

There's no magic bullet to avoid broken abstractions so my advice is usually to first painstakingly do all the different implementations (or at least a few of them). *After* you have something working and you have emerged victorous from horrible battles with the guts of these implementations, *then* you could start looking at what the different implementations have in common. At this point, you'll also be a bit wiser about where they differ – and you'll be able to take these important details into account, at which point you retire from just being an idiot drawing squares and arrows on a whiteboard.

This is why I recommend sticking with simple `Handle`s until you really need it. But naturally, sometimes we really need the extra power.

13.3.1. A Handle interface

So let's do the simplest thing that can possibly work. Consider the following definition of the `Handle` we discussed before:

```
module MyApp.Database
    ( Handle (..)  -- We now need to export this
    ) where

data Handle = Handle
```

```
    { createUser :: Text -> IO User
    , ...
    }
```

What's the type of `createUser` now?

```
createUser :: Handle -> Text -> IO User
```

It's exactly the same as before! This is pretty much a requirement: it means we can move our `Handles` to this approach when we need it, not when we envision that we will need it at some point in the future.

13.3.2. A Handle implementation

We can now create a concrete implementation for this abstract `Handle` type. We'll do this in a module like `MyApp.Database.Postgres`.

```
module MyApp.Database.Postgres where
import MyApp.Database

data Config = ...

new :: Config -> Logger.Handle -> ... -> IO Handle
```

The `Config` datatype and the `new` function have now moved to the implementation module, rather than the interface module.

Since we can have any number of implementation modules, it is worth mentioning that we will have multiple `Config` types and `new` functions (exactly one of each per implementation). Configurations are always specific to the concrete implementation. For example, an sqlite database may just have a `FilePath` in the configuration, but our `Postgres` implementation will have other details such as port, database, username and password.

In the implementation of `new`, we simply initialize a `Handle`:

```
new config dep1 dep2 ... depN = do
    -- Intialization of things inside the handle
    ...

    -- Construct record
    return Handle
        { createUser = \name -> do
            ...
        , ...
        }
```

Of course, we can manually float out the body of `createUser` since constructing these large records gets kind of ugly.

13.4. Compared to other approaches

We've presented an approach to modularize the effectful layer of medium- to large-scaled Haskell applications. There are many other approaches to tackling this, so any comparison I come up with would probably be inexhaustive.

Perhaps the most important advantage of using `Handles` is that they are first class values that we can freely mix and match. This often does not come for free when using more exotic strategies.

Consider the following type signature from a Hackage package – and I do not mean to discredit the author, the package works fine but simply uses a different approach than my personal preference:

```
-- | Create JSON-RPC session around conduits from transport layer.
-- When context exits session disappears.
runJsonRpcT
    :: (MonadLoggerIO m, MonadBaseControl IO m)
    => Ver                  -- ^ JSON-RPC version
    -> Bool                 -- ^ Ignore incoming requests/notifs
    -> Sink ByteString m () -- ^ Sink to send messages
    -> Source m ByteString  -- ^ Source to receive messages from
    -> JsonRpcT m a         -- ^ JSON-RPC action
    -> m a                  -- ^ Output of action
```

I'm a fairly experienced Haskeller and it still takes me a bit of eye-squinting to see how this will fit into my application, especially if I want to use this package with other libraries that do not use the `Sink/Source` or `MonadBaseControl` abstractions.

It is somewhat obvious that one running call to `runJsonRpcT` corresponds to being connected to one JSON-RPC endpoint, since it takes a single sink and source. But what if we want our application to be connected to multiple endpoints at the same time?

What if we need to have hundreds of thousands of these, and we want to store them in some priority queue and only consider the most recent ones in the general case. How would you go about that?

You could consider running a lightweight thread for every `runJsonRpcT`, but that means you now need to worry about thread overhead, communicating exceptions between threads and killing the threads after you remove them. Whereas with first-class handles, we would just have a `HashPSQ Text Int JsonRpc.Handle`, which is much easier to reason about.

So, I guess one of the oldest and most widely used approaches is MTL-style monad transformers. This uses a hierarchy of typeclasses to represent access to various sub-systems.

I love working with MTL-style transformers in the case of pure code, since they often allow us to express complex ideas concisely. For effectful code, on the other hand, they do not seem to offer many advantages and often make it harder to reason about code.

My personal preference for writing complex effectful code is to reify the effectful operations as a datatype and then write pure code manipulating these effectful operations. An interpreter can then simply use the `Handles` to perform the effects. For simpler effectful code, we can just use `Handles` directly.

I have implemented a number of these patterns in the (ever unfinished) example web application fugacious, in case you want to see them in action or if you want a more elaborate example than the short snippets in this blogpost. Finally, I would like to thank Alex Lang and Nicolas Mattia for proofreading, and Titouan Vervack for many corrections and typos.

14. The ReaderT Design Pattern - Michael Snoyman

Original article: [13]

Often times I'll receive or read questions online about "design patterns" in Haskell. A common response is that Haskell doesn't have them. What many languages address via patterns, in Haskell we address via language features (like built-in immutability, lambdas, laziness, etc). However, I believe there is still room for some high-level guidance on structuring programs, which I'll loosely refer to as a Haskell design pattern.

The pattern I'm going to describe today is what I've been referring to as the "ReaderT pattern" for a few years now, at least in informal discussions. I use it as the basis for the design of Yesod's `Handler` type, it describes the majority of the Stack code base, and I've recommended it to friends, colleagues, and customers regularly.

That said, this pattern is not universally held in the Haskell world, and plenty of people design their code differently. So remember that, like other articles I've written, this is highly opinionated, but represents my personal and FP Complete's best practices recommendations.

Let's summarize the pattern, and then get into details and exceptions:

- Your application should define a core data type (call it `Env` if you want).
- This data type will contain all runtime configuration and global functions that may be mockable (like logging functions or database access).
- If you must have some mutable state, put it in `Env` as a mutable reference (`IORef`, `TVar`, etc).
- Your application code will, in general, live in `ReaderT Env IO`. Define it as `type App = ReaderT Env IO` if you wish, or use a newtype wrapper instead of `ReaderT` directly.
- You can use additional monad transformers on occassion, but only for small subsets of your application, and it's best if those subsets are pure code.
- Optional: instead of directly using the `App` datatype, write your functions in terms of mtl-style typeclasses like `MonadReader` and `MonadIO`, which will allow you to recover some of the purity you think I just told you to throw away with `IO` and mutable references.

That's a lot to absorb all at once, some of it (like the mtl typeclasses) may be unclear, and other parts (especially that mutable reference bit) probably seems completely wrong. Let me motivate these statements and explain the nuances.

14.1. Better globals

Let's knock out the easy parts of this. Global variables are bad, and mutable globals are far worse. Let's say you have an application which wants to configure its logging level (e.g., should we print or swallow DEBUG level messages?). There are three common ways you might do this in Haskell:

1. Use a compile-time flag to control which logging code gets included in your executable
2. Define a global value which reads a config file (or environment variables) with `unsafePerformIO`
3. Read the config file in your `main` function and then pass the value (explicitly, or implicitly via `ReaderT`) to the rest of your code

(1) is tempting, but experience tells me it's a terrible solution. Every time you have conditional compilation in your codebase, you're adding in a fractal of possible build failures. If you have 5 conditionals, you now have 32 (2^5) possible build configurations. Are you sure you have the right set of import statements for all of those 32 configurations? It's just a pain to deal with. Moreover, do you really want to decide at *compile time* that you're not going to need debug information? I'd much rather be able to flip a `false` to `true` in a config file and restart my app to get more information while debugging.

(By the way, even better than this is the ability to signal the process *while it's running* to change debug levels, but we'll leave that out for now. The `ReaderT`+mutable variable pattern is one of the best ways to achieve this.)

OK, so you've agreed with me that you shouldn't conditionally compile. Now that you've written your whole application, however, you're probably hesitant to have to rewrite it to thread through some configuration value everywhere. I get that, it's a pain. So you decide you'll just use `unsafePerformIO`, since the file will be read once, it's a pure-ish value for the entire runtime, and everything seems mostly safe. However:

- You now have a slight level of non-determinism of where exceptions will occur. If you have a missing or invalid config file, where will the exception get thrown from? I'd much rather that occur immediately on app initialization.
- Suppose you want to run one small part of the application with louder debug information (because you know it's more likely to fail thant the rest of your code). You basically can't do this at all.
- Just wait until you use STM inside your config file parsing for some reason, and then your config value first gets evaluated inside a *different* STM block. (And there's no way that could ever happen.)
- Every time you use `unsafePerformIO`, a kitten dies.

It's time to just bite the bullet, define some `Env` data type, put the config file values in it, and thread it through your application. If you design your application from the beginning like this: great, no harm done. Doing it later in application development is

certainly a pain, but the pain can be mitigated by some of what I'll say below. And it is *absolutely* better to suck up a little pain with mechanical code rewriting than be faced with race conditions around `unsafePerformIO`. Remember, this is Haskell: we'd rather face compile time rather than runtime pain.

Once you've accepted your fate and gone all-in on (3), you're on easy street:

- Have some new config value to pass around? Easy, just augment the `Env` type.
- Want to temporarily bump the log level? Use `local` and you're golden

You're now far less likely to resort to ugly hacks like CPP code (1) or global variables (2), because you've eliminated the potential pain from doing it the Right Way.

14.1.1. Initializing resources

The case of reading a config value is nice, but even nicer is initializing some resources. Suppose you want to initialize a random number generator, open up a `Handle` for sending log messages to, set up a database pool, or create a temporary directory to store files in. These are all far more logical to do inside `main` than from some global position.

One advantage of the global variable approach for these kinds of initializations is that it can be defered until the first time the value is used, which is nice if you think some resources may not always be needed. But if that's what you want, you can use an approach like `runOnce`.

14.2. Avoiding WriterT and StateT

Why in the world would I ever recommend mutable references as anything but a last resort? We all know that purity in Haskell is paramount, and mutability is the devil. And besides, we have these wonderful things called `WriterT` and `StateT` if we have some values that need to change over time in our application, so why not use them?

In fact, early versions of Yesod did just that: they used a `StateT` kind of approach within `Handler` to allow you to modify the user session values and set response headers. However, I switched over to mutable references quite a while ago, and here's why:

Exception-survival If you have a runtime exception, you will *lose your state* in `WriterT` and `StateT`. Not so with a mutable reference: you can read the last available state before the runtime exception was thrown. We use this to great benefit in Yesod, to be able to set response headers even if the response fails with something like a `notFound`.

False purity We say `WriterT` and `StateT` are pure, and technically they are. But let's be honest: if you have an application which is entirely living within a `StateT`, you're not getting the benefits of restrained mutation that you want from pure code. May as well call a spade a spade, and accept that you have a mutable variable.

Concurrency What's the result of `put 4 >> concurrently (modify (+ 1)) (modify (+ 2)) >> get`? You may *want* to say that it will be `7`, but it definitely won't be. Your options, depending on how `concurrently` is implemented with regards to the state provided by `StateT`, are `4`, `5`, or `6`. Don't believe me? Play around with:

```
#!/usr/bin/env stack
-- stack --resolver lts-8.12 script
import Control.Concurrent.Async.Lifted
import Control.Monad.State.Strict

main :: IO ()
main = execStateT
    (concurrently (modify (+ 1)) (modify (+ 2)))
    4 >>= print
```

The issue is that we need to clone the state from the parent thread into both child threads, and then *arbitrarily pick* which child state will survive. Or, if we want to, we can just throw both child states away and continue with the original parent state. (By the way, if you think the fact that this code compiles is a bad thing, I agree, and suggest you use `Control.Concurrent.Async.Lifted.Safe`.)

Dealing with mutable state between different threads is a hard problem, but `StateT` doesn't *fix* the problem, it *hides* it. If you use a mutable variable, you'll be forced to think about this. What semantics do we want? Should we use an `IORef`, and stick to `atomicModifyIORef`? Should we use a `TVar`? These are fair questions, and ones we'll be forced to examine. For a `TVar`-like approach:

```
#!/usr/bin/env stack
-- stack --resolver lts-8.12 script
{-# LANGUAGE FlexibleContexts #-}
import Control.Concurrent.Async.Lifted.Safe
import Control.Monad.Reader
import Control.Concurrent.STM

modify :: (MonadReader (TVar Int) m, MonadIO m)
       => (Int -> Int)
       -> m ()
modify f = do
  ref <- ask
  liftIO $ atomically $ modifyTVar' ref f

main :: IO ()
main = do
  ref <- newTVarIO 4
  runReaderT (concurrently (modify (+ 1)) (modify (+ 2))) ref
  readTVarIO ref >>= print
```

And you can even get a little fancier with prebaked transformers.

WriterT is broken Don't forget that, as Gabriel Gonzalez has demonstrated, even the strict `WriterT` has a space leak.

Caveats I still do use `StateT` and `WriterT` sometimes. One prime example is Yesod's `WidgetT`, which is essentially a `WriterT` sitting on top of `HandlerT`. It makes sense in that context because:

- The mutable state is expected to be modified for a small subset of the application

- Although we can perform side-effects while building up the widget, the widget construction itself is a morally pure activity
- We don't need to let state survive an exception: if something goes wrong, we'll send back an error page instead
- There's no good reason to use concurrency when constructing a widget.
- Despite my space leak concerns, I thoroughly benchmarked `WriterT` against alternatives, and found it to be the fastest for this use case. (Numbers beat reasoning.)

The other great exception to this rule is pure code. If you have some subset of your application which can perform no `IO` but needs some kind of mutable state, absolutely, 100%, please use `StateT`.

14.3. Avoiding ExceptT

I'm already strongly on record as saying that `ExceptT` over `IO` is a bad idea. To briefly repeat myself: the contract of `IO` is that any exception can be thrown at any time, so `ExceptT` doesn't actually document possible exceptions, it misleads. You can see that blog post for more details.

I'm rehashing this here because some of the downsides of `StateT` and `WriterT` apply to `ExceptT` as well. For example, how do you handle concurrency in an `ExceptT`? With runtime exceptions, the behavior is clear: when using `concurrently`, if any child thread throws an exception, the other thread is killed and the exception rethrown in the parent. What behavior do you want with `ExceptT`?

Again, you can use `ExceptT` from pure code where a runtime exception is not part of the contract, just like you should use `StateT` in pure code. But once we've eliminated `StateT`, `WriterT`, and `ExceptT` from our main application transformers, we're left with...

14.4. Just ReaderT

And now you know why I call this "the ReaderT design pattern". `ReaderT` has a huge advantage over the other three transformers listed: *it has no mutable state*. It's simply a convenient manner of passing an extra parameter to all functions. And even if that parameter contains mutable references, that parameter itself is fully immutable. Given that:

- We get to ignore all of the state-overwriting issues I mentioned with concurrency. Notice how we were able to use the `.Safe` module in the example above. That's because it is *actually safe* to do concurrency with a `ReaderT`.
- Similarly, you can use the `monad-unlift` library package
- Deep monad transformer stacks are confusing. Knocking it all down to just one transformer reduces complexity, significantly.

- It's not just simpler for you. It's simpler for GHC too, which tends to have a much better time optimizing one-layer `ReaderT` code versus 5-transformer-deep code.

By the way, once you've bought into `ReaderT`, you can just throw it away entirely, and manually pass your `Env` around. Most of us don't do that, because it feels masochistic (imagine having to tell *every call to `logDebug`* where to get the logging function). But if you're trying to write a simpler codebase that doesn't require understanding of transformers, it's now within your grasp.

14.5. Has typeclass approach

Let's say we're going to expand our mutable variable example above to include a logging function. It may look something like this:

```
#!/usr/bin/env stack
-- stack --resolver lts-8.12 script
{-# LANGUAGE FlexibleContexts #-}
import Control.Concurrent.Async.Lifted.Safe
import Control.Monad.Reader
import Control.Concurrent.STM
import Say

data Env = Env
  { envLog :: !(String -> IO ())
  , envBalance :: !(TVar Int)
  }

modify :: (MonadReader Env m, MonadIO m)
       => (Int -> Int)
       -> m ()
modify f = do
  env <- ask
  liftIO $ atomically $ modifyTVar' (envBalance env) f

logSomething :: (MonadReader Env m, MonadIO m)
             => String
             -> m ()
logSomething msg = do
  env <- ask
  liftIO $ envLog env msg

main :: IO ()
main = do
  ref <- newTVarIO 4
  let env = Env
        { envLog = sayString
        , envBalance = ref
```

```
      }
  runReaderT
    (concurrently
      (modify (+ 1))
      (logSomething "Increasing account balance"))
    env
  balance <- readTVarIO ref
  sayString $ "Final balance: " ++ show balance
```

Your first reaction to this is probably that defining this `Env` data type for your application looks like overhead and boilerplate. You're right, it is. Like I said above though, it's better to just suck up some of the pain initially to make a better long-term application development practice. Now let me double down on that...

There's a bigger problem with this code: it's *too coupled*. Our `modify` function takes in an entire `Env` value, even though it never uses the logging function. And similarly, `logSomething` never uses the mutable variable it's provided. Exposing too much state to a function is bad:

- We can't, from the type signature, get any idea about what the code is doing
- It's more difficult to test. In order to see if `modify` is doing the right thing, we need to provide it some garbage logging function.

So let's double down on that boilerplate, and use the `Has` typeclass trick. This composes well with `MonadReader` and other mtl classes like `MonadThrow` and `MonadIO` to allow us to state *exactly* what our function needs, at the cost of having to define a lot of typeclasses upfront. Let's see how this looks:

```
#!/usr/bin/env stack
-- stack --resolver lts-8.12 script
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
import Control.Concurrent.Async.Lifted.Safe
import Control.Monad.Reader
import Control.Concurrent.STM
import Say

data Env = Env
  { envLog :: !(String -> IO ())
  , envBalance :: !(TVar Int)
  }

class HasLog a where
  getLog :: a -> (String -> IO ())
instance HasLog (String -> IO ()) where
  getLog = id
instance HasLog Env where
  getLog = envLog

class HasBalance a where
```

```
  getBalance :: a -> TVar Int
instance HasBalance (TVar Int) where
  getBalance = id
instance HasBalance Env where
  getBalance = envBalance

modify :: (MonadReader env m, HasBalance env, MonadIO m)
       => (Int -> Int)
       -> m ()
modify f = do
  env <- ask
  liftIO $ atomically $ modifyTVar' (getBalance env) f

logSomething :: (MonadReader env m, HasLog env, MonadIO m)
             => String
             -> m ()
logSomething msg = do
  env <- ask
  liftIO $ getLog env msg

main :: IO ()
main = do
  ref <- newTVarIO 4
  let env = Env
        { envLog = sayString
        , envBalance = ref
        }
  runReaderT
    (concurrently
      (modify (+ 1))
      (logSomething "Increasing account balance"))
    env
  balance <- readTVarIO ref
  sayString $ "Final balance: " ++ show balance
```

Holy creeping boilerplate batman! Yes, type signatures get longer, rote instances get written. But our type signatures are now deeply informative, and we can test our functions with ease, e.g.:

```
main :: IO ()
main = hspec $ do
  describe "modify" $ do
    it "works" $ do
      var <- newTVarIO (1 :: Int)
      runReaderT (modify (+ 2)) var
      res <- readTVarIO var
      res `shouldBe` 3
  describe "logSomething" $ do
    it "works" $ do
```

```
    var <- newTVarIO ""
    let logFunc msg = atomically $ modifyTVar var (++ msg)
        msg1 = "Hello "
        msg2 = "World\n"
    runReaderT (logSomething msg1 >> logSomething msg2) logFunc
    res <- readTVarIO var
    res `shouldBe` (msg1 ++ msg2)
```

And if defining all of these classes manually bothers you, or you're just a big fan of the library, you're free to use lens:

```
#!/usr/bin/env stack
-- stack --resolver lts-8.12 script
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
import Control.Concurrent.Async.Lifted.Safe
import Control.Monad.Reader
import Control.Concurrent.STM
import Say
import Control.Lens
import Prelude hiding (log)

data Env = Env
  { envLog :: !(String -> IO ())
  , envBalance :: !(TVar Int)
  }

makeLensesWith camelCaseFields ''Env

modify :: (MonadReader env m, HasBalance env (TVar Int), MonadIO m)
       => (Int -> Int)
       -> m ()
modify f = do
  env <- ask
  liftIO $ atomically $ modifyTVar' (env^.balance) f

logSomething :: (MonadReader env m, HasLog env (String -> IO ()), MonadIO m)
             => String
             -> m ()
logSomething msg = do
  env <- ask
  liftIO $ (env^.log) msg

main :: IO ()
main = do
  ref <- newTVarIO 4
```

```
  let env = Env
        { envLog = sayString
        , envBalance = ref
        }
  runReaderT
    (concurrently
      (modify (+ 1))
      (logSomething "Increasing account balance"))
    env
  balance <- readTVarIO ref
  sayString $ "Final balance: " ++ show balance
```

In our case, where `Env` doesn't have any immutable config-style data in it, the advantages of the lens approach aren't as apparent. But if you have some deeply nested config value, and especially want to play around with using `local` to tweak some values in it throughout your application, the lens approach can pay off.

So to summarize: this approach really is about biting the bullet and absorbing some initial pain and boilerplate. I argue that the myriad benefits you get from it during app development are well worth it. Remember: you'll pay that upfront cost *once*, you'll reap its rewards daily.

14.6. Regain purity

It's unfortunate that our `modify` function has a `MonadIO` constraint on it. Even though our real implementation requires `IO` to perform side-effects (specifically, to read and write a `TVar`), we've now infected all callers of the function by saying "we have the right to perform *any* side-effects, including launching the missiles, or worse, throwing a runtime exception". Can we regain some level of purity? The answer is yes, it just requires a bit more boilerplate:

```
#!/usr/bin/env stack
-- stack --resolver lts-8.12 script
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
import Control.Concurrent.Async.Lifted.Safe
import Control.Monad.Reader
import qualified Control.Monad.State.Strict as State
import Control.Concurrent.STM
import Say
import Test.Hspec

data Env = Env
  { envLog :: !(String -> IO ())
  , envBalance :: !(TVar Int)
  }

class HasLog a where
  getLog :: a -> (String -> IO ())
```

```
instance HasLog (String -> IO ()) where
  getLog = id
instance HasLog Env where
  getLog = envLog

class HasBalance a where
  getBalance :: a -> TVar Int
instance HasBalance (TVar Int) where
  getBalance = id
instance HasBalance Env where
  getBalance = envBalance

class Monad m => MonadBalance m where
  modifyBalance :: (Int -> Int) -> m ()
instance (HasBalance env, MonadIO m) => MonadBalance (ReaderT env m) where
  modifyBalance f = do
    env <- ask
    liftIO $ atomically $ modifyTVar' (getBalance env) f
instance Monad m => MonadBalance (State.StateT Int m) where
  modifyBalance = State.modify

modify :: MonadBalance m => (Int -> Int) -> m ()
modify f = do
  -- Now I know there's no way I'm performing IO here
  modifyBalance f

logSomething :: (MonadReader env m, HasLog env, MonadIO m)
             => String
             -> m ()
logSomething msg = do
  env <- ask
  liftIO $ getLog env msg

main :: IO ()
main = hspec $ do
  describe "modify" $ do
    it "works, IO" $ do
      var <- newTVarIO (1 :: Int)
      runReaderT (modify (+ 2)) var
      res <- readTVarIO var
      res `shouldBe` 3
  it "works, pure" $ do
      let res = State.execState (modify (+ 2)) (1 :: Int)
      res `shouldBe` 3
  describe "logSomething" $ do
    it "works" $ do
      var <- newTVarIO ""
      let logFunc msg = atomically $ modifyTVar var (++ msg)
          msg1 = "Hello "
```

```
    msg2 = "World\n"
runReaderT (logSomething msg1 >> logSomething msg2) logFunc
res <- readTVarIO var
res `shouldBe` (msg1 ++ msg2)
```

It's silly in an example this short, since the entirety of the `modify` function is now in a typeclass. But with larger examples, you can see how we'd be able to specify that entire portions of our logic perform no arbitrary side-effects, while still using the `ReaderT` pattern to its fullest.

To put this another way: the function `foo :: Monad m => Int -> m Double` may appear to be impure, because it lives in a `Monad`. But this isn't true: by giving it a constraint of "any arbitrary instance of `Monad`", we're stating "this has no real side-effects". After all, the type above unifies with `Identity`, which of course is pure.

This example may seem a bit funny, but what about:

```
parseInt :: MonadThrow m => Text -> m Int
```

You may think "that's impure, it throws a runtime exception". But the type unifies with `parseInt :: Text -> Maybe Int`, which of course is pure. We've gained a lot of knowledge about our function and can feel safe calling it.

So the take-away here is: if you can generalize your functions to mtl-style `Monad` constraints, do it, you'll regain a lot of the benfits you'd have with purity.

14.7. Analysis

While the technique here is certainly a bit heavy-handed, for any large-scale application or library development that cost will be amortized. I've found the benefits of working in this style to far outweigh the costs in many real world projects.

There are other problems it causes, like more confusing error messages and more cognitive overhead for someone joining the project. But in my experience, once someone is onboarded to the approach, it works out well.

In addition to the concrete benefits I listed above, using this approach automatically navigates you around many common monad transformer stack pain points that you'll see people experiencing in the real world. I encourage others to share their real-world examples of them. I personally haven't hit those problems in a long while, since I've stuck to this approach.

14.8. Post-publish updates

June 15, 2017 The comment below from Ashley about ImplicitParams spawned a Reddit discussion about the problems with that extension. Please do read the discussion yourself, but the takeaway for me is that `MonadReader` is a better choice.

Part II.

Posts on testing

15. Practical testing in Haskell - Jasper van der Jeugt

William Yao: *A short post about writing property tests for an LRU cache. Main takeaway is what Jasper terms the "Action trick": generating complicated data more easily by instead generating a sequence of events that could happen to the data and constructing a value accordingly. For instance, you could generate a binary search tree by generating a sequence of insertions and deletions.*

Original article: [14]

15.1. Introduction

There has been a theme of "Practical Haskell" in the last few blogposts I published, and when I published the last one, on how to write an LRU Cache, someone asked me if I could elaborate on how I would test or benchmark such a module. For the sake of brevity, I will constrain myself to testing for now, although I think a lot of the ideas in the blogpost also apply to benchmarking.

This post is written in Literate Haskell. It depends on the LRU Cache we wrote last time, so you need both modules if you want to play around with the code. Both can be found in this repo.

Since I use a different format for blogpost filenames than GHC expects for module names, loading both modules is a bit tricky. The following works for me:

```
$ ghci posts/2015-02-24-lru-cache.lhs \
    posts/2015-03-13-practical-testing-in-haskell.lhs
*Data.SimpleLruCache> :m +Data.SimpleLruCache.Tests
*Data.SimpleLruCache Data.SimpleLruCache.Tests>
```

Alternatively, you can of course rename the files.

15.2. Test frameworks in Haskell

There are roughly two kinds of test frameworks which are commonly used in the Haskell world:

- Unit testing, for writing concrete test cases. We will be using HUnit.
- Property testing, which allows you to test *properties* rather than specific *cases*. We will be using QuickCheck. Property testing is something that might be unfamiliar to people just starting out in Haskell. However, because there already are great tutorials out there on there on QuickCheck, I will not explain it in detail. smallcheck also falls in this category.

Finally, it's nice to have something to tie it all together. We will be using Tasty, which lets us run HUnit and QuickCheck tests in the same test suite. It also gives us plenty of convenient options, e.g. running only a part of the test suite. We could also choose to use test-framework or Hspec instead of Tasty.

15.3. A module structure for tests

Many Haskell projects start out by just having a `tests.hs` file somewhere, but this obviously does not scale well to larger codebases.

The way I like to organize tests is based on how we organize code in general: through the module hierarchy. If I have the following modules in `src/`:

```
AcmeCompany.AwesomeProduct.Database
AcmeCompany.AwesomeProduct.Importer
AcmeCompany.AwesomeProduct.Importer.Csv
```

I aim to have the following modules in `tests/`:

```
AcmeCompany.AwesomeProduct.Database.Tests
AcmeCompany.AwesomeProduct.Importer.Tests
AcmeCompany.AwesomeProduct.Importer.Csv.Tests
```

If I want to add some higher-level tests which basically test the entire product, I can usually add these higher in the module tree. For example, if I wanted to test our entire awesome product, I would write the tests in `AcmeCompany.AwesomeProduct.Tests`.

Every `.Tests` module exports a `tests :: TestTree` value. A `TestTree` is a tasty concept – basically a structured group of tests. Let's go to our motivating example: testing the LRU Cache I wrote in the previous blogpost.

Since I named the module `Data.SimpleLruCache`, we use `Data.SimpleLruCache.Tests` here.

```
{-# OPTIONS_GHC -fno-warn-orphans #-}
{-# LANGUAGE BangPatterns                #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Data.SimpleLruCache.Tests
    ( tests
    ) where
import           Control.Applicative     ((<$>), (<*>))
import           Control.DeepSeq         (NFData)
import           Control.Monad           (foldM_)
import           Data.Hashable           (Hashable (..))
import qualified Data.HashPSQ            as HashPSQ
import           Data.IORef              (newIORef, readIORef, writeIORef)
import           Data.List               (foldl')
import qualified Data.Set                as S
import           Prelude                 hiding (lookup)
import           Data.SimpleLruCache
import qualified Test.QuickCheck         as QC
import qualified Test.QuickCheck.Monadic as QC
```

```
import          Test.Tasty                (TestTree, testGroup)
import          Test.Tasty.HUnit          (testCase)
import          Test.Tasty.QuickCheck     (testProperty)
import          Test.HUnit                (Assertion, (@?=))
```

15.4. What to test

One of the hardest questions is, of course, which functions and modules should I test? If unlimited time and resources are available, the obvious answer is "everything". Unfortunately, time and resources are often scarce.

My rule of thumb is based on my development style. I tend to use GHCi a lot during development, and play around with datastructures and functions until they seem to work. These "it seems to work" cases I execute in GHCi often make great candidates for simple HUnit tests, so I usually start with that.

Then I look at invariants of the code, and try to model these as QuickCheck properties. This sometimes requires writing tricky `Arbitrary` instances; I will give an example of this later in this blogpost.

I probably don't have to say that the more critical the code is, the more tests should be added.

After doing this, it is still likely that we will hit bugs if the code is non-trivial. These bugs form good candidates for testing as well:

1. First, add a test case to reproduce the bug. Sometimes a test case will be a better fit, sometimes we should go with a property – it depends on the bug.
2. Fix the bug so the test case passes.
3. Leave in the test case for regression testing.

Using this strategy, you should be able to convince yourself (and others) that the code works.

15.5. Simple HUnit tests

Testing simple cases using HUnit is trivial, so we won't spend that much time here. `@?=` asserts that two values must be equal, so let's use that to check that trimming the empty `Cache` doesn't do anything evil:

```
testCache01 :: Assertion
testCache01 =
    trim (empty 3 :: Cache String Int) @?= empty 3
```

If we need to some I/O for our test, we can do so without much trouble in HUnit. After all,

```
Test.HUnit> :i Assertion
type Assertion = IO ()  -- Defined in 'Test.HUnit.Lang'
```

so `Assertion` is just `IO`!

```
testCache02 :: Assertion
testCache02 = do
    h  <- newHandle 10 :: IO (Handle String Int)
    v1 <- cached h "foo" (return 123)
    v1 @?= 123
    v2 <- cached h "foo" (fail "should be cached")
    v2 @?= 123
```

That was fairly easy.

As you can see, I usually give simple test cases numeric names. Sometimes there is a meaningful name for a test (for example, if it is a regression test for a bug), but usually I don't mind using just numbers.

15.6. Simple QuickCheck tests

Let's do some property based testing. There are a few properties we can come up with.

Calling `HashPSQ.size` takes $O(n)$ time, which is why are keeping our own counter, `cSize`. We should check that it matches `HashPSQ.size`, though:

```
sizeMatches :: (Hashable k, Ord k) => Cache k v -> Bool
sizeMatches c =
    cSize c == HashPSQ.size (cQueue c)
```

The `cTick` field contains the priority of our next element that we will insert. The priorities currently in the queue should all be smaller than that.

```
prioritiesSmallerThanNext :: (Hashable k, Ord k) => Cache k v -> Bool
prioritiesSmallerThanNext c =
    all (< cTick c) priorities
  where
    priorities = [p | (_, p, _) <- HashPSQ.toList (cQueue c)]
```

Lastly, the size should always be smaller than or equal to the capacity:

```
sizeSmallerThanCapacity :: (Hashable k, Ord k) => Cache k v -> Bool
sizeSmallerThanCapacity c =
    cSize c <= cCapacity c
```

15.7. Tricks for writing Arbitrary instances

15.7.1. The Action trick

Of course, if you are somewhat familiar with QuickCheck, you will know that the previous properties require an `Arbitrary` instance for `Cache`.

One way to write such instances is what I'll call the "direct" method. For us this would mean generating a list of `[(key, priority, value)]` pairs and convert that to a `HashPSQ`. Then we could compute the size of that and initialize the remaining fields.

However, writing an `Arbitrary` instance this way can get hard if our datastructure becomes more complicated, especially if there are complicated invariants. Additionally, if we take any shortcuts in the implementation of `arbitrary`, we might not test the edge cases well!

Another way to write the `Arbitrary` instance is by modeling use of the API. In our case, there are only two things we can do with a pure `Cache`: insert and lookup.

```
data CacheAction k v
    = InsertAction k v
    | LookupAction k
    deriving (Show)
```

This has a trivial Arbitrary instance:

```
instance (QC.Arbitrary k, QC.Arbitrary v) =>
        QC.Arbitrary (CacheAction k v) where
    arbitrary = QC.oneof
        [ InsertAction <$> QC.arbitrary <*> QC.arbitrary
        , LookupAction <$> QC.arbitrary
        ]
```

And we can apply these actions to our pure `Cache` to get a new `Cache`:

```
applyCacheAction
    :: (Hashable k, Ord k)
    => CacheAction k v -> Cache k v -> Cache k v
applyCacheAction (InsertAction k v) c = insert k v c
applyCacheAction (LookupAction k)   c = case lookup k c of
    Nothing      -> c
    Just (_, c') -> c'
```

You probably guessed where this was going by now: we can generate an arbitrary `Cache` by generating a bunch of these actions and applying them one by one on top of the `empty` cache.

```
instance (QC.Arbitrary k, QC.Arbitrary v, Hashable k, NFData v, Ord k) =>
        QC.Arbitrary (Cache k v) where
    arbitrary = do
        capacity <- QC.choose (1, 50)
        actions  <- QC.arbitrary
        let !cache = empty capacity
        return $! foldl' (\c a -> applyCacheAction a c) cache actions
```

Provided that we can model the complete user facing API using such an "action" datatype, I think this is a great way to write `Arbitrary` instances. After all, our `Arbitrary` instance should then be able to reach the same states as a user of our code.

An extension of this trick is using a separate datatype which holds the list of actions we used to generate the `Cache` as well as the `Cache`.

```
data ArbitraryCache k v = ArbitraryCache [CacheAction k v] (Cache k v)
    deriving (Show)
```

When a test fails, we can then log the list of actions which got us into the invalid state – very useful for debugging. Furthermore, we can implement the `shrink` method in order to try to reach a similar invalid state using less actions.

15.7.2. The SmallInt trick

Now, note that our `Arbitrary` instance is for `Cache k v`, i.e., we haven't chosen yet what we want to have as `k` and `v` for our tests. In this case `v` is not so important, but the choice of `k` is important.

We want to cover all corner cases, and this includes ensuring that we cover collisions. If we use `String` or `Int` as key type k, collisions are very unlikely due to the high cardinality of both types. Since we are using a hash-based container underneath, hash collisions must also be covered.

We can solve both problems by introducing a `newtype` which restricts the cardinality of `Int`, and uses a "worse" (in the traditional sense) hashing method.

```
newtype SmallInt = SmallInt Int
    deriving (Eq, Ord, Show)
instance QC.Arbitrary SmallInt where
    arbitrary = SmallInt <$> QC.choose (1, 100)
instance Hashable SmallInt where
    hashWithSalt salt (SmallInt x) = (salt + x) `mod` 10
```

15.8. Monadic QuickCheck

Now let's mix QuickCheck with monadic code. We will be testing the `Handle` interface to our cache. This interface consists of a single method:

```
cached
    :: (Hashable k, Ord k)
    => Handle k v -> k -> IO v -> IO v
```

We will write a property to ensure our cache retains and evicts the right key-value pairs. It takes two arguments: the capacity of the LRU Cache (we use a `SmallInt` in order to get more evictions), and a list of key-value pairs we will insert using `cached` (we use `SmallInt` so we will cover collisions).

```
historic
    :: SmallInt               -- ^ Capacity
    -> [(SmallInt, String)]   -- ^ Key-value pairs
    -> QC.Property            -- ^ Property
historic (SmallInt capacity) pairs = QC.monadicIO $ do
```

`QC.run` is used to lift `IO` code into the QuickCheck property monad `PropertyM` – so it is a bit like a more concrete version of `liftIO`. I prefer it here over `liftIO` because it makes it a bit more clear what is going on.

```
    h <- QC.run $ newHandle capacity
```

We will fold (`foldM_`) over the pairs we need to insert. The state we pass in this `foldM_` is the history of pairs we previously inserted. By building this up again using `(:)`, we ensure `history` contains a recent-first list, which is very convenient.

Inside every step, we call `cached`. By using an `IORef` in the code where we would usually actually "load" the value `v`, we can communicate whether or not the value

was already in the cache. If it was already in the cache, the write will not be executed, so the `IORef` will still be set to `False`. We store that result in `wasInCache`.

In order to verify this result, we reconstruct a set of the N most recent keys. We can easily do this using the list of recent-first key-value pairs we have in `history`.

```
    foldM_ (step h) [] pairs
  where
    step h history (k, v) = do
        wasInCacheRef <- QC.run $ newIORef True
        _             <- QC.run $ cached h k $ do
            writeIORef wasInCacheRef False
            return v
        wasInCache    <- QC.run $ readIORef wasInCacheRef
        let recentKeys = nMostRecentKeys capacity S.empty history
        QC.assert (S.member k recentKeys == wasInCache)
        return ((k, v) : history)
```

This is our auxiliary function to calculate the N most recent keys, given a recent-first key-value pair list.

```
nMostRecentKeys :: Ord k => Int -> S.Set k -> [(k, v)] -> S.Set k
nMostRecentKeys _ keys [] = keys
nMostRecentKeys n keys ((k, _) : history)
    | S.size keys >= n    = keys
    | otherwise           =
        nMostRecentKeys n (S.insert k keys) history
```

This test did not cover checking that the *values* in the cache are correct, but only ensures it retains the correct key-value pairs. This is a conscious decision: I think the retaining/evicting part of the LRU Cache code was the most tricky, so we should prioritize testing that.

15.9. Tying everything up

Lastly, we have our `tests :: TestTree`. It is not much more than an index of tests in the module. We use `testCase` to pass HUnit tests to the framework, and `testProperty` for QuickCheck properties.

Note that I usually tend to put these at the top of the module, but here I put it at the bottom of the blogpost for easier reading.

```
tests :: TestTree
tests = testGroup "Data.SimpleLruCache"
    [ testCase "testCache01" testCache01
    , testCase "testCache02" testCache02
    , testProperty "size == HashPSQ.size"
        (sizeMatches :: Cache SmallInt String -> Bool)
    , testProperty "priorities < next priority"
        (prioritiesSmallerThanNext :: Cache SmallInt String -> Bool)
    , testProperty "size < capacity"
```

```
        (sizeSmallerThanCapacity :: Cache SmallInt String -> Bool)
    , testProperty "historic" historic
    ]
```

The last thing we need is a `main` function for `cabal test` to invoke. I usually put this in something like `tests/Main.hs`. If you use the scheme which I described above, this file should look very neat:

```
module Main where

import           Test.Tasty (defaultMain, testGroup)

import qualified AcmeCompany.AwesomeProduct.Database.Tests
import qualified AcmeCompany.AwesomeProduct.Importer.Csv.Tests
import qualified AcmeCompany.AwesomeProduct.Importer.Tests
import qualified Data.SimpleLruCache.Tests

main :: IO ()
main = defaultMain $ testGroup "Tests"
    [ AcmeCompany.AwesomeProduct.Database.Tests.tests
    , AcmeCompany.AwesomeProduct.Importer.Csv.Tests.tests
    , AcmeCompany.AwesomeProduct.Importer.Tests.tests
    , Data.SimpleLruCache.Tests.tests
    ]
```

If you are still hungry for more Haskell testing, I would recommend looking into Haskell program coverage for mission-critical modules.

Special thanks to Alex Sayers, who beat everyone's expectations when he managed to stay sober for just long enough to proofread this blogpost.

16. Property-Based Testing in a Screencast Editor: Introduction - Oskar Wickström

> ***William Yao:*** *Probably the best series of posts I've read about using property-based testing in practice. Lots of interesting tricks and techniques explored. Highly recommended.*
>
> *Original article: [15]*

This is the first in a series of posts about using property-based testing (PBT) within *Komposition*, a screencast editor that I've been working on during the last year. It introduces PBT and highlights some challenges in testing properties of an application like Komposition.

Future posts will focus on individual case studies, covering increasingly complex components and how they are tested. I'll reflect on what I've learned in each case, what bugs the tests have found, and what still remains to be improved.

For example, I'll explain how using PBT helped me find and fix bugs in the specification and implementation of Komposition's video classifier. Those were bugs that would be very hard to find using example-based tests or using a static type system!

This series is not a tutorial on PBT, but rather a collection of motivating examples. That said, you should be able to follow along without prior knowledge of PBT.

16.1. Komposition

In early 2018 I started producing Haskell screencasts. A majority of the work involved cutting and splicing video by hand in a non-linear editing system (NLE) like Premiere Pro or Kdenlive. I decided to write a screencast editor specialized for my needs, reducing the amount of manual labor needed to edit the recorded material. Komposition was born.

Komposition is a modal GUI application built for editing screencasts. Unlike most NLE systems, it features a hierarchical timeline, built out of *sequences*, *parallels*, *tracks*, *clips* and *gaps*. To make the editing experience more efficient, it automatically classifies scenes in screen capture video, and sentences in recorded voice-over audio.

If you are curious about Komposition and want to learn more right away, check out its documentation. Some of the most complex parts of Komposition include focus and timeline transformations, video classification, video rendering, and the main application logic. Those are the areas in which I've spent most effort writing tests, using a combination of example-based and property-based testing.

I've selected the four most interesting areas where I've applied PBT in Komposition, and I'll cover one in each coming blog post:

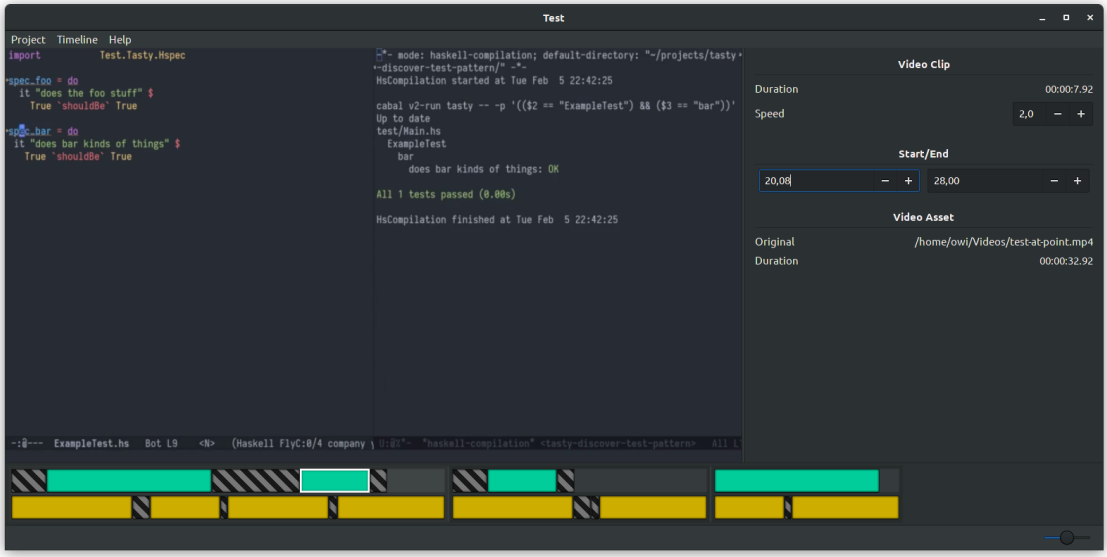

Figure 16.1.: Komposition's timeline mode

1. Timeline flattening
2. Video scene classification
3. Focus and timeline consistency
4. Symmetry of undo/redo

I hope these case studies will be motivating, and that they will show the value of properties all the way from unit testing to integration testing.

16.2. Property-Based Testing

To get the most out of this series, you need a basic understanding of what PBT is, so let's start there. For my take on a minimal definition, PBT is about:

1. Specifying your system under test in terms of properties, where properties describe invariants of the system based on its input and output.
2. Testing that those properties hold against a large variety of inputs.

It's worth noting that PBT is not equal to QuickCheck, or any other specific tool, for that matter. The set of inputs doesn't have to be randomly generated. You don't have to use "shrinking". You don't have to use a static type system or a functional programming language. PBT is a general idea that can be applied in many ways.

The following resources are useful if you want to learn more about PBT:

- The introductory articles on Hypothesis, although specific to Python.
- "What is Property Based Testing?" by David R. MacIver is a definition of what PBT is, and particularly what it isn't.

The code examples will be written in Haskell and using the Hedgehog testing system. You don't have to know Haskell to follow this series, as I'll explain the techniques primarily without code. But if you are interested in the Haskell specifics and in Hedgehog, check out "Property testing with Hedgehog" by Tim Humphries.

16.3. Properties of the Ugly Parts

When I started with PBT, I struggled with applying it to anything beyond simple functions. Examples online are often focused on the fundamentals. They cover concepts like reversing lists, algebraic laws, and symmetric encoders and decoders. Those are important properties to test, and they are good examples for teaching the foundations of PBT.

I wanted to take PBT beyond pure and simple functions, and leverage it on larger parts of my system. The "ugly" parts, if you will. In my experience, the complexity of a system often becomes much higher than the sum of its parts. The way subsystems are connected and form a larger graph of dependencies drives the need for integration testing at an application level.

Finding resources on integration testing using PBT is hard, and it might drive you to think that PBT is not suited for anything beyond the introductory examples. With the case studies in this blog series I hope to contribute to debunking such misconceptions.

16.4. Designing for Testability

In my case, it's a desktop multimedia application. What if we're working on a backend that connects to external systems and databases? Or if we're writing a frontend application with a GUI driven by user input? In addition to these kinds of systems being hard to test at a high level due to their many connected subsystems, they usually have stateful components, side effects, and non-determinism. How do we make such systems testable with properties?

Well, the same way we would design our systems to be testable with examples. Going back to "Writing Testable Code" by Miško Hevery from 2008, and Kent Beck's "Test-Driven Development by Example" from 2003, setting aside the OOP specifics, many of their guidelines apply equally well to code tested with properties:

Determinism: Make it possible to run the "system under test" deterministically, such that your tests can be reliable. This does not mean the code has to be pure, but you might need to stub or otherwise control side effects during your tests.

No global state: In order for tests to be repeatable and independent of execution order, you might have to rollback database transactions, use temporary directories for generated files, stub out effects, etc.

High cohesion: Strive for modules of high cohesion, with smaller units each having a single responsibility. Spreading closely related responsibilities thin across multiple modules makes the implementation harder to maintain and test.

Low coupling: Decrease coupling between interface and implementation. This makes it easier to write tests that don't depend on implementation details. You may then modify the implementation without modifying the corresponding tests.

I find these guidelines universal for writing testable code in any programming language I've used professionally, regardless of paradigm or type system. They apply to both example-based and property-based testing.

16.5. Patterns for Properties

Great, so we know how to write testable code. But how do we write properties for more complex units, and even for integration testing? There's not a lot of educational resources on this subject that I know of, but I can recommend the following starting points:

- "Choosing properties for property-based testing" by Scott Wlaschin, giving examples of properties within a set of common categories.
- The talk "Property-Based Testing for Better Code" by Jessica Kerr, with examples of generating valid inputs and dealing with timeouts.

Taking a step back, we might ask "Why it's so hard to come up with these properties?" I'd argue that it's because doing so forces us to understand our system in a way we're not used to. It's challenging understanding and expressing the general behavior of a system, rather than particular *anecdotes* that we've observed or come up with.

If you want to get better at writing properties, the only advice I can give you (in addition to studying whatever you can find in other projects) is to *practice*. Try it out on whatever you're working on. Talk to your colleagues about using properties in addition to example-based tests at work. Begin at a smaller scale, testing simple functions, and progress towards testing larger parts of your system once you're comfortable. It's a long journey, filled with reward, surprise, and joy!

16.6. Testing Case Studies

With a basic understanding of PBT, how we can write testable code, and how to write properties for our system under test, we're getting ready to dive into the case studies:

1. Introduction (this chapter)
2. Timeline Flattening (see chapter 17)
3. Video Scene Classification (see chapter 18)
4. Integration Testing (see chapter 19)

16.7. Credits

Thank you Chris Ford, Alejandro Serrano Mena, Tobias Pflug, Hillel Wayne, and Ulrik Sandberg for kindly providing your feedback on my drafts!

17. Case Study 1: Timeline Flattening - Oskar Wickström

Original article: [16]

In the first post of this series I introduced the Komposition screencast editor, and briefly explained the fundamentals of property-based testing (PBT). Furthermore, I covered how to write testable code, regardless of how you check your code with automated tests. Lastly, I highlighted some difficulties in using properties to perform component and integration testing.

If you haven't read the introductory post, I suggest doing so before continuing with this one. You'll need an understanding of what PBT is for this case study to make sense.

This post is the first case study in the series, covering the timeline flattening process in Komposition and how it's tested using PBT. The property tests aren't integration-level tests, but rather unit tests. This case study serves as a warm-up to the coming, more advanced, ones.

Before we look at the tests, we need to learn more about Komposition's hierarchical timeline and how the flattening process works.

17.1. The Hierarchical Timeline

Komposition's timeline is hierarchical. While many non-linear editing systems have support for some form of nesting[1]. they are primarily focused on flat timeline workflows. The timeline structure and the keyboard-driven editing in Komposition is optimized for the screencast editing workflow I use.

It's worth emphasizing that Komposition is not a general video editor. In addition to its specific editing workflow, you may need to adjust your recording workflow to use it effectively[2].

17.1.1. Video and Audio in Parallels

At the lowest level of the timeline are *clips* and *gaps*. Those are put within the video and audio *tracks of parallels*. The following diagram (figuare 17.1) shows a parallel consisting of two video clips and one audio clip. The tracks of a parallel are played

1 Final Cut Pro has compound clips, and Adobe Premiere Pro has nested sequences

2 The section on workflow in Komposition's documentation describes how to plan, record, and edit your screencast in way compatible with Komposition.

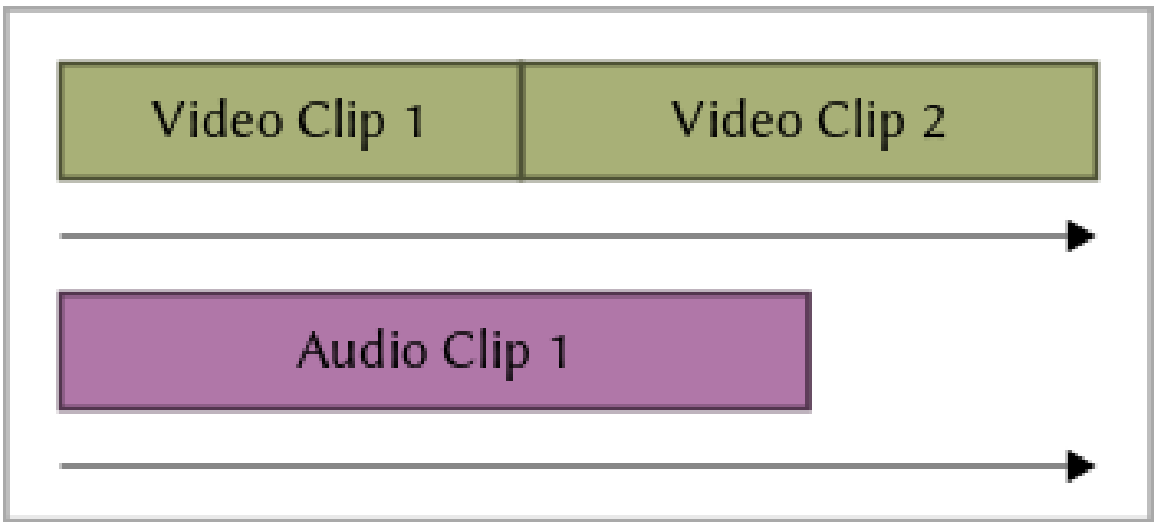

Figure 17.1.: Clips and gaps are placed in video and audio tracks

simultaneously (in parallel), as indicated by the arrows in the above diagram. The tracks start playing at the same time. This makes parallels useful to synchronize the playback of specific parts of a screencast, and to group closely related clips.

17.1.2. Gaps

When editing screencasts made up of separate video and audio recordings you often end up with differing clip duration. The voice-over audio clip might be longer than the corresponding video clip, or vice versa. A useful default behaviour is to extend the short clips. For audio, this is easy. Just pad with silence. For video, it's not so clear what to do. In Komposition, shorter video tracks are padded with repeated still frame sections called *gaps*.

The following diagram (figure 17.2) shows a parallel with a short video clip and a longer audio clip. The dashed area represents the implicit gap. You can also add gaps

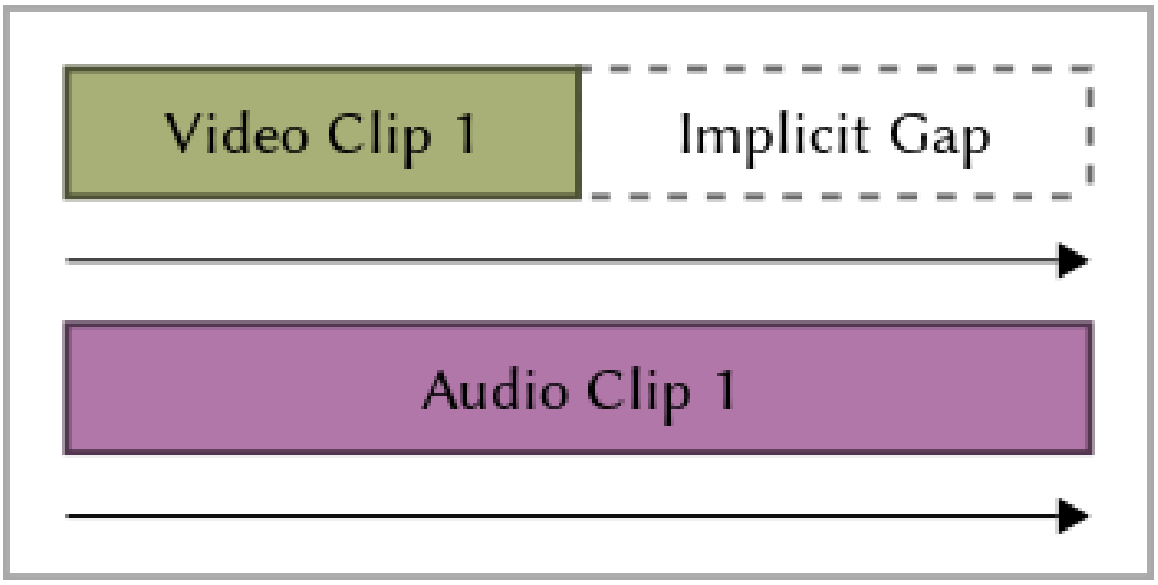

Figure 17.2.: Still frames are automatically inserted at implicit gaps to match track duration

manually, specifying a duration of the gap and inserting it into a video or audio track. The following diagram (figure 17.3) shows a parallel with manually added gaps in both video and audio tracks. Manually added gaps (called *explicit* gaps) are padded

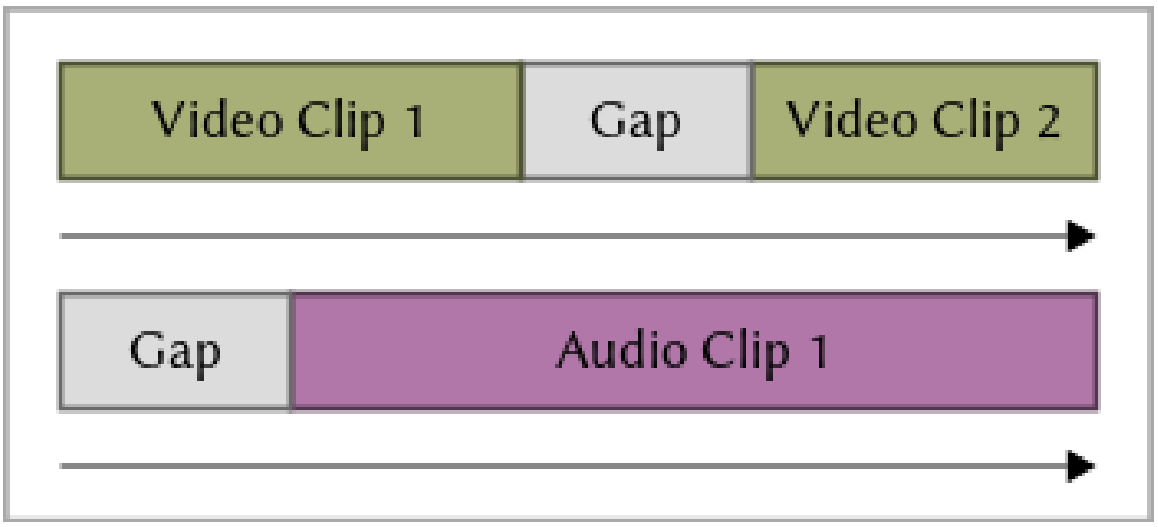

Figure 17.3.: Adding explicit gaps manually

with still frames or silence, just as implicit gaps that are added automatically to match track duration.

17.1.3. Sequences

Parallels are put in *sequences*. The parallels within a sequence are played sequentially; the first one is played in its entirety, then the next one, and so on. This behaviour is different from how parallels play their tracks. Parallels and sequences, with their different playback behaviors, make up the fundamental building blocks of the compositional editing in Komposition.

The following diagram (figure 17.4) shows a sequence of two parallels, playing sequentially:

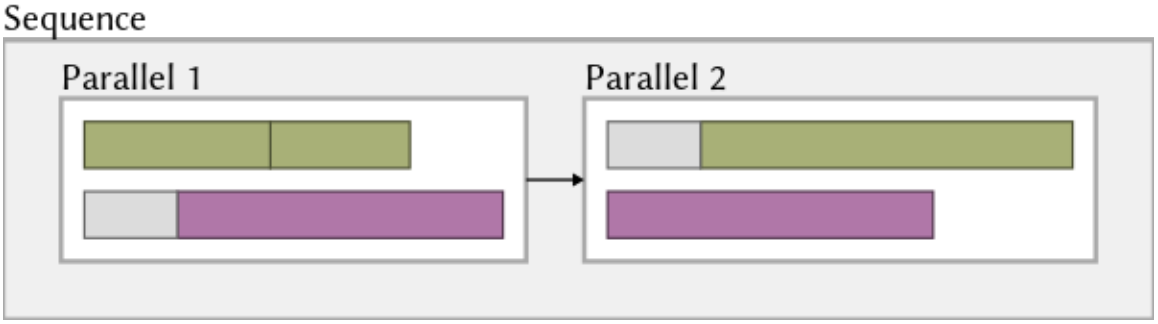

Figure 17.4.: A sequence containing two parallels

17.1.4. The Timeline

Finally, at the top level, we have the *timeline*. Effectively, the timeline is a sequence of sequences; it plays every child sequence in sequence. The reason for this level to exist

is for the ability to group larger chunks of a screencast within separate sequences. I

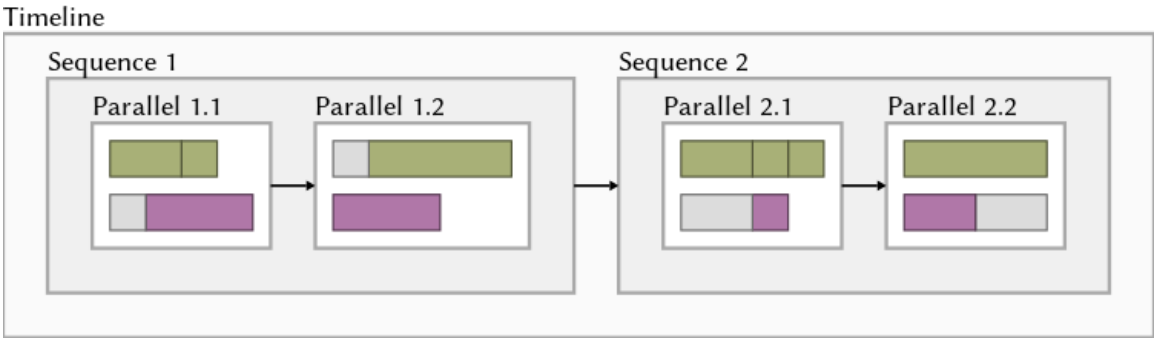

Figure 17.5.: A timeline containing two sequences, with two parallels each

use separate sequences within the timeline to delimit distinct parts of a screencast, such as the introduction, the different chapters, and the summary.

17.2. Timeline Flattening

Komposition currently uses FFmpeg to render the final media. This is done by constructing an `ffmpeg` command invocation with a filter graph describing how to fit together all clips, still frames, and silent audio parts.

FFmpeg doesn't know about hierarchical timelines; it only cares about video and audio streams. To convert the hierarchical timeline into a suitable representation to build the FFmpeg filter graph from, Komposition performs *timeline flattening*.

The flat representation of a timeline contains only two tracks; audio and video. All gaps are *explicitly* represented in those tracks. The following graph shows how a hierarchical timeline is flattened into two tracks. Notice in the graphic above how the

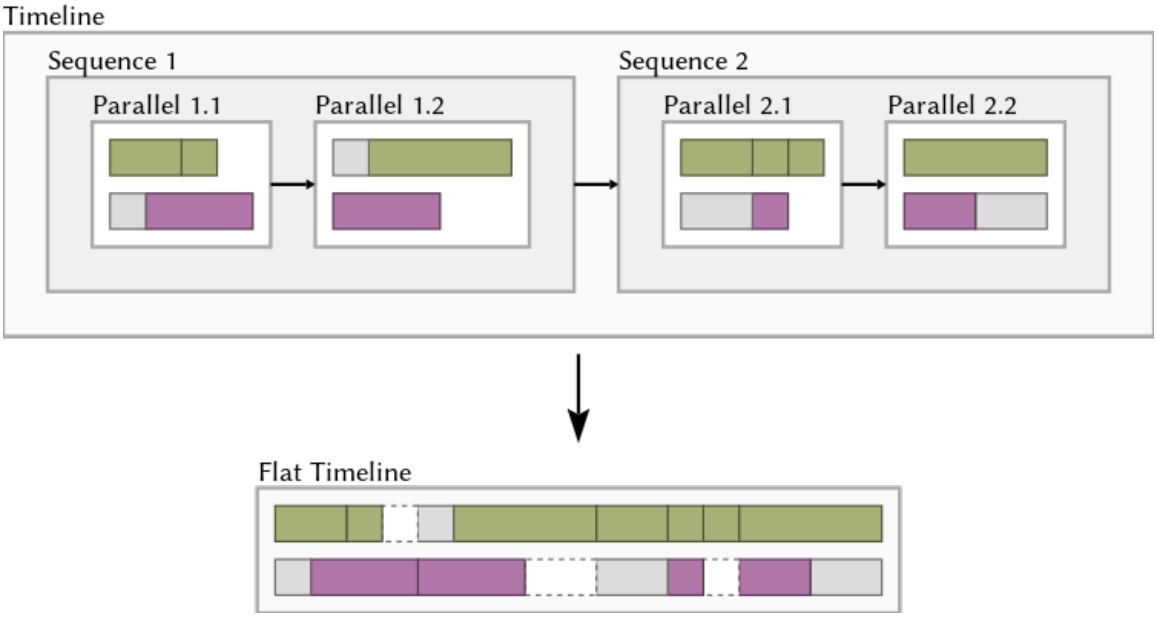

Figure 17.6.: Timeline flattening transforming a hierarchical timeline

implicit gaps at the ends of video and audio tracks get represented with explicit gaps in the flat timeline. This is because FFmpeg does not know how to render implicit gaps. All gaps are represented explicitly, and are converted to clips of still frames or silent audio when rendered with FFmpeg.

17.3. Property Tests

To test the timeline flattening, there's a number of properties that are checked. I'll go through each one and their property test code.

These properties were primarily written after I already had an implementation. They capture some general properties of flattening that I've come up with. In other cases, I've written properties before beginning on an implementation, or to uncover an existing bug that I've observed.

Thinking about your system's general behaviour and expressing that as executable property tests is hard. I believe, like with any other skill, that it requires a lot of practice. Finding general patterns for properties, like the ones Scott Wlaschin describe in Choosing properties for property-based testing (see chapter 20), is a great place to start. When you struggle with finding properties of your system under test, try applying these patterns and see which work for you.

17.3.1. Property: Duration Equality

Given a timeline t, where all parallels have at least one video clip, the total duration of the flattened t must be equal to the total duration of t. Or, in a more dense notation,

$$\forall t \in T \rightarrow duration(flatten(t)) = duration(t)$$

where T is the set of timelines with at least one video clip in each parallel.

The reason that all parallels must have at least one video clip is because currently the flattening algorithm can only locate still frames for video gaps from within the same parallel. If it encounters a parallel with no video clips, the timeline flattening fails. This limitation is discussed in greater detail at the end of this article.

The test for the duration equality property is written using Hedgehog, and looks like this:

```
hprop_flat_timeline_has_same_duration_as_hierarchical =
  property $ do
    -- 1. Generate a timeline with video clips in each parallel
    timeline' <- forAll $
        Gen.timeline (Range.exponential 0 5) Gen.parallelWithClips

    -- 2. Flatten the timeline and extract the result
    let Just flat = Render.flattenTimeline timeline'

    -- 3. Check that hierarchical and flat timeline duration are equal
    durationOf AdjustedDuration timeline'
      === durationOf AdjustedDuration flat
```

It generates a timeline using `forAll` and custom generators (1). Instead of generating timelines of *any* shape and filtering out only the ones with video clips in each parallel, which would be very inefficient, this test uses a custom generator to only obtain inputs that satisfy the invariants of the system under test.

The range passed as the first argument to `Gen.timeline` is used as the bounds of the generator, such that each level in the generated hierarchical timeline will have at most 5 children.

`Gen.timeline` takes as its second argument *another generator*, the one used to generate parallels, which in this case is `Gen.parallelWithClips`. With Hedgehog generators being regular values, it's practical to compose them like this. A "higher-order generator" can be a regular function taking other generators as arguments.

As you might have noticed in the assertion (3), `durationOf` takes as its first argument a value `AdjustedDuration`. What's that about? Komposition supports adjusting the playback speed of video media for individual clips. To calculate the final duration of a clip, the playback speed needs to taken into account. By passing `AdjustedDuration` we take playback speed into account for all video clips.

SIDETRACK: FINDING A BUG

Let's say I had introduced a bug in timeline flattening, in which all video gaps weren't added correctly to the flat video tracks. The flattening is implemented as a fold, and it would not be unthinkable that the accumulator was incorrectly constructed in a case. The test would catch this quickly and present us with a minimal counter-example (see figure 17.7).

Hedgehog prints the source code for the failing property. Below the `forAll` line the generated value is printed. The difference between the expected and actual value is printed below the failing assertion. In this case it's a simple expression of type `Duration`. In case you're comparing large tree-like structures, this diff will highlight only the differing expressions. Finally, it prints the following:

This failure can be reproduced by running:

```
> recheck (Size 23) (Seed 16495576598183007788 5619008431246301857) <property>
```

When working on finding and fixing the fold bug, we can use the printed size and seed values to deterministically rerun the test with the exact same inputs.

17.3.2. Property: Clip Occurence

Slightly more complicated than the duration equality property, the clip occurrence property checks that all clips from the hierarchical timeline, and no other clips, occur within the flat timeline. As discussed in the introduction on timeline flattening, implicit gaps get converted to explicit gaps and thereby add more gaps, but no video or audio clips should be added or removed.

```
hprop_flat_timeline_has_same_clips_as_hierarchical =
  property $ do
    -- 1. Generate a timeline with video clips in each parallel
    timeline' <- forAll $
        Gen.timeline (Range.exponential 0 5) Gen.parallelWithClips

    -- 2. Flatten the timeline
```

```
x <interactive> failed after 24 tests and 22 shrinks.

    ┏━━ test/Komposition/Render/CompositionTest.hs ━━
 24 ┃ hprop_flat_timeline_has_same_duration_as_hierarchical = property $ do
 25 ┃   timeline' <- forAll $
 26 ┃     Gen.timeline (Range.exponential 0 5) Gen.parallelWithClips
    ┃       Timeline
    ┃         { _sequences =
    ┃             Sequence
    ┃               { _sequenceAnnotation = ()
    ┃               , _parallels =
    ┃                   Parallel
    ┃                     { _parallelAnnotation = ()
    ┃                     , _videoTrack =
    ┃                         VideoTrack
    ┃                           { _videoTrackAnnotation = ()
    ┃                           , _videoParts =
    ┃                               [ VideoGap () (Duration 1 s)
    ┃                               , VideoGap () (Duration 1 s)
    ┃                               , VideoClip
    ┃                                   ()
    ┃                                   VideoAsset
    ┃                                     { _videoAssetMetadata =
    ┃                                         AssetMetadata
    ┃                                           { _path = OriginalPath { _unOriginalPath = "\NUL" }
    ┃                                           , _duration = Duration 1 s
    ┃                                           }
    ┃                                     , _videoAssetTranscoded = TranscodedPath { _unProxyPath = "\NUL" }
    ┃                                     , _videoAssetProxy = TranscodedPath { _unProxyPath = "\NUL" }
    ┃                                     , _videoSpeed = VideoSpeed { _unVideoSpeed = 0.1 }
    ┃                                     , _videoClassifiedScene = Nothing
    ┃                                     }
    ┃                                   TimeSpan { spanStart = Duration 0 s , spanEnd = Duration 1 s }
    ┃                                   VideoSpeed { _unVideoSpeed = 0.1 }
    ┃                               ]
    ┃                           }
    ┃                     , _audioTrack =
    ┃                         AudioTrack { _audioTrackAnnotation = () , _audioParts = [] }
    ┃                     } :|
    ┃                     []
    ┃               } :|
    ┃               []
    ┃         }
 27 ┃   let Just flat = Render.flattenTimeline timeline'
 28 ┃   durationOf AdjustedDuration timeline' === durationOf AdjustedDuration flat
    ┃   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    ┃   Failed (- lhs =/= + rhs)
    ┃   - Duration 12 s
    ┃   + Duration 11 s

This failure can be reproduced by running:
> recheck (Size 23) (Seed 16495576598183007788 5619008431246301857) <property>
```

Figure 17.7.: Hedgehog presenting a minimal counter-example

```
    let flat = Render.flattenTimeline timeline'

    -- 3. Check that all video clips occur in the flat timeline
    flat ^.. _Just . Render.videoParts . each . Render._VideoClipPart
        === timelineVideoClips timeline'

    -- 4. Check that all audio clips occur in the flat timeline
    flat ^.. _Just . Render.audioParts . each . Render._AudioClipPart
        === timelineAudioClips timeline'
```

The hierarchical timeline is generated and flattened like before (1, 2). The two assertions check that the respective video clips (3) and audio clips (4) are equal. It's using lenses to extract clips from the flat timeline, and the helper functions `timelineVideoClips` and `timelineAudioClips` to extract clips from the original hierarchical timeline.

17.4. Still Frames Used

In the process of flattening, the still frame source for each gap is selected. It doesn't assign the actual pixel data to the gap, but a value describing which asset the still frame should be extracted from, and whether to pick the first or the last frame (known as *still frame mode*). This representation lets the flattening algorithm remain a pure function, and thus easier to test. Another processing step runs the effectful action that extracts still frames from video files on disk.

The decision of still frame mode and source is made by the flattening algorithm based on the parallel in which each gap occur, and what video clips are present before or after. It favors using clips occurring after the gap. It only uses frames from clips before the gap in case there are no clips following it. To test this behaviour, I've defined three properties.

17.4.1. Property: Single Initial Video Clip

The following property checks that an initial single video clip, followed by one or more gaps, is used as the still frame source for those gaps.

```
hprop_flat_timeline_uses_still_frame_from_single_clip =
  property $ do
    -- 1. Generate a video track generator where the first video part
    --    is always a clip
    let genVideoTrack = do
          v1 <- Gen.videoClip
          vs <- Gen.list (Range.linear 1 5) Gen.videoGap
          pure (VideoTrack () (v1 : vs))

    -- 2. Generate a timeline with the custom video track generator
    timeline' <- forAll $ Gen.timeline
      (Range.exponential 0 5)
      (Parallel () <$> genVideoTrack <*> Gen.audioTrack)
```

```
    -- 3. Flatten the timeline
    let flat = Render.flattenTimeline timeline'

    -- 4. Check that any video gaps will use the last frame of a
    --    preceding video clip
    flat
      ^.. ( _Just
          . Render.videoParts
          . each
          . Render._StillFramePart
          . Render.stillFrameMode
          )
      &   traverse_ (Render.LastFrame ===)
```

The custom video track generator (1) always produces tracks with an initial video clip followed by one or more video gaps. The generated timeline (2) can contain parallels with any audio track shape, which may result in a *longer* audio track and thus an implicit gap at the end of the video track. In either case, all video gaps should padded with the last frame of the initial video clip, which is checked in the assertion (4).

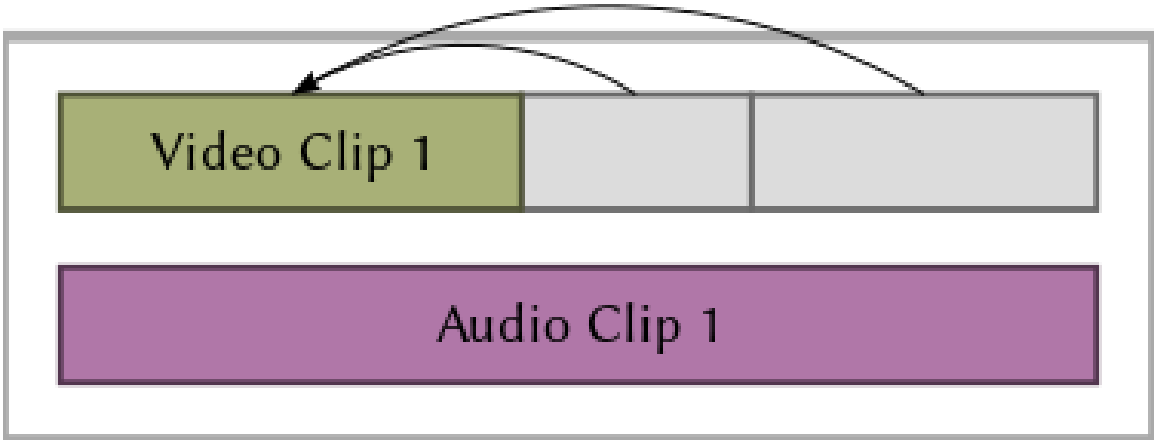

Figure 17.8.: Still frames being sourced from the single initial video clip

17.4.2. Property: Ending with a Video Clip

In case the video track ends with a video clip, and is longer than the audio track, all video gaps within the track should use the first frame of a following clip.

```
hprop_flat_timeline_uses_still_frames_from_subsequent_clips =
  property $ do
    -- 1. Generate a parallel where the video track ends with a video clip,
    --    and where the audio track is shorter
    let
      genParallel = do
        vt <-
```

```
        VideoTrack ()
          <$> (   snoc
              <$> Gen.list (Range.linear 1 10) Gen.videoPart
              <*> Gen.videoClip
              )
      at <- AudioTrack () . pure . AudioGap () <$> Gen.duration'
        (Range.linearFrac
          0
          (durationToSeconds (durationOf AdjustedDuration vt) - 0.1)
        )
      pure (Parallel () vt at)

  -- 2. Generate a timeline with the custom parallel generator
  timeline' <- forAll $ Gen.timeline (Range.exponential 0 5) genParallel

  -- 3. Flatten the timeline
  let flat = Render.flattenTimeline timeline'

  -- 4. Check that all gaps use the first frame of subsequent clips
  flat
    ^.. ( _Just
        . Render.videoParts
        . each
        . Render._StillFramePart
        . Render.stillFrameMode
        )
    &   traverse_ (Render.FirstFrame ===)
```

The custom generator (1) produces parallels where the video track is guaranteed to end with a clip, and where the audio track is 100 ms shorter than the video track. This ensures that there's no implicit video gap at the end of the video track. Generating (2) and flattening (3) is otherwise the same as before. The assertion (4) checks that all video gaps uses the first frame of a following clip.

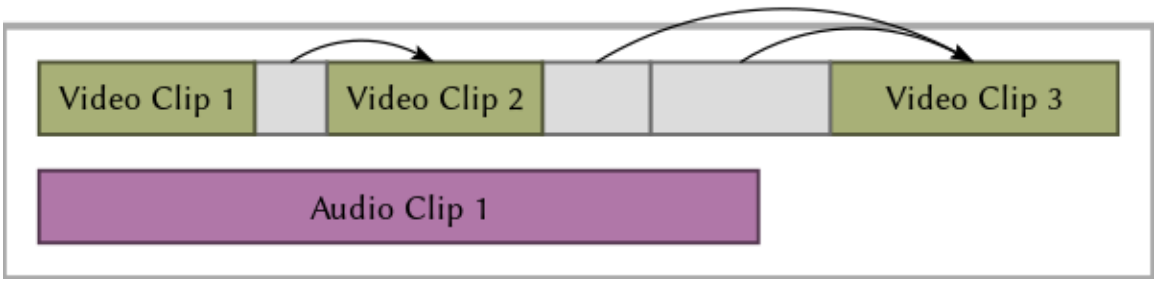

Figure 17.9.: Still frames being sourced from following video clips when possible

17.4.3. Property: Ending with an Implicit Video Gap

The last property on still frame usage covers the case where the video track is shorter than the audio track. This leaves an implicit gap which, just like explicit gaps inserted

by the user, are padded with still frames.

```
hprop_flat_timeline_uses_last_frame_for_automatic_video_padding =
  property $ do
    -- 1. Generate a parallel where the video track only contains a video
    --    clip, and where the audio track is longer
    let
      genParallel = do
        vt <- VideoTrack () . pure <$> Gen.videoClip
        at <- AudioTrack () . pure . AudioGap () <$> Gen.duration'
          (Range.linearFrac
            (durationToSeconds (durationOf AdjustedDuration vt) + 0.1)
            10
          )
        pure (Parallel () vt at)

    -- 2. Generate a timeline with the custom parallel generator
    timeline' <- forAll $ Gen.timeline (Range.exponential 0 5) genParallel

    -- 3. Flatten the timeline
    let flat = Render.flattenTimeline timeline'

    -- 4. Check that video gaps (which should be a single gap at the
    --    end of the video track) use the last frame of preceding clips
    flat
      ^.. ( _Just
          . Render.videoParts
          . each
          . Render._StillFramePart
          . Render.stillFrameMode
          )
      &   traverse_ (Render.LastFrame ===)
```

The custom generator (1) generates a video track consisting of video clips only, and an audio track that is 100ms longer. Generating the timeline (2) and flattening (3) are again similar to the previous property tests. The assertion (4) checks that all video gaps use the last frame of preceding clips, even if we know that there should only be one at the end.

17.5. Properties: Flattening Equivalences

The last property I want to show in this case study checks flattening at the sequence and parallel levels. While rendering a full project always flattens at the timeline, the *preview* feature in Komposition can be used to render and preview a single sequence or parallel.

There should be no difference between flattening an entire timeline and flattening all of its sequences or parallels and folding those results into a single flat timeline. This is what the *flattening equivalences* properties are about.

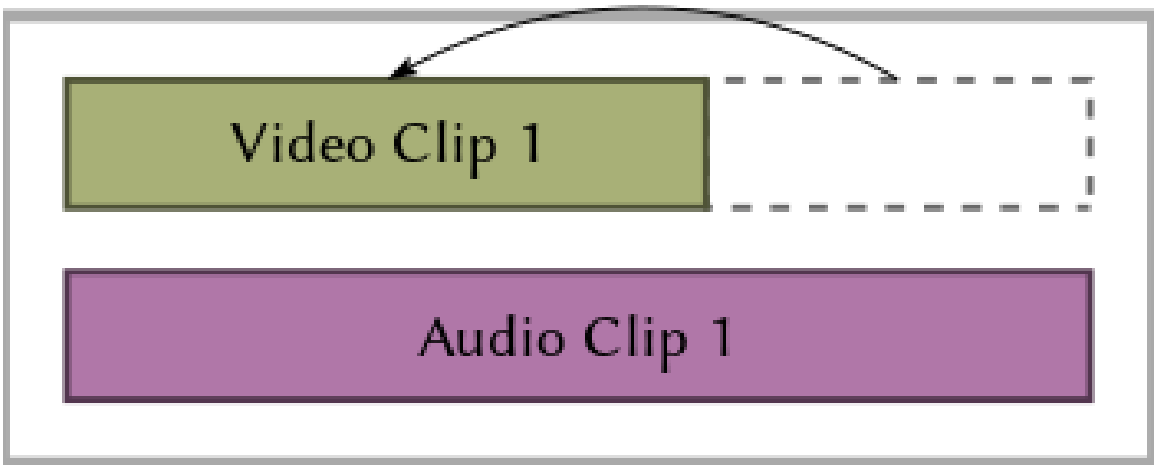

Figure 17.10.: Still frames being sourced from preceding video clip for last implicit gap

```
hprop_flat_timeline_is_same_as_all_its_flat_sequences =
  property $ do
    -- 1. Generate a timeline
    timeline' <- forAll $
      Gen.timeline (Range.exponential 0 5) Gen.parallelWithClips

    -- 2. Flatten all sequences and fold the resulting flat
    --    timelines together
    let flat = timeline' ^.. sequences . each
               & foldMap Render.flattenSequence

    -- 3. Make sure we successfully flattened the timeline
    flat /== Nothing

    -- 4. Flatten the entire timeline and compare to the flattened
    --    sequences
    Render.flattenTimeline timeline' === flat
```

The first property generates a timeline (1) where all parallels have at least one video clip. It flattens all sequences within the timeline and folds the results together (2). Folding flat timelines together means concatenating their video and audio tracks, resulting in a single flat timeline.

Before the final assertion, it checks that we got a result (3) and not Nothing. As it's using the `Gen.parallelWithClips` generator there should always be video clips in each parallel, and we should always successfully flatten and get a result. The final assertion (4) checks that rendering the original timeline gives the same result as the folded-together results of rendering each sequence.

The other property is very similar, but operates on parallels rather than sequences:

```
hprop_flat_timeline_is_same_as_all_its_flat_parallels =
  property $ do
    -- 1. Generate a timeline
    timeline' <- forAll $
```

```
    Gen.timeline (Range.exponential 0 5) Gen.parallelWithClips

  -- 2. Flatten all parallels and fold the resulting flat
  --    timelines together
  let flat = timeline' ^.. sequences . each . parallels . each
             & foldMap Render.flattenParallel

  -- 3. Make sure we successfully flattened the timeline
  flat /== Nothing

  -- 4. Flatten the entire timeline and compare to the flattened
  --    parallels
  Render.flattenTimeline timeline' === flat
```

The only difference is in the traversal (2), where we apply `Render.flattenParallel` to each parallel instead of applying `Render.flattenSequence` to each sequence.

17.6. Missing Properties

Whew! That was quite a lot of properties and code, especially for a warm-up. But timeline flattening could be tested more thoroughly! I haven't yet written the following properties, but I'm hoping to find some time to add them:

- **Clip playback timestamps are the same**. The "clip occurrence" property only checks that the hierarchical timeline's clips occur in the flat timeline. It doesn't check when in the flat timeline they occur. One way to test this would be to first annotate each clip in original timeline with its playback timestamp, and transfer this information through to the flat timeline. Then the timestamps could be included in the assertion.
- **Source assets used as still frame sources**. The "still frames used" properties only check the still frame mode of gaps, not the still frame sources. The algorithm could have a bug where it always uses the first video clip's asset as a frame source, and the current property tests would not catch it.
- **Same flat result is produced regardless of sequence grouping**. Sequences can be split or joined in any way without affecting the final rendered media. They are merely ways of organizing parallels in logical groups. A property could check that however you split or join sequences within a timeline, the flattened result is the same.

17.7. A Missing Feature

As pointed out earlier, parallels must have at least one video clip. The flattening algorithm can only locate still frame sources for video gaps from within the same parallel. This is an annoying limitation when working with Komposition, and the algorithm should be improved.

As the existing set of properties describe timeline flattening fairly well, changing the algorithm could be done with a TDD-like workflow:

1. Modify the property tests to capture the intended behaviour
2. Tests will fail, with the errors showing how the existing implementation fails to find still frame sources as expected
3. Change the implementation to make the tests pass

PBT is not only an after-the-fact testing technique. It can be used much like conventional example-based testing to drive development.

17.8. Obligatory Cliff-Hanger

In this post we've looked at timeline flattening, the simplest case study in the "Property-Based Testing in a Screencast Editor" series. The system under test was a module of pure functions, with complex enough behaviour to showcase PBT as a valuable tool. The tests are more closely related to the design choices and concrete representations of the implementation.

Coming case studies will dive deeper into the more complex subsystems of Komposition, and finally we'll see how PBT can be used for integration testing. At that level, the property tests are less tied to the implementation, and focus on describing the higher-level outcomes of the interaction between subsystems.

Next up is property tests for the video classifier. It's also implemented a pure function, but with slightly more complicated logic that is trickier to test. We're going to look at an interesting technique where we generate the *expected output* instead of the input.

18. Case Study 2: Video Scene Classification - Oskar Wickström

Original article: [17]

In the last case study on property-based testing (PBT) in Komposition we looked at timeline flattening. This post covers the video classifier, how it was tested before, and the bugs I found when I wrote property tests for it.

If you haven't read the introduction (see 16) or the first case study (see 17) yet, I recommend checking them out!

18.1. Classifying Scenes in Imported Video

Komposition can automatically classify *scenes* when importing video files. This is a central productivity feature in the application, effectively cutting recorded screencast material automatically, letting the user focus on arranging the scenes of their screencast. Scenes are segments that are considered *moving*, as opposed to *still* segments:

- A still segment is a sequence of at least S seconds of *near-equal* frames
- A moving segment is a sequence of *non-equal* frames, or a sequence of near-equal frames with a duration less than S

S is a preconfigured minimum still segment duration in Komposition. In the future it might be configurable from the user interface, but for now it's hard-coded.

Equality of two frames f_1 and f_2 is defined as a function $E(f_1, f_2)$, described informally as:

- comparing corresponding pixel color values of f_1 and f_2, with a small epsilon for tolerance of color variation, and
- deciding two frames equal when at least 99% of corresponding pixel pairs are considered equal.

In addition to the rules stated above, there are two edge cases:

1. The first segment is always a considered a moving segment (even if it's just a single frame)
2. The last segment may be a still segment with a duration less than S

The second edge case is not what I would call a desirable feature, but rather a shortcoming due to the classifier not doing any type of backtracking. This could be changed in the future.

18.2. Manually Testing the Classifier

The first version of the video classifier had no property tests. Instead, I wrote what I thought was a decent classifier algorithm, mostly messing around with various pixel buffer representations and parallel processing to achieve acceptable performance.

The only type of testing I had available, except for general use of the application, was a color-tinting utility. This was a separate program using the same classifier algorithm. It took as input a video file, and produced as output a video file where each frame was tinted green or red, for moving and still frames, respectively (Note from the editor: on the web-page there is a GIF showing the color tinting. For obvious reasons, an anmiated GIF cannot be included here).

In the (in this document omitted) recording above you see the color-tinted output video based on a recent version of the classifier. It classifies moving and still segments rather accurately. Before I wrote property tests and fixed the bugs that I found, it did not look so pretty, flipping back and forth at seemingly random places.

At first, debugging the classifier with the color-tinting tool way seemed like a creative and powerful technique. But the feedback loop was horrible, having to record video, process it using the slow color-tinting program, and inspecting it by eye. In hindsight, I can conclude that PBT is far more effective for testing the classifier.

18.3. Video Classification Properties

Figuring out how to write property tests for video classification wasn't obvious to me. It's not uncommon in example-based testing that tests end up mirroring the structure, and even the full implementation complexity, of the system under test. The same can happen in property-based testing.

With some complex systems it's very hard to describe the correctness as a relation between any valid input and the system's observed output. The video classifier is one such case. How do I decide if an output classification is correct for a specific input, without reimplementing the classification itself in my tests?

The other way around is easy, though! If I have a classification, I can convert that into video frames. Thus, the solution to the testing problem is to not generate the input, but instead generate the expected output. Hillel Wayne calls this technique "oracle generators" in his recent article[1].

The classifier property tests generate high-level representations of the expected classification output, which are lists of values describing the type and duration of segments. Next, the list of output segments is converted into a sequence of actual frames.

Figure 18.1.: A generated sequence of expected classified segments

Frames are two-dimensional arrays of RGB pixel values. The conversion is simple:

1 See the "Oracle Generators" section in Finding Property Tests (see 21).

- Moving segments are converted to a sequence of alternating frames, flipping between all gray and all white pixels
- Still frames are converted to a sequence of frames containing all black pixels

The example sequence in the diagram above, when converted to pixel frames with a frame rate of 10 FPS, can be visualized like in the following diagram, where each thin rectangle represents a frame: By generating high-level output and converting it to

Figure 18.2.: Pixel frames derived from a sequence of expected classified output segments

pixel frames, I have input to feed the classifier with, and I know what output it should produce. Writing effective property tests then comes down to writing generators that produce valid output, according to the specification of the classifier. In this post I'll show two such property tests.

18.4. Testing Still Segment Minimum Length

As stated in the beginning of this post, classified still segments must have a duration greater than or equal to S , where S is the minimum still segment duration used as a parameter for the classifier. The first property test we'll look at asserts that this invariant holds for all classification output.

```
hprop_classifies_still_segments_of_min_length = property $ do

  -- 1. Generate a minimum still segment length/duration
  minStillSegmentFrames <- forAll $ Gen.int (Range.linear 2 (2 * frameRate))
  let minStillSegmentTime = frameCountDuration minStillSegmentFrames

  -- 2. Generate output segments
  segments <- forAll $
    genSegments (Range.linear 1 10)
                (Range.linear 1
                              (minStillSegmentFrames * 2))
                (Range.linear minStillSegmentFrames
                              (minStillSegmentFrames * 2))
                resolution

  -- 3. Convert test segments to actual pixel frames
  let pixelFrames = testSegmentsToPixelFrames segments

  -- 4. Run the classifier on the pixel frames
  let counted = classifyMovement minStillSegmentTime (Pipes.each pixelFrames)
                & Pipes.toList
                & countSegments
```

```
  -- 5. Sanity check
  countTestSegmentFrames segments === totalClassifiedFrames counted

  -- 6. Ignore last segment and verify all other segments
  case initMay counted of
    Just rest ->
      traverse_ (assertStillLengthAtLeast minStillSegmentTime) rest
    Nothing -> success
  where
    resolution = 10 :. 10
```

This chunk of test code is pretty busy, and it's using a few helper functions that I'm not going to bore you with. At a high level, this test:

1. Generates a minimum still segment duration, based on a minimum frame count (let's call it n) in the range $[2, 20]$. The classifier currently requires that $n \geq 2$, hence the lower bound. The upper bound of 20 frames is an arbitrary number that I've chosen.
2. Generates valid output segments using the custom generator `genSegments`, where
   - moving segments have a frame count in $[1, 2n]$, and
   - still segments have a frame count in $[n, 2n]$.
3. Converts the generated output segments to actual pixel frames. This is done using a helper function that returns a list of alternating gray and white frames, or all black frames, as described earlier.
4. Count the number of consecutive frames within each segment, producing a list like `[Moving 18, Still 5, Moving 12, Still 30]`.
5. Performs a sanity check that the number of frames in the generated expected output is equal to the number of frames in the classified output. The classifier must not lose or duplicate frames.
6. Drops the last classified segment, which according to the specification can have a frame count less than n, and asserts that all other still segments have a frame count greater than or equal to n.

Let's run some tests.

```
> :{
| hprop_classifies_still_segments_of_min_length
|   & Hedgehog.withTests 10000
|   & Hedgehog.check
| :}

 ✓<interactive> passed 10000 tests.
```

Cool, it looks like it's working.

Sidetrack: Why generate the output?
Now, you might wonder why I generate output segments first, and then convert to pixel frames. Why not generate random pixel frames to begin with? The property test above only checks that the still segments are long enough!

The benefit of generating valid output becomes clearer in the next property test, where I use it as the expected output of the classifier. Converting the output to a sequence of pixel frames is easy, and I don't have to state any complex relation between the input and output in my property. When using oracle generators, the assertions can often be plain equality checks on generated and actual output.

But there's benefit in using the same oracle generator for the "minimum still segment length" property, even if it's more subtle. By generating valid output and converting to pixel frames, I can generate inputs that cover the edge cases of the system under test. Using property test statistics and coverage checks, I could inspect coverage, and even fail test runs where the generators don't hit enough of the cases I'm interested in[2].

Had I generated random sequences of pixel frames, then perhaps the majority of the generated examples would only produce moving segments. I could tweak the generator to get closer to either moving or still frames, within some distribution, but wouldn't that just be a variation of generating valid scenes? It would be worse, in fact. I wouldn't then be reusing existing generators, and I wouldn't have a high-level representation that I could easily convert from and compare with in assertions.

18.5. Testing Moving Segment Time Spans

The second property states that the classified moving segments must start and end at the same timestamps as the moving segments in the generated output. Compared to the previous property, the relation between generated output and actual classified output is stronger.

```
hprop_classifies_same_scenes_as_input = property $ do
  -- 1. Generate a minimum still still segment duration
  minStillSegmentFrames <- forAll $ Gen.int (Range.linear 2 (2 * frameRate))
  let minStillSegmentTime = frameCountDuration minStillSegmentFrames

  -- 2. Generate test segments
  segments <- forAll $ genSegments (Range.linear 1 10)
                                   (Range.linear 1
                                                 (minStillSegmentFrames * 2))
                                   (Range.linear minStillSegmentFrames
                                                 (minStillSegmentFrames * 2))
                                   resolution

  -- 3. Convert test segments to actual pixel frames
  let pixelFrames = testSegmentsToPixelFrames segments
```

2 John Hughes' talk Building on developers' intuitions goes into depth on this. There's also work being done to provide similar functionality for Hedgehog.

```
  -- 4. Convert expected output segments to a list of expected time spans
  --    and the full duration
  let durations = map segmentWithDuration segments
      expectedSegments = movingSceneTimeSpans durations
      fullDuration = foldMap unwrapSegment durations

  -- 5. Classify movement of frames
  let classifiedFrames =
        Pipes.each pixelFrames
        & classifyMovement minStillSegmentTime
        & Pipes.toList

  -- 6. Classify moving scene time spans
  let classified =
        (Pipes.each classifiedFrames
         & classifyMovingScenes fullDuration)
        >-> Pipes.drain
        & Pipes.runEffect
        & runIdentity

  -- 7. Check classified time span equivalence
  expectedSegments === classified

  where
    resolution = 10 :. 10
```

Steps 1–3 are the same as in the previous property test. From there, this test:

4. Converts the generated output segments into a list of time spans. Each time span marks the start and end of an expected moving segment. Furthermore, it needs the full duration of the input in step 6, so that's computed here.
5. Classify the movement of each frame, i.e. if it's part of a moving or still segment.
6. Run the second classifier function called classifyMovingScenes, based on the full duration and the frames with classified movement data, resulting in a list of time spans.
7. Compare the expected and actual classified list of time spans.

While this test looks somewhat complicated with its setup and various conversions, the core idea is simple. But is it effective?

18.6. Bugs! Bugs everywhere!

Preparing for a talk on property-based testing, I added the "moving segment time spans" property a week or so before the event. At this time, I had used Komposition to edit multiple screencasts. Surely, all significant bugs were caught already. Adding

property tests should only confirm the level of quality the application already had. Right?

Nope. First, I discovered that my existing tests were fundamentally incorrect to begin with. They were not reflecting the specification I had in mind, the one I described in the beginning of this post.

Furthermore, I found that the generators had errors. At first, I used Hedgehog to generate the pixels used for the classifier input. Moving frames were based on a majority of randomly colored pixels and a small percentage of equally colored pixels. Still frames were based on a random single color.

The problem I had not anticipated was that the colors used in moving frames were not guaranteed to be distinct from the color used in still frames. In small-sized examples I got black frames at the beginning and end of moving segments, and black frames for still segments, resulting in different classified output than expected. Hedgehog shrinking the failing examples' colors towards 0, which is black, highlighted this problem even more.

I made my generators much simpler, using the alternating white/gray frames approach described earlier, and went on to running my new shiny tests. Here's what I got: What? Where does 0s–0.6s come from? The classified time span should've been

```
── test/Komposition/Import/Video/FFmpegTest.hs ──
208 hprop_classifies_same_scenes_as_input = withTests 100 . property $ do
209   -- Generate test segments
210   segments <- forAll $ genSegments (Range.linear (frameRate * 1) (frameRate * 5)) resolution
    | [ Scene 10 ]
211   -- Convert test segments to timespanned ones, and actual pixel frames
212   let segmentsWithTimespans = segments
213                             & map segmentWithDuration
214                             & segmentTimeSpans
215       pixelFrames = testSegmentsToPixelFrames segments
216       fullDuration = foldMap
217                      (durationOf AdjustedDuration . unwrapSegment)
218                      segmentsWithTimespans
219   -- Run classifier on pixel frames
220   classified <-
221     (Pipes.each pixelFrames
222      & classifyMovement 2.0
223      & classifyMovingScenes fullDuration)
224     >-> Pipes.drain
225     & Pipes.runEffect
226   -- Check classified timespan equivalence
227   unwrapScenes segmentsWithTimespans === classified
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    | Failed (- lhs =/= + rhs)
    |   [
    | -   TimeSpan { spanStart = Duration 0 s , spanEnd = Duration 1 s }
    | +   TimeSpan { spanStart = Duration 0 s , spanEnd = Duration 0.6 s }
    |   ]
228
229   where resolution = 20 :. 20
```

Figure 18.3.: Hedgehog output

0s–1s, as the generated output has a single moving scene of 10 frames (1 second at 10 FPS). I started digging, using the `annotate` function in Hedgehog to inspect the generated and intermediate values in failing examples.

I couldn't find anything incorrect in the generated data, so I shifted focus to the implementation code. The end timestamp 0.6s was consistently showing up in failing

examples. Looking at the code, I found a curious hard-coded value 0.5 being bound and used locally in `classifyMovement`.

The function is essentially a *fold* over a stream of frames, where the accumulator holds vectors of previously seen and not-yet-classified frames. Stripping down and simplifying the old code to highlight one of the bugs, it looked something like this:

```
classifyMovement minStillSegmentTime =
  case ... of
    InStillState{..} ->
      if someDiff > minEqualTimeForStill
        then ...
        else ...
    InMovingState{..} ->
      if someOtherDiff >= minStillSegmentTime
        then ...
        else ...
  where
    minEqualTimeForStill = 0.5
```

Let's look at what's going on here. In the `InStillState` branch it uses the value `minEqualTimeForStill`, instead of always using the `minStillSegmentTime` argument. This is likely a residue from some refactoring where I meant to make the value a parameter instead of having it hard-coded in the definition.

Sparing you the gory implementation details, I'll outline two more problems that I found. In addition to using the hard-coded value, it incorrectly classified frames based on that value. Frames that should've been classified as "moving" ended up "still". That's why I didn't get 0s–1s in the output.

Why didn't I see 0s–0.5s, given the hard-coded value 0.5? Well, there was also an off-by-one bug, in which one frame was classified incorrectly together with the accumulated moving frames.

The `classifyMovement` function is 30 lines of Haskell code juggling some state, and I managed to mess it up in three separate ways at the same time. With these tests in place I quickly found the bugs and fixed them. I ran thousands of tests, all passing.

Finally, I ran the application, imported a previously recorded video, and edited a short screencast. The classified moving segments where *notably* better than before.

18.7. Summary

A simple streaming fold can hide bugs that are hard to detect with manual testing. The consistent result of 0.6, together with the hard-coded value 0.5 and a frame rate of 10 FPS, pointed clearly towards an off-by-one bug. I consider this is a great showcase of how powerful shrinking in PBT is, consistently presenting minimal examples that point towards specific problems. It's not just a party trick on ideal mathematical functions.

Could these errors have been caught without PBT? I think so, but what effort would it require? Manual testing and introspection did not work for me. Code review might have revealed the incorrect definition of `minEqualTimeForStill`, but perhaps not the off-by-one and incorrect state handling bugs. There are of course many other QA techniques, I won't evaluate all. But given the low effort that PBT requires in this setting,

the amount of problems it finds, and the accuracy it provides when troubleshooting, I think it's a clear win.

I also want to highlight the iterative process that I find naturally emerges when applying PBT:

1. Think about how your system is supposed to work. Write down your specification.
2. Think about how to generate input data and how to test your system, based on your specification. Tune your generators to provide better test data. Try out alternative styles of properties. Perhaps model-based or metamorphic testing fits your system better.
3. Run tests and analyze the minimal failing examples. Fix your implementation until all tests pass.

This can be done when modifying existing code, or when writing new code. You can apply this without having any implementation code yet, perhaps just a minimal stub, and the workflow is essentially the same as TDD.

18.8. Coming Up

The final post in this series will cover testing at a higher level of the system, with effects and multiple subsystems being integrated to form a full application. We will look at property tests that found many bugs and that made a substantial refactoring possible.

19. Case Study 3: Integration Testing - Oskar Wickström

Original article: [18]

This is the final case study in the "Property-Based Testing in a Screencast Editor" series. It covers property-based integration testing and its value during aggressive refactoring work within Komposition.

19.1. A History of Two Stacks

In Komposition, a project's state is represented using an in-memory data structure. It contains the hierarchical timeline, the focus, import and render settings, project storage file paths, and more. To let users navigate backwards and forwards in their history of project edits, for example when they have made a mistake, Komposition supplies *undo* and *redo* commands.

The undo/redo history was previously implemented as a data structure recording project states, compromised of:

- a *current state* variable
- a stack of *previous states*
- a stack of *possible future states*

The undo/redo history data structure held entire project state values. Each undoable and redoable user action created a new state value. Let's look a bit closer at how this worked.

19.1.1. Performing Actions

When a user performed an undoable/redoable action, the undo/redo history would:

- push the previous state onto the undo stack
- perform the action and replace the current state
- discard all states in the redo stack

This can be visualized as in the following diagram, where the state d is being replaced with a new state h, and d being pushed onto the undo stack. The undo/redo history to the left of the dividing line is the original, and the one to the right is the resulting history. Again, note that performing new actions discarded all states in the redo stack.

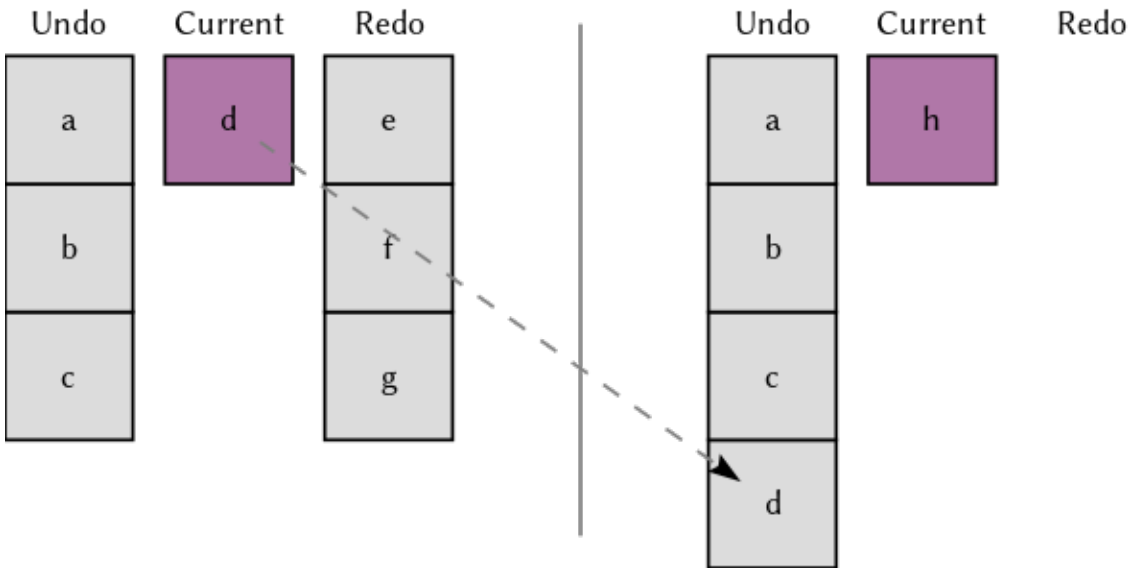

Figure 19.1.: Performing an action pushes the previous state onto the undo stack and discards the redo stack

19.1.2. Undoing Actions

When the user chose to undo an action, the undo/redo history would:

- pop the undo stack and use that state as the current state
- push the previous state onto the redo stack

The following diagram shows how undoing the last performed action's resulting state, *d*, pushes *d* onto the redo stack, and pops *c* from the undo stack to use that as the current state.

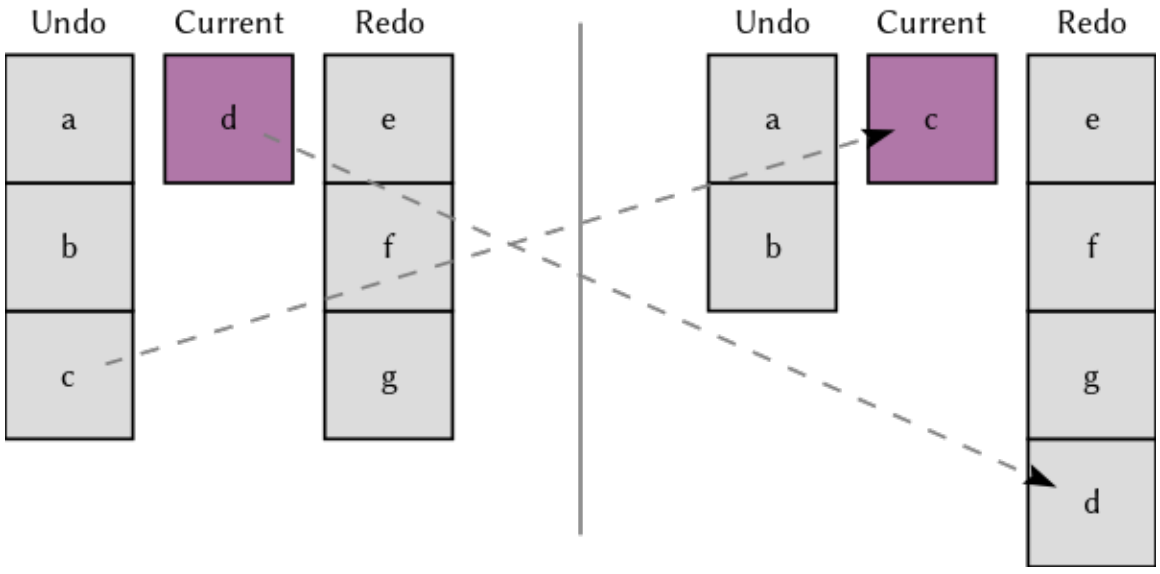

Figure 19.2.: Undoing pushes the previous state onto the redo stack and pops the undo stack for a current state

19.1.3. Redoing Actions

When the user chose to redo an action, the undo/redo history would:

- pop the redo stack and use that state as the current state
- push the previous state onto the undo stack

The last diagram shows how redoing, recovering a previously undone state, pops g from the redo stack to use that as the current state, and pushes the previous state d onto the undo stack. Note that not all user actions in Komposition are un-

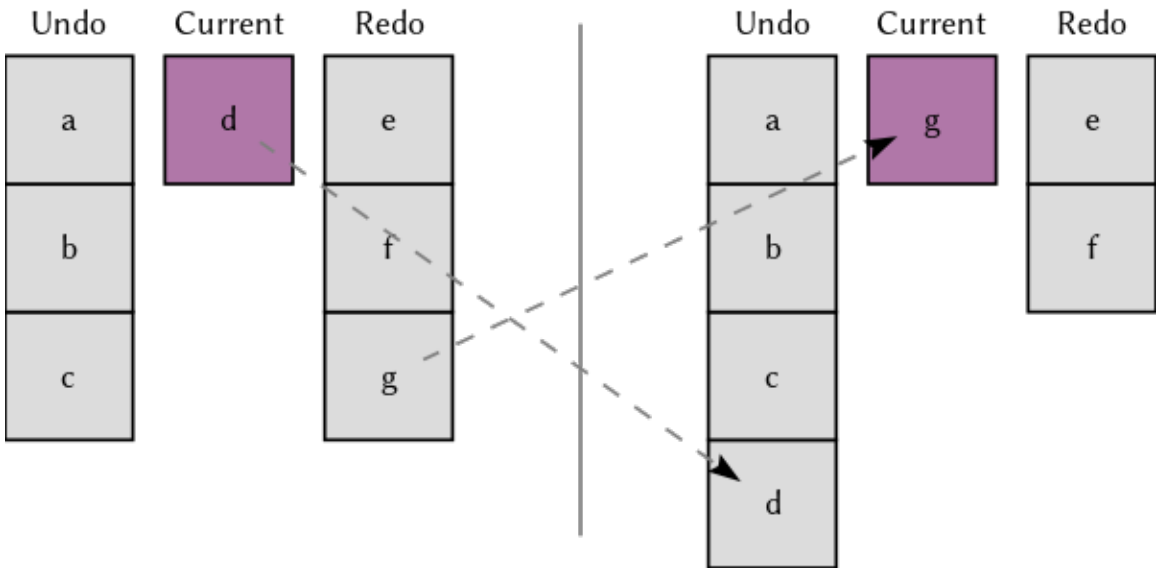

Figure 19.3.: Undoing pushes the previous state onto the redo stack and pops the undo stack for a current state

doable/redoable. Actions like navigating the focus or zooming are not recorded in the history.

19.1.4. Dealing With Performance Problems

While the "two stacks of states" algorithm was easy to understand and implement, it failed to meet my non-functional requirements. A screencast project compromised of hundreds or thousands of small edits would consume gigabytes of disk space when stored, take tens of seconds to load from disk, and consume many gigabytes of RAM when in memory.

Now, you might think that my implementation was incredibly naive, and that the performance problems could be fixed with careful profiling and optimization. And you'd probably be right! I did consider going down that route, optimizing the code, time-windowing edits to compact history on the fly, and capping the history at some fixed size. Those would all be interesting pursuits, but in the end I decided to try something else.

19.2. Refactoring with Property-Based Integration Tests

Instead of optimizing the current stack-based implementation, I decided to implement the undo/redo history in terms of inverse actions. In this model, actions not only modify the project state, they also return another action, its inverse, that *reverses* the effects of the original action. Instead of recording a new project state data structure for each edit, the history only records descriptions of the actions themselves.

I realized early that introducing the new undo/redo history implementation in Komposition was not going to be a small task. It would touch the majority of command implementation code, large parts of the main application logic, and the project binary serialization format. What it wouldn't affect, though, was the module describing user commands in abstract.

To provide a safety net for the refactoring, I decided to cover the undo/redo functionality with tests. As the user commands would stay the same throughout my modifications, I chose to test at that level, which can be characterized as integration-level testing. The tests run Komposition, including its top-level application control flow, but with the user interface and some other effects stubbed out. Making your application testable at this level is hard work, but the payoff can be huge.

With Komposition featuring close to twenty types of user commands, combined with a complex hierarchical timeline and navigation model, the combinatory explosion of possible states was daunting. Relying on example-based tests to safeguard my work was not satisfactory. While PBT couldn't cover the entire state space either, I was confident it would improve my chances of finding actual bugs.

19.2.1. Undo/Redo Tests

Before I began refactoring, I added tests for the inverse property of undoable/redoable actions. The first test focuses on undoing actions, and is structured as follows:

1. Generate an initial project and application state
2. Generate a sequence of undoable/redoable commands (wrapped in *events*)
3. Run the application with the initial state and the generated events
4. Run an undo command for each original command
5. Assert that final timeline is equal to the initial timeline

Let's look at the Haskell Hedgehog property test:

```
hprop_undo_actions_are_undoable = property $ do

  -- 1. Generate initial timeline and focus
  timelineAndFocus <- forAllWith showTimelineAndFocus $
    Gen.timelineWithFocus (Range.linear 0 10) Gen.parallel

  -- ... and initial application state
  initialState <- forAll (initializeState timelineAndFocus)
```

```
  -- 2. Generate a sequence of undoable/redoable commands
  events <- forAll $
    Gen.list (Range.exponential 1 100) genUndoableTimelineEvent

  -- 3. Run 'events' on the original state
  beforeUndos <- runTimelineStubbedWithExit events initialState

  -- 4. Run as many undo commands as undoable commands
  afterUndos <- runTimelineStubbedWithExit (undoEvent <$ events) beforeUndos

  -- 5. That should result in a timeline equal to the one we started
  -- with
  timelineToTree (initialState ^. currentTimeline)
    === timelineToTree (afterUndos ^. currentTimeline)
```

The second test, focusing on redoing actions, is structured very similarly to the previous test:

1. Generate an initial project and application state
2. Generate a sequence of undoable commands (wrapped in *events*)
3. Run the application with the initial state and the generated events
4. Run an undo commands for each original command
5. Run an redo commands for each original command
6. Assert that final timeline is equal to the timeline before undoing actions

The test code is also very similar:

```
hprop_undo_actions_are_redoable = property $ do

  -- 1. Generate the initial timeline and focus
  timelineAndFocus <- forAllWith showTimelineAndFocus $
    Gen.timelineWithFocus (Range.linear 0 10) Gen.parallel

  -- ... and the initial application state
  initialState <- forAll (initializeState timelineAndFocus)

  -- 2. Generate a sequence of undoable/redoable commands
  events <- forAll $
    Gen.list (Range.exponential 1 100) genUndoableTimelineEvent

  -- 3. Run 'events' on the original state
  beforeUndos <- runTimelineStubbedWithExit events initialState

  -- 4. Run undo commands corresponding to all original commands
  afterRedos  <-
    runTimelineStubbedWithExit (undoEvent <$ events) beforeUndos
```

```
    -- 5. Run redo commands corresponding to all original commands
    >>= runTimelineStubbedWithExit (redoEvent <$ events)

  -- 6. That should result in a timeline equal to the one we had
  -- before undoing actions
  timelineToTree (beforeUndos ^. currentTimeline)
    === timelineToTree (afterRedos ^. currentTimeline)
```

Note that these tests only assert on the equality of timelines, not entire project states, as undoable commands only operate on the timeline.

19.2.2. All Tests Passing, Everything Works

The undo/redo tests were written and run on the original stack-based implementation, kept around during a refactoring that took me two weeks of hacking during late nights and weekends, and finally run and passing with the new implementation based on inverse actions. Except for a few minimal adjustments to data types, these tests stayed untouched during the entire process.

The confidence I had when refactoring felt like a super power. Two simple property tests made the undertaking possible. They found numerous bugs, including:

- Off-by-one index errors in actions modifying the timeline
- Inconsistent timeline focus:
  - focus was incorrectly restored on undoing an action
  - focus was outside of the timeline bounds
- Non-inverse actions:
  - actions returning incorrectly constructed inverses
  - the inverse of splitting a sequence is joining sequences, and joining them back up didn't always work

After all tests passed, I ran the application with its GUI, edited a screencast project, and it all worked flawlessly. It's almost too good to be true, right?

Property testing is not a silver bullet, and there might still be bugs lurking in my undo/redo history implementation. The tests I run are never going to be exhaustive and my generators might be flawed. That being said, they gave me a confidence in refactoring that I've never had before. Or maybe I just haven't hit that disastrous edge case yet?

19.3. Why Test With Properties?

This was the last case study in the "Property-Based Testing in a Screencast Editor" series. I've had a great time writing these articles and giving talks on the subject. Before I wrap up, I'll summarize my thoughts on PBT in general and my experience with it in Komposition.

Property-based testing is not only for pure functions; you can use it to test effectful actions. It is not only for unit testing; you can write integration tests using properties.

It's not only for functional programming languages; there are good frameworks for most popular programming languages.

Properties describe the general behavior of the system under test, and verify its correctness using a variety of inputs. Not only is this an effective way of finding errors, it's also a concise way of documenting the system.

The iterative process in property-based testing, in my experience, comes down to the following steps:

1. Think about the specification of your system under test
2. Think about how generators and tests should work
3. Write or modify generators, tests, and implementation code, based on steps 1 and 2
4. Get minimal examples of failing tests
5. Repeat

Using PBT within Komposition has made it possible to confidently refactor large parts of the application. It has found errors in my thinking, my generators, my tests, and in my implementation code. Testing video scene classification went from a time consuming, repetitive, and manual verification process to a fast, effective, and automated task.

In short, it's been a joy, and I look forward to continue using PBT in my work and in my own projects. I hope I've convinced you of its value, and inspired you to try it out, no matter what kind of project you're working on and what programming language you are using. Involve your colleagues, practice writing property tests together, and enjoy finding complicated bugs before your users do!

20. Choosing properties for property-based testing - Scott Wlaschin

William Yao: *More starting points for figuring out what properties to test.*

Original article: [19]

UPDATE: I did a talk on property-based testing based on these posts. Slides and video here.

In the previous post, I described the basics of property-based testing, and showed how it could save a lot of time by generating random tests.

But here's a common problem. Everyone who sees a property-based testing tool like FsCheck or QuickCheck thinks that it is amazing...but when it times come to start creating your own properties, the universal complaint is: "what properties should I use? I can't think of any!"

The goal of this post is to show some common patterns that can help you discover the properties that are applicable to your code.

20.1. Categories for properties

In my experience, many properties can be discovered by using one of the seven approaches listed below.

- "Different paths, same destination" (see 20.1.1)
- "There and back again" (see 20.1.2)
- "Some things never change" (see 20.1.3)
- "The more things change, the more they stay the same" (see 20.1.4)
- "Solve a smaller problem first" (see 20.1.5)
- "Hard to prove, easy to verify" (see 20.1.6)
- "The test oracle" (see 20.1.7)

This is by no means a comprehensive list, just the ones that have been most useful to me. For a different perspective, check out the list of patterns that the PEX team at Microsoft have compiled.

20.1.1. "Different paths, same destination"

These kinds of properties are based on combining operations in different orders, but getting the same result. For example, in the diagram below, doing *X* then *Y* gives the same result as doing *Y* followed by *X* (see figure 20.1). The commutative property of

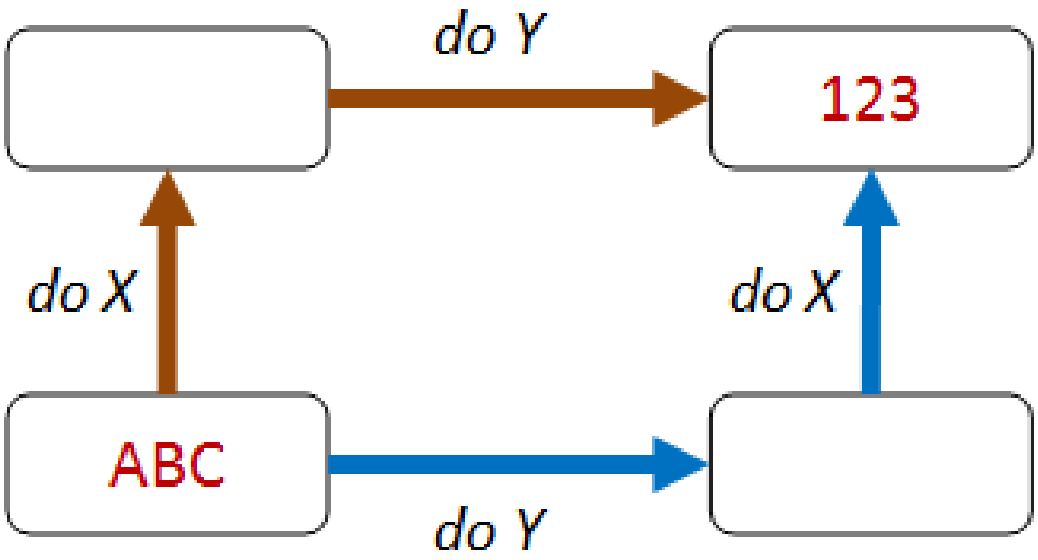

Figure 20.1.: Commutative Properties

addition is an obvious example of this pattern. For example, the result of add 1 then add 2 is the same as the result of add 2 followed by add 1.

This pattern, generalized, can produce a wide range of useful properties. We'll see some more uses of this pattern later in this post.

20.1.2. "There and back again"

These kinds of properties are based on combining an operation with its inverse, ending up with the same value you started with.

In the diagram below, doing *X* serializes ABC to some kind of binary format, and the inverse of *X* is some sort of deserialization that returns the same ABC value again (see figure 20.2). In addition to serialization/deserialization, other pairs of operations can be checked this way: `addition/subtraction`, `write/read`, `setProperty/getProperty`, and so on.

Other pair of functions fit this pattern too, even though they are not strict inverses, pairs such as `insert/contains`, `create/exists` , etc.

20.1.3. "Some things never change"

These kinds of properties are based on an invariant that is preserved after some transformation.

In the diagram below (figure 20.3), the transform changes the order of the items, but the same four items are still present afterwards. Common invariants include size of a collection (for `map` say), the contents of a collection (for sort say), the height or depth of something in proportion to size (e.g. balanced trees).

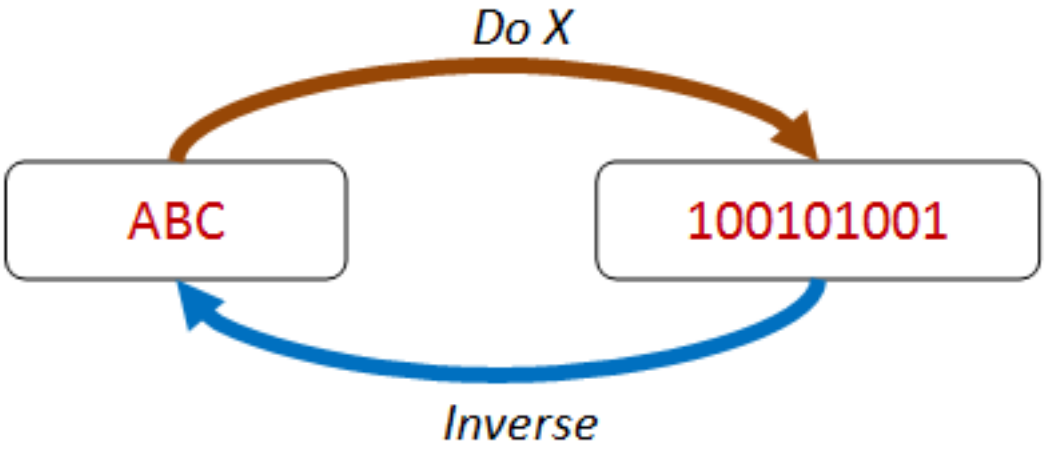

Figure 20.2.: Inverse Properties

Figure 20.3.: Invariant Properties

20.1.4. "The more things change, the more they stay the same"

These kinds of properties are based on "idempotence" – that is, doing an operation twice is the same as doing it once.

In the diagram below (figure 20.4), using distinct to filter the set returns two items, but doing distinct twice returns the same set again. Idempotence properties are very

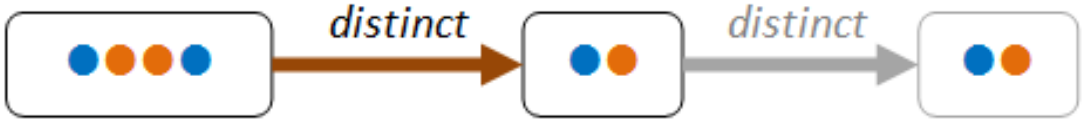

Figure 20.4.: Idempotent Properties

useful, and can be extended to things like database updates and message processing.

20.1.5. "Solve a smaller problem first"

These kinds of properties are based on "structural induction" – that is, if a large thing can be broken into smaller parts, and some property is true for these smaller parts, then you can often prove that the property is true for a large thing as well.

In the diagram below (figure 20.5), we can see that the four-item list can be partitioned into an item plus a three-item list, which in turn can be partitioned into an item

plus a two-item list. If we can prove the property holds for two-item list, then we can infer that it holds for the three-item list, and for the four-item list as well. Induction

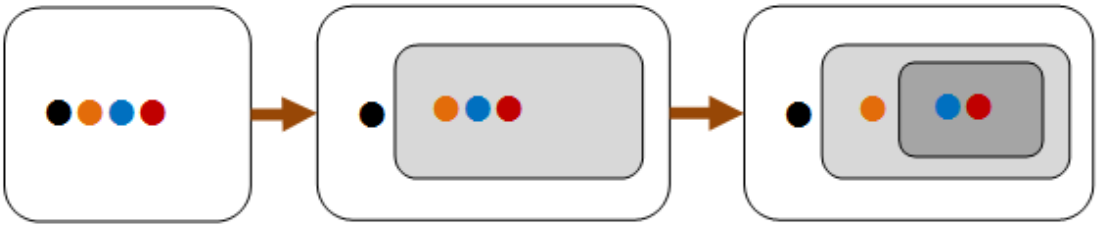

Figure 20.5.: Induction Properties

properties are often naturally applicable to recursive structures such as lists and trees.

20.1.6. "Hard to prove, easy to verify"

Often an algorithm to find a result can be complicated, but verifying the answer is easy.

In the diagram below (figure 20.6), we can see that finding a route through a maze is hard, but checking that it works is trivial! Many famous problems are of this sort,

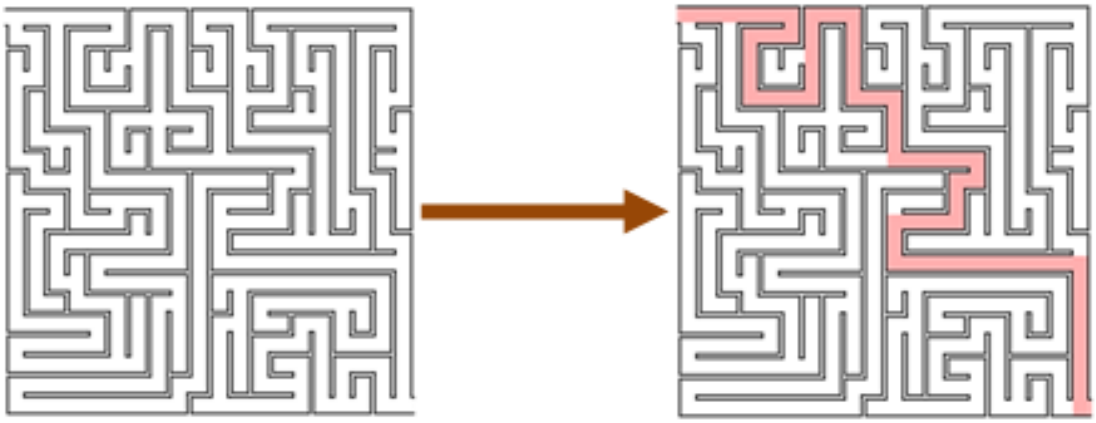

Figure 20.6.: Easy to Verify Properties

such as prime number factorization. But this approach can be used for even simple problems.

For example, you might check that a string tokenizer works by just concatenating all the tokens again. The resulting string should be the same as what you started with.

20.1.7. "The test oracle"

In many situations you often have an alternate version of an algorithm or process (a "test oracle") that you can use to check your results (figure 20.7). For example, you might have a high-performance algorithm with optimization tweaks that you want

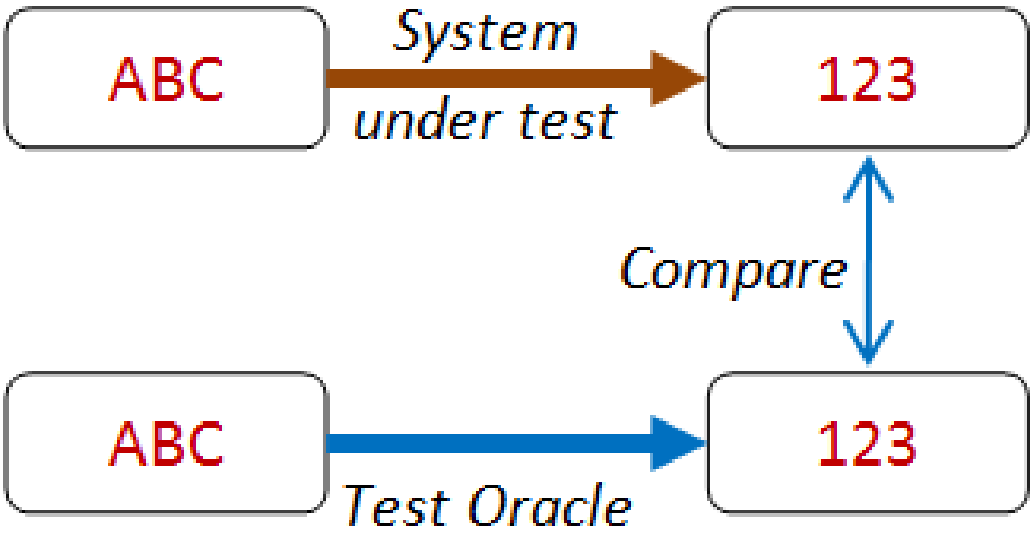

Figure 20.7.: Test Oracle

to test. In this case, you might compare it with a brute force algorithm that is much slower but is also much easier to write correctly.

Similarly, you might compare the result of a parallel or concurrent algorithm with the result of a linear, single thread version.

20.2. Putting the categories to work with some real examples

In this section, we'll apply these categories to see if we can come up with properties for some simple functions such as "sort a list" and "reverse a list".

20.2.1. "Different paths, same destination" applied to a list sort

Let's start with "different paths, same destination" and apply it to a "list sort" function.

Can we think of any way of combining an operation *before* `List.sort`, and another operation *after* `List.sort`, so that you should end up with the same result? That is, so that "going up then across the top" is the same as "going across the bottom then up". How about this?

- **Path 1**: We add one to each element of the list, then sort.
- **Path 2**: We sort, then add one to each element of the list.
- Both lists should be equal.

Here's some code that implements that property:

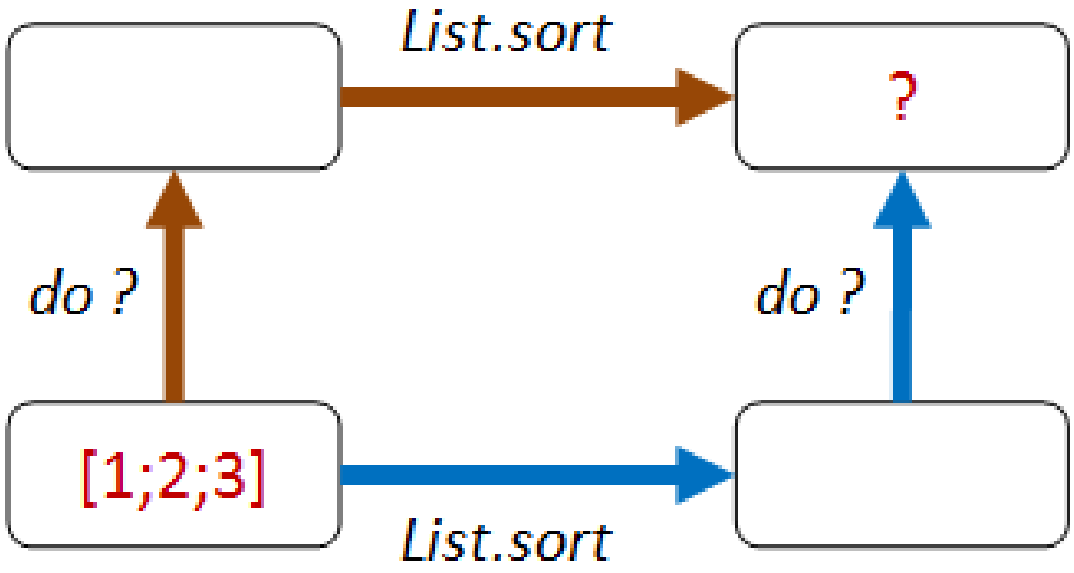

Figure 20.8.: List Sort Property

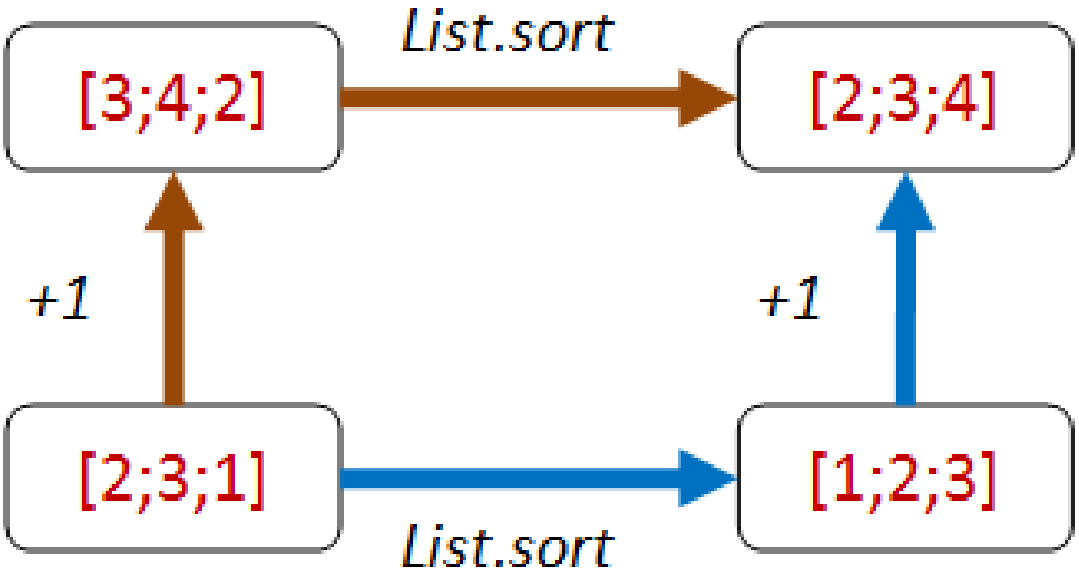

Figure 20.9.: List Sort Property +1

```
let ``+1 then sort should be same as sort then +1`` sortFn aList =
    let add1 x = x + 1

    let result1 = aList |> sortFn |> List.map add1
    let result2 = aList |> List.map add1 |> sortFn
    result1 = result2

// test
let goodSort = List.sort
Check.Quick (``+1 then sort should be same as sort then +1`` goodSort)
// Ok, passed 100 tests.
```

Well, that works, but it also would work for a lot of other transformations too. For example, if we implemented `List.sort` as just the identity, then this property would be satisfied equally well! You can test this for yourself:

```
let badSort aList = aList
Check.Quick (``+1 then sort should be same as sort then +1`` badSort)
// Ok, passed 100 tests.
```

The problem with this property is that it is not exploiting any of the "sortedness". We know that a sort will probably reorder a list, and certainly, the smallest element should be first.

How about adding an item that we know will come at the front of the list after sorting?

- **Path 1**: We append Int32.MinValue to the end of the list, then sort.
- **Path 2**: We sort, then prepend Int32.MinValue to the front of the list.
- Both lists should be equal.

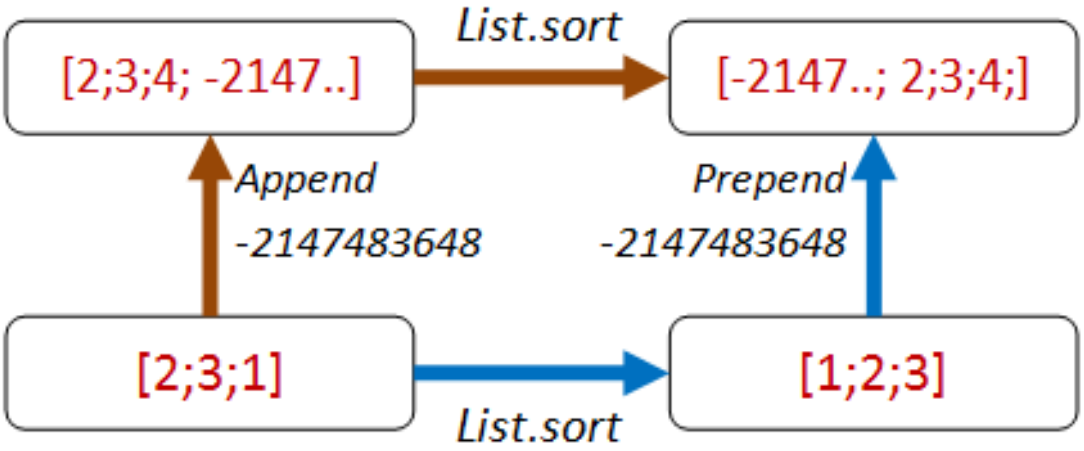

Figure 20.10.: List Sort Property with `Int32.MinValue`

Here's the code:

```
let ``append minValue then sort should be same as sort then prepend minValue``
    sortFn aList =
        let minValue = Int32.MinValue

        let appendThenSort = (aList @ [minValue]) |> sortFn
        let sortThenPrepend = minValue :: (aList |> sortFn)
        appendThenSort = sortThenPrepend

// test
Check.Quick (``append minValue then sort should be same as sort then
    prepend minValue`` goodSort)
// Ok, passed 100 tests.
```

The bad implementation fails now!

```
Check.Quick (``append minValue then sort should be same as sort then
    prepend minValue`` badSort)
// Falsifiable, after 1 test (2 shrinks)
// [0]
```

In other words, the bad sort of `[0; minValue]` is *not* the same as `[minValue; 0]`. So that's good!

But...we've got some hard coded things in there that the Enterprise Developer From Hell (see previous post) could take advantage of! The EDFH will exploit the fact that we always use `Int32.MinValue` and that we always prepend or append it to the test list.

In other words, the EDFH can identify which path we are on and have special cases for each one:

```
// The Enterprise Developer From Hell strikes again
let badSort2 aList =
        match aList with
        | [] -> []
        | _ ->
                let last::reversedTail = List.rev aList
                if (last = Int32.MinValue) then
                        // if min is last, move to front
                        let unreversedTail = List.rev reversedTail
                        last :: unreversedTail
                else
                        aList // leave alone
```

And when we check it...

```
// Oh dear, the bad implementation passes!
Check.Quick (``append minValue then sort should be same as sort then
    prepend minValue`` badSort2)
// Ok, passed 100 tests.
```

We could fix this by (a) picking a random number smaller than any number in the list and (b) inserting it at a random location rather than always appending it. But rather than getting too complicated, let's stop and reconsider.

An alternative approach which also exploits the "sortedness" is to first negate all the values, then on the path that negates *after* the sort, add an extra reverse as well.

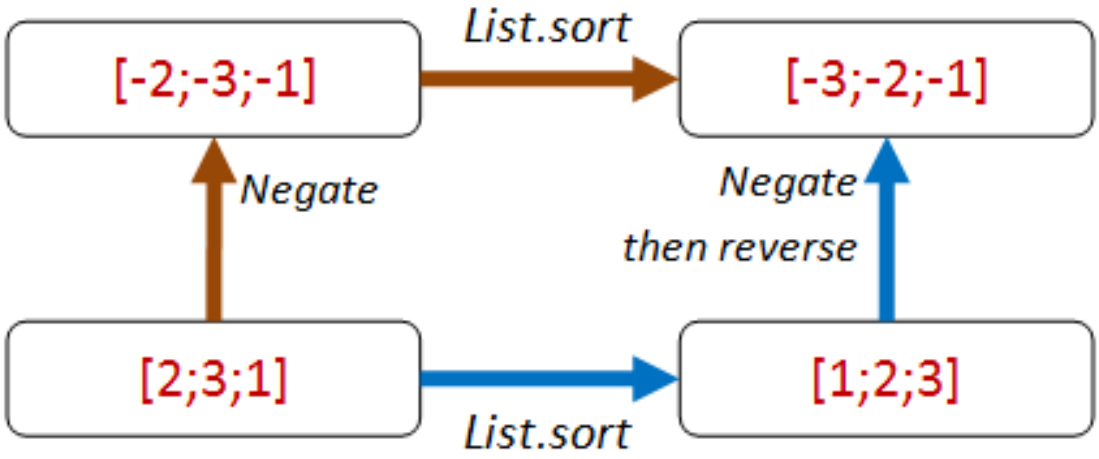

Figure 20.11.: List Sort Property with negate

```
let ``negate then sort should be same as sort then negate then reverse``
    sortFn aList =
        let negate x = x * -1

        let negateThenSort = aList |> List.map negate |> sortFn
        let sortThenNegateAndReverse = aList |> sortFn |> List.map
                negate |> List.rev
        negateThenSort = sortThenNegateAndReverse
```

This property is harder for the EDFH to beat because there are no magic numbers to help identify which path you are on:

```
// test
Check.Quick ( ``negate then sort should be same as sort then negate then
    reverse`` goodSort)
// Ok, passed 100 tests.

// test
Check.Quick ( ``negate then sort should be same as sort then negate then
    reverse``  badSort)
// Falsifiable, after 1 test (1 shrinks)
// [1; 0]

// test
Check.Quick ( ``negate then sort should be same as sort then negate then
    reverse``  badSort2)
// Falsifiable, after 5 tests (3 shrinks)
// [1; 0]
```

You might argue that we are only testing sorting for lists of integers. But the `List.sort` function is generic and knows nothing about integers per se, so I have high confidence that this property does test the core sorting logic.

Applying "different paths, same destination" to a list reversal function

Ok, enough of `List.sort`. What about applying the same ideas to the list reversal function?

We can do the same append/prepend trick:

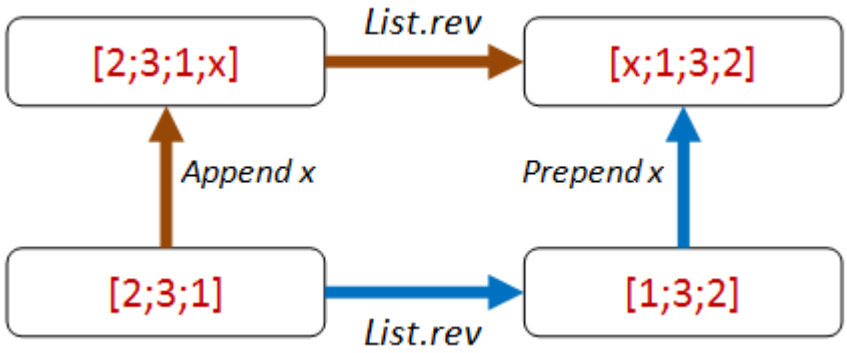

Figure 20.12.: List reverse

Here's the code for the property:

```
let ``append any value then reverse should be same as reverse then prepend
    same value`` revFn anyValue aList =

        let appendThenReverse = (aList @ [anyValue]) |> revFn
        let reverseThenPrepend = anyValue :: (aList |> revFn)
        appendThenReverse = reverseThenPrepend
```

Here are the test results for the correct function and for two incorrect functions:

```
// test
let goodReverse = List.rev
Check.Quick (``append any value then reverse should be same as reverse
    then prepend same value`` goodReverse)
// Ok, passed 100 tests.

// bad implementation fails
let badReverse aList = []
Check.Quick (``append any value then reverse should be same as reverse
    then prepend same value`` badReverse)
// Falsifiable, after 1 test (2 shrinks)
// true, []

// bad implementation fails
let badReverse2 aList = aList
Check.Quick (``append any value then reverse should be same as reverse
```

```
    then prepend same value`` badReverse2)
// Falsifiable, after 1 test (1 shrinks)
// true, [false]
```

You might notice something interesting here. I never specified the type of the list. The property works with *any* list.

In cases like these, FsCheck will generate random lists of bools, strings, ints, etc.

In both failing cases, the `anyValue` is a bool. So FsCheck is using lists of bools to start with.

Here's an exercise for you: Is this property good enough? Is there some way that the EDFH can create an implementation that will pass?

20.3. "There and back again"

Sometimes the multi-path style properties are not available or too complicated, so let's look at some other approaches. We'll start with properties involving inverses.

Let's start with list sorting again. Is there an inverse to sorting? Hmmm, not really. So we'll skip sorting for now. What about list reversal? Well, as it happens, reversal is its own inverse!

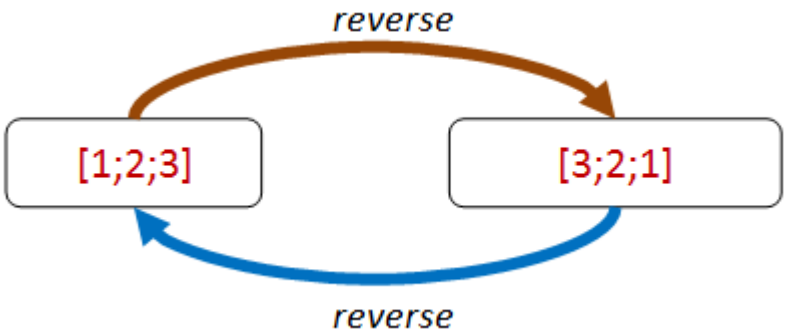

Figure 20.13.: List reverse with inverse

Let's turn that into a property:

```
let ``reverse then reverse should be same as original`` revFn aList =
        let reverseThenReverse = aList |> revFn |> revFn
        reverseThenReverse = aList
```

And it passes:

```
let goodReverse = List.rev
Check.Quick (``reverse then reverse should be same as original`` goodReverse)
// Ok, passed 100 tests.
```

Unfortunately, a bad implementation satisfies the property too!

```
let badReverse aList = aList
Check.Quick (``reverse then reverse should be same as original`` badReverse)
// Ok, passed 100 tests.
```

Nevertheless, the use of properties involving inverses can be very useful to verify that your inverse function (such as deserialization) does indeed "undo" the primary function (such as serialization).

We'll see some real examples of using this in the next post.

20.4. "Hard to prove, easy to verify"

So far we've been testing properties without actually caring about the end result of an operation. But of course in practice, we do care about the end result!

Now we normally can't really tell if the result is right without duplicating the function under test. But often we can tell that the result is *wrong* quite easily. In the maze diagram from above, we can easily check whether the path works or not.

If we are looking for the *shortest* path, we might not be able to check it, but at least we know that we have *some* valid path. This principle can be applied quite generally.

For example, let's say that we want to check whether a `string split` function is working. We don't have to write a tokenizer – all we have to do is ensure that the tokens, when concatenated, give us back the original string!

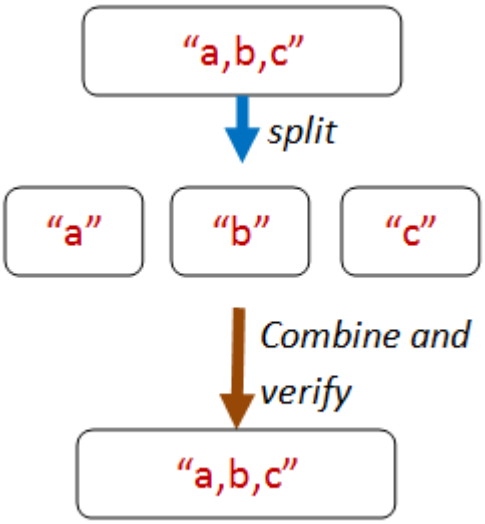

Figure 20.14.: String split property

Here's the core code from that property:

```
let concatWithComma s t = s + "," + t

let tokens = originalString.Split [| ',' |]
let recombinedString =
        // can use reduce safely because there is always at least one token
        tokens |> Array.reduce concatWithComma

// compare the result with the original
originalString = recombinedString
```

But how can we create an original string? The random strings generated by FsCheck are unlikely to contain many commas! There are ways that you can control exactly how FsCheck generates random data, which we'll look at later.

For now though, we'll use a trick. The trick is to let FsCheck generate a list of random strings, and then we'll build an `originalString` from them by concatting them together. So here's the complete code for the property:

```
let ``concatting the elements of a string split by commas recreates the
    original string`` aListOfStrings =
        // helper to make a string
        let addWithComma s t = s + "," + t
        let originalString = aListOfStrings |> List.fold addWithComma ""

        // now for the property
        let tokens = originalString.Split [| ',' |]
        let recombinedString =
            // can use reduce safely because there is always at least
            // one token
            tokens |> Array.reduce addWithComma

        // compare the result with the original
        originalString = recombinedString
```

When we test this we are happy:

```
Check.Quick ``concatting the elements of a string split by commas recreates
    the original string``
// Ok, passed 100 tests.
```

"Hard to prove, easy to verify" for list sorting

So how can we apply this principle to a sorted list? What property is easy to verify?

The first thing that pops into my mind is that for each pair of elements in the list, the first one will be smaller than the second.

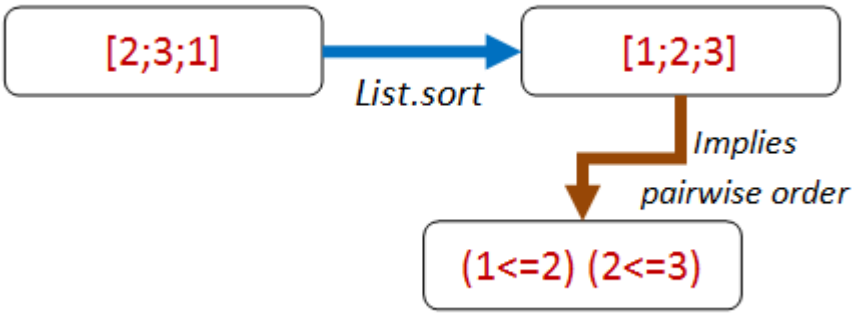

Figure 20.15.: Pairwise property

So let's make that into a property:

```
let ``adjacent pairs from a list should be ordered`` sortFn aList =
        let pairs = aList |> sortFn |> Seq.pairwise
        pairs |> Seq.forall (fun (x,y) -> x <= y )
```

But something funny happens when we try to check it. We get an error!

```
let goodSort = List.sort
Check.Quick (``adjacent pairs from a list should be ordered`` goodSort)
```

```
System.Exception: Geneflect: type not handled System.IComparable
   at FsCheck.ReflectArbitrary.reflectObj@102-4.Invoke(String message)
   at Microsoft.FSharp.Core.PrintfImpl.go@523-3[b,c,d](String fmt,
    Int32 len, FSharpFunc`2 outputChar, FSharpFunc`2 outa, b os,
    FSharpFunc`2 finalize, FSharpList`1 args, Int32 i)
   at Microsoft.FSharp.Core.PrintfImpl.run@521[b,c,d](FSharpFunc`2
    initialize, String fmt, Int32 len, FSharpList`1 args)
```

What does `System.Exception: type not handled System.IComparable` mean? It means that FsCheck is trying to generate a random list, but all it knows is that the elements must be `IComparable`. But `IComparable` is not a type than can be instantiated, so FsCheck throws an error.

How can we prevent this from happening? The solution is to specify a particular type for the property, such as `int list`, like this:

```
let ``adjacent pairs from a list should be ordered`` sortFn
    (aList:int list) =
        let pairs = aList |> sortFn |> Seq.pairwise
        pairs |> Seq.forall (fun (x,y) -> x <= y )
```

This code works now.

```
let goodSort = List.sort
Check.Quick (``adjacent pairs from a list should be ordered`` goodSort)
// Ok, passed 100 tests.
```

Note that even though the property has been constrained, the property is still a very general one. We could have used `string list` instead, for example, and it would work just the same.

```
let ``adjacent pairs from a string list should be ordered`` sortFn
    (aList:string list) =
        let pairs = aList |> sortFn |> Seq.pairwise
        pairs |> Seq.forall (fun (x,y) -> x <= y )

Check.Quick (``adjacent pairs from a string list should be ordered`` goodSort)
// Ok, passed 100 tests.
```

TIP: If FsCheck throws "type not handled", add explicit type constraints to your property

Are we done now? No! One problem with this property is that it doesn't catch malicious implementations by the EDFH.

```
// bad implementation passes
let badSort aList = []
Check.Quick (``adjacent pairs from a list should be ordered`` badSort)
// Ok, passed 100 tests.
```

Is it a surprise to you that a silly implementation also works? Hmmm. That tells us that there must be some property *other than pairwise ordering* associated with sorting that we've overlooked. What are we missing here?

This is a good example of how doing property-based testing can lead to insights about design. We thought we knew what sorting meant, but we're being forced to be a bit stricter in our definition. As it happens, we'll fix this particular problem by using the next principle!

20.5. "Some things never change"

A useful kind of property is based on an invariant that is preserved after some transformation, such as preserving length or contents.

They are not normally sufficient in themselves to ensure a correct implementation, but they *do* often act as a counter-check to more general properties. For example, in the previous post, we created commutative and associative properties for addition, but then noticed that simply having an implementation that returned zero would satisfy them just as well! It was only when we added `x + 0 = x` as a property that we could eliminate that particular malicious implementation.

And in the "list sort" example above, we could satisfy the "pairwise ordered" property with a function that just returned an empty list! How could we fix that? Our first attempt might be to check the length of the sorted list. If the lengths are different, then the sort function obviously cheated!

```
let ``sort should have same length as original`` sortFn (aList:int list) =
        let sorted = aList |> sortFn
        List.length sorted = List.length aList
```

We check it and it works:

```
let goodSort = List.sort
Check.Quick (``sort should have same length as original`` goodSort )
// Ok, passed 100 tests.
```

And yes, the bad implementation fails:

```
let badSort aList = []
Check.Quick (``sort should have same length as original`` badSort )
// Falsifiable, after 1 test (1 shrink)
// [0]
```

Unfortunately, the BDFH is not defeated and can come up with another compliant implementation! Just repeat the first element N times!

```
// bad implementation has same length
let badSort2 aList =
        match aList with
        | [] -> []
        | head::_ -> List.replicate (List.length aList) head

// for example
// badSort2 [1;2;3] => [1;1;1]
```

Now when we test this, it passes:

```
Check.Quick (``sort should have same length as original`` badSort2)
// Ok, passed 100 tests.
```

What's more, it also satisfies the pairwise property too!

```
Check.Quick (``adjacent pairs from a list should be ordered`` badSort2)
// Ok, passed 100 tests.
```

20.5.1. Sort invariant - 2nd attempt

So now we have to try again. What is the difference between the real result `[1;2;3]` and the fake result `[1;1;1]`?

Answer: the fake result is throwing away data. The real result always contains the same contents as the original list, but just in a different order.

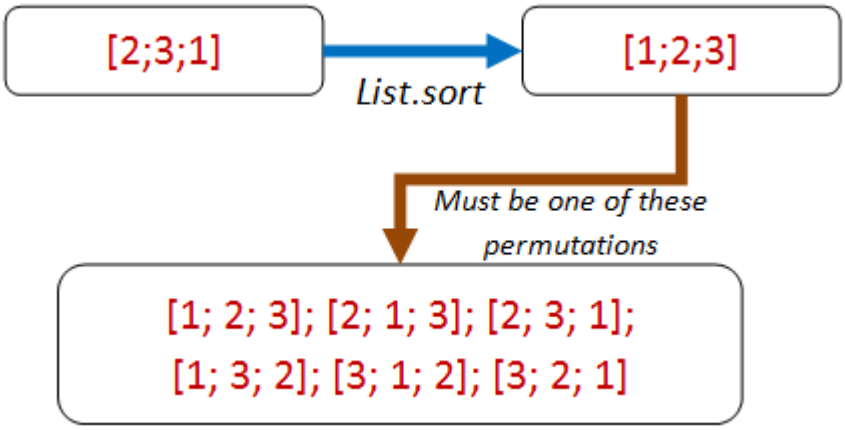

Figure 20.16.: Permutation property

That leads us to a new property: a sorted list is always a permutation of the original list. Aha! Let's write the property in terms of permutations now:

```
let ``a sorted list is always a permutation of the original list``
    sortFn (aList:int list) =
        let sorted = aList |> sortFn
        let permutationsOfOriginalList = permutations aList

        // the sorted list must be in the seq of permutations
        permutationsOfOriginalList
        |> Seq.exists (fun permutation -> permutation = sorted)
```

Great, now all we need is a permutation function. Let's head over to StackOverflow and ~~steal~~ borrow an implementation. Here it is:

```
/// given aList and anElement to insert,
/// generate all possible lists with anElement
/// inserted into aList
let rec insertElement anElement aList =
        // From http://stackoverflow.com/a/4610704/1136133
        seq {
                match aList with
                // empty returns a singleton
                | [] -> yield [anElement]
                // not empty?
                | first::rest ->
                        // return anElement prepended to the list
                        yield anElement::aList
                        // also return first prepended to all the sublists
                        for sublist in insertElement anElement rest do
                                yield first::sublist
                }

/// Given a list, return all permutations of it
let rec permutations aList =
        seq {
                match aList with
                | [] -> yield []
                | first::rest ->
                        // for each sub-permutation,
                        // return the first inserted into it somewhere
                        for sublist in permutations rest do
                                yield! insertElement first sublist
                }
```

Some quick interactive tests confirm that it works as expected:

```
permutations ['a';'b';'c'] |> Seq.toList
// [['a'; 'b'; 'c']; ['b'; 'a'; 'c']; ['b'; 'c'; 'a']; ['a'; 'c'; 'b'];
// ['c'; 'a'; 'b']; ['c'; 'b'; 'a']]

permutations ['a';'b';'c';'d'] |> Seq.toList
// [['a'; 'b'; 'c'; 'd']; ['b'; 'a'; 'c'; 'd']; ['b'; 'c'; 'a'; 'd'];
// ['b'; 'c'; 'd'; 'a']; ['a'; 'c'; 'b'; 'd']; ['c'; 'a'; 'b'; 'd'];
// ['c'; 'b'; 'a'; 'd']; ['c'; 'b'; 'd'; 'a']; ['a'; 'c'; 'd'; 'b'];
// ['c'; 'a'; 'd'; 'b']; ['c'; 'd'; 'a'; 'b']; ['c'; 'd'; 'b'; 'a'];
// ['a'; 'b'; 'd'; 'c']; ['b'; 'a'; 'd'; 'c']; ['b'; 'd'; 'a'; 'c'];
// ['b'; 'd'; 'c'; 'a']; ['a'; 'd'; 'b'; 'c']; ['d'; 'a'; 'b'; 'c'];
// ['d'; 'b'; 'a'; 'c']; ['d'; 'b'; 'c'; 'a']; ['a'; 'd'; 'c'; 'b'];
// ['d'; 'a'; 'c'; 'b']; ['d'; 'c'; 'a'; 'b']; ['d'; 'c'; 'b'; 'a']]
```

```
permutations [3;3] |> Seq.toList
// [[3; 3]; [3; 3]]
```

Excellent! Now let's run FsCheck:

```
Check.Quick (``a sorted list is always a permutation of the original
    list`` goodSort)
```

Hmmm. That's funny, nothing seems to be happening. And my CPU is maxing out for some reason. What's going on?

What's going on is that you are going to be sitting there for a long time! If you are following along at home, I suggest you right-click and cancel the interactive session now. The innocent looking `permutations` is really *really* slow for any normal sized list. For example, a list of just 10 items has 3,628,800 permutations. While with 20 items, you are getting to astronomical numbers. And of course, FsCheck will be doing hundreds of these tests! So this leads to an important tip:

TIP: Make sure your property checks are very fast. You will be running them a LOT!

We've already seen that even in the best case, FsCheck will evaluate the property 100 times. And if shrinking is needed, even more. So make sure your tests are fast to run! But what happens if you are dealing with real systems such as databases, networks, or other slow dependencies?

In his (highly recommended) video on using QuickCheck, John Hughes tells of when his team was trying to detect flaws in a distributed data store that could be caused by network partitions and node failures. Of course, killing real nodes thousands of times was too slow, so they extracted the core logic into a virtual model, and tested that instead. As a result, the code was *later refactored* to make this kind of testing easier. In other words, property-based testing influenced the design of the code, just as TDD would.

20.5.2. Sort invariant - 3rd attempt

Ok, so we can't use permutations by just looping through them. So let's use the same idea but write a function that is specific for this case, a `isPermutationOf` function.

```
let ``a sorted list has same contents as the original list`` sortFn
    (aList:int list) =
        let sorted = aList |> sortFn
        isPermutationOf aList sorted
```

Here's the code for `isPermutationOf` and its associated helper functions:

```
/// Given an element and a list, and other elements previously
/// skipped,
/// return a new list without the specified element.
/// If not found, return None
let rec withoutElementRec anElement aList skipped =
        match aList with
        | [] -> None
        | head::tail when anElement = head ->
```

```
                // matched, so create a new list from the skipped and
                // the remaining
                // and return it
                let skipped' = List.rev skipped
                Some (skipped' @ tail)
        | head::tail  ->
                // no match, so prepend head to the skipped and recurse
                let skipped' = head :: skipped
                withoutElementRec anElement tail skipped'

/// Given an element and a list
/// return a new list without the specified element.
/// If not found, return None
let withoutElement x aList =
        withoutElementRec x aList []

/// Given two lists, return true if they have the same contents
/// regardless of order
let rec isPermutationOf list1 list2 =
        match list1 with
        | [] -> List.isEmpty list2 // if both empty, true
        | h1::t1 ->
                match withoutElement h1 list2 with
                | None -> false
                | Some t2 ->
                        isPermutationOf t1 t2
```

Let's try the test again. And yes, this time it completes before the heat death of the universe.

```
Check.Quick (``a sorted list has same contents as the original list``
    goodSort)
// Ok, passed 100 tests.
```

What's also great is that the malicious implementation now fails to satisfy this property!

```
Check.Quick (``a sorted list has same contents as the original list``
    badSort2)
// Falsifiable, after 2 tests (5 shrinks)
// [1; 0]
```

In fact, these two properties, `adjacent pairs from a list should be ordered` and `a sorted list has same contents as the original list` should indeed ensure that any implementation is correct.

20.6. Sidebar: Combining properties

Just above, we noted that there were *two* properties needed to define the "is sorted" property. It would be nice if we could combine them into one property `is sorted` so that we can have a single test.

Well, of course we can always merge the two sets of code into one function, but it's preferable to keep functions as small as possible. Furthermore, a property like `has same contents` might be reusable in other contexts as well. What we want then, is an equivalent to `AND` and `OR` that is designed to work with properties.

FsCheck to the rescue! There are built in operators to combine properties: `.&.` for `AND` and `.|.` for `OR`. Here is an example of them in use:

```
let ``list is sorted``sortFn (aList:int list) =
        let prop1 = ``adjacent pairs from a list should be ordered`` sortFn
            aList
        let prop2 = ``a sorted list has same contents as the original list``
            sortFn aList
        prop1 .&. prop2
```

When we test the combined property with a good implementation of `sort`, everything works as expected.

```
let goodSort = List.sort
Check.Quick (``list is sorted`` goodSort )
// Ok, passed 100 tests.
```

And if we test a bad implementation, the combined property fails as well.

```
let badSort aList = []
Check.Quick (``list is sorted`` badSort )
// Falsifiable, after 1 test (0 shrinks)
// [0]
```

But there's a problem now. Which of the two properties failed?

What we would like to do is add a "label" to each property so that we can tell them apart. In FsCheck, this is done with the `|@` operator:

```
let ``list is sorted (labelled)``sortFn (aList:int list) =
    let prop1 = ``adjacent pairs from a list should be ordered`` sortFn
            aList
                |@ "adjacent pairs from a list should be ordered"
    let prop2 = ``a sorted list has same contents as the original list``
            sortFn aList
                |@ "a sorted list has same contents as the original list"
    prop1 .&. prop2
```

And now, when we test with the bad sort, we get a message `Label of failing property: a sorted list has same contents as the original list:`

```
Check.Quick (``list is sorted (labelled)`` badSort )
// Falsifiable, after 1 test (2 shrinks)
// Label of failing property: a sorted list has same contents as the
// original list
// [0]
```

For more on these operators, see the FsCheck documentation under "And, Or and Labels".

And now, back to the property-divising strategies.

20.7. "Solving a smaller problem"

Sometimes you have a recursive data structure or a recursive problem. In these cases, you can often find a property that is true of a smaller part.

For example, for a sort, we could say something like:

```
A list is sorted if:
* The first element is smaller (or equal to) the second.
* The rest of the elements after the first element are also sorted.
```

Here is that logic expressed in code:

```
let rec ``First element is <= than second, and tail is also sorted``
        sortFn (aList:int list) =
    let sortedList = aList |> sortFn
    match sortedList with
    | [] -> true
    | [first] -> true
    | [first;second] ->
        first <= second
    | first::second::tail ->
        first <= second &&
        let subList = second::tail
        ``First element is <= than second, and tail is also sorted``
            sortFn subList
```

This property is satisfied by the real sort function:

```
let goodSort = List.sort
Check.Quick (``First element is <= than second, and tail is also sorted``
    goodSort )
// Ok, passed 100 tests.
```

But unfortunately, just like previous examples, the malicious implementations also pass.

```
let badSort aList = []
Check.Quick (``First element is <= than second, and tail is also sorted``
    badSort )
// Ok, passed 100 tests.

let badSort2 aList =
        match aList with
        | [] -> []
        | head::_ -> List.replicate (List.length aList) head

Check.Quick (``First element is <= than second, and tail is also sorted``
    badSort2)
// Ok, passed 100 tests.
```

So as before, we'll need another property (such as the `has same contents` invariant) to ensure that the code is correct.

If you do have a recursive data structure, then try looking for recursive properties. They are pretty obvious and low hanging, when you get the hang of it.

20.8. Is the EDFH really a problem?

In the last few examples, I've noted that trivial but wrong implementations often satisfy the properties as well as good implementations. But should we *really* spend time worrying about this? I mean, if we ever really released a sort algorithm that just duplicated the first element it would be obvious immediately, surely?

So yes, it's true that truly malicious implementations are unlikely to be a problem. On the other hand, you should think of property-based testing not as a *testing* process, but as a *design* process – a technique that helps you clarify what your system is really trying to do. And if a key aspect of your design is satisfied with just a simple implementation, then perhaps there is something you have overlooked – something that, when you discover it, will make your design both clearer and more robust.

20.9. "The more things change, the more they stay the same"

Our next type of property is "idempotence". Idempotence simply means that doing something twice is the same as doing it once. If I tell you to "sit down" and then tell you to "sit down" again, the second command has no effect.

Idempotence is essential for reliable systems and is a key aspect of service oriented and message-based architectures. If you are designing these kinds of real-world systems it is well worth ensuring that your requests and processes are idempotent. I won't go too much into this right now, but let's look at two simple examples.

First, our old friend `sort` is idempotent (ignoring stability) while `reverse` is not, obviously.

```
let ``sorting twice gives the same result as sorting once`` sortFn
    (aList:int list) =
        let sortedOnce = aList |> sortFn
        let sortedTwice = aList |> sortFn |> sortFn
        sortedOnce = sortedTwice

// test
let goodSort = List.sort
Check.Quick (``sorting twice gives the same result as sorting once``
    goodSort )
// Ok, passed 100 tests.
```

In general, any kind of query should be idempotent, or to put it another way: "asking a question should not change the answer". In the real world, this may not be the case. A simple query on a datastore run at different times may give different results.

Here's a quick demonstration. First we'll create a `NonIdempotentService` that gives different results on each query.

```
type NonIdempotentService() =
    let mutable data = 0
    member this.Get() =
        data
```

```
    member this.Set value =
        data <- value

let ``querying NonIdempotentService after update gives the same result``
    value1 value2 =
        let service = NonIdempotentService()
        service.Set value1

        // first GET
        let get1 = service.Get()

        // another task updates the data store
        service.Set value2

        // second GET called just like first time
        let get2 = service.Get()
        get1 = get2
```

But if we test it now, we find that it does not satisfy the required idempotence property:

```
Check.Quick ``querying NonIdempotentService after update gives the same result``
// Falsifiable, after 2 tests
```

As an alternative, we can create a (crude) `IdempotentService` that requires a timestamp for each transaction. In this design, multiple GETs using the same timestamp will always retrieve the same data.

```
type IdempotentService() =
    let mutable data = Map.empty
    member this.GetAsOf (dt:DateTime) =
        data |> Map.find dt
    member this.SetAsOf (dt:DateTime) value =
        data <- data |> Map.add dt value

let ``querying IdempotentService after update gives the same result``
    value1 value2 =
        let service = IdempotentService()
        let dt1 = DateTime.Now.AddMinutes(-1.0)
        service.SetAsOf dt1 value1

        // first GET
        let get1 = service.GetAsOf dt1

        // another task updates the data store
        let dt2 = DateTime.Now
        service.SetAsOf dt2 value2

        // second GET called just like first time
        let get2 = service.GetAsOf dt1
        get1 = get2
```

And this one works:

```
Check.Quick ``querying IdempotentService after update gives the same result``
// Ok, passed 100 tests.
```

So, if you are building a REST GET handler or a database query service, and you want idempotence, you should consider using techniques such as etags, "as-of" times, date ranges, etc. If you need tips on how to do this, searching for idempotency patterns will turn up some good results.

20.10. "Two heads are better than one"

And finally, last but not least, we come to the "test oracle". A test oracle is simply an alternative implementation that gives the right answer, and that you can check your results against.

Often the test oracle implementation is not suitable for production – it's too slow, or it doesn't parallelize, or it's too poetic, etc., but that doesn't stop it being very useful for testing. So for "list sort", there are many simple but slow implementations around. For example, here's a quick implementation of insertion sort:

```
module InsertionSort =

        // Insert a new element into a list by looping over the list.
        // As soon as you find a larger element, insert in front of it
        let rec insert newElem list =
                match list with
                | head::tail when newElem > head ->
                        head :: insert newElem tail
                | other -> // including empty list
                        newElem :: other

        // Sorts a list by inserting the head into the rest of the list
        // after the rest have been sorted
        let rec sort list =
                match list with
                | []   -> []
                | head::tail ->
                        insert head (sort tail)

        // test
        // insertionSort [5;3;2;1;1]
```

With this in place, we can write a property that tests the result against insertion sort.

```
let ``sort should give same result as insertion sort`` sortFn (aList:int list) =
        let sorted1 = aList |> sortFn
        let sorted2 = aList |> InsertionSort.sort
        sorted1 = sorted2
```

When we test the good sort, it works. Good!

```
let goodSort = List.sort
Check.Quick (``sort should give same result as insertion sort`` goodSort)
// Ok, passed 100 tests.
```

And when we test a bad sort, it doesn't. Even better!

```
let badSort aList = aList
Check.Quick (``sort should give same result as insertion sort`` badSort)
// Falsifiable, after 4 tests (6 shrinks)
// [1; 0]
```

20.11. Generating Roman numerals in two different ways

We can also use the test oracle approach to cross-check two different implementations when you're not sure that *either* implementation is right!

For example, in my post "Commentary on 'Roman Numerals Kata with Commentary'" I came up with two completely different algorithms for generating Roman Numerals. Can we compare them to each other and test them both in one fell swoop? The first algorithm was based on understanding that Roman numerals were based on tallying, leading to this simple code:

```
let arabicToRomanUsingTallying arabic =
   (String.replicate arabic "I")
        .Replace("IIIII","V")
        .Replace("VV","X")
        .Replace("XXXXX","L")
        .Replace("LL","C")
        .Replace("CCCCC","D")
        .Replace("DD","M")
        // optional substitutions
        .Replace("IIII","IV")
        .Replace("VIV","IX")
        .Replace("XXXX","XL")
        .Replace("LXL","XC")
        .Replace("CCCC","CD")
        .Replace("DCD","CM")
```

Another way to think about Roman numerals is to imagine an abacus. Each wire has four "unit" beads and one "five" bead. This leads to the so-called "bi-quinary" approach:

```
let biQuinaryDigits place (unit,five,ten) arabic =
  let digit =  arabic % (10*place) / place
  match digit with
  | 0 -> ""
  | 1 -> unit
  | 2 -> unit + unit
```

```
  | 3 -> unit + unit + unit
  | 4 -> unit + five // changed to be one less than five
  | 5 -> five
  | 6 -> five + unit
  | 7 -> five + unit + unit
  | 8 -> five + unit + unit + unit
  | 9 -> unit + ten  // changed to be one less than ten
  | _ -> failwith "Expected 0-9 only"

let arabicToRomanUsingBiQuinary arabic =
  let units = biQuinaryDigits 1 ("I","V","X") arabic
  let tens = biQuinaryDigits 10 ("X","L","C") arabic
  let hundreds = biQuinaryDigits 100 ("C","D","M") arabic
  let thousands = biQuinaryDigits 1000 ("M","?","?") arabic
  thousands + hundreds + tens + units
```

We now have two completely different algorithms, and we can cross-check them with each other to see if they give the same result.

```
let ``biquinary should give same result as tallying`` arabic =
        let tallyResult = arabicToRomanUsingTallying arabic
        let biquinaryResult = arabicToRomanUsingBiQuinary arabic
        tallyResult = biquinaryResult
```

But if we try running this code, we get a `ArgumentException: The input must be non-negative` due to the `String.replicate` call.

```
Check.Quick ``biquinary should give same result as tallying``
// ArgumentException: The input must be non-negative.
```

So we need to only include inputs that are positive. We also need to exclude numbers that are greater than 4000, say, since the algorithms break down there too. How can we implement this filter?

We saw in the previous post that we could use preconditions. But for this example, we'll try something different and change the generator. First we'll define a *new* arbitrary integer called `arabicNumber` which is filtered as we want (an "arbitrary" is a combination of a generator algorithm and a shrinker algorithm, as described in the previous post).

```
let arabicNumber = Arb.Default.Int32() |> Arb.filter (fun i -> i > 0
    && i <= 4000)
```

Next, we create a new property *which is constrained to only use "arabicNumber"* by using the `Prop.forAll` helper.

We'll give the property the rather clever name of "for all values of arabicNumber, biquinary should give same result as tallying".

```
let ``for all values of arabicNumber biquinary should give same result
    as tallying`` =
        Prop.forAll arabicNumber ``biquinary should give same result as
            tallying``
```

Now finally, we can do the cross-check test:

```
Check.Quick ``for all values of arabicNumber biquinary should give same
    result as tallying``
// Ok, passed 100 tests.
```

And we're good! Both algorithms work correctly, it seems.

20.12. "Model-based" testing

"Model-based" testing, which we will discuss in more detail in a later post, is a variant on having a test oracle. The way it works is that, in parallel with your (complex) system under test, you create a simplified model. Then, when you do something to the system under test, you do the same (but simplified) thing to your model. At the end, you compare your model's state with the state of the system under test. If they are the same, you're done. If not, either your SUT is buggy or your model is wrong and you have to start over!

20.13. Interlude: A game based on finding properties

With that, we have come to the end of the various property categories. We'll go over them one more time in a minute – but first, an interlude.

If you sometimes feel that trying to find properties is a mental challenge, you're not alone. Would it help to pretend that it is a game? As it happens, there *is* a game based on property-based testing. It's called Zendo and it involves placing sets of objects (such as plastic pyramids) on a table, such that each layout conforms to a pattern – a rule – or as we would now say, *a property*!.

The other players then have to guess what the rule (property) is, based on what they can see. Here's a picture of a Zendo game in progress (figure 20.17).

The white stones mean the property has been satisfied, while black stones mean failure. Can you guess the rule here? I'm going to guess that it's something like "a set must have a yellow pyramid that's not touching the ground".

Alright, I suppose Zendo wasn't really inspired by property-based testing, but it is a fun game, and it has even been known to make an appearance at programming conferences. If you want to learn more about Zendo, the rules are here.

20.14. Applying the categories one more time

With all these categories in hand, let's look at one more example problem, and see if we can find properties for it. This sample is based on the well-known `Dollar` example described in Kent Beck's "TDD By Example" book.

Nat Pryce, of *Growing Object-Oriented Software Guided by Tests* fame, wrote a blog post about property-based testing a while ago ("Exploring Test-Driven Development with QuickCheck"). In it, he expressed some frustration about property-based testing being useful in practice. So let's revisit the example he referenced and see what we can do with it.

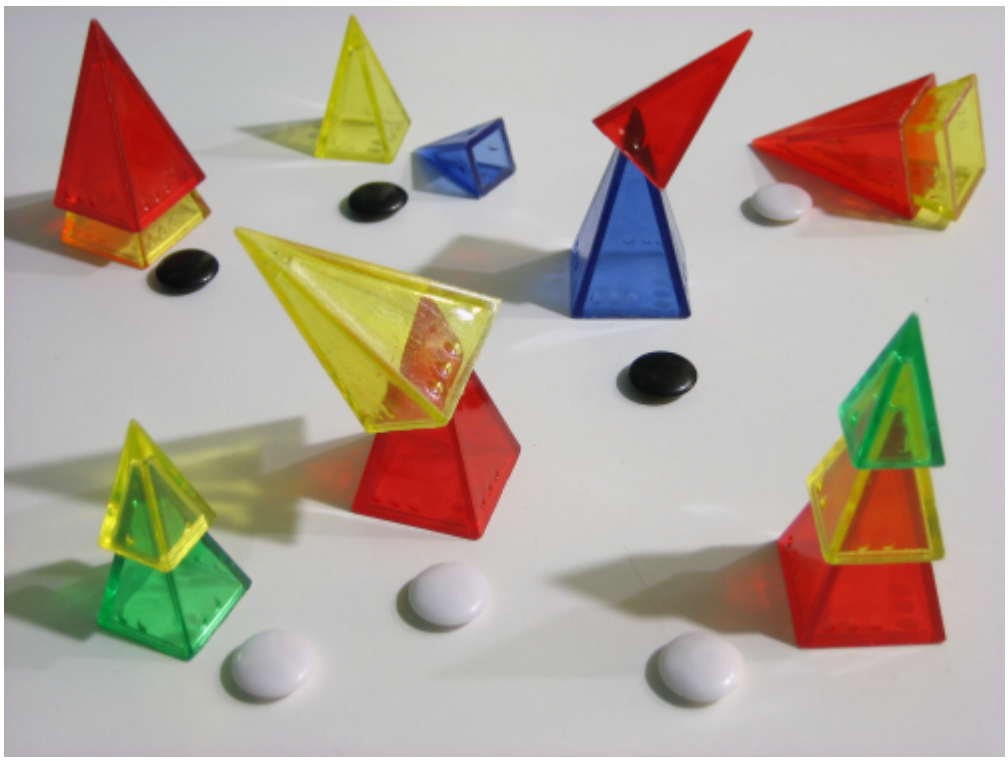

Figure 20.17.: Zendo

We're not going to attempt to critique the design itself and make it more type-driven – others have done that. Instead, we'll take the design as given and see what properties we can come up with. So what do we have?

- A `Dollar` class that stores an `Amount`.
- Methods `Add` and `Times` that transform the amount in the obvious way.

```
// OO style class with members
type Dollar(amount:int) =
        member val Amount  = amount with get, set
        member this.Add add =
                this.Amount <- this.Amount + add
        member this.Times multiplier  =
                this.Amount <- this.Amount * multiplier
        static member Create amount  =
                Dollar amount
```

So, first let's try it out interactively to make sure it works as expected:

```
let d = Dollar.Create 2
d.Amount  // 2
d.Times 3
d.Amount  // 6
d.Add 1
d.Amount  // 7
```

But that's just playing around, not real testing. So what kind of properties can we think of? Let's run through them all again:

- Different paths to same result
- Inverses
- Invariants
- Idempotence
- Structural induction
- Easy to verify
- Test oracle

Let's skip the "different paths" one for now. What about inverses? Are there any inverses we can use? Yes, the setter and getter form an inverse that we can create a property from:

```
let ``set then get should give same result`` value =
        let obj = Dollar.Create 0
        obj.Amount <- value
        let newValue = obj.Amount
        value = newValue

Check.Quick ``set then get should give same result``
// Ok, passed 100 tests.
```

Idempotence is relevant too. For example, doing two sets in a row should be the same as doing just one. Here's a property for that:

```
let ``set amount is idempotent`` value =
        let obj = Dollar.Create 0
        obj.Amount <- value
        let afterFirstSet = obj.Amount
        obj.Amount <- value
        let afterSecondSet = obj.Amount
        afterFirstSet = afterSecondSet

Check.Quick ``set amount is idempotent``
// Ok, passed 100 tests.
```

Any "structural induction" properties? No, not relevant to this case. Any "easy to verify" properties? Not anything obvious. Finally, is there a test oracle? No. Again not relevant, although if we really were designing a complex currency management system, it might be very useful to cross-check our results with a third party system.

20.14.1. Properties for an immutable Dollar

A confession! I cheated a bit in the code above and created a mutable class, which is how most OO objects are designed.

But in "TDD by Example" , Kent quickly realizes the problems with that and changes it to an immutable class, so let me do the same. Here's the immutable version:

```
type Dollar(amount:int) =
        member val Amount  = amount
        member this.Add add =
                Dollar (amount + add)
        member this.Times multiplier  =
                Dollar (amount * multiplier)
        static member Create amount  =
                Dollar amount

// interactive test
let d1 = Dollar.Create 2
d1.Amount  // 2
let d2 = d1.Times 3
d2.Amount  // 6
let d3 = d2.Add 1
d3.Amount  // 7
```

What's nice about immutable code is that we can eliminate the need for testing of setters, so the two properties we just created have now become irrelevant! To tell the truth they were pretty trivial anyway, so it's no great loss.

So then, what new properties can we devise now? Let's look at the `Times` method. How can we test that? Which one of the strategies can we use? I think the "different paths to same result" is very applicable. We can do the same thing we did with "sort" and do a times operation both "inside" and "outside" and see if they give the same result.

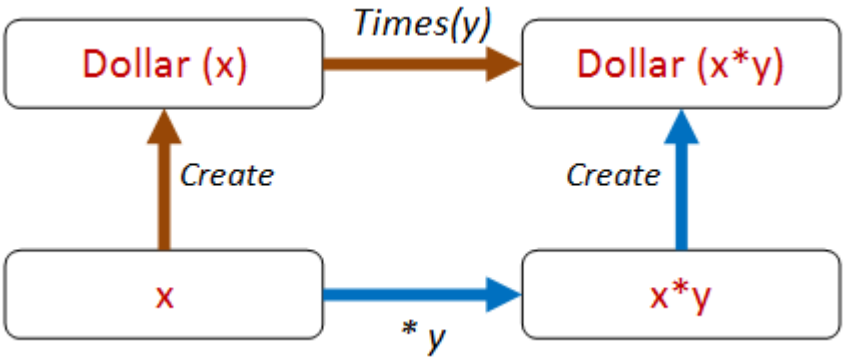

Figure 20.18.: Dollar times

Here's that property expressed in code:

```
let ``create then times should be same as times then create`` start
    multiplier =
```

```
        let d0 = Dollar.Create start
        let d1 = d0.Times(multiplier)
        let d2 = Dollar.Create (start * multiplier)
        d1 = d2
```

Great! Let's see if it works!

```
Check.Quick ``create then times should be same as times then create``
// Falsifiable, after 1 test
```

Oops – it doesn't work! Why not? Because we forgot that `Dollar` is a reference type and doesn't compare equal by default! As a result of this mistake, we have discovered a property that we might have overlooked! Let's encode that before we forget.

```
let ``dollars with same amount must be equal`` amount =
        let d1 = Dollar.Create amount
        let d2 = Dollar.Create amount
        d1 = d2

Check.Quick ``dollars with same amount must be equal``
// Falsifiable, after 1 test
```

So now we need to fix this by adding support for `IEquatable` and so on. You can do that if you like – I'm going to switch to F# record types and get equality for free!

20.14.2. Dollar properties – version 3

Here's the `Dollar` rewritten again:

```
type Dollar = {amount:int }
        with
        member this.Add add =
                {amount = this.amount + add }
        member this.Times multiplier  =
                {amount = this.amount * multiplier }
        static member Create amount  =
                {amount=amount}
```

And now our two properties are satisfied:

```
Check.Quick ``dollars with same amount must be equal``
// Ok, passed 100 tests.

Check.Quick ``create then times should be same as times then create``
// Ok, passed 100 tests.
```

We can extend this approach for different paths. For example, we can extract the amount and compare it directly, like this: The code looks like this:

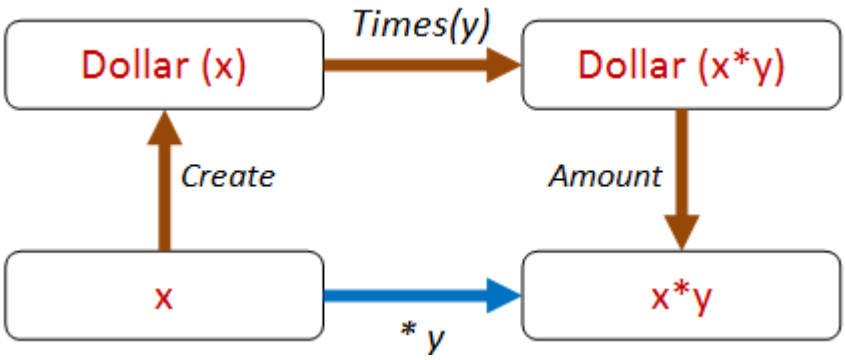

Figure 20.19.: Dollar times

```
let ``create then times then get should be same as times`` start multiplier =
        let d0 = Dollar.Create start
        let d1 = d0.Times(multiplier)
        let a1 = d1.amount
        let a2 = start * multiplier
        a1 = a2

Check.Quick ``create then times then get should be same as times``
// Ok, passed 100 tests.
```

And we can also include `Add` in the mix as well.

For example, we can do a `Times` followed by an `Add` via two different paths, like this:

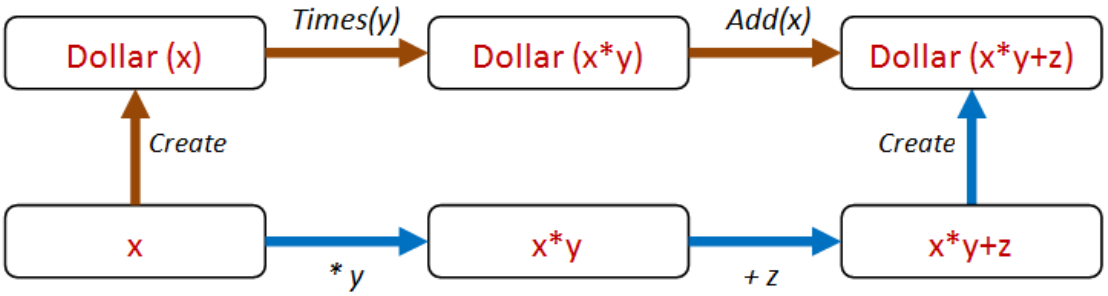

Figure 20.20.: Dollar times

And here's the code:

```
let ``create then times then add should be same as times then add then
    create`` start multiplier adder =
        let d0 = Dollar.Create start
        let d1 = d0.Times(multiplier)
        let d2 = d1.Add(adder)
        let directAmount = (start * multiplier) + adder
        let d3 = Dollar.Create directAmount
        d2 = d3
```

```
Check.Quick ``create then times then add should be same as times then
    add then create``
// Ok, passed 100 tests.
```

So this "different paths, same result" approach is very fruitful, and we can generate *lots* of paths this way.

20.14.3. Dollar properties – version 4

Shall we call it done then? I would say not! We are beginning to get a whiff of a code smell. All this `(start * multiplier) + adder` code seems like a bit of duplicated logic, and could end up being brittle.

Can we abstract out some commonality that is present all these cases? If we think about it, our logic is *really* just this:

- Transform the amount on the "inside" in some way.
- Transform the amount on the "outside" in the same way.
- Make sure that the results are the same.

But to test this, the `Dollar` class is going to have to support an arbitrary transform! Let's call it `Map`! Now all our tests can be reduced to this one property:

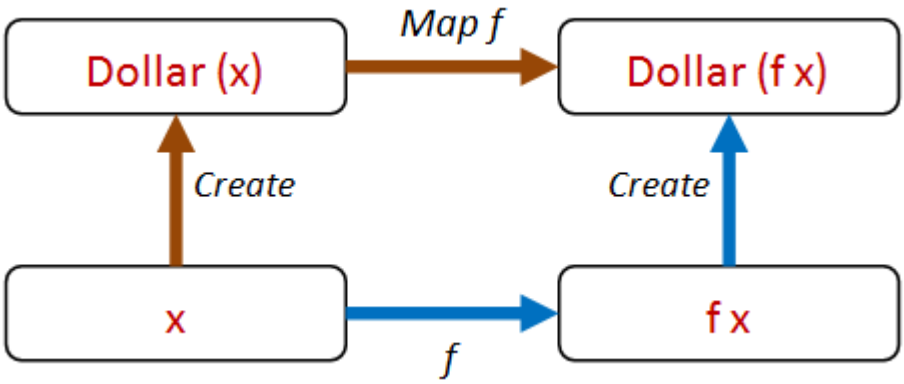

Figure 20.21.: Dollar map

Let's add a `Map` method to `Dollar`. And we can also rewrite `Times` and `Add` in terms of `Map`:

```
type Dollar = {amount:int }
        with
        member this.Map f =
                {amount = f this.amount}
        member this.Times multiplier =
                this.Map (fun a -> a * multiplier)
        member this.Add adder =
                this.Map (fun a -> a + adder)
        static member Create amount  =
                {amount=amount}
```

Now the code for our property looks like this:

```
let ``create then map should be same as map then create`` start f =
        let d0 = Dollar.Create start
        let d1 = d0.Map f
        let d2 = Dollar.Create (f start)
        d1 = d2
```

But how can we test it now? What functions should we pass in? Don't worry! FsCheck has you covered! In cases like this, FsCheck will actually generate random functions for you too! Try it – it just works!

```
Check.Quick ``create then map should be same as map then create``
// Ok, passed 100 tests.
```

Our new "map" property is much more general than the original property using "times", so we can eliminate the latter safely.

20.14.4. Logging the function parameter

There's a little problem with the property as it stands. If you want to see what the function is that FsCheck is generating, then Verbose mode is not helpful.

```
Check.Verbose ``create then map should be same as map then create``
```

Gives the output:

```
0:
18
<fun:Invoke@3000>
1:
7
<fun:Invoke@3000>
-- etc
98:
47
<fun:Invoke@3000>
99:
36
<fun:Invoke@3000>
Ok, passed 100 tests.
```

We can't tell what the function values actually were.

However, you can tell FsCheck to show more useful information by wrapping your function in a special `F` case, like this:

```
let ``create then map should be same as map then create2`` start (F (_,f)) =
        let d0 = Dollar.Create start
        let d1 = d0.Map f
        let d2 = Dollar.Create (f start)
        d1 = d2
```

And now when you use Verbose mode...

```
Check.Verbose ``create then map should be same as map then create2``
```

... you get a detailed log of each function that was used:

```
0:
0
{ 0->1 }
1:
0
{ 0->0 }
2:
2
{ 2->-2 }
-- etc
98:
-5
{ -5->-52 }
99:
10
{ 10->28 }
Ok, passed 100 tests.
```

Each `{ 2->-2 }`, `{ 10->28 }`, etc., represents the function that was used for that iteration.

20.15. TDD vs. property-based testing

How does property-based testing (PBT) fit in with TDD? This is a common question, so let me quickly give you my take on it.

First off, TDD works with *specific examples*, while PBT works with *universal properties*.

As I said in the previous post, I think examples are useful as a way into a design, and can be a form of documentation. But in my opinion, relying *only* on example-based tests would be a mistake.

Property-based approaches have a number of advantages over example-based tests:

- Property-based tests are more general, and thus are less brittle.
- Property-based tests provide a better and more concise description of requirements than a bunch of examples.
- As a consequence, one property-based test can replace many, many, example-based tests.
- By generating random input, property-based tests often reveal issues that you have overlooked, such as dealing with nulls, missing data, divide by zero, negative numbers, etc.
- Property-based tests force you to think.

- Property-based tests force you to have a clean design.

These last two points are the most important for me. Programming is not a matter of writing lines of code, it is about creating a design that meets the requirements. So, anything that helps you think deeply about the requirements and what can go wrong should be a key tool in your personal toolbox!

For example, in the Roman Numeral section, we saw that accepting `int` was a bad idea (the code broke!). We had a quick fix, but really we should model the concept of a `PositiveInteger` in our domain, and then change our code to use that type rather than just an `int`. This demonstrates how using PBT can actually improve your domain model, not just find bugs.

Similarly, introducing a `Map` method in the Dollar scenario not only made testing easier, but actually improved the usefulness of the Dollar "api". Stepping back to look at the big picture, though, TDD and property-based testing are not at all in conflict. They share the same goal of building correct programs, and both are really more about design than coding (think "Test-driven *design*" rather than "Test-driven *development*").

20.16. The end, at last

So that brings us to the end of another long post on property-based testing! I hope that you now have some useful approaches that you can take away and apply to your own code base. Next time, we'll look at some real-world examples, and how you can create custom generators that match your domain.

The code samples used in this post are available on GitHub.

21. Finding Property Tests - Hillel Wayne

William Yao: *More starting points for figuring out what properties to test.*

Original article: [20]

A while back I was ranting about APLs and included this python code to get the mode of a list:

```
def mode(l):
  max = None
  count = {}
  for x in l:
    if x not in count:
      count[x] = 0
    count[x] += 1
    if not max or count[x] > count[max]:
      max = x
  return max
```

There's a bug in it. Do you see it? If not, try running it on the list `[0, 0, 1]`:

```
>>> mode([0, 0, 1])
1
```

The issue is that 0 is falsy, so if max is 0, `if not max` is true.

I could say this bug was a result of carelessness. I didn't write any tests for this function, just tried a few obvious examples and thought "yeah it works". But I don't think writing bespoke manual unit tests would have caught this. To surface the bug, the mode of a list must be a falsy value like 0 or `[]` *and* the last value of the list must be something else. It's a small intersection of Python's typing and the mechanics of `mode`, making it too unusual a case to be found by standard unit testing practice.

A different testing style is property based testing (PBT) with contracts. By generating a random set of inputs, we cover more of the state space than we'd do manually. The problem with PBT, though, is that it can be hard to find good properties. I'd like to take mode as a case study in what properties we could come up with. There are a few things I'm looking for:

- The property test should find the bug.
- The test should be *simple*. I'm presumably not putting a lot of effort into testing such a simple function, so a complex test doesn't accurately capture what I'd do.

- The test should be *obvious*. I'm looking for a natural test that finds the bug, not a post-hoc one. Catching a bug with tests is much less believable if you already know what you're looking for.

So, let's talk some tests and contracts! I'm using hypothesis for the property tests, dpcontracts for my contracts library, pytest for the runner[1]. For the sake of this problem, assume we're only passing in nonempty lists.

```
@require("l must not be empty", lambda args: len(args.l) > 0)
def mode(l):
  ...
```

21.1. Contract-wise

One property of a function is "all of the contracts are satisfied". We can use this to write "thin" tests, where we don't put any assertions in the test itself. If any of our contracts raise an error then the test will fail.

```
@given(lists(integers(), min_size=1))
def test_mode(l):
    mode(l)
```

21.1.1. Types

Typically in dynamic programming languages, contracts are used as a poor replacement of a static type system. Instead of checking the type at compile time, people check the type at runtime. Most contract libraries are heavily geared towards this kind of use.

```
@ensure("result is an int", lambda _, r: isinstance(r, int))
```

This is a bad contract for three reasons:

1. It requires the function to return integers when it's currently generic. We *could* try to make it generic by doing something like `type(a.l) == type(r)`, but *ugh*.
2. We should be using mypy for type checks anyway.
3. It doesn't actually find the error. The problem isn't the type, it's that we got the wrong result.

1 Oddly enough I'm getting increasingly less sold on using pytest, purely because I want to experiment with weird janky metaprogramming in my tests and pytest doesn't really support that.

Sanity Checking

We can go further than replicating static types. One common type of contract is a "sanity check": some property that does not fully specify our code, but is easy to express and should hold true anyway. For example, we know the mode will be an element of the list, so why not check that we're returning an element of the list?

```
@ensure("result is in list", lambda a, r: r in a.l)
```

This is a pretty good contract! It tells us useful things about the function, and it's not easily replacable with a typecheck. If I was writing production code I'd probably write a lot of contracts like this. But it also doesn't find the problem, so we need to go further.

21.1.2. First element

Our sanity check was only minimally related to our function. There are lots of functions that return elements in the list: `head`, `random_element`, `last`, etc. The issue is a subtle bug in our implementation. Our contract should express some important property about our function. In mode's case, it should relate to the count of the value.

One *extremely* useful property is adding bounds. The mode of a list is the element that occurs most frequently. Every element of the list should occur less often, or as often, as the mode[2]. One good arbitrary element is the first element:

```
@ensure("result > arbitrary",
  lambda a, r: a.l.count(r) >= a.l.count(a.l[0]))
```

This finds the bug!

```
args = ([0, 0, 1],), kwargs = {}, rargs = Args(l=[0, 0, 1])
result = 1
E           dpcontracts.PostconditionError: result > arbitrary
```

Personally, I'd prefer this as a property test clause instead of a contract clause. It doesn't feel "right" to me. I think it's more an aesthetic judgment than a technical one here.

```
@given(lists(integers(), min_size=1))
def test_mode(l):
    x = mode(l)
    assert l.count(x) >= l.count(l[0])
```

Either way, this is only a partial contract: while it will catch *some* incorrect outputs, it won't catch them all. We could get this with `[1, 0, 0, 2]`: `count(0) > count(2) >= count(1)`, but our broken function would return 2. In some cases, this is all we can feasibly get. For simpler functions, though, we can rule out all incorrect outputs. We want a total contract, one which always raises on an incorrect output and never raises on a correct one.

2 We have to say "less than or *equal* to" for two reasons. First, the mode is not strictly more frequent than other elements, like in `[1, 1, 2, 2]`. Second, what if the mode *is* `l[0]`?

21.1.3. The dang definition

Why not just use the definition itself?

```
@ensure("result is the mode",
  lambda a, r: all((a.l.count(r) >= a.l.count(x) for x in a.l)))
```

To make this nicer we can extract this into a dedicated helper contract:

```
def is_mode(l, m):
    return all((l.count(m) >= l.count(x) for x in l))

@ensure("result is the mode", lambda a, r: is_mode(a.l, r))
```

This also catches the error.

```
args = ([0, 0, 1],), kwargs = {}, rargs = Args(l=[0, 0, 1])
result = 1
```

This is the same faulty input as before. Property-based Testing libraries **shrink** inputs to find the smallest possible error, which is `[0, 0, 1]`.

Compare to an Oracle

If we had another way of getting the answer that we knew was correct, we could just compare the two results and see if they're the same. This is called using an **oracle**. Oracles are often a good choice when you're trying to refactor a complex function or optimize an expensive one. For our purposes, it goes too far.

```
from collections import Counter

def math_mode(l):
    c = Counter(l)
    return c.most_common(1)[0][0]

@require("l must not be empty", lambda args: len(args.l) > 0)
@ensure("result matches oracle", lambda a, r: r == math_mode(a.l))
def mode(l):
  ...
```

This is too heavy. Not only is it cumbersome, but it overconstrains what the mode can be. We see this in the error it finds: it finds an error with a smaller input than the other two!

```
args = ([0, 1],), kwargs = {}, rargs = Args(l=[0, 1]), result = 1
```

We haven't precisely defined the semantics of `mode`. If there are two values which tie for the most elements, which is the mode? Our prior contracts didn't say: as long as we picked an element that had at least as many instances as any other element, we were good. With `math_mode`, we're arbitrarily choosing one of them as the "real" mode and checking that our `mode` picked that arbitrary element. We can see this better by writing a manual test:

```
def test_mode():
    mode([3, 2, 2, 3])

...

args = ([3, 2, 2, 3],), kwargs = {}, rargs = Args(l=[3, 2, 2, 3])
result = 2
```

Whereas with our previous contract passes on this.

21.2. Property-wise

Our contract approach converged on "testing the definition" as the best result. There are many cases where code-under-test does not have a nice mathematical definition. Contracts are still useful here, as they can rule out bad cases, but you'll need additional tests.

Hypothetically contracts can express all possible properties of a function. In practice you're limited to what your framework can express and check. For most complicated properties we're better off sticking it in a dedicated test.

"Property-wise" property tests have several advantages over "contract-wise" property tests:

1. We can test properties that aren't "ergonomic" in our contract framework.
2. We can test properties that involve effects.
3. We can test metamorphic relations which involve comparing multiple function calls.

For this `mode` problem we don't need any of them, but let's show off some possible tactics anyway.

21.2.1. Preserving Transformation

We find some transformation of the input that should give the same output. For lists, a good transformation is sorting[3]. The mode of a list doesn't change if you sort it:

```
@given(lists(integers(), min_size=1))
def test_sorting_preserves_mode(l):
    assert mode(l) == mode(sorted(l))

...

l = [0, 0, -1]
```

We could also reverse the list instead of sort it, but that gives us an error case of `[1, -1]`, which again is due to overconstraints.

Or we could assert that the mode doesn't change if we add it again to the list:

3 Unless you're trying to test a sorting function.

```
def test_can_add_to_mode(l):
    m = mode(l)
    assert mode(l + [m]) == m
```

This does *not* find the bug, though.

21.2.2. Controlled Transformation

Instead of finding a solution that doesn't change the answer, we could find one that changes it in a known way. One of them might be "doubling all of the numbers doubles the mode":

```
@given(lists(integers(), min_size=1))
def test_doubling_doubles_mode(l):
    doubled = [x * 2 for x in l]
    assert 2*mode(l) == mode(doubled)
```

This does not find the bug. We could also try "adding 1 to every element adds 1 to the mode":

```
@given(lists(integers(), min_size=1))
def test_incrementing_increments_mode(l):
    incremented = [x + 1 for x in l]
    assert mode(l)+1 == mode(incremented)

...

l = [0, 1]
```

It gives the same output as in our overconstrained case, but it only "is wrong" when we have 0's anyway. If we restrict the list to only positive integers, it will pass (unlike our oracle contract).

If we wanted to be extra thorough we could generatively pick both a slope and an increment:

```
@given(lists(integers(), min_size=1), integers())
def test_affine_relation(l, m, b):
    transformed = [m*x+b for x in l]
    assert  m*mode(l)+b == mode(transformed)

...

l = [0, 1], m = 1, b = 1
```

It depends on how paranoid you want to get.

21.2.3. Oracle Generators

The big advantage of manual tests to generative ones is that you can come up with the appropriate outputs for a given input. Since we can't easily do that in PBT, we're stuck testing properties instead of oracles.

Or we could go in reverse: take a random output and generate a corresponding input. One way we can do that:

1. generate pairs of elements and counts. Make sure that the elements are unique
2. construct a list from that
3. pass in both the list and the corresponding mode, selected from the pair.

```
@composite
def list_and_mode(draw):
    out = []
    pairs_max_10 = tuples(integers(), integers(min_value=1, max_value=10))
    counts = draw(lists(pairs_max_10,
        min_size=1,
        max_size=5,
        unique_by= lambda x: x[0]))
    for number, count in counts:
        out += ([number] * count)
    mode_of_out = max(counts, key=lambda x: x[1])[0]
    return out, mode_of_out

@given(list_and_mode())
def test_can_find_mode(lm):
    l, m = lm
    assert mode(l) == m

...

lm = ([0, 1], 0)
```

This overconstrains (we're not ruling out two pairs having the same counts), but it *does not* raise a false positive for `[3, 2, 2, 3]`. This is because we construct the list in the same way `max` interprets the list. If we do

```
-   for number, count in counts:
+   for number, count in reversed(counts):
```

Then it raises `[3, 2]` as a "counterexample". Between the cumbersomeness and the overconstraining, making an oracle generator is not a good choice for this problem. There are some cases, though, where it can be more useful.

21.3. Limitations

Here's a fixed version that *looks* like it will work:

```
def mode(l):
  max = None
  count = {}
  for x in l:
    if x not in count:
      count[x] = 0
    count[x] += 1
+   if max is None or count[x] > count[max]:
-   if not max or count[x] > count[max]:
      max = x
  return max
```

And this passes all of our tests. But there's still a bug in it. Again, take a minute to see if you can find it. If you can't, try the following:

```
mode([None, None, 2])
```

This will select the mode as 2, when it really should be `None`. The problem isn't in our contracts or assertions. It's in our test generator: we're only testing with lists of integers. Hypothesis can generate heterogenous lists, but you still have to explicitly list the types you want to be in the list. In order to find this bug we'd have to explicitly realize that `None` might be a problem for us.

If we only want to call `mode` on homogenous lists, we should instead use a type-checker to catch the bug:

```
+ def mode(l: List[T]) -> T:
- def mode(l):
    max = None
+   count = {} # type: Dict[T, int]
-   count = {}
```

This will raise a spurious error, saying that the return value is actually an `Optional[T]`. If we change `max = None` to `max = l[0]` both the error and the bug go away. But we can change the return value to `Optional[T]` and the bug remains- mypy can't actually detect if we're passing in a heterogenous list. More type-oriented languages can ban heterogenous lists outright but even those will miss the bugs our contracts caught. Static and dynamic analysis are complementary, not contradictory[4].

21.3.1. Summary

This was a pretty short dive into what makes a good property or contract. It also focused on just pure functions: a lot of languages use contracts to maintain class invariants or monitor the side effects of procedures.

If you're interested in learning more about properties, chapter 20 is a canonical article on abstract properties and chapter 16 is a series on applying it to business

4 Both contracts and properties can be checked statically, but *most* people using them will be checking them at runtime. This is because static analysis of contracts quickly turns into formal verification, which is really, really hard.

problems. If you're interested in learning more about contracts, I'd recommend... actually, I can't think of anything that's not language-specific. Kind of surprising given how useful they are.

22. Using types to unit-test in Haskell - Alexis King

> ***William Yao:*** *Introduces a way of doing more "traditional" unit testing for code with side effects, using mocks and hand-written tests.*
>
> *Original article: [21]*

Object-oriented programming languages make unit testing easy by providing obvious boundaries between units of code in the form of classes and interfaces. These boundaries make it easy to stub out parts of a system to test functionality in isolation, which makes it possible to write fast, deterministic test suites that are robust in the face of change. When writing Haskell, it can be unclear how to accomplish the same goals: even inside pure code, it can become difficult to test a particular code path without also testing all its collaborators.

Fortunately, by taking advantage of Haskell's expressive type system, it's possible to not only achieve parity with object-oriented testing techniques, but also to provide stronger static guarantees as well. Furthermore, it's all possible without resorting to extra-linguistic hacks that static object-oriented languages sometimes use for mocking, such as dynamic bytecode generation.

22.1. First, an aside on testing philosophy

Testing methodology is a controversial topic within the larger programming community, and there are a multitude of different approaches. This blog post is about *unit testing*, an already nebulous term with a number of different definitions. For the purposes of this post, I will define a unit test as a test that stubs out collaborators of the code under test in some way. Accomplishing that in Haskell is what this is primarily about.

I want to be clear that I do not think that unit tests are the only way to write tests, nor the best way, nor even always an applicable way. Depending on your domain, rigorous unit testing might not even make sense, and other forms of tests (end-to-end, integration, benchmarks, etc.) might fulfill your needs.

In practice, though, implementing those other kinds of tests seems to be well-documented in Haskell compared to pure, object-oriented style unit testing. As my Haskell applications have grown, I have found myself wanting a more fine-grained testing tool that allows me to both test a piece of my codebase in isolation and also use my domain-specific types. This blog post is about that.

With that disclaimer out of the way, let's talk about testing in Haskell.

22.2. Drawing seams using types

One of the primary attributes of unit tests in object-oriented languages, especially statically-typed ones, is the concept of "seams" within a codebase. These are internal boundaries between components of a system. Some boundaries are obvious—interactions with a database, manipulation of the file system, and performing I/O over the network, to name a few examples—but others are more subtle. Especially in larger codebases, it can be helpful to isolate two related but distinct pieces of functionality as much as possible, which makes them easier to reason about, even if they're actually part of the same codebase.

In OO languages, these seams are often marked using interfaces, whether explicitly (in the case of static languages) or implicitly (in the case of dynamic ones). By programming to an interface, it's possible to create "fake" implementations of that interface for use in unit tests, effectively making it possible to stub out code that isn't directly relevant to the code being tested.

In Haskell, representing these seams is a lot less obvious. Consider a fairly trivial function that reverses a file's contents on the file system:

```
reverseFile :: FilePath -> IO ()
reverseFile path = do
  contents <- readFile path
  writeFile path (reverse contents)
```

This function is impossible to test without testing against a real file system. It simply performs I/O directly, and there's no way to "mock out" the file system for testing purposes. Now, admittedly, this function is so trivial that a unit test might not seem worth the cost, but consider a slightly more complicated function that interacts with a database:

```
renderUserProfile :: Id User -> IO HTML
renderUserProfile userId = do
  user <- fetchUser userId
  posts <- fetchRecentPosts userId

  return $ div
    [ h1 (userName user <> "'s Profile")
    , h2 "Recent Posts"
    , ul (map (li . postTitle) posts)
    ]
```

It might now be a bit more clear that it could be useful to test the above function without running a real database and doing all the necessary context setup before each test case. Indeed, it would be nice if a test could just provide stubbed implementations for `fetchUser` and `fetchRecentPosts`, then make assertions about the output.

One way to solve this problem is to pass the results of those two functions to `renderUserProfile` as arguments, turning it into a pure function that could be easily tested. This becomes obnoxious for functions of even just slightly more complexity, though (it is not unreasonable to imagine needing a handful of different queries to render a user's profile page), and it requires significantly restructuring code simply because the tests need it.

The above code is not only difficult to test, however—it has another problem, too. Specifically, both functions return `IO` values, which means they can effectively do *anything*. Haskell has a very strong type system for typing terms, but it doesn't provide any guarantees about effects beyond a simple yes/no answer about function purity. Even though the `renderUserProfile` function should really only need to interact with the database, it could theoretically delete files, send emails, make HTTP requests, or do any number of other things.

Fortunately, it's possible to solve *both* problems—a lack of testability and a lack of type safety—using the same general technique. This approach is reminiscent of the interface-based seams of object-oriented languages, but unlike most object-oriented approaches, it provides additional type safety guarantees without the need to explicitly modify the code to support some kind of dependency injection.

22.2.1. Making implicit interfaces explicit

Statically typed, object-oriented languages provide interfaces as a language construct to encode certain kinds of contracts into the type system, and Haskell has something similar. Typeclasses are, in many ways, an analog to OO interfaces, and they can be used in a similar way. In the above case, let's write down interfaces that the `reverseFile` and `renderUserProfile` functions can use:

```
class Monad m => MonadFS m where
  readFile :: FilePath -> m String
  writeFile :: FilePath -> String -> m ()

class Monad m => MonadDB m where
  fetchUser :: Id User -> m User
  fetchRecentPosts :: Id User -> m [Post]
```

The really nice thing about these interfaces is that our function implementations don't have to change *at all* to take advantage of them. In fact, all we have to change is their types:

```
reverseFile :: MonadFS m => FilePath -> m ()
reverseFile path = do
  contents <- readFile path
  writeFile path (reverse contents)

renderUserProfile :: MonadDB m => Id User -> m HTML
renderUserProfile userId = do
  user <- fetchUser userId
  posts <- fetchRecentPosts userId

  return $ div
    [ h1 (userName user <> "'s Profile")
    , h2 "Recent Posts"
    , ul (map (li . postTitle) posts)
    ]
```

This is pretty neat, since we haven't had to alter our code at all, but we've managed to completely decouple ourselves from `IO`. This has the direct effect of both making our code more abstract (we no longer rely on the "real" file system or a "real" database, which makes our code easier to test) and restricting what our functions can do (just from looking at the type signatures, we know what side-effects they can perform).

Of course, since we're now coding against an interface, our code doesn't actually do much of anything. If we want to actually use the functions we've written, we'll have to define instances of `MonadFS` and `MonadDB`. When actually running our code, we'll probably still use `IO` (or some monad transformer stack with `IO` at the bottom), so we can define trivial instances for that existing use case:

```
instance MonadFS IO where
  readFile = Prelude.readFile
  writeFile = Prelude.writeFile

instance MonadDB IO where
  fetchUser = SQL.fetchUser
  fetchRecentPosts = SQL.fetchRecentPosts
```

Even if we go no further, **this is already incredibly useful**. By restricting the sorts of effects our functions can perform at the type level, it becomes a lot easier to see which code is interacting with what. This can be invaluable when working in a part of a moderately large codebase that you are unfamiliar with. Even if the only instance of these typeclasses is `IO`, the benefits are immediately apparent.

Of course, this blog post is about testing, so we're going to go further and take advantage of these seams we've now drawn. The question is: how?

22.3. Testing with typeclasses: an initial attempt

Given that we now have functions depending on an interface instead of `IO`, we can create separate instances of our typeclasses for use in tests. Let's start with the `renderUserProfile` function. We'll create a simple wrapper around the `Identity` type, since we don't actually care much about the "effects" of our `MonadDB` methods:

```
import Data.Functor.Identity

newtype TestM a = TestM (Identity a)
  deriving (Functor, Applicative, Monad)

unTestM :: TestM a -> a
unTestM (TestM (Identity x)) = x
```

Now, we'll create a trivial instance of `MonadDB` for `TestM`:

```
instance MonadDB TestM where
  fetchUser _ = return User { userName = "Alyssa" }
  fetchRecentPosts _ = return
    [ Post { postTitle = "Metacircular Evaluator" } ]
```

With this instance, it's now possible to write a simple unit test of the `renderUserProfile` function that doesn't need a real database running at all:

```
spec = describe "renderUserProfile" $ do
  it "shows the user's name" $ do
    let result = unTestM (renderUserProfile (intToId 1234))
    result `shouldContainElement` h1 "Alyssa's Profile"

  it "shows a list of the user's posts" $ do
    let result = unTestM (renderUserProfile (intToId 1234))
    result `shouldContainElement` ul [ li "Metacircular Evaluator" ]
```

This is pretty nice, and running the above tests reveals a nice property of these kinds of isolated test cases: the test suite runs *really, really fast*. Communicating with a database, even in extremely simple ways, takes a measurable amount of time, especially with dozens of tests. In contrast, even with hundreds of tests, our unit test suite runs in less than a tenth of a second.

This all seems to be successful, so let's try and apply the same testing technique to `reverseFile`.

22.3.1. Testing side-effectful code

Looking at the type signature for `reverseFile`, we have a small problem:

```
reverseFile :: MonadFS m => FilePath -> m ()
```

Specifically, the return type is `()`. Making any assertions against the result of this function would be completely worthless, given that it's guaranteed to be the same exact thing each time. Instead, `reverseFile` is inherently side-effectful, so we want to be able to test that it properly interacts with the file system in the correct way.

In order to do this, a simple wrapper around `Identity` won't be enough, but we can replace it with something more powerful: `Writer`. Specifically, we can use a writer monad to "log" what gets called in order to test side-effects. We'll start by creating a new `TestM` type, just like last time:

```
newtype TestM a = TestM (Writer [String] a)
  deriving (Functor, Applicative, Monad, MonadWriter [String])

logTestM :: TestM a -> [String]
logTestM (TestM w) = execWriter w
```

Using this slightly more powerful type, we can write a useful instance of `MonadFS` that will track the argument given to `writeFile`:

```
instance MonadFS TestM where
  readFile _ = return "hello"
  writeFile _ contents = tell [contents]
```

Again, the instance is quite simple, but it now enables us to write a straightforward unit test for `reverseFile`:

```
spec = describe "reverseFile" $
  it "reverses a file's contents on the filesystem" $ do
    let calls = logTestM (reverseFile "foo.txt")
    calls `shouldBe` ["olleh"]
```

Again, quite simple to both implement and use, and the test itself is blindingly fast. There's another problem, though, which is that we have technically left part of `reverseFile` untested: we've completely ignored the `path` argument.

In this contrived example, it may seem silly to test something so trivial, but in real code, it's quite possible that one would care very much about testing multiple different aspects about a single function. When testing `renderUserProfile`, this was not hard, since we could reuse the same `TestM` type and `MonadDB` instance for both test cases, but in the `reverseFile` example, we've ignored the path entirely.

We *could* adjust our `MonadFS` instance to also track the path provided to each method, but this has a few problems. First, it means every test case would depend on all the various properties we are testing, which would mean updating every test case when we add a new one. It would also be simply impossible if we needed to track multiple types—in this particular case, it turns out that `String` and `FilePath` are actually the same type, but in practice, there may be a handful of disparate, incompatible types.

Both of the above issues could be fixed by creating a sum type and manually filtering out the relevant elements in each test case, but a much more intuitive approach would be to simply have a separate instance for each case. Unfortunately, in Haskell, creating a new instance means creating an entirely new type. To illustrate how much duplication that would entail, we could create the following type and instance for testing proper propagation of the `path` argument:

```
newtype TestM' a = TestM' (Writer [FilePath] a)
  deriving (Functor, Applicative, Monad, MonadWriter [FilePath])

logTestM' :: TestM' a -> [FilePath]
logTestM' (TestM' w) = execWriter w

instance MonadFS TestM' where
  readFile path = tell [path] >> return ""
  writeFile path _ = tell [path]
```

Now it's possible to add an extra test case that asserts that the proper path is provided to the two filesystem functions:

```
spec = describe "reverseFile" $ do
  it "reverses a file's contents on the filesystem" $ do
    let calls = logTestM (reverseFile "foo.txt")
    calls `shouldBe` ["olleh"]

  it "operates on the file at the provided path" $ do
    let paths = logTestM' (reverseFile "foo.txt")
    paths `shouldBe` ["foo.txt", "foo.txt"]
```

This works, but it's ultimately unacceptably complicated. Our test harness code is now significantly larger than the actual tests themselves, and the amount of boilerplate is frustrating. Verbose test suites are especially bad, since forcing programmers

to jump through hoops just to implement a single test reduces the likelihood that people will actually write good tests, if they write tests at all. In contrast, if writing tests is easy, then people will naturally write more of them.

The above strategy to writing tests is not good enough, but it does reveal a particular problem: in Haskell, typeclass instances are not first-class values that can be manipulated and abstracted over, they are static constructs that can only be managed by the compiler, and users do not have a direct way to modify them. With some cleverness, however, we can actually create an approximation of first-class typeclass dictionaries, which will allow us to dramatically simplify the above testing mechanism.

22.4. Creating first-class typeclass instances

In order to provide an easy way to construct instances, we need a way to represent instances as ordinary Haskell values. This is not terribly difficult, given that instances are conceptually just records containing a collection of functions. For example, we could create a datatype that represents an instance of the `MonadFS` typeclass:

```
data MonadFSInst m = MonadFSInst
  { _readFile :: FilePath -> m String
  , _writeFile :: FilePath -> String -> m ()
  }
```

To avoid namespace clashes with the actual method identifiers, the record fields are prefixed with an underscore, but otherwise, the translation is remarkably straightforward. Using this record type, we can easily create values that represent the two instances we defined above:

```
contentInst :: MonadWriter [String] m => MonadFSInst m
contentInst = MonadFSInst
  { _readFile = \_ -> return "hello"
  , _writeFile = \_ contents -> tell [contents]
  }

pathInst :: MonadWriter [FilePath] m => MonadFSInst m
pathInst = MonadFSInst
  { _readFile = \path -> tell [path] >> return ""
  , _writeFile = \path _ -> tell [path]
  }
```

These two values represent two different implementations of `MonadFS`, but since they're ordinary Haskell values, they can be manipulated and even *extended* like any other records. This can be extremely useful, since it makes it possible to create a sort of "base" instance, then have individual test cases override individual pieces of functionality piecemeal.

Of course, although we've written these two instances, we have no way to actually use them. After all, Haskell does not provide a way to explicitly provide typeclass dictionaries. Fortunately, we can create a sort of "proxy" type that will use a reader to thread the dictionary around explicitly, and the instance can defer to the dictionary's implementation.

22.4.1. Creating an instance proxy

To represent our proxy type, we'll use a combination of a `Writer` and a `ReaderT`; the former to implement the logging used by instances, and the latter to actually thread around the dictionary. Our type will look like this:

```
newtype TestM log a =
    TestM (ReaderT (MonadFSInst (TestM log)) (Writer log) a)
  deriving ( Functor, Applicative, Monad
           , MonadReader (MonadFSInst (TestM log))
           , MonadWriter log
           )

logTestM :: MonadFSInst (TestM log) -> TestM log a -> log
logTestM inst (TestM m) = execWriter (runReaderT m inst)
```

This might look rather complicated, and it is, but let's break down exactly what it's doing.

1. The `TestM` type includes two type parameters. The first is the type of value that will be logged (hence the name `log`), which corresponds to the argument to `Writer` from previous incarnations of `TestM`. Unlike those types, though, we want this version to work with any `Monoid`, so we'll make it a type parameter. The second parameter is simply the type of the current monadic value, as before.

2. The type itself is defined as a wrapper around a small monad transformer stack, the first of which is `ReaderT`. The state threaded around by the reader is, in this case, the instance dictionary, which is `MonadFSInst`.

   However, recall that `MonadFSInst` accepts a type variable—the type of a monad itself—so we must provide `TestM log` as an argument to `MonadFSInst`. This slight bit of indirection allows us to tie the knot between the mutually dependent instances and proxy type.

3. The base monad in the transformer stack is `Writer`, which is used to actually implement the logging functionality, just like in prior cases. The only difference now is that the `log` type parameter now determines what the writer actually produces.

4. Finally, as before, we use `GeneralizedNewtypeDeriving` to derive all the relevant `mtl` classes, adding the somewhat wordy `MonadReader` constraint to the list.

Using this single type, we can now implement a `MonadFS` instance that defers to the dictionary carried around within `TestM`'s reader state:

```
instance Monoid log => MonadFS (TestM log) where
  readFile path = do
    f <- asks _readFile
    f path
  writeFile path contents = do
    f <- asks _writeFile
    f path contents
```

This may seem somewhat boilerplate-y, and it is to some extent, but the important consideration is that this boilerplate only needs to be written *once*. With this in place, it's now possible to write an arbitrary number of first-class instances that use the above mechanism without extending the mechanism at all.

To see what actually using this code would look like, let's update the `reverseFile` tests to use the new `TestM` implementation, as well as the `contentInst` and `pathInst` dictionaries from earlier:

```
spec = describe "reverseFile" $ do
  it "reverses a file's contents on the filesystem" $ do
    let calls = logTestM contentInst (reverseFile "foo.txt")
    calls `shouldBe` ["olleh"]

  it "operates on the file at the provided path" $ do
    let paths = logTestM pathInst (reverseFile "foo.txt")
    paths `shouldBe` ["foo.txt", "foo.txt"]
```

We can do a little bit better, though. Really, the definitions of `contentInst` and `pathInst` are specific to each test case. With ordinary typeclass instances, we cannot scope them to any particular block, but since `MonadFSInst` is just an ordinary Haskell datatype, we can manipulate them just like any other Haskell values. Therefore, we can just inline those instances' definitions into the test cases themselves to keep them closer to the actual tests.

```
spec = describe "reverseFile" $ do
  it "reverses a file's contents on the filesystem" $ do
    let contentInst = MonadFSInst
          { _readFile = \_ -> return "hello"
          , _writeFile = \_ contents -> tell [contents]
          }
    let calls = logTestM contentInst (reverseFile "foo.txt")
    calls `shouldBe` ["olleh"]

  it "operates on the file at the provided path" $ do
    let pathInst = MonadFSInst
          { _readFile = \path -> tell [path] >> return ""
          , _writeFile = \path _ -> tell [path]
          }
    let paths = logTestM pathInst (reverseFile "foo.txt")
    paths `shouldBe` ["foo.txt", "foo.txt"]
```

This is pretty good. We're now able to create inline instances of our `MonadFS` typeclass, which allows us to write extremely concise unit tests using ordinary Haskell typeclasses as system seams. We've managed to cut down on the boilerplate considerably, though we still have a couple problems. For one, this example only uses a single typeclass containing only two methods. A real `MonadFS` typeclass would likely have at least a dozen methods for performing various filesystem operations, and writing out the instance dictionaries for every single method, even the ones that aren't used within the code under test, would be pretty frustratingly verbose.

This problem is solvable, though. Since instances are just ordinary Haskell records, we can create a "base" instance that just throws an exception whenever the method is called:

```
baseInst :: MonadFSInst m
baseInst = MonadFSInst
  { _readFile = error "unimplemented instance method `_readFile'"
  , _writeFile = error "unimplemented instance method `_writeFile'"
  }
```

Then code that only uses `readFile` could only override that particular method, for example:

```
let myInst = baseInst { _readFile = ... }
```

Normally, of course, this would be a terrible idea. However, since this is all just test code, it can be extremely useful in quickly figuring out what methods need to be stubbed out for a particular test case. Since all the code actually gets run at test time, attempts to use unimplemented instance methods will immediately raise an error, informing the programmer which methods need to be implemented to make the test pass. This can also help to significantly cut down on the amount of effort it takes to implement each test.

Another problem is that our approach is specialized exclusively to `MonadFS`. What about functions that use both `MonadFS` *and* `MonadDB`, for example? Fortunately, that is not hard to solve, either. We can adapt the `MonadFSInst` type to include fields for all of the typeclasses relevant to our system, turning it into a generic test fixture of sorts:

```
data FixtureInst m = FixtureInst
  { -- MonadFS
    _readFile :: FilePath -> m String
  , _writeFile :: FilePath -> String -> m ()

    -- MonadDB
  , _fetchUser :: Id User -> m User
  , _fetchRecentPosts :: Id User -> m [Post]
  }
```

Updating `TestM` to use `FixtureInst` instead of `MonadFSInst` is trivial, and all the rest of the infrastructure still works. However, this means that every time a new typeclass is added, three things need to be updated:

1. Its methods need to be added to the `FixtureInst` record.
2. Those methods need to be given error-raising defaults in the `baseInst` value.
3. An actual instance of the typeclass needs to be written for `TestM` that defers to the `FixtureInst` value.

Furthermore, most of this manual manipulation of methods is required every time a particular typeclass changes, whether that means adding a method, removing a

method, renaming a method, or changing a method's type. This is especially frustrating given that all this code is really just mechanical boilerplate that could all be derived by the set of typeclasses being tested.

That last point is especially important: aside from the instances themselves, every piece of boilerplate above is obviously possible to generate from existing types alone. With that piece of information in mind, we can do even better: we can use Template Haskell.

22.5. Removing the boilerplate using `test-fixture`

The above code was not only rather boilerplate-heavy, it was pretty complicated. Fortunately, you don't actually have to write it. Enter the library `test-fixture`:

```
import Control.Monad.TestFixture
import Control.Monad.TestFixture.TH

mkFixture "FixtureInst" [''MonadFS, ''MonadDB]

spec = describe "reverseFile" $ do
  it "reverses a file's contents on the filesystem" $ do
    let contentInst = def
          { _readFile = \_ -> return "hello"
          , _writeFile = \_ contents -> log contents
          }
    let calls = logTestFixture (reverseFile "foo.txt") contentInst
    calls `shouldBe` ["olleh"]

  it "operates on the file at the provided path" $ do
    let pathInst = def
          { _readFile = \path -> log path >> return ""
          , _writeFile = \path _ -> log path
          }
    let paths = logTestFixture (reverseFile "foo.txt") pathInst
    paths `shouldBe` ["foo.txt", "foo.txt"]
```

That's it. The above code automatically generates everything you need to write fast, simple, deterministic unit tests in Haskell. The `mkFixture` function is a Template Haskell macro that expands into a definition quite similar to the `FixtureInst` type we wrote by hand, but since it's automatically generated from the typeclass definitions, it never needs to be updated.

The `logTestFixture` function replaces the `logTestM` function we wrote by hand, but it works exactly the same. The `Control.Monad.TestFixture` library also exports a `log` function that is a synonym for `tell . singleton`, but using `tell` directly still works if you prefer.

The `mkFixture` function also generates a `Default` instance, which replaces the `baseInst` value defined earlier. It functions the same way, though, producing useful error messages that refer to the names of unimplemented typeclass methods that have not been stubbed out.

This blog post is not a `test-fixture` tutorial—indeed, it is much more complicated than a `test-fixture` tutorial would be, since it covers what the library is really doing under the hood—but if you're interested, I would highly recommend you take a look at the `test-fixture` documentation on Hackage.

22.6. Conclusion, credits, and similar techniques

This blog post came about as the result of a need my coworkers and I found when writing Haskell code; we wanted a way to write unit tests quickly and easily, but we didn't find much advice from the rest of the Haskell ecosystem. The `test-fixture` library is the result of that exploratory work, and we currently use it to test a significant portion of our Haskell code.

It would be extremely unfair to suggest that I was the inventor of this technique or the inventor of the library. Two of my coworkers, Joe Vargas and Greg Wiley, came up with the general approach and wrote `Control.Monad.TestFixture`, and I simply wrote the Template Haskell macro to eliminate the boilerplate. With that in mind, I think I can say with some fairness that I think this technique is a joy to use when unit testing is a desirable goal, and I would definitely recommend it if you are interested in doing isolated testing in Haskell.

The general technique of using typeclasses to emulate effects was in part inspired by the well-known `mtl` library. An alternate approach to writing unit-testable Haskell code is using free monads, but overall, I prefer this approach over free monads because the typeclass constraints add type safety in ways that free monads do not (at least not without additional boilerplate), and this approach also lends itself well to static analysis-based boilerplate reduction techniques. It has its own tradeoffs, though, so if you've had success with free monads, then I certainly make no claim this is a superior approach, just one that I've personally found pleasant.

As a final note, if you *do* check out `test-fixture`, feel free to leave feedback by opening issues on the GitHub issue tracker—even things like confusing documentation are worth a bug report.

23. Time Travelling and Fixing Bugs with Property-Based Testing - Oskar Wickström

> ***William Yao:*** *Another large-ish case study of using property-based testing, this time introducing a technique to help ensure that you're genuinely testing enough of your code's input space.*
>
> *Original article: [22]*

Property-based testing (PBT) is a powerful testing technique that helps us find edge cases and bugs in our software. A challenge in applying PBT in practice is coming up with useful properties. This tutorial is based on a simple but realistic system under test (SUT), aiming to show some ways you can test and find bugs in such logic using PBT. It covers refactoring, dealing with non-determinism, testing generators themselves, number of examples to run, and coupling between tests and implementation. The code is written in Haskell and the testing framework used is Hedgehog.

This tutorial was originally written as a book chapter, and later extracted as a standalone piece. Since I'm not expecting to finish the PBT book any time soon, I decided to publish the chapter here.

23.1. System Under Test: User Signup Validation

The business logic we'll test is the validation of a website's user signup form. The website requires users to sign up before using the service. When signing up, a user must pick a valid username. Users must be between 18 and 150 years old. Stated formally, the validation rules are:

$$\begin{aligned} 0 < \text{length}(\text{name}) \leq 50 \\ 18 < \text{age} \leq 150 \end{aligned} \tag{1}$$

The signup and its validation is already implemented by previous programmers. There have been user reports of strange behaviour, and we're going to locate and fix the bugs using property tests. Poking around the codebase, we find the data type representing the form:

```
data SignupForm = SignupForm
  { formName  :: Text
  , formAge   :: Int
  } deriving (Eq, Show)
```

And the existing validation logic, defined as `validateSignup`. We won't dig into to the implementation yet, only its type signature:

```
validateSignup
  :: SignupForm -> Validation (NonEmpty SignupError) Signup
```

It's a pure function, taking `SignupForm` data as an argument, and returning a `Validation` value. In case the form data is valid, it returns a `Signup` data structure. This data type resembles `SignupForm` in its structure, but refines the age as a `Natural` when valid:

```
data Signup = Signup
  { name  :: Text
  , age   :: Natural
  } deriving (Eq, Show)
```

In case the form data is invalid, `validateSignup` returns a non-empty list of `SignupError` values. `SignupError` is a union type of the possible validation errors:

```
data SignupError
  = NameTooShort Text
  | NameTooLong Text
  | InvalidAge Int
  deriving (Eq, Show)
```

23.1.1. The Validation Type

The `Validation` type comes from the validation package. It's parameterized by two types:

1. the type of validation failures
2. the type of a successfully validated value

The `Validation` type is similar to the Either type. The major difference is that it *accumulates* failures, rather than short-circuiting on the first failure. Failures are accumulated when combining multiple `Validation` values using `Applicative`.

Using a non-empty list for failures in the `Validation` type is common practice. It means that if the validation fails, there's at least one error value.

23.2. Validation Property Tests

Let's add some property tests for the form validation, and explore the existing implementation. We begin in a new test module, and we'll need a few imports:

```
import           Data.List.NonEmpty (NonEmpty (..))
import           Data.Text          (Text)
import           Data.Validation
import           Hedgehog
import qualified Hedgehog.Gen       as Gen
import qualified Hedgehog.Range     as Range
```

Also, we'll need to import the implementation module:

```
import Validation
```

We're now ready to define some property tests.

23.2.1. A Positive Property Test

The first property test we'll add is a *positive* test. That is, a test using only valid input data. This way, we know the form validation should always be successful. We define `prop_valid_signup_form_succeeds`:

```
prop_valid_signup_form_succeeds = property $ do
  let genForm = SignupForm <$> validName <*> validAge  -- (1)
  form <- forAll genForm                               -- (2)

  case validateSignup form of                          -- (3)
    Success{}        -> pure ()
    Failure failure' -> do
      annotateShow failure'
      failure
```

First, we define `genForm` (1), a generator producing form data with valid names and ages. Next, we generate `form` values from our defined generator (2). Finally, we apply the `validateSignup` function and pattern match on the result (3):

- In case it's successful, we have the test pass with `pure ()`
- In case it fails, we print the `failure'` and fail the test

The `validName` and `validAge` generators are defined as follows:

```
validName :: Gen Text
validName = Gen.text (Range.linear 1 50) Gen.alphaNum

validAge :: Gen Int
validAge = Gen.integral (Range.linear 1 150)
```

Recall the validation rules (eq. 1). The ranges in these generators yielding valid form data are defined precisely in terms of the validation rules.

The character generator used for names is `alphaNum`, meaning we'll only generate names with alphabetic letters and numbers. If you're comfortable with regular expressions, you can think of `genValidName` as producing values matching `[a-zA-Z0-9]+`. Let's run some tests:

```
λ> check prop_valid_signup_form_succeeds
     ✓<interactive> passed 100 tests.
```

Hooray, it works.

23.2.2. Negative Property Tests

In addition to the positive test, we'll add *negative* tests for the name and age, respectively. Opposite to positive tests, our negative tests will only use invalid input data. We can then expect the form validation to always fail.

First, let's test invalid names.

```
prop_invalid_name_fails = property $ do
  let genForm = SignupForm <$> invalidName <*> validAge -- (1)
  form <- forAll genForm

  case validateSignup form of                          -- (2)
    Failure (NameTooLong{}  :| []) -> pure ()
    Failure (NameTooShort{} :| []) -> pure ()
    other                          -> do               -- (3)
      annotateShow other
      failure
```

Similar to our the positive property test, we define a generator `genForm` (1). Note that we use `invalidName` instead of `validName`. Again, we pattern match on the result of applying `validateSignup` (2). In this case we expect failure. Both `NameTooLong` and `NameTooShort` are expected failures. If we get anything else, the test fails (3).

The test for invalid age is similar, expect we use the `invalidAge` generator, and expect only `InvalidAge` validation failures:

```
prop_invalid_age_fails = property $ do
  let genForm = SignupForm <$> validName <*> invalidAge
  form <- forAll genForm
  case validateSignup form of
    Failure (InvalidAge{} :| []) -> pure ()
    other                        -> do
      annotateShow other
      failure
```

The `invalidName` and `invalidAge` generators are also defined in terms of the validation rules (eq. 1), but with ranges ensuring no overlap with valid data:

```
invalidName :: Gen Text
invalidName =
  Gen.choice [mempty, Gen.text (Range.linear 51 100) Gen.alphaNum]

invalidAge :: Gen Int
invalidAge = Gen.integral (Range.linear minBound 0)
```

Let's run our new property tests:

```
λ> check prop_invalid_name_fails
     ✓<interactive> passed 100 tests.

λ> check prop_invalid_age_fails
     ✓<interactive> passed 100 tests.
```

All good? Maybe not. The astute reader might have noticed a problem with one of our generators. We'll get back to that later.

23.2.3. Accumulating All Failures

When validating the form data, we want *all* failures returned to the user posting the form, rather than returning only one at a time. The `Validation` type accumulates failures when combined with `Applicative`, which is exactly what we want. Yet, while the hard work is handled by `Validation`, we still need to test that we're correctly combining validations in `validateSignup`.

We define a property test generating form data, where all fields are invalid (1). It expects the form validation to fail, returning two failures (2).

```
prop_two_failures_are_returned = property $ do
  let genForm = SignupForm <$> invalidName <*> invalidAge -- (1)
  form <- forAll genForm
  case validateSignup form of
    Failure failures | length failures == 2 -> pure ()     -- (2)
    other -> do
      annotateShow other
      failure
```

This property is weak. It states nothing about *which* failures should be returned. We could assert that the validation failures are equal to some expected list. But how do we know if the name is too long or too short? I'm sure you'd be less thrilled if we replicated all of the validation logic in this test.

Let's define a slightly stronger property. We pattern match, extract the two failures (1), and check that they're not equal (2).

```
prop_two_different_failures_are_returned = property $ do
  let genForm = SignupForm <$> invalidName <*> invalidAge
  form <- forAll genForm
  case validateSignup form of
    Failure (failure1 :| [failure2]) ->              -- (1)
      failure1 /== failure2                          -- (2)
    other                              -> do
      annotateShow other
      failure
```

We're still not being specific about which failures should be returned. But unlike `prop_two_failures_are_returned`, this property at least makes sure there are no duplicate failures.

23.3. The Value of a Property

Is there a faulty behaviour that would slip past `prop_two_different_failures_are_returned`? Sure. The implementation could have a typo or copy-paste error, and always return `NameTooLong` failures, even if the name is too short. Does this mean our property is bad? Broken? Useless? In itself, this property doesn't give us strong confidence in the correctness of `validateSignup`. In conjuction with our other properties, however, it provides value. Together they make up a stronger test suite.

Let's look at it in another way. What are the *benefits* of weaker properties over stronger ones? In general, weak properties are beneficial in that they are:

1. easier to define
2. likely to catch simple mistakes early
3. less coupled to the SUT

A small investment in a set of weak property tests might catch a lot of mistakes. While they won't precisely specify your system and catch the trickiest of edge cases, their power-to-weight ratio is compelling. Moreover, a set of weak properties is better than no properties at all. If you can't formulate the strong property you'd like, instead start simple. Lure out some bugs, and improve the strength and specificity of your properties over time.

Coming up with good properties is a skill. Practice, and you'll get better at it.

23.4. Testing Generators

Remember how in section 23.2.2 we noted that there's a problem? The issue is, we're not covering all validation rules in our tests. But the problem is not in our property definitions. It's in one of our *generators*, namely `genInvalidAge`. We're now in a perculiar situation: we need to test our tests.

One way to test a generator is to define a property specifically testing the values it generates. For example, if we have a generator `positive` that is meant to generate only positive integers, we can define a property that asserts that all generated integers are positive:

```
positive :: Gen Int
positive = Gen.integral (Range.linear 1 maxBound)

prop_integers_are_positive = property $ do
  n <- forAll positive
  assert (n >= 1)
```

We could use this technique to check that all values generated by `validAge` are valid. How about `invalidAge`? Can we check that it generates values such that all boundaries of our validation function are hit? No, not using this technique. Testing the correctness of a generator using a property can only find problems with *individual* generated values. It can't perform assertions over *all* generated values. In that sense, it's a *local* assertion.

Instead, we'll find the generator problem by capturing statistics on the generated values and performing *global* assertions. Hedgehog, and a few other PBT frameworks, can measure the occurences of user-defined *labels*. A label in Hedgehog is a `Text` value, declared with an associated condition. When Hedgehog runs the tests, it records the percentage of tests in which the condition evaluates to `True`. After the test run is complete, we're presented with a listing of percentages per label.

We can even have Hedgehog fail the test unless a certain percentage is met. This way, we can declare mininum coverage requirements for the generators used in our property tests.

23.4.1. Adding Coverage Checks

Let's check that we generate values covering enough cases, based on the validation rules in eq. 1 . In `prop_invalid_age_fails`, we use `cover` to ensure we generate values outside the boundaries of valid ages. 5% is enough for each, but realistically they could both get close to 50%.

```
prop_invalid_age_fails = property $ do
  let genForm = SignupForm <$> validName <*> invalidAge
  form <- forAll genForm
  cover 5 "too young" (formAge form <= 0)
  cover 5 "too old"   (formAge form >= 151)
  case validateSignup form of
    Failure (InvalidAge{} :| []) -> pure ()
    other                        -> do
      annotateShow other
      failure
```

Let's run some tests again.

```
λ> check prop_invalid_age_fails
  ✗ <interactive> failed
    after 100 tests.
    too young 100% ████████████████████ ✓ 5%
  ⚠ too old     0% ···················· ✗ 5%

    ┏━━ test/Validation/V1Test.hs ━━━
 63 ┃ prop_invalid_age_fails = property $ do
 64 ┃   let genForm = SignupForm <$> validName <*> invalidAge
 65 ┃   form <- forAll genForm
 66 ┃   cover 5 "too young" (formAge form <= 0)
 67 ┃   cover 5 "too old"   (formAge form >= 151)
    ┃   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    ┃   | Failed (0% coverage)
 68 ┃   case validateSignup form of
 69 ┃     Failure (InvalidAge{} :| []) -> pure ()
 70 ┃     other                        -> do
 71 ┃       annotateShow other
 72 ┃       failure

    Insufficient coverage.
```

Figure 23.1.: Hedgehog fails

100% too young and 0% too old. The `invalidAge` generator is clearly not good enough. Let's have a look at its definition again:

```
invalidAge :: Gen Int
invalidAge = Gen.integral (Range.linear minBound 0)
```

We're only generating invalid ages between the minimum bound of `Int` and `0`. Let's fix that, by using `Gen.choice` and another generator for ages greater than 150:

```
invalidAge :: Gen Int
invalidAge = Gen.choice
  [ Gen.integral (Range.linear minBound 0)
  , Gen.integral (Range.linear 151 maxBound)
  ]
```

Running tests again, the coverage check stops complaining. But there's another problem:

```
λ> check prop_invalid_age_fails
  ✗ <interactive> failed at test/Validation/V1Test.hs:75:7
    after 3 tests and 2 shrinks.
    too young 67% ██████████ ..... ✓ 5%
  ⚠ too old    0% ..................... ✗ 5%

    ┏━━ test/Validation/V1Test.hs ━━━
 66 ┃ prop_invalid_age_fails = property $ do
 67 ┃   let genForm = SignupForm <$> validName <*> invalidAge
 68 ┃   form <- forAll genForm
    ┃   │ SignupForm { formName = "a" , formAge = 151 }
 69 ┃   cover 5 "too young" (formAge form <= 0)
 70 ┃   cover 5 "too old"   (formAge form >= 151)
 71 ┃   case validateSignup form of
 72 ┃     Failure (InvalidAge{} :| []) -> pure ()
 73 ┃     other                        -> do
 74 ┃       annotateShow other
    ┃       │ Success Signup { name = "a" , age = 151 }
 75 ┃       failure
    ┃       ^^^^^^^
```

Figure 23.2.: Hedgehog fails

OK, we have an actual bug. When the age is 151 or greater, the form is deemed valid. It should cause a validation failure. Looking closer at the implementation, we see that a pattern guard is missing the upper bound check:

```
validateAge age' | age' > 0  = Success (fromIntegral age')
                 | otherwise = Failure (pure (InvalidAge age'))
```

If we change it to `age' > 0 && age' <= 150`, and rerun the tests, they pass.

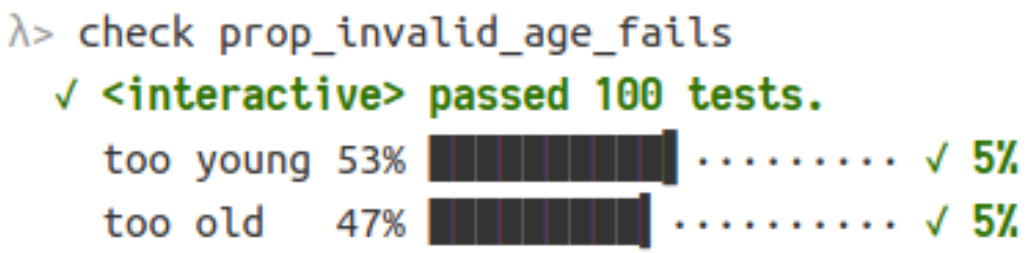

Figure 23.3.: Hedgehog passes

We've fixed the bug. Measuring and declaring requirements on coverage is a powerful tool in Hedgehog. It gives us visibility into the generative tests we run, making it practical to debug generators. It ensures our tests meet our coverage requirements, even as implementation and tests evolve over time.

23.5. From Ages to Birth Dates

So far, our efforts have been successful. We've fixed real issues in both implementation and tests. Management is pleased. They're now asking us to modify the signup system, and use our testing skills to ensure quality remains high.

Instead of entering their age, users will enter their birth date. Let's suppose this information is needed for something important, like sending out birthday gifts. The form validation function must be modified to check, based on the supplied birth date date, if the user signing up is old enough.

First, we import the `Calendar` module from the `time` package:

```
import           Data.Time.Calendar
```

Next, we modify the `SignupForm` data type to carry a `formBirthDate` of type `Date`, rather than an `Int`.

```
data SignupForm = SignupForm
  { formName      :: Text
  , formBirthDate :: Day
  } deriving (Eq, Show)
```

And we make the corresponding change to the `Signup` data type:

```
data Signup = Signup
  { name      :: Text
  , birthDate :: Day
  } deriving (Eq, Show)
```

We've also been requested to improve the validation errors. Instead of just `InvalidAge`, we define three constructors for various invalid birthdates:

```
data SignupError
  = NameTooShort Text
  | NameTooLong Text
```

```
  | TooYoung Day
  | TooOld Day
  | NotYetBorn Day
  deriving (Eq, Show)
```

Finally, we need to modify the `validateSignup` function. Here, we're faced with an important question. How should the validation function obtain *today's date*?

23.5.1. Keeping Things Deterministic

We could make `validateSignup` a non-deterministic action, which in Haskell would have the following type signature:

```
validateSignup
  :: SignupForm -> IO (Validation (NonEmpty SignupError) Signup)
```

Note the use of `IO`. It means we could retrieve the current time from the system clock, and extract the `Day` value representing today's date. But this approach has severe drawbacks.

If `validateSignup` uses `IO` to retrieve the current date, we can't test it with other dates. What it there's a bug that causes validation to behave incorrectly only on a particular date? We'd have to run the tests on that specific date to trigger it. If we introduce a bug, we want to know about it *immediately*. Not weeks, months, or even years after the bug was introduced. Furthermore, if we find such a bug with our tests, we can't easily reproduce it on another date. We'd have to rewrite the implementation code to trigger the bug again.

Instead of using `IO`, we'll use a simply technique for keeping our function pure: take all the information the function needs as arguments. In the case of `validateSignup`, we'll pass today's date as the first argument:

```
validateSignup
  :: Day -> SignupForm -> Validation (NonEmpty SignupError) Signup
```

Again, let's not worry about the implementation just yet. We'll focus on the tests.

23.5.2. Generating Dates

In order to test the new `validateSignup` implementation, we need to generate `Day` values. We're going to use a few functions from a separate module called `Data.Time.Gen`, previously written by some brilliant developer in our team. Let's look at their type signatures. The implementations are not very interesting.

The generator, `day`, generates a day within the given range:

```
day :: Range Day -> Gen Day
```

A day range is constructed with `linearDay`:

```
linearDay :: Day -> Day -> Range Day
```

Alternatively, we might use `exponentialDay`:

```
exponentialDay :: Day -> Day -> Range Day
```

The `linearDay` and `exponentialDay` range functions are analoguous to Hedgehog's `linear` and `exponential` ranges for integral numbers.

To use the generator functions from `Data.Time.Gen`, we first add an import, qualified as `Time`:

```
import qualified Data.Time.Gen      as Time
```

Next, we define a generator `anyDay`:

```
anyDay :: Gen Day
anyDay =
  let low  = fromGregorian 1900 1 1
      high = fromGregorian 2100 12 31
  in  Time.day (Time.linearDay low high)
```

The date range [1900-01-01,2100-12-31] is arbitrary. We could pick any centuries we like, provided the `time` package supports the range. But why not make it somewhat realistic?

23.5.3. Rewriting Existing Properties

Now, it's time to rewrite our existing property tests. Let's begin with the one testing that validating a form with all valid data succeeds:

```
prop_valid_signup_form_succeeds = property $ do
  today <- forAll anyDay                                    -- (1)
  let genForm = SignupForm <$> validName <*> validBirthDate today
  form <- forAll genForm                                    -- (2)

  case validateSignup today form of
    Success{}        -> pure ()
    Failure failure' -> do
      annotateShow failure'
      failure
```

A few new things are going on here. We're generating a date representing today (1), and generating a form with a birth date based on today's date (2). Generating today's date, we're effectively time travelling and running the form validation on that date. This means our `validBirthDate` generator must know which date is today, in order to pick a valid birth date. We pass today's date as a parameter, and generate a date within the range of 18 to 150 years earlier:

```
validBirthDate :: Day -> Gen Day
validBirthDate today = do
  n <- Gen.integral (Range.linear 18 150)
  pure (n `yearsBefore` today)
```

We define the helper function `yearsBefore` in the test suite. It offsets a date backwards in time by a given number of years:

```
yearsBefore :: Integer -> Day -> Day
yearsBefore years = addGregorianYearsClip (negate years)
```

The `Data.Time.Calendar` module exports the `addGregorianYearsClip` function. It adds a number of years, clipping February 29th (leap days) to February 28th where necessary.

Let's run tests:

```
λ> check prop_valid_signup_form_succeeds
     ✓ <interactive> passed 100 tests.
```

Let's move on to the next property, checking that invalid birth dates do *not* pass validation. Here, we use the same pattern as before, generating today's date, but use `invalidBirthDate` instead:

```
prop_invalid_age_fails = property $ do
  today <- forAll anyDay
  form <- forAll (SignupForm <$> validName <*> invalidBirthDate today)

  cover 5 "not yet born" (formBirthDate form > today)
  cover 5 "too young" (formBirthDate form > 18 `yearsBefore` today)
  cover 5 "too old" (formBirthDate form < 150 `yearsBefore` today)

  case validateSignup today form of
    Failure (TooYoung{}   :| []) -> pure ()
    Failure (NotYetBorn{} :| []) -> pure ()
    Failure (TooOld{}     :| []) -> pure ()
    other                        -> do
      annotateShow other
      failure
```

Notice that we've also adjusted the coverage checks. There's a new label, "not born yet," for birth dates in the future. Running tests, we see the label in action:

```
λ> check prop_invalid_age_fails
  ✓ <interactive> passed 100 tests.
    not yet born 18% ■■■■ ..... ✓ 5%
    too young    54% ■■■■■■■■■■ ..... ✓ 5%
    too old      46% ■■■■■■■■■ ..... ✓ 5%
```

Figure 23.4.: Hedgehog results

Good coverage, all tests passing. We're not quite done, though. There's a particular set of dates that we should be sure to cover: "today" dates and birth dates that are close to, or exactly, 18 years apart.

Within our current property test for invalid ages, we're only sure that generated birth dates include at least 5% too old, and at least 5% too young. We don't know how far away from the "18 years" validation boundary they are.

We could tweak our existing generators to produce values close to that boundary. Given a date T, exactly 18 years before today's date, then:

- `invalidBirthDate` would need to produce birth dates just after but not equal to T
- `validBirthDate` would need to produce birth dates just before or equal to T

There's another option, though. Instead of defining separate properties for valid and invalid ages, we'll use a *single* property for all cases. This way, we only need a single generator.

23.6. A Single Validation Property

In Building on developers' intuitions to create effective property-based tests, John Hughes talks about "one property to rule them all." Similarly, we'll define a single property `prop_validates_age` for birth date validation. We'll base our new property on `prop_invalid_age_fails`, but generalize to cover both positive and negative tests:

```
prop_validates_age = property $ do
  today <- forAll anyDay
  form  <- forAll (SignupForm <$> validName <*> anyBirthDate today) -- (1)

  let tooYoung        = formBirthDate form > 18 `yearsBefore` today -- (2)
      notYetBorn      = formBirthDate form > today
      tooOld          = formBirthDate form < 150 `yearsBefore` today
      oldEnough       = formBirthDate form <= 18 `yearsBefore` today
      exactly age = formBirthDate form == age `yearsBefore` today
      closeTo age =
        let diff' =
                diffDays (formBirthDate form) (age `yearsBefore` today)
        in  abs diff' `elem` [0 .. 2]

  cover 10 "too young"    tooYoung
  cover 1  "not yet born" notYetBorn
  cover 1  "too old"      tooOld

  cover 20 "old enough"   oldEnough                                  -- (3)
  cover 1  "exactly 18"   (exactly 18)
  cover 5  "close to 18"  (closeTo 18)

  case validateSignup today form of                                  -- (4)
    Failure (NotYetBorn{} :| []) | notYetBorn -> pure ()
    Failure (TooYoung{} :| []) | tooYoung -> pure ()
    Failure (TooOld{} :| []) | tooOld -> pure ()
    Success{} | oldEnough             -> pure ()
    other                             -> annotateShow other >> failure
```

There are a few new things going on here:

1. Instead of generating exclusively invalid or valid birth dates, we're now generating *any* birth date based on today's date
2. The boolean expressions are used both in coverage checks and in asserting, so we separate them in a `let` binding
3. We add three new labels for the valid cases
4. Finally, we assert on both valid and invalid cases, based on the same expressions used in coverage checks

Note that our assertions are more specific than in `prop_invalid_age_fails`. The failure cases only pass if the corresponding label expressions are true. The `oldEnough` case covers all valid birth dates. Any result other than the four expected cases is considered incorrect.

The `anyBirthDate` generator is based on today's date:

```
anyBirthDate :: Day -> Gen Day
anyBirthDate today =
  let                                                    -- (1)
      inPast range = do
        years <- Gen.integral range
        pure (years `yearsBefore` today)
      inFuture = do
        years <- Gen.integral (Range.linear 1 5)
        pure (addGregorianYearsRollOver years today)
      daysAroundEighteenthYearsAgo = do
        days <- Gen.integral (Range.linearFrom 0 (-2) 2)
        pure (addDays days (18 `yearsBefore` today))
  in                                                     -- (2)
      Gen.frequency
        [ (5, inPast (Range.exponential 1 150))
        , (1, inPast (Range.exponential 151 200))
        , (2, inFuture)
        , (2, daysAroundEighteenthYearsAgo)
        ]
```

We defines helper functions (1) for generating dates in the past, in the future, and close to 18 years ago. Using those helper functions, we combine four generators, with different date ranges, using a `Gen.frequency` distribution (1). The weights we use are selected to give us a good coverage. Let's run some tests (see figure 23.5)

Looks good! We've gone from testing positive and negative cases separately, to instead have a single property covering all cases, based on a single generator. It's now easier to generate values close to the valid/invalid boundary of our SUT, i.e. around 18 years from today's date.

```
λ> check prop_validates_age
  ✓ <interactive> passed 100 tests.
    too young     62% ▇▇▇▇▇▇▇▇▇▇ ..... ✓ 10%
    not yet born  20% ▇▇▇▇ ..... ✓  1%
    too old        4% ▇ ..... ✓  1%
    old enough    38% ▇▇▇▇▇▇▇ ..... ✓ 20%
    exactly 18    16% ▇▇▇ ..... ✓  1%
    close to 18   21% ▇▇▇▇ ..... ✓  5%
```

Figure 23.5.: Hedgehog results

23.7. February 29th

For the fun of it, let's run some more tests. We'll crank it up to 20000 (see figure 23.6 Failure! Chaos! What's going on here? Let's examine the test case:

- Today's date is 1956-02-29
- The birth date is 1938-03-01
- The validation function considers this *valid* (it returns a `Success` value)
- The test does considers this *invalid* (`oldEnough` is `False`)

This means that when the validation runs on a leap day, February 29th, and the person would turn 18 years old the day after (on March 1st), the validation function incorrectly considers the person old enough. We've found a bug.

23.7.1. Test Count and Coverage

Two things led us to find this bug:

1. Most importantly, that we generate today's date and pass it as a parameter. Had we used the actual date, retrieved with an IO action, we'd only be able to find this bug every 1461 days. Pure functions are easier to test.
2. That we ran more tests than the default of 100. We might not have found this bug until much later, when the generated dates happened to trigger this particular bug. In fact, running 20000 tests does not always trigger the bug.

Our systems are often too complex to be tested exhaustively. Let's use our form validation as an example. Between 1900-01-01 and 2100-12-31 there are 73,413 days. Selecting today's date and the birth date from that range, we have more than five billion combinations. Running that many Hedgehog tests in GHCi on my laptop (based on some quick benchmarks) would take about a month. And this is a simple pure validation function!

```
λ> check (withTests 20000 prop_validates_age)
  ✗ <interactive> failed at test/Validation/V3Test.hs:141:64
    after 17000 tests and 25 shrinks.
    too young     60% ██████████········ ✓ 10%
    not yet born  20% ████··············· ✓  1%
    too old        9% ██················ ✓  1%
    old enough    40% ████████············ ✓ 20%
    exactly 18    14% ██················ ✓  1%
    close to 18   21% ████ ·············· ✓  5%

       ╭── test/Validation/V3Test.hs ──
   114 │ prop_validates_age = property $ do
   115 │   today <- forAll anyDay
       │   | 1956 - 02 - 29
   116 │   form  <- forAll (SignupForm <$> validName <*> anyBirthDate today)
       │   | SignupForm { formName = "aa" , formBirthDate = 1938 - 03 - 01 }
   117 │
   118 │   let tooYoung        = formBirthDate form > 18 `yearsBefore` today
   119 │       notYetBorn      = formBirthDate form > today
   120 │       tooOld          = formBirthDate form < 150 `yearsBefore` today
   121 │       oldEnough       = formBirthDate form <= 18 `yearsBefore` today
   122 │       exactlyEighteen = formBirthDate form == 18 `yearsBefore` today
   123 │       closeToEighteen =
   124 │         let diff' =
   125 │                 diffDays (formBirthDate form) (18 `yearsBefore` today)
   126 │         in  abs diff' `elem` [0 .. 2]
   127 │
   128 │   cover 10 "too young"    tooYoung
   129 │   cover 1  "not yet born" notYetBorn
   130 │   cover 1  "too old"      tooOld
   131 │
   132 │   cover 20 "old enough"   oldEnough
   133 │   cover 1  "exactly 18"   exactlyEighteen
   134 │   cover 5  "close to 18"  closeToEighteen
   135 │
   136 │   case validateSignup today form of
   137 │     Failure (NotYetBorn{} :| []) | notYetBorn -> pure ()
   138 │     Failure (TooYoung{} :| []) | tooYoung -> pure ()
   139 │     Failure (TooOld{} :| []) | tooOld -> pure ()
   140 │     Success{} | oldEnough             -> pure ()
   141 │     other                             -> annotateShow other >> failure
       │     | Success Signup { name = "aa" , birthDate = 1938 - 03 - 01 }
       │     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
       │
```

Figure 23.6.: Hedgehog results

To increase coverage, even if it's not going to be exhaustive, we can increase the number of tests we run. But how many should we run? On a continuous integration server we might be able to run more than we do locally, but we still want to keep a tight feedback loop. And what if our generators never produce inputs that reveal existing bugs, regardless of the number of tests we run?

If we can't test exhaustively, we need to ensure our generators cover interesting combinations of inputs. We need to carefully design and measure our tests and generators, based on the edge cases we already know of, as well as the ones that we discover over time. PBT without measuring coverage easily turns into a false sense of security.

In the case of our leap day bug, we can catch it with fewer tests, and on every test run. We need to make sure we cover leap days, used both as today's date and as the birth date, even with a low number of tests.

23.7.2. Covering Leap Days

To generate inputs that cover certain edge cases, we combine specific generators using `Gen.frequency`:

```
(today, birthDate') <- forAll
  (Gen.frequency
    [ (5, anyDayAndBirthDate)                                  -- (1)

    , (2, anyDayAndBirthDateAroundYearsAgo 18)                 -- (2)
    , (2, anyDayAndBirthDateAroundYearsAgo 150)

    , (1, leapDayAndBirthDateAroundYearsAgo 18)                -- (3)
    , (1, leapDayAndBirthDateAroundYearsAgo 150)

    , (1, commonDayAndLeaplingBirthDateAroundYearsAgo 18) -- (4)
    , (1, commonDayAndLeaplingBirthDateAroundYearsAgo 150)
    ]
  )
```

Arbitrary values for today's date and the birth date are drawn most frequently (1), with a weight of 5. Next, with weights of 2, are generators for cases close to the boundaries of the validation function (2). Finally, with weights of 1, are generators for special cases involving leap days as today's date (3) and leap days as birth date (4).

Note that these generators return pairs of dates. For most of these generators, there's a strong relation between today's date and the birth date. For example, we can't first generate *any* today's date, pass that into a generator function, and expect it to always generate a leap day that occured 18 years ago. Such a generator would have to first generate the leap day and then today's date.

Let's define the generators. The first one, `anyDayAndBirthDate`, picks any today's date within a wide date range. It also picks a birth date from an even wider date range, resulting in some future birth dates and some ages above 150.

```
anyDayAndBirthDate :: Gen (Day, Day)
anyDayAndBirthDate = do
```

```
  today <- Time.day
    (Time.linearDay (fromGregorian 1900 1 1)
                    (fromGregorian 2020 12 31)
    )
  birthDate' <- Time.day
    (Time.linearDay (fromGregorian 1850 1 1)
                    (fromGregorian 2050 12 31)
    )
  pure (today, birthDate')
```

Writing automated tests with a hard-coded year 2020 might scare you. Won't these tests fail when run in the future? No, not these tests. Remember, the validation function is deterministic. We control today's date. The *actual* date on which we run these tests doesn't matter.

Similar to the previous generator is `anyDayAndBirthDateAroundYearsAgo`. First, it generates any date as today's date (1). Next, it generates an arbitrary date approximately some number of years ago (2), where the number of years is an argument of the generator.

```
anyDayAndBirthDateAroundYearsAgo :: Integer -> Gen (Day, Day)
anyDayAndBirthDateAroundYearsAgo years = do
  today <- Time.day                                    -- (1)
    (Time.linearDay (fromGregorian 1900 1 1)
                    (fromGregorian 2020 12 31)
    )
  birthDate' <- addingApproxYears (negate years) today  -- (2)
  pure (today, birthDate')
```

The `addingApproxYearsAgo` generator adds a number of years to a date, and offsets it between two days back and two days forward in time.

```
addingApproxYears :: Integer -> Day -> Gen Day
addingApproxYears years today = do
  days <- Gen.integral (Range.linearFrom 0 (-2) 2)
  pure (addDays days (addGregorianYearsRollOver years today))
```

The last two generators used in our `frequency` distribution cover leap day edge cases. First, let's define the `leapDayAndBirthDateAroundYearsAgo` generator. It generates a leap day used as today's date, and a birth date close to the given number of years ago.

```
leapDayAndBirthDateAroundYearsAgo :: Integer -> Gen (Day, Day)
leapDayAndBirthDateAroundYearsAgo years = do
  today      <- leapDay (Range.linear 1904 2020)
  birthDate' <- addingApproxYears (negate years) today
  pure (today, birthDate')
```

The `leapDay` generator uses `mod` to only generate years divisible by 4 and constructs dates on February 29th. That alone isn't enough to only generate valid leap days, though. Years divisible by 100 but not by 400 are not leap years. To keep the generator simple, we discard those years using the already existing `isLeapDay` predicate as a filter.

```
leapDay :: Range Integer -> Gen Day
leapDay yearRange = Gen.filter isLeapDay $ do
  year <- Gen.integral yearRange
  pure (fromGregorian (year - year `mod` 4) 2 29)
```

In general, we should be careful about discarding generated values using `filter`. If we discard too much, Hedgehog gives up and complains loudly. In this particular case, discarding a few generated dates is fine. Depending on the year range we pass it, we might not discard any date.

Finally, we define the `commonDayAndLeaplingBirthDateAroundYearsAgo` generator. It first generates a leap day used as the birth date, and then a today's date approximately the given number of years after the birth date.

```
commonDayAndLeaplingBirthDateAroundYearsAgo :: Integer -> Gen (Day, Day)
commonDayAndLeaplingBirthDateAroundYearsAgo years = do
  birthDate' <- leapDay (Range.linear 1904 2020)
  today <- addingApproxYears years birthDate'
  pure (today, birthDate')
```

That's it for the generators. Now, how do we know that we're covering the edge cases well enough? With coverage checks!

```
cover 5     -- (1)
      "close to 18, validated on common day"
      (closeTo 18 && not (isLeapDay today))
cover 1
      "close to 18, validated on leap day"
      (closeTo 18 && isLeapDay today)

cover 5     -- (2)
      "close to 150, validated on common day"
      (closeTo 150 && not (isLeapDay today))
cover 1
      "close to 150, validated on leap day"
      (closeTo 150 && isLeapDay today)

cover 5     -- (3)
      "exactly 18 today, born on common day"
      (exactly 18 && not (isLeapDay birthDate'))
cover       -- (4)
  1
  "legally 18 today, born on leap day"
  (  isLeapDay birthDate'
  && (addGregorianYearsRollOver 18 birthDate' == today)
  )
```

We add new checks to the property test, checking that we hit both leap day and regular day cases around the 18th birthday (1) and the 150th birthday (2). Notice that we had similar checks before, but we were not discriminating between leap days and common days.

Finally, we check the coverage of two leap day scenarios that can occur when a person legally turns 18: a person born on a common day turning 18 on a leap day (3), and a leapling turning 18 on a common day (4).

Running the modified property test, we get the leap day counter-example every time, even with as few as a hundred tests. For example, we might see today's date being 1904-02-29 and the birth date being 1886-03-01. The validation function deems the person old enough. Again, this is incorrect.

Now that we can quickly and reliably reproduce the failing example we are in a great position to find the error. While we could use a fixed seed to reproduce the particular failing case from the 20000 tests run, we are now more confident that the property test would catch future leap day-related bugs, if we were to introduce new ones. Digging into the implementation, we'll find a boolean expression in a pattern guard being the culprit:

```
birthDate' <= addGregorianYearsRollOver (-18) today
```

The use of `addGregorianYearsRollOver` together with adding a negative number of years is the problem, rolling over to March 1st instead of clipping to February 28th. Instead, we should use `addGregorianYearsClip`:

```
birthDate' <= addGregorianYearsClip (-18) today
```

Running 100 tests again, we see that they all pass, and that our coverage requirements are met.

```
λ> Hedgehog.check prop_validates_age
  ✓ <interactive> passed 100 tests.
    too young                                17% ▇▇▇ ····· ✓ 10%
    not yet born                              7% ▇ ····· ✓ 1%
    too old                                  19% ▇▇▇ ····· ✓ 1%
    old enough                               83% ▇▇▇▇▇▇▇▇▇▇▇▇▇ ··· ✓ 20%
    close to 18, validated on common day     30% ▇▇▇▇▇ ····· ✓ 5%
    close to 18, validated on leap day        2% ▇ ····· ✓ 1%
    close to 150, validated on common day    31% ▇▇▇▇▇ ····· ✓ 5%
    close to 150, validated on leap day       6% ▇ ····· ✓ 1%
    exactly 18 today, born on common day     17% ▇▇▇ ····· ✓ 5%
    legally 18 today, born on leap day        5% ▇ ····· ✓ 1%
```

Figure 23.7.: Hedgehog results

23.8. Summary

In this tutorial, we started with a simple form validation function, checking the name and age of a person signing up for an online service. We defined property tests for

positive and negative tests, learned how to test generators with coverage checks, and found bugs in both the test suite and the implementation.

When requirements changed, we had to start working with dates. In order to keep the validation function deterministic, we had to pass in today's date. This enabled us to simulate the validation running on any date, in combination with any reported birth date, and trigger bugs that could otherwise take years to find, if ever. Had we not made it deterministic, we would likely not have found the leap day bug later on.

To generate inputs that sufficiently test the validation function's boundaries, we rewrote our separate positive and negative properties into a single property, and used coverage checks to ensure the quality of our generators. The trade-off between multiple disjoint properties and a single more complicated property is hard.

With multiple properties, for example split between positive and negative tests, both generators and assertions can be simpler and more targeted. On the other hand, you run a risk of missing certain inputs. The set of properties might not cover the entire space of inputs. Furthermore, performing coverage checks across multiple properties, using multiple targeted generators, can be problematic.

Ensuring coverage of generators in a single property is easier. You might even get away with a naive generator, depending on the system you're testing. If not, you'll need to combine more targeted generators, for example with weighted probabilities. The drawback of using a single property is that the assertion not only becomes more complicated, it's also likely to mirror the implementation of the SUT. As we saw with our single property testing the validation function, the assertion duplicated the validation rules. You might be able to reuse the coverage expressions in assertions, but still, there's a strong coupling.

The choice between single or multiple properties comes down to *how* you want to cover the boundaries of the SUT. Ultimately, both approaches can achieve the same coverage, in different ways. They both suffer from the classic problem of a test suite mirroring the system it's testing.

Finally, running a larger number of tests, we found a bug related to leap days. Again, without having made the validation function deterministic, this could've only been found on a leap day. We further refined our generators to cover leap day cases, and found the bug reliably with as few as 100 tests. The bug was easy to find and fix when we had the inputs pointing directly towards it.

That's it for this tutorial. Thanks for reading, and happy property testing and time travelling!

24. Metamorphic Testing - Hillel Wayne

> ***William Yao:*** *Another more general PBT post. The motivating problem: vanilla PBT assumes it's easy to generate inputs to our code. Sometimes it's not. For instance, what if you're testing an image classifier neural net? You can't randomly generate images, because you don't know what the output classification should be for a random image. So we might only have a small set of manually-classified test inputs. Metamorphic testing is a way of expanding our set of test inputs programmatically by transforming the inputs we do have in some way and finding relationships between the original result and the transformed result. For instance, if we invert the color of one of our test images, our classifier should probably give us the same result. If you make a property out of that, you now have more test cases for free, and that catches more bugs.*
>
> *Original article: [23]*

Confession: I read the ACM Magazine. This makes me a dweeb even in programming circles. One of the things I found in it is "Metamorphic Testing". I've never heard of it, and nobody I knew heard about it either. But the academic literature was shockingly impressive: many incredibly successful case studies in wildly different fields. So why haven't we heard of it before? There's only one article anywhere targeted at people outside academia. Let's make it two.

24.1. Background

Most written tests use **oracles**. That's where you know the answer and are explicitly checking that the computation gives you the answer.

```
def test_dist():
    p1 = (0, 3)
    p2 = (4, 0)
    assert dist(p1, p2) == 5
```

In addition to being an oracle test, it's also a manual test. Somebody sat down and decided specific inputs and specific outputs. As systems get more complex, bespoke manual tests become less and less useful. Each one only hits a single point in a larger state space, and we want something that covers the state space.

This gives us **generative testing**: writing tests that hit a random set of the statespace. The most popular style of generative testing is **property based testing**, or PBT. We find a "property" of the function and then generate inputs and see if the outputs match that property.

```
def test_dist():
    p1 = random_point()
    p2 = random_point()
    assert dist(p1, p2) >= 0
```

The advantage of PBT is that it gives more coverage. The downside is that we've lost specificity. This is *not* an oracle test anymore! We don't know what the answer should be, and the function might be broken in a way that has the same property. We rely on heuristics here.

One big problem with PBT is finding good properties. Most functions have simple, general properties and complex, specific properties. General properties (see chapter 20) can be applied to a wider variety of functions but don't give us much information. More specific properties give more information, but are harder to find and only apply to specific problem domains. If you had a function that determined whether or not a graph is acyclic, what property tests would you write? Would they give you confidence your function is right?

24.2. Motivation

Now take a more complex problem. Imagine we're trying to write an English speech-to-text (STT) processor. It takes a sound file and outputs the text. How would you test it?

The simplest way is with a manual oracle. Read out a sentence and confirm it gives you that sentence. But this isn't nearly enough! The range of human speech is *enormous*. It'd be better if we could instead test 1,000 or 10,000 different sound files. Manually transcribing oracles is going to be way too expensive. This means we have to use property-based testing instead.

But how do we generate the inputs? One way would be to create random strings, then run them through a text-to-speech processor (TTS), and then check our STT gives the same text. But, once again, this gives us a very limited range of human speech. Will our TTS give us changes in tone, slurred words, strong accents? If we don't handle those, is our STT actually that useful? We're better off sweeping for "wild" text, such as from radio, podcasts, online videos.

Now we have a new problem. Using a TTS meant we started with the transcription. We don't have that with "wild" text, and we still don't want to transcribe it ourselves. We're restricted to using properties instead. So what properties should we test? Some simple ones might be "it doesn't crash on any input" (good) or "It doesn't turn acoustic music into words" (maybe?). These properties don't really cover the "intent" of the program, and don't increase confidence all that much.

So we have two problems. One, we need a wide variety of speech inputs. Two, we need a way to know make them into useful tests without spending hours manually transcribing the speech into oracles.

24.3. Metamorphic Testing

That all treats the output in isolation. What if we embed it in a broader context? For example, if a given soundclip transcibes to output `out`, then we should *still* get output `out` if we:

- Double the volume, or
- Raise the pitch, or
- Increase the tempo, or
- Add some background static, or
- Add some traffic noises, or
- Do any combination of the above.

All of these are "straightforward" transformations we can easily test. For example, for the "traffic noises" test, we can take 10 traffic samples, overlay them on a soundclip, and see that all 11 versions match. We can double or half the volume to turn 11 versions into 33 versions, and double the tempo to get 66 versions. Then we can then scale this up to every soundclip in our database, which helps augment the space of our inputs.

Having 66 versions to compare is useful enough. However, there's something else here: we don't need to know what the output is. If all 66 transformations return `out`, the test passes, and if any return something different, the test fails. At no point do we need to check what `out` is. This is really, really big. It dramatically increases the range we can test with very little human effort. We could, for example, download an episode of *This American Life*, run the transformations, and see if they all match[1]. We have useful tests *without listening to the voice clip.* We can now generate complex, deep tests without the use of an oracle!

The two inputs, along with their outputs, are all connected to each other. This kind of property spanning multiple inputs/outputs is called a **metamorphic relation**[2]. Testing that leverages this is called **metamorphic testing**. For complex systems, it can be easier to find interesting metamorphic relations than interesting single input/output properties.

To be a bit more formal: if we have `x` and `f(x)`, we can make some transformation on `x` to get `x2` and `f(x2)`. In the STT case, we just checked `f(x) = f(x2)`, but we can use whatever relations we want between the two. You could also have MRs like `f(x2) > f(x)` or "`f(x2)/f(x)` is an integer". Similarly, we can also span more than two inputs, using `f(x)` and `f(x3)`. One example of this might be comparing search engine results with no filters to engine results with one filter and two filters. Most of the case studies I read only use two inputs, because even that is enough to find crazy bugs.

24.4. The Case Studies

Speaking of case studies: How effective is MT in practice? It's one thing to talk about a technique in abstract, or provide toy examples. Reading case studies is useful for

1 Okay, there's obvious problems here, because the podcast might have music, samples in other languages, etc. But the theory is sound: given we have a way of acquiring speech samples, we can use it as part of tests without having to manually label it first.

2 The corresponding idea in specifications is **hyperproperties**, properties on sets of behaviors instead of individual behaviors. Most HP research is concerned with security hyperproperties. As I understand it HPs are a superset of MRs.

three reasons. First, it shows whether or not this actually works. Next, it shares some potential gotchas if we try to use MT. Finally, it gives us ideas on *how* we can use it. Any MR a case study uses is something we might be able to adapt for our own purposes.

"Metamorphic Testing: A Review of Challenges and Opportunities" lists a lot of studies, but they're all academic papers. Here are a few of the most interesting ones. Articles marked `(pdf)` are, unsurprisingly, PDFs.

METTLE: A Metamorphic Testing Approach To Validating Unsupervised Machine Learning Methods (pdf) Defines 11 different MRs for testing unsupervised clustering, like "do we get the same result if we shuffle the inputs?" and "do additional inputs at cluster boundaries belong to those clusters?" Different models changed under dfferent relations. For example, about 5% of tested k-means models had a mean clustering error of 20% under shuffling the order of input points

DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars (pdf) Subject was car vision systems, MRs were things like "adding a rain filter" or "slightly tilting the image". Authors put sample results here: Pretty much all the systems they tested collapsed under the MR changes.

Automated Testing of Graphics Shader Compilers (pdf) Injecting dead code and runtime-constants into shaders made things in pictures disappear or turn to noise. The researchers made a startup called GraphicsFuzz off their work, which was acquired by Google and the site taken down.

Metamorphic Testing of RESTful Web APIs (pdf) Do you get the same items when you change the pagination? What if you order them by date? A whole bunch of errors in Spotify and Youtube in this paper.

An innovative approach for testing bioinformatics programs using metamorphic testing (pdf, but now not) Finding mistakes in bioinformatics stuff? Look I barely understand bioinformatics, but it's demonstrating how MR is useful in specialist domains.

24.4.1. The Problem

Huh, they're all PDFs.

Finding all of those took several hours. And that ties into the biggest drag on MT adoption: All of the above are **preprints**, or first drafts of eventual academic papers. When I dig into obscure techniques, I always ask "why is it obscure?" Sometimes there's an obvious reason, sometimes it's a complex set of subtle reasons, sometimes it's just bad luck.

In the case of MT the problem is obvious. **Almost all of the info is behind academic paywalls.** If you want to learn about MT, you either need journal access or spend hours hunting down preprints[3].

3 I had a second, refuted hypothesis: since a lot of the major researchers are from China and Hong Kong, maybe the technique was more well-known in Mandarin-language programming communities than English-language ones. Brian Ng was kind enough to check for me and didn't find significant use.

24.4.2. Learning More

The inventor of MT is TY Chen. He's also the driver of a lot of the research. Other names are Zhi Quan Zhou and Sergio Segura, both of whom have put all of their preprints online. Most of the research is by one of those.

The best starting resource are probably Metamorphic Testing: A Review of Challenges and Opportunities and A Survey on Metamorphic Testing. While this article was about Metamorphic *Testing*, researchers have also been applying Metamorphic Relationships in general to a wide variety of other disciplines, such as formal verification and debugging. I have not researched those other uses in depth, but they're probs also worth looking into.

In terms of application, it should be theoretically possible to adapt most PBT libraries to check metamorphic properties. In fact the first example in the Quickcheck tests a MR, and this essay on PBT implicitly uses an MR. *In general* it seems to me that most PBT research focuses on how we effectively generate and shrink inputs, while MT research is more focused on determining what we actually want to test. As such they are probably complementary techniques.

Thanks to Brian Ng for help researching this.

24.4.3. PS: Request

It's not actually that surprising that I never heard of this before. There's a lot of really interesting, useful techniques that never leave their tiny bubble. Learning about MT was more luck than any action on my part.

If you know of anything you think deserves wider use, please email me.

25. Unit testing effectful Haskell with monad-mock

William Yao:

Introduces a way of doing more "traditional" unit testing, but focused more on doing white-box testing, checking whether the code under test performed certain operations, rather than just expecting on the output. Since this is Haskell, doing that is a little bit more unusual.

Personally, I'm not a fan of writing tests like this because they feel like they couple the tests to the implementation too tightly, and calcify design decisions too quickly. Still, if there's something that's legitimately too difficult to sandbox for your testing environment, it's a useful pattern to be aware of.

Original article: [24]

Nearly eight months ago (see chapter 22), I wrote a blog post about unit testing effectful Haskell code using a library called test-fixture. That library has served us well, but it wasn't as easy to use as I would have liked, and it worked better with certain patterns than others. Since then, I've learned more about Haskell and more about testing, and I'm pleased to announce that I am releasing an entirely new testing library, monad-mock.

25.1. A first glance at monad-mock

The monad-mock library is, first and foremost, designed to be *easy*. It doesn't ask much from you, and it requires almost zero boilerplate.

The first step is to write an mtl-style interface that encodes an effect you want to mock. For example, you might want to test some code that interacts with the filesystem:

```
class Monad m => MonadFileSystem m where
  readFile :: FilePath -> m String
  writeFile :: FilePath -> String -> m ()
```

Now you just have to write your code as normal. For demonstration purposes, here's a function that defines copying a file in terms of `readFile` and `writeFile`:

```
copyFile :: MonadFileSystem m => FilePath -> FilePath -> m ()
copyFile a b = do
  contents <- readFile a
  writeFile b contents
```

Making this function work on the real filesystem is trivial, since we just need to define an instance of `MonadFileSystem` for `IO`:

```
instance MonadFileSystem IO where
  readFile = Prelude.readFile
  writeFile = Prelude.writeFile
```

But how do we test this? Well, we *could* run some real code in `IO`, which might not be so bad for such a simple function, but this seems like a bad idea. For one thing, a bad implementation of `copyFile` could do some pretty horrible things if it misbehaved and decided to overwrite important files, and if you're constantly running a test suite whenever a file changes, it's easy to imagine causing a lot of damage. Running tests against the real filesystem also makes tests slower and harder to parallelize, and it only gets much worse once you are doing more complex effects than interacting with the filesystem.

Using monad-mock, we can test this function in just a couple of lines of code:

```
import Control.Exception (evaluate)
import Control.Monad.Mock
import Control.Monad.Mock.TH
import Data.Function ((&))
import Test.Hspec

makeMock "FileSystemAction" [ts| MonadFileSystem |]

spec = describe "copyFile" $
  it "reads a file and writes its contents to another file" $
    evaluate $ copyFile "foo.txt" "bar.txt"
      & runMock [ ReadFile "foo.txt" :-> "contents"
                , WriteFile "bar.txt" "contents" :-> () ]
```

That's it! The last two lines of the above snippet are the real interesting bits, which specify the actions that are expected to be executed, and it couples them with their results. You will find that if you tweak the list in any way, such as reordering the actions, eliminating one or both of them, or adding an additional action to the end, the test will fail. We could even turn this into a property-based test that generated arbitrary file paths and file contents.

Admittedly, in this trivial example, the mock is a little silly, since converting this into a property-based test would demonstrate how much we've basically just reimplemented the function in our test. However, once our function starts to do somewhat more complicated things, then our tests become more meaningful. Here's a similar function that only copies a file if it is nonempty:

```
copyNonemptyFile :: MonadFileSystem m => FilePath -> FilePath -> m ()
copyNonemptyFile a b = do
  contents <- readFile a
  unless (null contents) $
    writeFile b contents
```

This function has some logic which is very clearly *not* expressed in its type, and it would be difficult to encode that information into the type in a safe way. Fortunately, we can guarantee that it works by writing some tests:

```
describe "copyNonemptyFile" $ do
  it "copies a file with contents" $
    evaluate $ copyNonemptyFile "foo.txt" "bar.txt"
      & runMock [ ReadFile "foo.txt" :-> "contents"
                , WriteFile "bar.txt" "contents" :-> () ]

  it "does nothing with an empty file" $
    evaluate $ copyNonemptyFile "foo.txt" "bar.txt"
      & runMock [ ReadFile "foo.txt" :-> "" ]
```

These tests are much more useful, and they have some actual value to them. Imagine we had accidentally written `when` instead of `unless`, an easy typo to make. Our tests would fail with some useful error messages:

```
1) copyNonemptyFile copies a file with contents
     uncaught exception: runMockT: expected the following unexecuted actions to be ru
       WriteFile "bar.txt" "contents"

2) copyNonemptyFile does nothing with an empty file
     uncaught exception: runMockT: expected end of program, called writeFile
       given action: WriteFile "bar.txt" ""
```

You now know enough to write tests with monad-mock.

25.2. Why unit test?

When the issue of testing is brought up in Haskell, it is often treated with a certain distaste by a portion of the community. There are some points I've seen a number of times, and though they take different forms, they boil down to two ideas:

1. "Haskell code does not need tests because the type system can prove correctness."
2. "Testing in Haskell is trivial because it is a pure language, and testing pure functions is easy."

I've been writing Haskell professionally for over a year now, and I can happily say that there *is* some truth to both of those things! When my Haskell code typechecks, I feel a confidence in it that I would not feel were I using a language with a less powerful type system. Furthermore, Haskell encourages a "pure core, impure shell" approach to system design that makes testing many things pleasant and straightforward, and it completely eliminates the worry of subtle nondeterminism leaking into tests.

That said, Haskell is not a proof assistant, and its type system cannot guarantee everything, especially for code that operates on the boundaries of what Haskell can control. For much the same reason, I find that my pure code is the code I am *least* likely to need to test, since it is also the code with the strongest type safety guarantees, operating on types in my application's domain. In contrast, the effectful code is often what I find the most value in extensively testing, since it often contains the most subtle complexity, and it is frequently difficult or even impossible to encode into types.

Haskell has the power to provide remarkably strong correctness guarantees with a surprisingly small amount of effort by using a combination of tests and types, using each to accommodate for the other's weaknesses and playing to each technique's strengths. Some code is test-driven, other code is type-driven. Most code ends up being a mix of both. Testing is just a tool like any other, and it's nice to feel confident in one's ability to effectively structure code in a decoupled, testable manner.

25.3. Why mock?

Even if you accept that testing is good, the question of whether or not to *mock* is a subtler issue. To some people, "unit testing" is synonymous with mocks. This is emphatically not true, and in fact, overly aggressive mocking is one of the best ways to make your test suite completely worthless. The monad-mock approach to mocking is a bit more principled than mocking in many dynamic, object-oriented languages, but it comes with many of the same drawbacks: mocks couple your tests to your implementation in ways that make them less valuable and less meaningful.

For the `MonadFileSystem` example above, I would actually probably *not* use a mock. Instead, I would use a **fake**, in-memory filesystem implementation:

```
newtype FakeFileSystemT m a = FakeFileSystemT (StateT [(FilePath, String)] m a)
  deriving (Functor, Applicative, Monad)

fakeFileSystemT :: Monad m => [(FilePath, String)]
                -> FakeFileSystemT m a -> m (a, [(FilePath, String)])
fakeFileSystemT fs (FakeFileSystemT x) = second sort <$> runStateT x fs

instance Monad m => MonadFileSystem (FakeFileSystemT m) where
  readFile path = FakeFileSystemT $ get >>= \fs -> lookup path fs &
    maybe (fail $ "readFile: no such file ‘" ++ path ++ "’") return
  writeFile path contents = FakeFileSystemT . modify $ \fs ->
    (path, contents) : filter ((/= path) . fst) fs
```

The above snippet demonstrates how easy it is to define a `MonadFileSystem` implementation in terms of `StateT`, and while this may seem like a lot of boilerplate, it really isn't. You have to write a fake *once* per interface, and the above block is a minuscule twelve lines of code. With this technique, you are still able to write tests that depend on the state of the filesystem before and after running the implementation, but you decouple yourself from the precise process of getting there:

```
describe "copyNonemptyFile" $ do
  it "copies a file with contents" $ do
    let ((), fs) = runIdentity $ copyNonemptyFile "foo.txt" "bar.txt"
          & fakeFileSystemT [ ("foo.txt", "contents") ]
    fs `shouldBe` [ ("bar.txt", "contents"), ("foo.txt", "contents") ]

  it "does nothing with an empty file" $ do
    let ((), fs) = runIdentity $ copyNonemptyFile "foo.txt" "bar.txt"
          & fakeFileSystemT [ ("foo.txt", "") ]
    fs `shouldBe` [ ("foo.txt", "") ]
```

This is better than using a mock, and I would highly recommend doing it if you can! However, a lot of real applications have to interact with services of much greater complexity than an idealized filesystem, and creating that sort of in-memory fake is not always practical. One such situation might be interacting with AWS CloudFormation, for example:

```
class Monad m => MonadAWS m where
  createStack :: StackName -> StackTemplate -> m (Either AWSError StackId)
  listStacks :: m (Either AWSError [StackSummaries])
  describeStack :: StackId -> m (Either AWSError StackInfo)
  -- and so on...
```

AWS is a very complex system, and it can do dozens of different things (and fail in dozens of different ways) based on an equally complex set of inputs. For example, in the above API, `createStack` needs to parse its template, which can be YAML or JSON, in order to determine which of many possible errors and behaviors can be produced, both on the initial call and on subsequent ones.

Creating a fake implementation of *AWS* is hardly feasible, and this is where a mock can be useful. By simply writing `makeMock "AWSAction" [ts| MonadAWS |]`, we can test functions that interact with AWS in a pure way without necessarily needing to replicate all of its complexity.

25.3.1. Isolating mocks

Of course, tests that use mocks provide less value than tests that use "smarter" fakes, since they are far more tightly coupled to the implementation, and it's dramatically more likely that you will need to change the tests when you change the logic. To avoid this, it can be helpful to create multiple interfaces to the same thing: a high-level interface and a low-level one. If our above `MonadAWS` is a low-level interface, we could create a high-level counterpart that does precisely what our application needs:

```
class Monad m => MonadDeploy m where
  executeDeployment :: Deployment -> m (Either DeployError ())
```

When running our application "for real", we would use `MonadAWS` to implement `MonadDeploy`:

```
executeDeploymentImpl :: MonadAWS m => Deployment -> m (Either DeployError ())
executeDeploymentImpl = ...
```

The nice thing about this is we can actually test `executeDeploymentImpl` using a `MonadAWS` mock, so we can still have unit test coverage of the code on the boundaries of our system! Additionally, by containing the mock to a single place, we can test the rest of our code using a smarter fake implementation of `MonadDeploy`, helping to decouple our code from AWS's complex API and improve the reliability and usefulness of our test suite.

They key point here is that mocking is just a small piece of the larger testing puzzle in *any* language, and that is just as true in Haskell. An overemphasis on mocking is an easy way to end up with a test suite that feels useless, probably because it is. Use mocks as a technique to insulate your application from the complexity in others' APIs, then use more domain-specific testing techniques and type-level assertions to ensure the correctness of your logic.

25.4. How monad-mock works

If you've read this far and are convinced that monad-mock is useful, you may safely stop reading now. However, if you are interested in the details of what it actually does and what makes it tick, the rest of this blog post is going to focus on how the implementation works and how it compares to other techniques.

The centerpiece of monad-mock's API is its monad transformer, `MockT`, which is a type constructor that accepts three types:

```
newtype MockT (f :: * -> *) (m :: * -> *) (a :: *)
```

The `m` and `a` type variables obviously correspond to the usual monad transformer arguments, which represent the underlying monad and the result of the monadic computation, respectively. The `f` variable is more interesting, since it's what makes `MockT` work at all, and it isn't even a type: it's a type constructor with kind `* -> *`. What does it mean?

Looking at the type signature of `runMockT` gives us a little bit more information about what that `f` actually represents:

```
runMockT :: (Action f, Monad m) => [WithResult f] -> MockT f m a -> m a
```

This type signature provides two pieces of key information:

1. The `f` parameter is constrained by the `Action f` constraint.
2. Running a mocked computation requires supplying a list of `WithResult f` values. This list corresponds to the list of expectations provided to `runMock` in earlier examples.

To understand both of these things, it helps to examine the definition of an actual datatype that can have an `Action` instance. For the filesystem example, the action datatype looks like this:

```
data FileSystemAction r where
  ReadFile :: FilePath -> FileSystemAction String
  WriteFile :: FilePath -> String -> FileSystemAction ()
```

Notice how each constructor clearly corresponds to one of the methods of `MonadFileSystem`, with a type to match. Now the purpose of the type provided to the `FileSystemAction` constructor (in this case `r`) should hopefully become clear: it represents the type of the value *produced* by each method. Also note that the type is completely phantom—it does not appear in negative position in any of the constructors.

With this in mind, we can take a look at the definition of `WithResult`:

```
data WithResult f where
  (:->) :: f r -> r -> WithResult f
```

This is what defines the `(:->)` constructor from earlier in the blog post, and you can see that it effectively just represents a tuple of an action and a value of its associated result. It's completely type-safe, since it ensures the result matches the type argument to the action.

Finally, this brings us to the `Action` class, which is not complex, but is unfortunately necessary:

```
class Action f where
  eqAction :: f a -> f b -> Maybe (a :~: b)
  showAction :: f a -> String
```

Notice that these methods are effectively just `(==)` and `show`, lifted to type constructors of kind `* -> *`. One significant difference is that `eqAction` produces `Maybe (a :~: b)` instead of `Bool`, where `(:~:)` is from `Data.Type.Equality`. This is a type equality witness, which means a successful equality between two values allows the compiler to be sure that the two *types* are equal. This is necessary for the implementation of `runMockT` due to the phantom type in actions—in order to convince GHC that we can properly return the result of a mocked action, we need to assure it that the value we're going to return is actually of the proper type.

Implementing this typeclass is not particularly burdensome, but it's entirely boilerplate, so even if you want to define your own action type (that is, you don't want to use `makeMock`), you can use the `deriveAction` function from `Control.Monad.Mock.TH` to derive an `Action` instance on an existing datatype.

25.4.1. Connecting the mock to its class

Now that we have an action with which to mock a class, we need to actually define an instance of that class for `MockT`. For this process, monad-mock provides a `mockAction` function with the following type:

```
mockAction :: (Action f, Monad m) => String -> f r -> MockT f m r
```

This function accepts two arguments: the name of the method being mocked and the action that represents the current call. This is easier to illustrate with an actual instance of `MonadFileSystem` using `MockT` and our `FileSystemAction` type:

```
instance Monad m => MonadFileSystem (MockT FileSystemAction m) where
  readFile a = mockAction "readFile" (ReadFile a)
  writeFile a b = mockAction "writeFile" (WriteFile a b)
```

This allows `readFile` and `writeFile` to defer to the mock, and providing the names of the functions as strings helps monad-mock to produce useful error messages upon failure. Internally, `MockT` is a `StateT` that keeps track of a list of `WithResult f` values as its state. Each call to the mock checks the action against the internal list of calls, and if they match, it returns the associated result. Otherwise, it throws an exception.

This scheme is simple, but it seems to work remarkably well. There are some obvious enhancements that will probably be eventually necessary, like allowing action results that run in the underlying monad `m` in order to support things like `throwError` from `MonadError`, but so far, it hasn't been necessary for what we've been using it for. Certain tricky signatures defy this simple technique, such as signatures where a monadic action appears in a negative position (that is, the signatures you need things like monad-control or monad-unlift for), but we've found that most of our effects don't have any reason to include such signatures.

25.5. A brief comparison with free(r) monads

At this point, astute readers will likely be thinking about free monads, which parts of this technique greatly resemble. The representation of actions as GADTs is especially similar to freer, which does something extremely similar. Indeed, you can think of this technique as something that combines a freer-style representation with mtl-style classes. Given that freer already does this, you might ask yourself what the point is.

If you are already sold on free monads, monad-mock may very well be uninteresting to you. From the perspective of theoretical novelty, monad-mock is not anything new or different. However, there are a variety of practical reasons to prefer mtl over free, and it's nice to see how easy it is to enjoy the testing benefits of free without too much extra effort.

An in-depth comparison between mtl and free is well outside the scope of this blog post. However, the key point is that this technique *only* affects test code, so the real runtime implementation will not be affected in any way. This means you can take advantage of the performance benefits and ecosystem support of mtl without sacrificing simple, expressive testing.

25.6. Conclusion

To cap things off, I want to emphasize monad-mock's role as a single part of a larger initiative we've been making for the better part of the past eighteen months. Haskell is a language with ever-evolving techniques and style, and it's sometimes dizzying to figure out how to use all the pieces together to develop robust, maintainable applications. While monad-mock might not be anything drastically different from existing testing techniques, my hope is that it can provide an opinionated mechanism to make testing easy and accessible, even for complex interactions with other services and systems.

I've made an effort to make it abundantly clear in this blog post that monad-mock is *not* a silver bullet to testing, and in fact, I would prefer other techniques for ensuring correctness whenever possible. Even so, mocking is a nice tool to have in your toolbox, and it's a good fallback to get even the worst APIs under test coverage.

If you want to try out monad-mock for yourself, take a look at the documentation on Hackage and start playing around! It's still early software, so it's not the most proven or featureful, but we've managed to get mileage out of it already, all the same. If you find any problems, have a use case it does not support, or just find something about it unclear, please do not hesitate to open an issue on the GitHub repository—we obviously can't fix issues we don't know about.

Thanks as always to the many people who have contributed ideas that have shaped my philosophy and approach to testing and have helped provide the tools that make this library work. Happy testing!

Bibliography

[1] William Yao: A list of Haskell articles on good design, good testing – https://williamyaoh.com/posts/2019-11-24-design-and-testing-articles.html. Accessed: 27.11.2019. 1

[2] Matt Parsons: Type Safety Back and Forth – https://www.parsonsmatt.org/2017/10/11/type_safety_back_and_forth.html. Accessed: 27.11.2019. 3

[3] Matt Parsons: Keep your types small... – https://www.parsonsmatt.org/2018/10/02/small_types.html. Accessed: 27.11.2019. 4

[4] David Luposchainsky: Algebraic Blindness – https://github.com/quchen/articles/blob/master/algebraic-blindness.md. Accessed: 27.11.2019. 5

[5] Alexis King: Parse, don't validate – https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/. Accessed: 27.11.2019. 6

[6] Jasper van der Jeugt: On Ad-hoc Datatypes – https://jaspervdj.be/posts/2016-05-11-ad-hoc-datatypes.html. Accessed: 27.11.2019. 7

[7] Tom Ellis: Good design and type safety in Yahtzee – http://h2.jaguarpaw.co.uk/posts/good-design-and-type-safety-in-yahtzee/. Accessed: 27.11.2019. 8

[8] Tom Ellis: Using our brain less in refactoring Yahtzee – http://h2.jaguarpaw.co.uk/posts/using-brain-less-refactoring-yahtzee/. Accessed: 27.11.2019. 9

[9] Michael Snoyman: Weakly Typed Haskell – https://www.fpcomplete.com/blog/2018/01/weakly-typed-haskell. Accessed: 27.11.2019. 10

[10] Matt Parsons: The Trouble with Typed Errors – https://www.parsonsmatt.org/2018/11/03/trouble_with_typed_errors.html. Accessed: 27.11.2019. 11

[11] Sandy Maguire: Type-Directed Code Generation – https://reasonablypolymorphic.com/blog/type-directed-code-generation/. Accessed: 27.11.2019. 12

[12] Jasper van der Jeugt: The Handle Pattern – https://jaspervdj.be/posts/2018-03-08-handle-pattern.html. Accessed: 11.12.2019. 13

[13] Michael Snoyman: The ReaderT Design Pattern – https://www.fpcomplete.com/blog/2017/06/readert-design-pattern. 14

[14] Jasper van der Jeugt: Practical testing in Haskell – https://jaspervdj.be/posts/2015-03-13-practical-testing-in-haskell.html. Accessed: 27.11.2019. 15

[15] Oskar Wickström: Property-Based Testing in a Screencast Editor: Introduction – https://wickstrom.tech/programming/2019/03/02/property-based-testing-in-a-screencast-editor-introduction.html. Accessed: 27.11.2019. 16

[16] Oskar Wickström: Property-Based Testing in a Screencast Editor, Case Study 1: Timeline Flattening – https://wickstrom.tech/programming/2019/03/24/property-based-testing-in-a-screencast-editor-case-study-1.html. Accessed: 27.11.2019. 17

[17] Oskar Wickström: Property-Based Testing in a Screencast Editor, Case Study 2: Video Scene Classification – https://wickstrom.tech/programming/2019/04/17/property-based-testing-in-a-screencast-editor-case-study-2.html. Accessed: 27.11.2019. 18

[18] Oskar Wickström: Property-Based Testing in a Screencast Editor, Case Study 3: Integration Testing – https://wickstrom.tech/programming/2019/06/02/property-based-testing-in-a-screencast-editor-case-study-3.html. Accessed: 27.11.2019. 19

[19] Scott Wlaschin: Choosing properties for property-based testing – https://fsharpforfunandprofit.com/posts/property-based-testing-2/. Accessed: 27.11.2019. 20

[20] Hillel Wayne: Findin Property Tests – https://www.hillelwayne.com/post/contract-examples/. Accessed: 27.11.2019. 21

[21] Alexis King: Using types to unit-test in Haskell – https://lexi-lambda.github.io/blog/2016/10/03/using-types-to-unit-test-in-haskell/. Accessed: 27.11.2019. 22

[22] Oskar Wickström: Time Travelling and Fixing Bugs with Property-Based Testing – https://wickstrom.tech/programming/2019/11/17/time-travelling-and-fixing-bugs-with-property-based-testing.html. Accessed: 27.11.2019. 23

[23] Hillel Wayne: Metamorphic Testing – https://www.hillelwayne.com/post/metamorphic-testing/. Accessed: 27.11.2019. 24

[24] Alexis King: Unit testing effectful Haskell with monad-mock – https://lexi-lambda.github.io/blog/2017/06/29/unit-testing-effectful-haskell-with-monad-mock/. Accessed: 27.11.2019. 25