Interactive code snippets not yet available for SoH 2.0, see our Status of of School of Haskell 2.0 blog post (https://www.fpcomplete.com/blog/2016/01/soh-status)

How to Replace Failure by a Heap of Successes

2 May 2015 Edward Kmett (https://www.schoolofhaskell.com/user/edwardk)
View Markdown source (https://www.schoolofhaskell.com/tutorial-raw/4906/0f6911c474fa38006c6b1ba653b47020740cadcf)

Previous content: Unlifted Structures (https://www.schoolofhaskell.com/user/edwardk/unlifted-structures)

Next content: PHOAS For Free (https://www.schoolofhaskell.com/user/edwardk/phoas)

See all content by Edward Kmett (https://www.schoolofhaskell.com/user/edwardk)

Sections

- Conclusion

One problem that libraries like `pipes` and `machines` often have to deal with is the notion of leftovers (https://hackage.haskell.org/package/pipes-parse-3.0.2/docs/Pipes-Parse.html). What do you do with the stuff you haven't consumed? As parsers (and stream transducers) are usually implemented, we wind up losing contravariance on the input argument and/or we have to clutter our code with a separate type argument or transformer for handling leftovers. How can we do better?

Danel Ahman and Tarmo Uustalu gave us a nice general theory of Update Monads (http://homepages.inf.ed.ac.uk/s1225336/papers/types13postproc.pdf). Today I want to go and tie that notion back to the idea of writing parsing combinators. I've been explaining this technique to folks since long before I'd ever heard the term "update monad," but never took the time to write it up.

Why now? I also then want to showcase a new way to exploit the limited structure of the updates to make a more efficient `Applicative` for parsing, and also talk a bit about how this same general design can be used to address the issue of leftovers in streaming models.

Monoid Actions

You can always update any state to any new (or old) state in the state monad. We have `get` and `put` after all.

Something like

```
undo m = do
  s <- get
  a <- m
  put s
  return a
```

will let us roll back to a previous state after we make a change, no matter what it is.

But what if you want to restrict the space of updates you are allowed to make to the state? What if some actions should be irreversible or you are interested in a very small space of updates which can be expressed in tightly bounded space that is much smaller than the entire new state itself?

An example where you might want an update to be irreversible is if you are keeping state about a bunch of file handles. Closing a file handle may affect that state, but you can't meaningfully "reopen it" just by reverting to a previous state involving the status of all of

your file handles.

In practice we usually try to hide the whole state from the user by not exporting it, and then providing a limited palette of operations we can perform on top of our now-opaque representation.

This isn't the only option we have!

Another way to do this is to define some language of updates and of how updates compose. The easiest such language to use that will fit nicely with the needs of a `Monad` is to make it so a chain of such updates can be composed associatively, and such that there is a unit update. In other words, the right vocabulary for "updating" our state is probably a `Monoid`. Then we need to figure out how to apply that `Monoid` to our state.

For this we can appeal to the notion of a (right) monoid action on a set (http://en.wikipedia.org/wiki/Semigroup_action).

That is to say we want:

```
class Monoid m => RightMonoidAction s m where
  act :: s -> m -> s
```

such that

```
act s mempty = s
act s (mappend m n) = act (act s m) n
```

All this says is that the identity element changes nothing, and that the composition of monoid elements results in the composition of the effects on the state.

The laws can be stated slightly more elegantly for left monoid actions, `m -> s -> s`, where we can think of `act` as monoid homomorphism from `m` to `Endo s`, but it is convenient to think of time as advancing to the right, so we'll stick to right monoid actions for now. Left monoid actions are discussed in a fairly practical setting in Brent Yorgey's very pleasant functional pearl Monoids: Themes and Variations (http://www.cis.upenn.edu/~byorgey/pub/monoid-pearl.pdf).

Update vs. State

So then what is an update monad?

Well, let's consider the old fashioned state monad again:

```
newtype State s a = State { runState :: s -> (a, s) }
```

This looks like a composition of two functors, `(->) s` and `(,) s`, although the second is flipped, but it doesn't *act* like the composition of reader and writer!

The notion of an `Update` monad came out of trying to find something that felt more like that composition.

What if we break up the need for the two `s`s to be the same? Done one way you get the Bob Atkey-style parameterized monad for State:

```
newtype State i j a = State { runState :: i -> (a, j) }
```

This is what we usually do, but another way to more loosely couple the types of the two `s`'s is to take the update to be some answer in a `Monoid` that has an action on our state.

```
newtype Update s m a = Update { runUpdate :: s -> (m, a) }
```

This has the same type as above (except for the fact that I needlessly shuffled the pair), but we want to give it a completely different `Monad`. Instead of a parameterized monad, we want a regular `Monad`. Instead of matching indices, we're applying the updates to our state.

The result you give back for how to manipulate the state is a mere update that has an appropriate monoid action on the state, rather than a whole new state:

```
instance RightMonoidAction s m => Applicative (Update s m) where
  pure a = Update $ \s -> (mempty, a)
  Update ff <*> Update fa = Update $ \s -> case ff s of
    (m, f) -> case fa (act s m) of
      (n, a) -> (mappend m n, f a)

instance RightMonoidAction s m => Monad (Update s m) where
  return a = Update $ \s -> (mempty, a)
  Update f >>= k = Update $ \s -> case f s of
    (m, a) -> case runUpdate (k a) (act s m) of
      (n, b) -> (mappend m n, b)
```

So what examples might we come up with?

We can always `get` in an update monad:

```
get = Update $ \s -> (mempty, s)
```

And it is easy to define an action for `Endo s` on `s`. Using that choice of `Monoid` lets us easily recover something with the full power of `State`.

```
instance RightMonoidAction s (Endo s) where
  act s (Endo f) = f s

put s = Update $ \_ -> (Endo (const s), ())
```

We can also recover the power of `State` by using `Last` from `Data.Monoid` with an appropriate action.

```
instance RightMonoidAction s (Last s) where
  act s (Last m) = fromMaybe s m

put s = Update $ \_ -> (Last (Just s), ())
```

We can recover `Writer m` with

```
instance Monoid m => RightMonoidAction () m where
  act () m = ()
```

or recover `Reader e` with

```
instance RightMonoidAction e () where
  act e () = e
```

For a simple example, let's say you have a monotonically increasing counter for fresh variables. You could have the update language consist of how many times you bump the counter, the relative change, rather than the new counter value itself.

Ahman and Uustalu have a bunch of other examples and we'll build one more below.

As an aside: We can also derive a coupdate comonad. This can be useful for defining a variant notion of a lens where we restrict updates to updates that can be made in some monoidal language. This works because coupdate is analogous to the store comonad restricted to updates in some monoidal language. We might revisit that concept in a future post, but Danel Ahman and Tarmo Uustalu wrote up an incredibly brief summary (http://homepages.inf.ed.ac.uk/s1225336/papers/types14.pdf) of them.

Sadly, the MPTC for `RightMonoidAction` is rather annoying to use in practice. We might want `RightMonoidAction m m` for every `Monoid`, but we likely also want `RightMonoidAction s (Endo s)`, which results in overlap, incoherence and conflict. You can work around this to some extent with `newtype` noise.

Consequently, we won't actually be using the type given above, but we'll be applying it in

spirit.

One benefit of thinking in terms of update monads is that now you can expose all of the guts of your application, and nobody can violate your state change invariants anyways. This resolves the false dichotomy between encapsulation and code reuse (https://www.reddit.com/r/haskell/comments/2uoton/edward_kmett_encapsulation_vs_code_reuse/) for this one application domain.

A List of Successes

To keep the code approachable, I'll go back to the simplest and cleanest design of a parser I know. The title of this post is a riff on Philip Wadler's 1985 paper How to Replace Failure by a List of Successes (https://rkrishnan.org/files/wadler-1985.pdf) in which he talked about a very straightforward example of a parser that works particularly well in a lazy language.

So what does a "list of successes" parser look like?

```
data Parser a = Parser { runParser :: String -> [(a, String)] }
```

Today we'd recognize it as just `StateT String []`, but monad transformers didn't exist back then.

To try to get away from arbitrary state, we should ask ourselves the question, "what actions do we actually want to be able to apply to the `String`?"

Well, a nice parser will only ever drop characters, so we could switch out the state `String` for one with a monoid action on it such as

```
newtype Drop = Drop Int

instance Monoid Drop where
  mempty = Drop 0
  mappend (Drop a) (Drop b) = Drop (a + b)

instance RightMonoidAction [a] Drop where
  act s (Drop n) = drop n s
```

Note: evil parsers that do things like push back input they haven't seen cause problems for parser combinators that avoid backtracking on consumption unless under `try` and the like, such as Parsec, so this is a fairly sound assumption.

This is just the `Sum Int` monoid, with a carefully chosen action. The action is valid as long as we limit ourselves to non-negative drops in aggregate smaller than the maximum

size of an `Int`.

Now we can consider the corresponding "update monad transformer," which would give us something like

```
data Parser a = Parser { runParser :: String -> [(Drop, a)] }
```

But since we're using the notion of an update monad in spirit rather than in actuality we'll drop the newtype for `Drop` and just use `Int`. We'll agree to just "think" `Drop` really hard when we see it in the future.

```
data Parser a = Parser { runParser :: String -> [(Int, a)] }
```

The one thing that we've really gained here is that we can know that no action randomly replaces the input string with another string. They all consume the same source.

One win we could have, if `String` was replaced by `ByteString` (or `Text`) here, is that you could exploit this to allow the user to recognize an identifier using any combination of actions and then `slice` the `ByteString` (or `Text`) over the range that sub-parser matched. This would give you efficient sharing with the source, rather than breaking the input down into characters and then rebuilding a new bytestring that shares nothing. Slicing in `trifecta` is done in this spirit, but on fingertrees of bytestrings instead.

Moreover, we've gained something else critical. We've gained information about exactly how many characters we've consumed in a way that could let us work smarter for actions in the applicative and for `(>>)`.

A Heap of Successes

If we grouped the results up by the `Int` worth of characters we are dropping, this would tell us the offset of everything in that group, regardless of parse result.

We could do this with something like:

```
data Parser a = Parser { runParser :: String -> IntMap [a] }
```

But what we really want is cheap access to the next element, not random access so this sounds a lot more like a heap to me.

We could go grab something like a pairing heap from an older post I wrote on Heaps of Performance (https://www.fpcomplete.com/user/edwardk/revisiting-matrix-multiplication/part-5), or grab a more standard heap construction.

For now, I'm going to punt on that and decide to proceed with a simple `[(Int, a)]` representation where I maintain one invariant: The list is sorted by the number of elements we're dropping.

```
type Heap a = [(Int, a)]
```

When I'm referring to a heap for now, I'll be referring to a list in this form. A sorted list is a *unary* heap replete with the heap property and everything, so I'm not even lying!

Feel free to replace it with something with better performance characteristics, though.

We can merge heaps:

```
-- fair interleaving, because, well, why not? i guess we'll see.
merge :: Heap a -> Heap a -> Heap a
merge [] as = as
merge as [] = as
merge aas@(a:as) bbs@(b:bs)
  | fst a <= fst b = a : merge bbs as
  | otherwise      = b : merge bs aas
```

and we can `gather` results by key.

```
gather :: Heap a -> Heap [a]
gather [] = []
gather ((i0, a0) : as0) = go i0 [a0] as0 where
  go i acc [] = [(i,acc)]
  go i acc ((j, a) : as)
    | i == j    = go i (a:acc) as
    | otherwise = (i, acc) : go j [a] as
```

So, let's write a parser:

```
newtype Parser i o = Parser { runParser :: [i] -> Heap o }
  deriving Functor
```

`State s a` is neither covariant or contravariant in `s`, because `s` occurs in both positive and negative position, but we can parameterize the Parser covariantly on its input type, because unlike the usual `Parser` we can make this a `Profunctor`: The `i` only occurs in negative position, and the `RightMonoidAction` on the lists we read as input for `Drop` doesn't care about the type parameter of the list.

```
instance Profunctor Parser where
  dimap f g (Parser m) = Parser $ map (second g) . m . map f
```

But we can start to see a bigger win when it comes to the `Applicative` instance:

```
instance Applicative (Parser i) where
  pure a = Parser $ \_ -> [(0,a)]
  Parser mf <*> Parser ma = Parser $ \s0 -> go 0 s0 (gather (mf s0)) [] where
    go i s ((j, fs) : fss) acc
      | s' <- drop (j-i) s = go j s' fss
                           $ merge acc
                           $ ma s' >>= \(k,a) -> fmap (\f -> (j+k,f a)) fs
    go _ _ [] acc = acc
```

In the expression `m <*> n`, the parser `n` doesn't care about the value returned by `m`. It only cares about how many characters it consumed. So what we do here is:

1. First, gather up all parses of the same length.
2. Starting with an empty heap as an accumulator, loop over the different lengths of parses in ascending order, dropping the delta from the previous drop from a working state, and feeding it to the second parser. When we're done we take the heap, shift everything in it by the number of elements we dropped to get there, and merge it with an accumulator.
3. Emit the accumulated heap.

```
instance Alternative (Parser i) where
  empty = Parser $ \_ -> []
  Parser m <|> Parser n = Parser $ \s -> m s `merge` n s
```

The `Alternative` just merges the two result heaps.

Turning to the `Monad`, we don't get the benefits we had with `(<*>)` in the `(>>=)` case. Subsequent parsing steps can now care about the values seen so far, so `gather` doesn't help. We can still at least drive the subsequent parses by dropping incrementally, as before, though.

```
instance Monad (Parser i) where
  return a = Parser $ \_ -> [(0,a)]
  Parser ma >>= amb = Parser $ \s0 -> go 0 s0 (ma s0) [] where
    go i s ((j, a) : fss) acc
      | s' <- drop (j-i) s = go j s' fss
                           $ merge acc
                           $ (\(k,b) -> (j+k,b)) <$> runParser (amb a) s'
    go _ _ [] acc = acc
  (>>) = (*>)
  fail _ = empty

instance MonadPlus (Parser i) where
  mzero = empty
  mplus = (<|>)
```

With that we can go and write instances to make this compatible with the operations in my `parsers` package:

We can do basic parsing:

```
instance Parsing (Parser i) where
  -- do or do not, there is no try
  try = id
  m <?> _ = m
  unexpected _ = empty
  eof = Parser $ \s -> case s of
    [] -> [(0,())]
    _  -> []
  notFollowedBy (Parser m) = Parser $ \s ->
    if null (m s)
    then [(0,())]
    else []
```

We can recognize characters:

```
instance (c ~ Char) => CharParsing (Parser c) where
  satisfy p = Parser $ \s -> case s of
    c:_ | p c -> [(1,c)]
    _         -> []
```

And this parser can gracefully support `lookAhead`.

```
instance LookAheadParsing (Parser i) where
  lookAhead (Parser m) = Parser $ \s -> first (const 0) <$> m s
```

And we can write a helper combinator that converts to a more traditional "list of

successes" form.

```
parse :: Parser i o -> [i] -> [(o, [i])]
parse (Parser m) s0 = go 0 s0 (m s0) where
  go i s ((j,o):xs) | s' <- drop (j-i) s = (o,s') : go j s' xs
  go _ _ [] = []
```

Caveats

The `Update` monad conversion isn't free. Left associated binds in an `Update` monad now apply two different actions to the same source, whereas normally we apply the two actions in series. Compare

```
(drop 10000 s, drop (10000+123) s)
```

with

```
let s' = drop 10000 s in (s', drop 123 s')`
```

You'd generally rather have the latter, so `State` still has a reason to exist!

Here we mitigate that to some extent by gathering all of our updates in a particular order that lets us share some work between them to reduce this cost when working across multiple results, but we still run into this for left associated binds.

There are several workarounds an update monad might employ to control for this phenomenon:

- We could also work around this by hitting this with the `Codensity` monad to force everything into right association (http://comonad.com/reader/2011/free-monads-for-less/).
- Or we can work smarter by finding a better representation for our set or Monoid. For example, if we had a cheaper drop by using Leonardo (https://www.fpcomplete.com/user/edwardk/fibonacci/leonardo) or skew-binary (https://www.fpcomplete.com/user/edwardk/online-lca#skew-binary) random access lists to reduce it to *O(log n)*, then we could greatly reduce the costs of applying our monoid action.

On Leftovers

Given the structure above, we *could* define a `Category` for parsing with (in part)

```
instance Category Parser where
  id = Parser $ \ case
    a:_ -> [(1,a)]
    _   -> []
  ...
```

Then we can drive `(f . g)` by having the right parser consume input and spit out a single output token, and repeating to generate a lexer that generates tokens consumed by the next parser. This is problematic though, we're generating a list of lists of tokens due to the non-determinism possible in the lexing phase, and they share prefixes, not suffixes. This is much better served by switching away from the simple parser model we have here where we know everything up front, and instead starting to talk about something like a `Conduit`, `Iteratee`, `Pipe` or `Machine`.

Exercise: Why can't we have `Arrow`, `Strong`, `Choice`, etc?

Now, I don't like leftovers. This same mechanism whereby we spot the fact that we are only doing a limited form of update (e.g. dropping) extends to scrapping "leftovers" from these streaming models as well as models where we know the entire state up front. We just need an appropriate notion of "action".

Conclusion

To get here what did we do? We used the fact that an update monad can have a smaller, simpler update language than a `State` monad. Then we took an existing `StateT`-based monad for parsing and converted it to an appropriate `UpdateT` monad. Afterwards we realized we could impose an invariant on the order it gave back its results in the list to gather up like updates and enable us to have a more efficient `Applicative`.

I leave as an exercise the task of swapping out `[(Int, a)]` for a min-heap. We don't need to maintain a full sorted list. We can punt the task of sorting to a heap.

I also leave as an exercise how to extract the proper leftmost parse first. `merge` being needlessly fair destroyed this property along with the truth of the `Monad` laws (unless you quotient out the order in which you get results). Heaps will also destroy this property without some extra tagging to help resolve ties. `gather` also needs to reverse the accumulators to ensure this property holds. I'm not bothering with either of these things in the code above.

Exercise: Why would we need to tag? How could we do it?

Moreover, I've only converted a fairly simple parser to this format above. Doing so with

proper error handling is a more complex affair.

I've collapsed the code above into a single gist (https://gist.github.com/ekmett/578eaf3e5a37f7315e6c).

-Edward Kmett (mailto:ekmett@gmail.com)

May 1, 2015

/share?url=https://www.schoolofhaskell.com
/user/edwardk/heap-of-successes)
