A Type-Safe Approach to Categorized Data

Eli Peery

Recently I was working on a personal project involving the US democratic primaries. The goal was to create a quiz that helped the user weigh candidates based on how each reflected the user's personal values. One interesting problem I encountered very early on was figuring out how best to represent the questions in the quiz. Since there were a lot of questions, over 95 in total, I knew I wanted the user to be able to choose what type of questions they answered. For example, if the user cared a lot about the environment and didn't pay much attention to the economy, they shouldn't have to answer questions about the current state of NAFTA. It was clear I needed some way of separating the questions into categories; exactly how, I wasn't sure.

First, I thought to use a simple record type to represent the question. It would have fields for the question text, the topic of the question, and a few other pieces of quiz-specific information. Users could be shown a subset of the total questions by filtering on the "topic" field of each question. I could use floating point values to represent opinions about the questions. Candidates could then be represented as a list containing pairings of questions and opinions. Simple enough.

```
data Question
  = Question
      { questionInfo :: Text,
        questionTopic :: Text
      }

type Opinion = Double
type Candidate = [(Question, Opinion)]

-- Example question and candidate
--                                  question info              question topic
--                                        ↓                          ↓
freeLunch = Question "There is no such thing as a free lunch" "Economics"

friedman :: Candidate
friedman = [(freeLunch, 1.0)]
```

Not long after I implemented this approach, a major design flaw became visible. There was nothing making sure the questions were actually assigned to the

correct topics, or if the topic they were assigned to even exists! Since topics are represented as text, you could set the topic to whatever you want and the compiler wouldn't know the difference. Typos or other little mistakes could lead to some very annoying bugs.

It's clear that this representation leaves a lot to be desired. If you're like me and you enjoy getting the compiler to do your work for you, you're probably thinking that we need some way of letting the compiler know about all valid questions and a way of assigning those questions to topics. That way the compiler could check our code for us and we'd be sure we didn't make any careless errors.

The initial way I went about doing this was simple. First, we'll create types representing the questions.

```
--       topic            _____ questions _____
--         ↓            ↓                     ↓
data Environment = ParisAgreement | GreenNewDeal

data Economics = EstateTax | SupportNAFTA

data Healthcare = SinglePayerSystem | PublicHealthInsurance
```

Then we'll create a data type that we'll use to combine the questions. The reason we need is because `List` in Haskell can only contain items of the same type. Without using `Topics` as a wrapper, there's no easy way to store and present questions as a group.

```
data Topics
  = TEnvironment Environment
  | TEconomics Economics
  | THealthcare Healthcare
```

Lastly, we'll create a function called `getQuestion` which we'll use to convert our concrete representation of questions, `Topics`, into our more flexible `Question` record type.

```
getQuestion :: Topics -> Question
getQuestion (TEnvironment ParisAgreement) =
  Question "The U.S. should rejoin the Paris Climate Agreement" "Environment"
getQuestion (TEnvironment GreenNewDeal) =
  Question "The Green New Deal is a good idea" "Environment"
getQuestion (TEconomics EstateTax) =
  Question "I am in favor of a national estate tax" "Economics"
getQuestion (TEconomics SupportNAFTA) =
  Question "The US should support NAFTA" "Economics"
getQuestion (THealthcare SinglePayerSystem) =
  Question "The U.S. should have a single-payer healthcare system" "Healthcare"
getQuestion (THealthcare PublicHealthInsurance) =
  Question "The U.S. should have some form of public health insurance" "Healthcare"
```

Now we can change our representation of candidates to reflect our newly created types.

```
type Candidate = [(Topics, Opinion)]
```

This seemed like a pretty good solution at the time. Questions were well defined and I could still leverage the power of record types to store information. But as I started to extend the quiz with more questions, topics, and functions that relied on those topic, a growing amount of the program would need to change to include cases for the new data types. This situation is referred to as the Expression Problem, a name given to it by Philip Wadler.

The Expression Problem basically states that programs are made up of data types and functions that act on those data types. As a program grows, the number of functions that rely on the structure of those data types also grows. Because of this, extending the program or adding more functionality becomes exponentially more difficult.

A commonly cited solution to the Expression Problem is a paper written by Wouter Swierstra titled Data Types á la Carte. The paper is specifically about the Expression Problem as it relates to programming language interpreters but a simplified version of its solution fits our problem quite nicely.

First, we'll create a simple data type which is basically the same as the standard `Either` type. We use an extension called TypeOperators so that our type constructor can be represented as a `(+)` symbol that will make our types easier to read than if we used `Either`.

```
-- Standard Either type
data Either a b = Left a | Right b

example1 :: Either Int (Either String Bool)
example1 = Right (Right True)

-- Our custom type
data a + b = InL a | InR b

infixr 8 +

example2 :: Int + String + Bool
example2 = InR (InR True)
```

We use the `infixr` keyword to specify that our operator is right associative. This makes it so we can omit unnecessary parentheses in our types and further increase our code readability.

Now that we have this new data type we can use it to replace our previous `Topics` data type.

```
type Topics = Environment + Economics + Healthcare
```

And we can redefine our functions to work on our new version of `Topics`.

```
getQuestion :: Topics -> Question
getQuestion (InL ParisAgreement) =
  Question "The U.S. should rejoin the Paris Climate Agreement" "Environment"
getQuestion (InL GreenNewDeal) =
  Question "The Green New Deal is a good idea" "Environment"
getQuestion (InR (InL EstateTax)) =
  Question "I am in favor of a national estate tax" "Economics"
getQuestion (InR (InL SupportNAFTA)) =
  Question "The US should support NAFTA" "Economics"
getQuestion (InR (InR SinglePayerSystem)) =
  Question "The U.S. should have a single-payer healthcare system" "Healthcare"
getQuestion (InR (InR PublicHealthInsurance)) =
  Question "The U.S. should have some form of public health insurance" "Healthcare"
```

This doesn't seem to fix our problem though. We still have one big function that we need to go back and update if we add more topics. Along with this, the nested data constructors get unwieldy very quickly. To get around this problem, let's instead define an ad hoc typeclass that encapsulates the idea of converting a topic into a question.

```
class IsTopic a where
  getQuestion :: a -> Question
```

We can now make our topics instances of this typeclass.

```
instance IsTopic Environment where
  getQuestion ParisAgreement =
    Question "The U.S. should rejoin the Paris Climate Agreement" "Environment"
  getQuestion GreenNewDeal =
    Question "The Green New Deal is a good idea" "Environment"

instance IsTopic Economics where
  getQuestion EstateTax =
    Question "I am in favor of a national estate tax" "Economics"
  getQuestion SupportNAFTA =
    Question "The US should support NAFTA" "Economics"

instance IsTopic Healthcare where
  getQuestion SinglePayerSystem =
    Question "The U.S. should have a single-payer healthcare system" "Healthcare"
  getQuestion PublicHealthInsurance =
    Question "The U.S. should have some form of public health insurance" "Healthcare"
```

This is the general way that we will add more functions in the future. Instead of creating monolithic functions that encapsulate all of our data types, we create type classes for the functionality we want and make our data types instances of that type class.

We can even create an instance for our new sum data type!

```
instance (IsTopic a, IsTopic b) => IsTopic (a + b) where
  getQuestion (InL x) = getQuestion x
  getQuestion (InR y) = getQuestion y
```

Here we say that if both the left element and the right element are topics, then we can convert each to a question by stripping off the excess data constructors and calling `getQuestion` on the inner value.

This lets us map our `getQuestion` function over a list of `Topics` to get back a list of `Questions`.

```
topics :: [Topics]
topics =
  [ InL ParisAgreement
  , InL GreenNewDeal
  , InR (InL EstateTax)
  , InR (InL SupportNAFTA)
  , InR (InR SinglePayerSystem)
  , InR (InR PublicHealthInsurance)
  ]

questions :: [Question]
questions = getQuestion <$> topics
```

This is great and all but there's still one last issue we've yet to address. Whenever we want something of type `Topics`, we have to manually type out a long chain of `InL`s and `InR`s.

In the actual application there are 15 topics so you can imagine how annoying it would be to work with `InR (InR (InR (InR (InR (InR (InR (InR (InR (InR (InR (InR (InR (InR EqualityAct))))))))))))))`.

To get around this the paper makes use of a clever typeclass.

```
class a :<: b where
  inj :: a -> b

instance a :<: a where
  inj = id

instance a :<: (a + b) where
  inj = InL

instance {-# OVERLAPPABLE #-} (a :<: c) => a :<: (b + c) where
  inj = InR . inj
```

This typeclass does a couple things. Its type tells us that `a` is contained within `b`, and the function `inj` allows for automatic injection into our custom sum type.

For our pragmatic purposes of being able to use this approach in programs we write, it's not critical that you're able to instantly grok how this works. (It took me a while before I understood exactly why it was able to do what it does.) What is most important is that you understand that we can replace our `InL`s and `InR`s from before with our new inj function and it will automate away the busywork.

This lets us remove all the excess boilerplate from before and instead define a list of `Topics` like this.

```
topics :: [Topics]
topics =
  [ inj ParisAgreement
  , inj GreenNewDeal
  , inj EstateTax
  , inj SupportNAFTA
  , inj SinglePayerSystem
  , inj PublicHealthInsurance
  ]
```

Now it doesn't matter how deeply nested our topics are. We just use the `inj` function and it takes care of everything for us. Much nicer.

In the next post we'll talk about ways we can use type-level programming to query this data type. We'll go over things like filtering by category, retrieving all questions of a category, and parsing strings we get from our API into concrete values.

Notes

1 - Got ideas on how this could be better implemented? Things that I missed? If you have any suggestions for how I can do to improve this post please let me know.

2 - In order for the code to in this post to compile, you'll need to enable these language extensions.

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
```

3 - The actual implementation I ended up using is a bit more involved than that described here. For more information, I'll refer you to this wonderful blog post by Sandy Maguire: Better Data Type á la Carte.