Algebraic Data Types Done Differently

Joachim Breitner

3.6.2006

For a while I have been thinking: Isn't there a way to get rid of the intermediate **Maybe** construct in a commen expression like **fromMaybe default . lookup**.

It seems that a way to do that would be to pass more information to the **Maybe**-generating function: What to do with a **Just**-Value, and what to return in case of **Nothing**.

This leads to a new definion of the **Maybe** data type as a function. Later I discovered that this seems to work for any algebraic data type.

This is probably nothing new, but I am offline at the moment, so I can't check. This means that this might also be total rubbish. Enjoy.

Contents

1. Preparation 1

2. Implementation of our Maybe 2

3. Same with Bool 3

4. Other data types 4

5. And what is it good for? 4

A. Proofs of the Monad law 4

1. Preparation

Assuming we have a function `f :: Maybe a` that returns a **Maybe** containing a variable of type `a`. Usually, we do not really care about this encapsulation, but we rather care to handle two cases – one with data and one without – separately. If we want to get rid of the **Maybe**, we obviously need to pass a default return value to the function. Therefore, we have `f :: a -> a`. But that is not sufficient: We still want to be able to distinguish between **Nothing** and **Just default**. So we move the processing of the **Just** value into the function. Because the result type of that processing does not matter to `f`, the type becomes `f :: forall b. (a -> b) -> b -> b`.

Is this equivalent to Prelude.Maybe-? We can check by trying to implement the various functions.

2. Implementation of our Maybe

We first hide some functions from **Prelude** that we are going to redefine, and import some standard modules. The `forall` construct is not standard Haskell98, so we tell ghc to not be so picky.

```
{-# OPTIONS_GHC -fglasgow-exts #-}
import Prelude hiding (Maybe, Bool, maybe, not)
import Control.Monad (MonadPlus, mzero, mplus)
```

Now let's define the type. We could have used **type** instead, and omitted the `M` constructor, but that would prevent us from declaring type class instansnces for our **Maybe**, so we have to take the long road with **newtype** and a constructor.

```
newtype Maybe a = M (forall b. (a -> b) -> b -> b)
```

We also need replacements for the standard constructors **Just** and **Maybe**. These are now mere functions, so we have to write them in lower case.

`just` just applies the given function to the stored value, ignoring the default. `nothing` just returns the default value, ignoring the function.

```
just :: a -> Maybe a
just x = M (\f d -> f x)

nothing :: Maybe a
nothing = M (\f d -> d)
```

We also want to be able to retrieve a value from **Maybe**, so we implement the **maybe** function as seen in the **Prelude**.

```
maybe d f (M a) = a f d
```

Because of the similarity between **maybe** and our definition of **Maybe**, some of the following code could be written using this **maybe** function. I chose not to, to keep the structure visible. On the other hand this means that any implementation of **Maybe** just needs `just`, `nothing` and **maybe**, and all the other functions can be done based on that, independently of the implementation.

Currently, our **Maybe**s are just uncomprehensible functions. We want to make them visible, so we define an instance for **Show**.

```
instance Show a => Show (Maybe a) where
        show (M a) = a (("Just␣"++) . show) "Nothing"
```

Monads are alwasy interesting, so we implement that instance as well. I use the **maybe** function once out of convenience, otherwise this might be an ugly **let** expression.

```
instance Monad Maybe where
        return = just
        fail _ = nothing
        (M a) >>= b = M (\f d -> a (maybe d f . b) d)
```

Is this really a monad? We can find out by proving the monad laws, but the proofs are ugly. See the end of this document if you want to know.

Just for fun we can define other class instances.

```
instance MonadPlus Maybe where
        mzero = nothing
        mplus (M a) (M b) = M (\f d -> a f (b f d))

instance Functor Maybe where
        fmap f' (M a) = M (\f d -> a (f.f') d)
```

A demonstration of how to implement a "regular" **Maybe** function here, see **maybeToList**:

```
maybeToList (M a) = a (:[]) []
```

3. Same with Bool

After having done this, I wonder: Can we also implement other data types this way? And it seems we can. Let's try **Bool**. We usually work with boolean values to later on decide which of two possible values we want. So why not pass these values directly to our **Bool** and skip the middle man? So we get:

```
newtype Bool = B (forall a. a -> a -> a)
```

Again, we implement the constructors as functions. Just for fun I'll use point-free style here, instead of lambda-notation. A true value will return the first argument, a false value the second.

```
true :: Bool
true = B const

false :: Bool
false = B (flip const)
```

Some logical operations:

```
not (B a) = B (flip a)

(B a) && (B b) = B (\t f -> a (b t f) f)
(B a) || (B b) = B (\t f -> a  t     (b t f))
```

Usually a **Bool** ends up in an **if** expression. But the standard **if** requires, hardcodedly, a **Bool**. If the **if** construct were a regular function, we could re-define that. But it is not, so we create our own **if'**.

```
if' (B a) t f = a t f
```

It seems that **if'** is to our **Bool** what **maybe** is to our **maybe**. Also, all implementations of an boolean value would need to only provide `true`, `false` and some kind of **if**.

The instance to show these values is straight forward

```
instance Show Bool where
        show (B a) = a "True" "False"
```

To make our **Bool**s compareable, we need to implement **Eq**. Unfortnately, this hard-codes the result type **Prelude.Bool**, so we are leaving our world here. We could define our own **Eq'** with `(==)` returning our **Bool**.

```
instance Eq Bool where
        (B a) == (B b) =  a (b True False) (b False True)
```

To combine our two types, we implement **isJust** and **isNothing**.

```
isJust (M a) = B (\t f -> a (const t) f)
isNothing    = not.isJust
```

4. Other data types

It seems that all algebraic data types can be represented this way. There will be one parameter per constructor which itself takes as many parameters as the constructor takes arguments. So examples might be:

```
newtype Either a b = E (forall c. (a -> c) -> (b -> c) -> c)
newtype (,) a b = E (forall c. (a -> b -> c) -> c)
```

5. And what is it good for?

No idea. I was hoping for higher efficiency because there is no need to pack the data in a **Maybe** or similar, but intuitively this implementation will be much more expensive because of all the functions to pass around. I also expect functions whose result is needed more than once to be run more often than necesary, although the compiler could optimize that away. **Maybe**.

A. Proofs of the Monad law

Sorry for being lazy and not commenting every step. And yes, they are ugly

```
   unM (M a) = a                                        — shortcut
   unM . M == \x -> unM (M x) == \x -> x == id          — properties
   M . unM == id :: Maybe a -> Maybe a

   return a >>= k
== just a >>= k                                         — def. return
== M (\f d -> f a) >>= k                                — def. just
== M (\f d -> (\f' d' -> f a) (maybe d f . b) d)        — def. bind
== M (\f d -> (maybe d f . b) a)                        — apply lambda
== M (\f d -> maybe d f (b a))                          — def. (.)
== M (\f d -> maybe d f (M (unM (b a))))                — shortcut
```

```
== M (\f d -> unM (b a) f d)                          — def. maybe
== M (unM (b a))                                      — eta reduction (?)
== b a                                                — shortcut
```

```
   (M m) >>= return
== (M m) >>= just
== (M m) >>= (\a -> just a))
== (M m) >>= (\a -> M (\f d -> f a))
== M (\f d -> m (maybe d f . (\a -> M (\f' d' -> f' a)) d))
== M (\f d -> m (\a -> maybe b f ( M (\f' d' -> f' a)) d))
== M (\f d -> m (\a -> (\f' d' -> f' a) f d) d))
== M (\f d -> m (\a -> f a) d)
== M (\f d -> m f d)
== M m
```

```
   (M m) >>= (\x -> k x >>= h)
== M (\f d -> m (maybe d f . (\x -> k x >>= h)) d)
== M (\f d -> m (\a -> (maybe d f . (\x -> k x >>= h)) a) d)
== M (\f d -> m (\a -> (maybe d f ((\x -> k x >>= h) a))) d)
== M (\f d -> m (\a -> (maybe d f (k a >>= h))) d)
== M (\f d -> m (\a -> (maybe d f (M (unM (k a >>= h))))) d)
== M (\f d -> m (\a -> (unM (k a >>= h)) f d) d)
== M (\f d -> m (\a -> (let M k' = (k a >>= h) in unM (M k')) f d) d)
== M (\f d -> m (\a -> (let M k' = (k a >>= h) in k') f d) d)
== M (\f d -> m (\a -> (let M k' = M (\f' d' -> unM (k a) (maybe d' f' . h)) in k') f d) d
== M (\f d -> m (\a -> (\f' d' -> unM (k a) (maybe d' f' . h) d') f d) d)
== M (\f d -> m (\a -> unM (k a) (maybe d f . h) d) d)
== M (\f d -> m (\a -> unM (k a) (maybe d f . h) d) d)
== M (\f d -> m (\a -> maybe d (maybe d f . h) (M (unM (k a)))) d)
== M (\f d -> m (\a -> maybe d (maybe d f . h) (k a)) d)
== M (\f d -> m (\a -> (maybe d (maybe d f . h) . k) a) d)
== M (\f d -> m (maybe d (maybe d f . h) . k) d)
== M (\f d -> (\f' d' -> m (maybe d' f' . k) d') (maybe d f . h) d)
== M (\f d -> let M mk' = M (\f' d' -> m (maybe d' f' . k) d') in mk' (maybe d f . h) d)
== M (\f d -> let M mk' = (M m >>= k) in mk' (maybe d f . h) d)
== M (\f d -> let M mk' = (M m >>= k) in unM (M mk') (maybe d f . h) d)
== M (\f d -> unM ((M m) >>= k) (maybe d f . h) d)
== ((M m) >>= k) >>= (M h)
```