why is Data.Set not a Monad?

Dan Doel dan.doel at gmail.com
Sun May 6 16:00:09 EDT 2007

- Previous message: why is Data.Set not a Monad?
- Next message: why is Data.Set not a Monad?y
- **Messages sorted by:** [date] [thread] [subject] [author]

```
On Sunday 06 May 2007, Frederik Eaton wrote:
> Hello,
>
> Anyone know why Data.Set is not a Monad? Or Data.Map?
>
> It seems Data.Map is a Functor, but not Data.Set...
>
> I am confused. Am I not importing the right modules? Thanks,

These questions all have slightly different answers, I think. Somewhat out of
order:

1) Data.Map isn't a monad because it isn't one. Consider:

  return :: a -> Map k a

What key will return choose to inject a value into a map? The only option that
leaps to mind is:

  return a = singleton undefined a

But that's hardly useful.

2) Data.Set is a monad, but you can't convince Haskell's type system of that
the way things are currently structured. This is because set operations
require elements to be instances of the Ord typeclass, and the Monad
typeclass signature doesn't allow for that. In the current typeclass, you
have:

  return :: a -> m a

While Set requires:

  return :: Ord a => a -> m a -- return = singleton

It is possible to structure things so that Set can be given a Monad instance,
but it may require some extensions. Here's one such way. Consider the
following typeclass:

  class Monad m a where
    return :: a -> m a
    (>>=) :: (Monad m b) => m a -> (a -> m b) -> m b

Here, 'm' is the monad type constructor, and 'a' will be the types it works
on. An instance is needed for each such allowable a. In Set's case, with
undecidable instances, this is easy:

  instance (Ord a) => Monad' Set a where
    return a = singleton a
    ...
```

The (Ord a) context is provided for return. However, bind is still a problem, because the obvious definition:

```
  m >>= f = fold (union . f) empty m
        :: (Ord b) => Set a -> (a -> Set b) -> Set b
```

Has an (Ord b) context that we can't provide. One way (not the only one, I'm sure) to solve this is to rebuild Set as a GADT, so that the (Ord b) context is packaged with the set. This can be simulated by wrapping the existing set (suppose the current set is imported qualified):

```
  data Set a where
    Empty :: Set a
    Wrap :: Ord a => Set a -> Set a

  singleton :: Ord a => a -> Set a
  singleton a = Wrap (Set.singleton a)

  union :: Set a -> Set a -> Set a
  union Empty t = t
  union s Empty = s
  union (Wrap s) (Wrap t) = Wrap (Set.union s t)

  fold :: (a -> b -> b) -> b -> Set a -> b
  fold _ z Empty = z
  fold f z (Wrap s) = Set.fold f z s
```

Now, since the GADT union doesn't require an Ord context, we can write:

```
  instance Ord a => Monad Set a where
    return a = singleton a -- The same as before
    s >>= f = fold (union . f) Empty s
```

So, we're finally at a 'valid' Monad instance for Set, and it only took us multi-parameter type classes, undecidable instances, and GADTs. :) Existing monads can be declared members of the revised class like so:

```
  instance Monad [] a where
    return a = [a]
    l >>= f = foldr ((++) . f) [] l
```

Simply not restricting the parameter 'a' leaves you with the case we currently have.

Other approaches have been suggested, I think, but this is one. I'm not sure it's an advisable road to take, as it's rather complicated, but it's an option.

3) As for Functors, it's easy to define an fmap operation for Map k v. You take your function of type (v -> u) and apply it to each element, storing the result at the same key. However, consider the type of the obvious fmap implementation for Set a:

```
  fmap f s = fold (insert . f) empty s :: Ord b => (a -> b) -> Set a -> Set b
```

This requires an (Ord b) context that is absent from the Functor method's signature, just as we ran into problems with the signatures in the current Monad class. In fact, Set's 'map' function requires Ord contexts for a and b both.

The problem here is that you can't use the same tricks as above to provide the (Ord b) context for fmap via a GADT (insert still requires a provided context). I suppose you could, of course, take *both* a and b as parameters to the class, so that you could place constraints on both. The same thing would work for Monad above, off the top of my head, and you could avoid the

```
GADT. However, I suspect if you start down a road of taking a parameter for
each distinct type variable in your method signatures, and having to declare
(whether explicitly or implicitly) n^m instances for a class, rather than 1,
things are going to get hairy.

Data.Set does provide a 'mapMonotonic' function of type:

  (a -> b) -> Set a -> Set b

which is the right type, but it appears to assume that the function respects
the ordering, such that:

  a1 `compare` a2 == f a1 `compare` f a2

or something like that. You could, therefore, pass in a function that doesn't
follow that, and up with a Set that isn't a set. Thus, it's unsuitable for
use as fmap.

Anyhow, I hope that made some sense at least, and answered some of your
questions. I'll attach a file that has a slightly more fleshed out
implementation of the GADT set wrapper, along with a monad instance, in case
you want to play with it. Some things are renamed a bit compared to the
above, since, for example, there already is a Monad typeclass.

Cheers.

-- Dan
-------------- next part --------------
A non-text attachment was scrubbed...
Name: GADTSet.hs
Type: text/x-hssrc
Size: 1167 bytes
Desc: not available
Url : http://www.haskell.org/pipermail/libraries/attachments/20070506/27d0d4b1/GADTSet.bin
```

- Previous message: why is Data.Set not a Monad?
- Next message: why is Data.Set not a Monad?y
- **Messages sorted by:** [date] [thread] [subject] [author]

More information about the Libraries mailing list