The Monad.Reader Issue 17

by Douglas M. Auclair ⟨dauclair@hotmail.com⟩
and Neil Brown ⟨neil@twistedsquare.com⟩
and Petr Pudlák ⟨petr.mvd@gmail.com⟩

January 9, 2011

Brent Yorgey, editor.

Contents

Editorial

by Brent Yorgey ⟨byorgey@cis.upenn.edu⟩

You may have noticed this edition of The Monad.Reader is rather late. This is because for the past eight months I have been the prisoner of giant, evil, imperative-programming-obsessed naked mole rats. I have managed to stay connected to the outside world by training earthworms to transmit IP packets, but as you may imagine, this is rather slow. The earthworms have been working nonstop to bring you this issue, which consists of three fun articles I hope you will enjoy: Doug Auclair explains the ins and outs of **difference lists**, a functional list representation allowing fast concatenation and appending; Neil Brown presents his new **InterleaveT** abstraction for flexible interleaving of streams of behavior; and Petr Pudlák explains **abstraction elimination** – the process carried out by lambdabot's `@pl` command – in terms of the `Reader` monad.

By the way, if you happen to meet "me" in real life, beware! It is actually an evil robotic replica.

List Leads Off with the Letter λ

by Douglas M. Auclair ⟨dauclair@hotmail.com⟩

The list data type provides a powerful way to operate with a collection of objects using a very simple syntax. For the most part, this is enough, but for specialized cases the structure of the list type makes processing slow or difficult. This article examines the list data type through the functional lens, allowing us to look at lists differently and increasing their expressivity and power without sacrificing their syntactic simplicity.

I was revisiting the Project Euler [1] problems recently, and one of the problems addressed lexicographic permutations.

Well, problem solved if you're using C++ [2] or other languages that have `lexicographic_permutation` packaged with their standard library set. Some languages do and some don't, but that discussion is not à propos to this article, nor even what lexicographic permutations are or how to go about generating them [3].

But something common enough did come up in the solving of that problem, which is the issue of appending to the end of lists effectively.

Now, I have a prejudice: I love lists, and by that I mean to say that in more than a few functional and logical programming languages, the List Data Structure (uttered with reverence, and with not even (barely) a hint of mocking) provides a facility for grouping and then operating on sets of objects that I find unparalleled in other programming languages and paradigms. In Haskell, for example, I can create a list simply by typing `[0..9]` or `["Mary", "Sue", "Bob"]`, and if I'm feeling old-fashioned and gutsy, I can iterate through those lists inductively with the code in Figure 1.

Or if I wish to embrace the modern era of programming (like, oh, since the 1970ish-es [4]), I can use `map` or `fold` to go anywhere I want to with a list. I mean, come on! `fold` is, after all, the "program transformer function". If I don't like my

```
> doSomething []    = []
> doSomething (h:t) = f h : doSomething t
```

Figure 1: Example of a (hand-constructed) function operating on a list

list as it is now (or if I don't even like it being a list at all), I just `fold` over it until it's exactly how I want it to be.

For example, the above function is really just a specialization of `map`, isn't it? `map` could be implemented as in Figure 2:

```
> map _ []    = []
> map f (h:t) = f h : map f t
```

Figure 2: An implementation of the function map

So, `doSomething` is really just `map f`, as Figure 3 shows:

```
> doSomething = map f
```

Figure 3: doSomething as a specialization of the map function

One could take this into even more general territory, as `map` is a specialization of `fold`, as shown by Figure 4. In fact, hold onto the composite function `((:) . f)`, as you will see it come up again in relation to operating with functional lists.

```
> map f = foldr ((:) . f) []
```

Figure 4: map is a folding function

So, yes, lists rock my world: they are easy to construct and easy to operate on.

Now, you may say that in Programming Language X, lists are one of the standard collection data types. This comes as no surprise. The need for lists is universal, so each programming language has its own implementation. Where functional and logic programming languages excel (like, for example, Lisp, Haskell and Prolog) is the ease with which lists are created and destructured. I've also used Programming Language X quite a bit myself, and I find list processing not a pleasure but a chore, for to construct the self-same list of names in this example programming language, I find myself required to do something like Figure 5. Destructuring requires a

```
BidirectionalLinkedList<String> list
     = new BidirectionalLinkedList<String>();
list.add("Mary");
list.add("Sue");
list.add("Bob");
```

Figure 5: List construction in Programming Language X

```
BidirectionalLinkedList<String> newlist
     = new BidirectionalLinkedList<String>();
for(String elem : list) { newlist.add(doit(elem)); }
```

Figure 6: List processing in Programming Language X

similar level of effort (Figure 6).

And this implementation is only after the stunning advances of the STL, grafting functionals onto the language with type parameterization. This first became publicly available for C++ around 1994 [5] and was later adopted by other similar languages when they expanded to include type parameterization.

Which is all well and good, but have you seen the work necessary to create the correct types in the parameters? And heaven help you if you get the declaration wrong!

Enough of that.

So lists are structures or collections, and structures can be viewed as objects, and that is a very useful and powerful way to view them and to define them:

```
> data List t = [] | (t : List t)
```

Figure 7: List declaration and definition

No problem with that, and everything is right with the world.

. . . until it isn't.

This definitely works, and works well, for lists in general, and it also works great most of the time for working with the elements of the list. After all, in practice, we are most concerned with the element we have worked on most recently, so, in most cases, the element we just put in is the element we'd most likely to be retrieving . . . AND (generally) our interest diminishes the further (in time) we are from an element, and that translates directly into where elements are found in a list.

Generally.

So, in general, lists work just fine; great, in fact.

There are specific cases where what we care about is not the most recent element, but another ordering is important. Two cases spring immediately to mind: first, queuing, and secondly (and commonly and specifically), document assembly.

In these two cases we wish to push onto the end of the list new elements, and the above definition doesn't give us access to the last element or position of the list. And to get that access, we must reverse the list so the last element becomes the first, or prepend to a reversed list. Either of these options has at least a linear time cost when `reverse` is (eventually) called.

Otherwise, we must call `(++)` (the list concatenation function) or a function like it to append elements to the end of the working list; this also has linear cost. Either way, we pay the full price.

Now, there are objects that do give us immediate access to that last element and position: deques and queues, for example. But when we go there, we give up the ease of composition and decomposition that we have with lists. We pay a penalty in expressivity with these types, or in performance when using lists against their nature.

Or do we?

Well, one way of looking at lists is as objects, and above we gave a definition of lists as objects, but this is not the only way to view lists. I propose another way of looking at lists: lists can be viewed as functions [6] as we see in Figure 8.

```
> (:) :: t -> [t] -> [t]
```

Figure 8: Definition of the list constructor function

The above definition says that `(:)` (pronounced "cons") takes an element and a list and gives a list. This is a well-known function in the list-lexicon, so what's the magic here?

I propose we look at this function in an entirely different way, as in Figure 9:

```
> (:) :: t -> ([t] -> [t])
```

Figure 9: The list constructor viewed functionally

In this case, `(:)` constructs a list **function** from a seed element. This allows us to use this function in a novel but perfectly acceptable way, as we see in Figure 10.

What we are saying here is that `|>` (pronounced "put to front") is an operator that takes an object `x` and puts that value on the front of `list`, just like `(:)` does.

```
> x |> list = (x:) . list
```

Figure 10: The "put to front" operator

The difference here (heh: "difference") (sorry, mathematician humor) is what `list` is. For `(:)` used normally, `list` is of type `[a]` (or, directly and tautologically: `list` is a list), but for `|>`, `list` is of type `[a] -> [a]`. Or, translating: `list` is a **function**.

Do you see significance here? Earlier I asked you to hold onto the the function that related `fold` to `map` which was `((:) . f)`. Now that we see `list` here is a function, we see that this function of the `map`–`fold` relation is of the same form for list-function construction in the points-free style. So, it looks like I've just reinvented the `(:)`-wheel here, expressing that function in a different way. So what's the big deal?

Well, what I've actually done is to reinvent the wheel as a meta-wheel, and so the big deal is as Figure 11:

```
> list <| x = list . (x:)
```

Figure 11: The "put to back" operator

What we have here with `<|` (pronounced "put to back") is a function that adds an element to the end of a list in – wait for it! – constant time. For "regular" lists the best you can do is linear time, and so my work of constructing a document by appending to the end of a list has just gone from an $O(n^2)$ to a linear-time operation. Not a big deal for a small document, but we found that once a document became more than a page or two, the operation went from "a blink of an eye" to "keeping your eyes closed until the cows came home . . . that had been eaten by wolves". This is not the case with lists as functions: document construction became a reasonable endeavor (that is, it occurred so quickly for us mere humans, living in this non-nanosecond-time, we didn't notice the elapsed time).

So, I've been a sly thing in one respect. I still haven't told you what a (functional) list is. I've defined the object view of lists, and I've declared what a functional view of lists looks like, but haven't defined them.

Here. Allow me to define them now in Figure 12.

There you go.

There are the "definitions". I say "definitions" because with that sole definition, everything works, for to ground a functional list, we simply pass it an empty list, as Figure 13 shows.

```
> empty = id
```

Figure 12: The "definitions" of functional lists (specifically: the empty list)

```
*Data.DiffList> empty []
[]
```

Figure 13: Converting a functional list to a "ground" list

Or even whatever list we are working with as Figure 14 shows:

```
*Data.DiffList> empty [1,2,3]
[1,2,3]
```

Figure 14: Appending a "ground" list to a functional list

Do you see what's going on here? Because I missed it the first time myself: `(++)` needs no definition for functional lists, for to append a ground list to a functional list, all that is required is application. And to append a functional list, all that is required is function composition [7].

So, when you append something to an empty list, you get that something back, and empty works as the seed of the 'put to' operators as Figure 15 shows.

It all works!

Summary

In this article we've demonstrated that there's more than one way to look at lists. The standard way is to view them is as objects, and in most cases, this works, and works well: this view provides a syntax that makes list processing simple and intuitive.

We've also shown that that is not the sole way to view things, and this view can be constraining, particularly when lists are played against type as a queue or a document assembler. In these cases it becomes simpler, declaratively, to view lists as functions, and not only does that provide a syntax for simple list construction either at the beginning or the end of the list, but it also provides constant-time construction at either end. Furthermore, defining this view is as simple as viewing the empty list as `id` and then using the partial function of `(:)` ("cons"). That's the generative view. Then, to extract the ("real" or "ground") list from that view,

```
*Data.DiffList> (5 |> 6 |> 2 |> empty) [1,2,4]
[5,6,2,1,2,4]
(empty <| 1 <| 2 <| 3) [4,5,6]
[1,2,3,4,5,6]
```

Figure 15: Concatenation with putting elements to the front and the back of functional lists

it's as simple as sending a list (for example, `[]`) to that functional list to ground the value for disposition.

In these special cases of working at the end of a list, which aren't all that rare, the functional view of list processing gives the programmer expressivity and efficiency, eliminating the chore of appending to the end or reversing the target list. I quite enjoy list processing this way: it gives me a fresh perspective on an old friend and makes list processing in these cases easy and fun.

About the Author

Douglas M. Auclair has built rule-based systems that processed over one million transactions per day as well as mission planning software for satellite image capture and processing. His favorite rejoinder to the rallying cry of "C'mon, folks, you don't have to be a rocket scientist to figure this out!" is "... but it sure helps." – Yes, he has friends in NASA, too.

He is currently working on the automation of extraction of cohesive ontologies from documents to allow for anything from querying the knowledge base to testing the strength of suppositions to draw (or to refute) conclusions. Neat stuff!

He maintains a Haskell-ish/Mathematical-ish blog at `http://logicaltypes.blogspot.com/`.

References

[1] `http://projecteuler.net/`

[2] `http://www2.research.att.com/~bs/C++.html`

[3] There is an article at `http://wordaligned.org/articles/next-permutation` that provides a simple, clear explanation of the lexicographic permutation algorithm with a very nice demonstrative example if you are so inclined to investigate.

[4] Laurent Siklóssy, **Let's Talk Lisp** talks about `MAPCAR` in chapter 6. And, if I may: this book is one of the rare ones. It's one of those books like Smullyan's **To Mock**

a Mockingbird or van der Linden's **Deep C Secrets** (with a **fish** on the orange book cover, no less!) or, and let us not forget the ultimate: Hofstadter's **Gödel, Escher and Bach: the Eternal Golden Braid** (I mean, even the **title** is a pun; Doug doesn't waste time in having fun, does he? He gets right to it!) that make you a better person but you don't notice it, but everybody else does, because you are constantly busting out laughing or grinning from ear-to-ear at the cleverness of their prose. Why do books about esoterica have to be so **heavy**! These esoteric books show that they don't have to be heavy in the lightness they deal with the subject (and deal with dealing with these esoteric subjects).

[5] `http://www.sgi.com/tech/stl/`

[6] Such a view of lists has a name: these functional list types are called difference lists from the Prolog community. The standard Haskell library does not have `Data.DList` – sad loss, I say! – but this type is provided by Don Stewart of Galois in the `dlist` package on Hackage.

`Data.DList` works great, but I've implemented my own system that is more than twice as fast, as I provide several specialized implementations of difference list operators that take a more direct approach in their implementations (`foldr` is elegant but can be a costly operation). If you'd like to use my `Data.DiffList` package, I've included the implementation with this article, available from `http://code.haskell.org/~byorgey/TMR/Issue17/DiffList.lhs`.

[7] Smullyan's **To Mock a Mockingbird** has a functional representation of numbers in combinator logic. There is also a paper available at `http://dkeenan.com/Lambda/` called **To Dissect a Mockingbird** that has a section on the end of the paper on functional numbers and logic.

The set of functional numbers in combinator logic has the interesting property that multiplication is simply application (or juxtaposing one number to another, just as is often expressed in regular algebra). Here, for functional lists, simply composing (functional) lists gives us concatenation for free, just as it is expressed in the common-sense "real world". We would think that `[1,2,3] . [4,5,6]` results in `[1,2,3,4,5,6]`, and for functional lists, this is precisely the case. Neat!

The InterleaveT Abstraction: Alternative with Flexible Ordering

by Neil Brown ⟨neil@twistedsquare.com⟩

The Alternative type class in Haskell supports the notion of choice between items such as parser actions or channel communications. If, instead of making a one-off choice between items, you need to choose several in an unknown order (e.g. parsing different sections in a file which can be in arbitrary order), the resulting code spelling out all the orderings can become quite intricate. This article explains InterleaveT, a simple combinator abstraction on top of Alternative and Monad which captures the flexibly-ordered interleaving of several streams of behaviour at a high level, without resorting to writing in a state machine-like style.

Introduction

Programs sometimes need to engage in different behaviours in an unknown order. For example, parsing `"top left"` or `"left top"` without worrying about the order of the two words, or communicating with thread A and thread B, without minding who comes first. This is almost invariably programmed by waiting for a choice of the events, and based on which happens first, deciding what will be done next. For example, if the parser starts by accepting `"top"` *or* `"left"`, and finds `"top"`, it will then only look for `"left"`.

When the choices become more complicated, for example, accepting one of `"front"` or `"back"` with one of `"top"` or `"bottom"` and one of `"left"` or `"right"`, spelling out all the options is verbose, combinatorially large, and often features duplication. All of these make the code harder to maintain.

This article introduces the InterleaveT abstraction, which uses a combinator-based approach to deal with different orderings (with a slightly more flexible approach than previous solutions in parsing). Parsing is a well-known example, but

the approach has utility in any setting that uses choice: we will see examples featuring channel communications, software transactions and card games.

The combinators introduced in this article are solely based on the existing Monad and Alternative type classes; anything that is an instance of both can use the combinators straight away. We start with a recap of these type classes, then explore the combinator API, examples of their use, and finally the implementation. The code is available on Hackage as the `interleave` library [1].

Type Class Recap

This article will make use of several existing type classes in the `base` library which can have instances for some type constructor `f` with kind $* \to *$, such as Maybe, [] and IO. To briefly recap:

- Functor: Gives you `fmap :: (a -> b) -> f a -> f b`, also written infix as `<$>`, which allows you to apply a pure function to a wrapped value.
- Applicative: Gives you `pure :: a -> f a`, which allows you to wrap a value, and the `(<*>) :: f (a -> b) -> f a -> f b` operator, which (together with `pure`) allows you to apply a function to several wrapped values.
- Monad: Gives you the `(>>=) :: f a -> (a -> f b) -> f b` operator and **do**-notation.
- Alternative: Supports a choice operator `(<|>) :: f a -> f a -> f a` and a never-available choice, `empty :: f a`.

These classes effectively form a hierarchy:

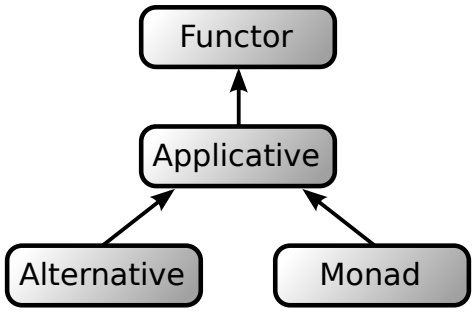

Figure 1: The effective hierarchy of type classes. For historical reasons, Monad does not actually depend on Applicative, but every Monad can be made an Applicative.

The Monad and Alternative classes are orthogonal: types that have an instance for Applicative may be instances of just one of Monad and Alternative, of both, or of neither. This article is concerned with types that are instances of *both*.

A Starting Example

As a motivating challenge, think about writing a regular expression that accepts a string with an unlimited number of `'a'`s, at most 6 `'b'`s and ends on the second `'c'`. (This is, in effect, the *interleaving* of three different behaviours.) With classic regular expressions you may end up in a horrible mess like this:

```
a*(c|ca*b|ca*ba*b|...|ba*ca*b|ba*ba*c|...)a*c
```

Regular expressions aren't well suited to this challenge, admittedly (now you have two problems, right? [2]). Consider the challenge in Parsec [3] instead; a typical solution counts the number of `'b'`s and `'c'`s found:

```
abcCount = abcCount' (0, 0)
  where
    abcCount' counts@(bCount, cCount)
      | cCount == 2 = return ()
      | bCount == 6 = (       (char 'a' >> return counts)
                        <|> (char 'c' >> return (bCount, cCount + 1)
                      ) >>= abcCount'
      | otherwise = (       (char 'a' >> return counts)
                      <|> (char 'b' >> return (bCount + 1, cCount))
                      <|> (char 'c' >> return (bCount, cCount + 1)
                    ) >>= abcCount'
```

The other approach, which this article covers, is to move to a higher level of abstraction that allows the problem to be specified in an obvious way by using combinators on top of Parsec:

```
engageMany
  [unlimited_ (char 'a')
  ,upTo 6 (char 'b')
  ,endAfter (upTo 2 (char 'c'))]
```

The next section explains the API that features the above combinators.

API

We will define an API for combinators that sit on top of an existing Monad and Alternative type, typically labelled `f`, and allow easy specification of interleaved behaviours. This section defines the API and behaviour of the types and functions; we will return to the implementation later on. We will start by defining a new wrapper/transformer type for supporting our combinators:

```
data InterleaveT f a = ...

instance Functor f => Functor (InterleaveT f) where ...
instance Alternative f => Applicative (InterleaveT f) where ...
```

The f parameter is the Monad/Alternative that we are wrapping, and the a parameter is the return type. Note that InterleaveT is an instance of Functor and Applicative but not Monad or Alternative. At the outer-most point we run such an item with the engage function:

```
engage :: Monad f => InterleaveT f a -> f a
```

Unlimited Repetition

The unlimited combinator allows an action to be performed an unlimited number of times:

```
unlimited :: Functor f => f a -> InterleaveT f [a]
unlimited_ :: Functor f => f a -> InterleaveT f ()
```

If we use engage to run the unlimited combinator, we will perform the given action repeatedly without ever terminating. This gives rise to an obvious law:

```
engage (unlimited_ x) == forever x
```

Limited Repetition

There are many functions that are all variants on the same theme: performing particular actions a limited number of times:

```
once :: Functor f => f a -> InterleaveT f (Maybe a)
upTo :: Functor f => Int -> f a -> InterleaveT f [a]
inOrder :: Functor f => [f a] -> InterleaveT f [a]
```

Note that all these functions indicate an upper limit on the amount of times the action may be executed, but not a lower limit. It is possible, via combinators for terminating the engage call early, that the actions may not take place at all (hence the Maybe return type of once). These combinators have various laws:

```
engage (once x) == Just <$> x
once == fmap listToMaybe . upTo 1
upTo n == inOrder . replicate n
engage (inOrder xs) == sequence xs
```

Note that when there are no actions left in a limited repetition, engage terminates. If you have an unlimited repetition, you never run out of actions and the call to engage will not terminate.

It is important to note the difference between once (sequence xs) and inOrder xs. The former offers sequence xs as a single choice to the Alternative instance, and if chosen will execute all of xs before the next item in the interleaving. In contrast, inOrder xs will begin by offering just the first item in the list xs to the Alternative instance. If this is chosen, then next time it will offer the second item in xs and so on. So once (sequence xs) will not interleave xs with anything else, but inOrder xs will interleave the list with all the other items being offered.

Interleaving

So far, so simple. This is not yet interleaving – just a single stream of actions. The combinator that really unlocks the power of the approach is alongside, which allows composition of two streams of behaviour:

```
alongside :: Alternative f =>
  InterleaveT f a -> InterleaveT f b -> InterleaveT f (a, b)
alongside_ :: Alternative f =>
  InterleaveT f a -> InterleaveT f b -> InterleaveT f ()
alongsideMerge :: (Alternative f, Monoid a) =>
  InterleaveT f a -> InterleaveT f a -> InterleaveT f a
```

This combinator is the one that actually interleaves different behaviours. It is worth noting that these alongside functions inherit the bias of the <|> operator. Most instances of Alternative use left-bias, so alongside is usually left-biased. Each individual action from either side of the alongside combinator may execute next, and afterwards the appropriate choices are offered again. Our next law indicates some of its behaviour:

```
engage (unlimited_ x `alongside_` unlimited_ y) == forever (x <|> y)
```

The laws for anything more complicated are too long to express – which is the whole motivation behind using the combinator approach! For example, consider the code:

```
engage (once x `alongside_` unlimited_ y)
```

This offers x <|> y repeatedly, but once x happens, it only performs y forever after. We will look at more examples of the combinator later on.

We can also combine alongside with engage to offer a more convenient function:

```
engageMany :: Alternative f => [InterleaveT f a] -> f [a]
```

This combinator takes a list of actions, runs them all alongside each other and returns the results in a list with corresponding order. Note that any alongside or engageMany call with an unlimited repetition in it will never run out of actions, and thus will never terminate without additional terminator combinators.

Termination

Termination can be explicitly instigated by wrapping a particular action with the endAfter combinator:

```
endAfter :: Functor f => InterleaveT f a -> InterleaveT f a
```

Whenever the given item has no more actions to perform, the entire combinator system that has been passed to the outer engage function will terminate. Since unlimited always has more actions to perform, wrapping an unlimited behaviour with endAfter has no effect. This all gives rise to our next laws:

```
endAfter (unlimited x) == unlimited x
engage (endAfter (once x)) == Just <$> x
engage (endAfter (once x) `alongside_` endAfter (once y)) == Just <$> (x <|> y)
engage (endAfter (once x) `alongside_` unlimited_ y) ==
  let m = (y >> return True <|> x >> return False) >>= \b -> when b m) in m
```

The combinators are becoming more powerful – which is indicated by the right-hand side of our laws becoming longer than the left! The latter example can be used to, for example, parse a sequence of unlimited 'a's ended by a 'b':

```
engage (unlimited_ (char 'a') `alongside_` endAfter (once (char 'b')))
```

If you instead want to terminate when a particular state is reached, you can use the endWhen function:

```
endWhen :: Functor f => (a -> Bool) -> InterleaveT f a -> InterleaveT f a
```

This acts like the given event but when the current return value satisfies the given function, the whole interleaving terminates. For example, you may want to accept up to six 'a's, unlimited 'b's, and terminate when there are more 'b's than 'a's:

```
engage $ endWhen (\(as,bs) -> length bs > length as)
  (upTo 6 'a' `alongside` unlimited (char 'b'))
```

State

A further combinator generalises unlimited to allow explicit state passing:

```
unlimitedRecurse :: Functor f => (a -> f (b, a)) -> a -> InterleaveT f [b]
unlimitedRecurse_ :: Functor f => (a -> f a) -> a -> InterleaveT f ()
```

The first allows output (besides the state) to be generated for the result. It is easy to use these to support a state monad transformer. For example, using StateT from the `transformers` library:

```
embedStateT :: StateT s f a -> s -> InterleaveT f [a]
embedStateT m x = unlimitedRecurse (runStateT m) x

embedStateT_ :: StateT s f a -> s -> InterleaveT f ()
embedStateT_ m x = unlimitedRecurse_ (runStateT m) x
```

This makes programing behaviours with state simpler, and allows the use of a more familiar API (the state-transformer monad). For example, you may want a parser that checks properties of pseudo-randomised sequences (*e.g.* trials in a psychology experiment) where you do not permit more than three identical letters in a row:

```
checkSeq :: StateT (Char, Int) ParseM ()
checkSeq = do (c, n) <- get
              c' <- if n < 3
                      then letter 'A' <|> letter 'B'
                      else if c == 'A' then letter 'B' else letter 'A'
              put $ if c == c' then (c, n + 1) else (c', 1)

checkSeqIT :: InterleaveT ParseM ()
checkSeqIT = embedStateT_ checkSeq (undefined, 0)
```

Note that the checkSeq parser is written solely using a ParseM monad and the StateT monad, without knowledge of the InterleaveT abstraction, but checkSeqIT wraps it simply into an InterleaveT item that can be interleaved with other actions.

Examples

The previous section showed the API of our combinators. In the following subsections we will look at a few examples of using the combinators.

Communicating Haskell Processes

Communicating Haskell Processes (CHP) is a library for message-passing [4]. The Alternative instance can be used to choose between reads and writes on different channels, as well as synchronisations on barriers.

CHP can be used to write simulations, often following a "tick" pattern; the simulation is separated into discrete time-frames by a global tick event that all agents in the simulation synchronise on together. Between the tick events, the agents carry out all the actions that they wish/can. This is usually managed by having each agent offering all the actions that they could perform, alongside the lower-priority tick event. If two agents both offer to either perform event a or perform the tick event, the higher priority of the a event (or rather, the lower priority of the tick event) ensures that they will perform a. The agents will then subsequently offer the tick event.

In code, this simple case might look like this, without InterleaveT:

```
((True <$ syncBarrier a) <|> (False <$ syncBarrier tick))
  >>= flip when (syncBarrier tick)
```

(This definition uses a handy little function from the Data.Functor module:

```
(<$) :: Functor f => a -> f b -> f a
(<$) x f = fmap (const x) f
```

If you ever find yourself writing m >> return x for a short monadic function, you can instead write x <$ m.)

With InterleaveT we have instead:

```
engageMany [once (syncBarrier a), endAfter (once (syncBarrier tick))]
```

The benefit increases when you need to add a third or fourth action. For example, in a particular blood clotting simulation [5], sites are connected to their neighbours via communication channels (over which platelets can be communicated, to move down the pipeline). The values communicated over the channels are Maybe Platelet; a Nothing value is sent when a site is empty during a time-step. Therefore, during a time-step an empty site must offer to read in platelets from the site behind it, and send the empty signal to the site ahead of it. This can be coded as:

```
fst <$> engage (once (readChannel prev)
                `alongside` once (writeChannel next Nothing)
                `alongside` endAfter (once (syncBarrier tick)))
```

Due to the right-associativity of alongside, you can use fst to pull off the return value of the left-most item: the value read from the prev channel above.

Parser

We saw earlier that parsing is one application of the InterleaveT abstraction. The InterleaveT pattern allows expression of multiple strands of parsing. For example, you may want to parse a file that has an optional definitions section, a mandatory

main section, and as many comment sections as you want. This could be expressed as:

```
engage $ once definitions `alongside` unlimited_ comment
         `alongside` endAfter (inOrder [main, eof])
```

Similarly, you may want to parse a location such as "top" or "bottom left" or "right top" (*i.e.* any of eight locations, with the words in any order):

```
engage $ once (word "top" <|> word "bottom")
         `alongside` once (word "left" <|> word "right")
```

One important point to note about parsers is that for efficiency reasons, most major Haskell parsing libraries (*e.g.* Parsec [3], attoparsec [6]) provide an instance for Alternative that only switches to try the right-hand side if the left-hand side fails without consuming any input. Therefore examples like those above may require additions like a try combinator; this is not an issue added by the interleave combinators, but rather a consideration which was already present in using the parsing libraries.

Drawing Cards

The DrawM monad from the `game-probability` library [7] supports randomly drawing cards from a deck [8]. In fact, it supports not only actually drawing cards, but also calculating the probabilities of various outcomes. In addition to Monad, it is an instance of Alternative: if one draw cannot be satisfied by the next card, the alternative is tried instead.

Let's take an example from the card game Dominion [9]. There is an adventurer card which allows you to keep drawing cards until you have drawn two treasure cards. This is easy to express using our combinators:

```
engage $ unlimited (drawWhere (not . isTreasure))
         `alongsideMerge` endAfter (upTo 2 (drawWhere isTreasure))
```

This function gives back a list of the cards drawn. This code can be used either to draw cards (for example, in a computerised version of the card game) or to calculate the chances of getting different numbers of cards given a particular starting deck.

Software Transactional Memory

Software Transactional Memory (STM) allows multiple transactional variables to be manipulated as part of a single atomic transaction. Using STM, it is possible to construct buffered communication channels for communicating between different

threads. However, this is subtly but crucially different from the CHP example above. An STM transaction fetches data from channels in one transaction. The data is either already there or it is not; the transaction will process it all at once, atomically. This is in contrast to CHP, which performs one communication at a time, and cannot roll back previously performed communications afterwards.

Note: At the time of writing, STM has no Applicative or Alternative instances. It is trivial to define these in your own code for now:

```
instance Applicative STM where
  pure = return
  (<*>) = ap
instance Alternative STM where
  empty = retry
  (<|>) = orElse
```

The STM library has a TChan abstraction, which represents a buffered channel. The readTChan ::TChan a ->STM a function reads from such a channel, failing if the channel is currently empty. We can use our combinators to wrap calls to this function. Let's imagine that you want a transaction where you read everything possible from channel req, one item from channel control (if present), and only succeed if you can also read from channel update. That can be coded as:

```
engage $ unlimited (readTChan req)
         `alongside` once (readTChan control)
         `alongside` endAfter (once (readTChan update))
```

Note that this code makes crucial use of the left-bias that the alongside operator has inherited from STM's <|> implementation (orElse). The combinators will first drain the req channel, and only when a read from that fails (due to it being empty) will control be tried, and after that (successful or not), update will be read from. Note that if the read from update fails, the semantics of STM mean that the entire transaction will retry, meaning that it will sleep until at least one of the channels are written to, at which point it will try the whole thing again from scratch. From the point of view of the process, the transaction simply does not complete until the interleaved behaviour has terminated (by successfully reading from the update channel). This is again a case of InterleaveT inheriting the semantics of its underlying monad. Similar code in the CHP monad will not cancel the earlier reads, and will simply wait for the next communication to take place.

Error Transformers

The InterleaveT abstraction also works well with error monads (or error monad transformers), also known as exception monads. There are many different error

monad transformers; for the purposes of illustration, we will define a typical example:

```
newtype ExT e m a = ExT { runExT :: m (Either e a)}

instance Monad m => Monad (ExT e m) where
  return = ExT . return . Right
  (>>=) m k = ExT $ runExT m >>= either (return . Left) (runExT . k)

abort :: Monad m => e -> ExT e m a
abort = ExT . return . Left

liftEx :: Functor m => m a -> ExT e m a
liftEx = ExT . fmap Right
```

The Functor, Monad and Applicative instances are trivial, but to be clear we give the Alternative instance below:

```
instance Alternative f => Alternative (ExT e f) where
  empty = ExT empty
  (<|>) a b = ExT (runExT a <|> runExT b)
```

This transformer can be used, for example, with the STM monad. You can define a helper function to run it at the top level:

```
atomicallyEx :: ExT e STM a -> IO (Either e a)
atomicallyEx = atomically . runExT
```

Now imagine that in the middle of an InterleaveT item in a transaction, you may want to terminate the entire transaction. STM itself offers no such facilities (retry does as its name suggests and will wait to retry, not abort), but by using ExT on top of STM we can terminate the transaction by returning an error/exception value.

The way that InterleaveT and ExT are implemented means that if an exception occurs in an action, the entire engage call terminates with the exception immediately. For example, you may have an application which reads Maybe values from a channel, but if it reads a Nothing value, that is a signal that the transaction should end. Adapting our earlier example, we can watch for Nothing values on the req channel and abort if one is found:

```
atomicallyEx $ engage $
  unlimited (readChanMaybe req)
  `alongside` once (liftEx $ readTChan control)
  `alongside` endAfter (once (liftEx $ readTChan update))
```

```
readChanMaybe :: TChan (Maybe a) -> ExT () STM a
readChanMaybe c = do x <- readTChan c
                     case x of
                       Nothing -> abort ()
                       Just x -> return x
```

This will attempt to read unlimited values from the req channel, at most one value from the control channel and end if it can then read from the update channel. If the update channel cannot be read from, the transaction will retry. If the transaction discovers a Nothing value on the req channel, either the first time or during any retry, it will abort the transaction regardless of the state of the update channel.

State Transformers

The InterleaveT abstraction can become more powerful (and more intricate) if combined with state monad transformers, such as the one found in the `transformers` library. For example, imagine we have a CHP system where we are willing to send out our status, and to accept new status values – but only, say, 3 updates per frame:

```
proc :: Chanin Status -> Chanout Status -> Status -> CHP Status
proc input output
  = fmap snd . runStateT (engage $ unlimited_ send `alongside_` upTo 3 recv)
  where
    send :: StateT Status CHP ()
    send = lift (get >>= writeChannel output)

    recv :: StateT Status CHP ()
    recv :: lift (readChannel input >>= put)
```

The two separate actions can be programmed separately, but interact via the state. The usefulness here stems from being able to easily alter (in the third line) how often each action can occur without altering the behaviour of the individual action, or its partner. That is, we could easily accept unlimited updates (or restrict the number of sends) without altering the code for send or recv.

Applicability and Testing

The InterleaveT abstraction is not useful for all types that have an Alternative instance. If the actions in the monad are deterministic (*i.e.* always have the same result), the combinators become trivial. For example, consider this code in the Maybe monad:

```
engage (unlimited_ x `alongside` endAfter (once y))
```

Both x and y are of type Maybe a. If x is Nothing, then this call is equivalent to y; if x is Just then the code will never terminate. So the combinators do not make sense here; they only make sense if the outcome of the events and choices between them can vary each time they are executed.

There is one exception to this, and that is for testing the InterleaveT abstraction. For this purpose we can use a writer-transformer monad on top of the list monad to get a list of possible options a behaviour would take. Our basic definitions are (using WriterT from the `transformers` library):

```
type M = WriterT String []

run :: M a -> [String]
run = execWriterT

action :: Char -> M ()
action c = tell [c]

a, b, c, d :: M ()
a = action 'a'
b = action 'b'
c = action 'c'
d = action 'd'
```

Then with the help of a function that checks for equality on sorted lists, we can write a test:

```
assertEqualS :: (Ord a, Show a) => String -> [a] -> [a] -> Assertion
assertEqualS msg = assertEqual msg `on` sort

assertEqualS "inOrder"
  ["abcd", "acbd", "acdb", "cdab", "cadb", "cabd"]
  (run $ engage $ inOrder [a, b] `alongsideMerge` inOrder [c, d])
```

This checks that running the given InterleaveT produces all six possible results we would expect. What this code is doing, in effect, is treating the InterleaveT abstraction as a grammar and then generating all possible strings – the dual of parsing, to which we have already seen InterleaveT applied. The only problem with this technique is that it cannot be used to support unlimited actions, as they will produce an infinite string (which cannot be properly checked for equality). Techniques such as observable sharing [10] could perhaps be used to circumvent this problem.

Implementation

In this section we will look at the implementation of the InterleaveT type and the associated combinators. The InterleaveT type is as follows:

```
data InterleaveT f a = Terminate { result :: a }
                     | NoMore    { result :: a }
                     | Continue  { result :: a, rest :: f (InterleaveT f a) }
```

The Terminate constructor indicates that the current interleaving should terminate with the given result regardless of what other choices are available. As we will see, this constructor is "viral"; it overrides Continue in all cases, so that if any combinator wants to terminate, then it will do so, no matter how deeply it is nested. In contrast, the NoMore constructor indicates that there are no further actions that could be executed in this branch, but if actions are available in other branches then the behaviour continues. The alternative to these is the Continue constructor, which has the result that would currently be returned if we stop now (due to a Terminate somewhere), and the choice to offer if we are instead continuing. Once executed, this rest choice will return the next set of choices to engage in.

The top-level engage function is very simple:

```
engage :: Monad f => InterleaveT f a -> f a
engage (Terminate x) = return x
engage (NoMore x) = return x
engage (Continue _ m) = m >>= engage
```

The work is all done by the combinators (and facilitated by the chosen representation); engage is a very hollow function.

The unlimited_ function is as follows:

```
unlimited_ :: Functor f => f a -> InterleaveT f ()
unlimited_ m = Continue () (unlimited_ m <$ m)
```

What the far-right-hand expression does is to execute the action m and then give back the result unlimited_ m as the next action to try to execute, and therefore unlimited always does exactly the same thing on each iteration of the interleaving. In figure 2, we take these definitions of engage and unlimited_ to derive one of our earlier laws.

The alongside combinator uses the Alternative instance of the wrapped type:

```
alongside :: Alternative f =>
  InterleaveT f a -> InterleaveT f b -> InterleaveT f (a, b)
alongside oa@(Continue a fa) ob@(Continue b fb)
  = Continue (a, b) ((flip alongside ob <$> fa) <|> (alongside oa <$> fb))
```

```
alongside (NoMore a) ob = (,) a <$> ob
alongside oa (NoMore b) = flip (,) b <$> oa
alongside oa ob = Terminate (result oa, result ob)
```

The result if we finish now is simply the pair of the results of the two sides. If both want to continue (the first case), we choose between the left-hand action (fa) – in which case we will pair it with the untouched right-hand side (ob) – and the right-hand action (fb), in which case we will pair it with the untouched left-hand side (oa). If either has no more actions, the result is simply added to the other side accordingly. If either or both wants to terminate (the last case), we propagate this accordingly. We use this definition to derive an earlier law in figure 3.

The endAfter function terminates when there are no more actions left to run; this translates to simply transforming the NoMore constructor into the Terminate constructor:

```
endAfter :: Functor f => InterleaveT f a -> InterleaveT f a
endAfter (NoMore x) = Terminate x
endAfter (Terminate x) = Terminate x
endAfter (Continue x m) = Continue x (endAfter <$> m)
```

The endWhen function generalises this to turn things into a Terminate constructor only when the given condition is satisfied:

```
endWhen :: Functor f => (a -> Bool) -> InterleaveT f a -> InterleaveT f a
endWhen _ (Terminate x) = Terminate x
endWhen f (NoMore x) = if f x then Terminate x else NoMore x
endWhen f (Continue x m)
  = if f x then Terminate x else Continue x (endWhen f <$> m)
```

The inOrder function is as follows:

```
inOrder :: Functor f => [f a] -> InterleaveT f [a]
inOrder [] = NoMore []
inOrder (m:ms) = Continue [] ((\x -> (x:) <$> inOrder ms) <$> m)
```

When the list of actions is empty, it returns NoMore with an empty list. Until then it Continues processing the list, and each result is added to the list of later results that results from future calls.

Instances

The Functor instance for InterleaveT defines fmap as a simple recursive function:

```
instance Functor f => Functor (InterleaveT f) where
  fmap f (Terminate x) = Terminate (f x)
  fmap f (NoMore x) = NoMore (f x)
```

```
  fmap f (Continue x m) = Continue (f x) (fmap f <$> m)
```

The Applicative instance uses the NoMore constructor for the implementation of pure; for the <∗> operator it uses the alongside combinator to run the two parts alongside each other, and then merges them by applying the function (left-hand side) to the argument (right-hand side):

```
instance Alternative f => Applicative (InterleaveT f) where
  pure = NoMore
  (<∗>) a b = uncurry ($) <$> (a `alongside` b)
```

The Applicative instance is useful in explaining why there is no Monad instance for InterleaveT. In the case of the <∗> operator, it is possible for either the left-hand side or right-hand side to actually execute first (or repeatedly). The results are then merged afterwards. The Monad instance would require the right-hand side to know the return value of the left-hand side before it could execute, which obviously is not possible if the right-hand side can execute first.

The Applicative instance can also be used to provide an implementation of the engageMany function that illustrates the behaviour of the instance. Recall that the engageMany function runs many behaviours alongside each other; it is defined as follows:

```
engageMany :: Alternative f => [InterleaveT f a] -> f [a]
engageMany = engage . sequenceA
```

Note that sequenceA does not actually denote sequencing of the events themselves. Rather, it acts as a short-hand for something like foldr1 alongsideMerge, *i.e.* taking a list of actions and indicating that they should all be run alongside each other, but with the values returned in the same *sequence* as the actions were passed. To help the reader's intuition, it is perhaps helpful to look at the instance above with this definition of sequenceA for lists:

```
sequenceA :: Applicative f => [f a] -> f [a]
sequenceA (x:xs) = (:) <$> x <∗> sequenceA xs
sequenceA [] = pure []
```

This effectively translates, for InterleaveT, to:

```
sequenceA :: Alternative f => [InterleaveT f a] -> InterleaveT f [a]
sequenceA (x:xs) = uncurry (:) <$> alongside x (sequenceA xs)
sequenceA [] = NoMore []
```

That is, all the events in the list are run alongside each other, and then merged back into a list once they have finished, as explained earlier for the Applicative instance.

Related Work: Parsing Permutations

The CHP example described earlier in the article was the motivation for the original combinators – they were known as behaviours and were explained (along with some notes on grammars) in a blog post [11]. They were originally specific to CHP but it was subsequently realised that they would work for anything that supported both Monad and Alternative – which eventually led to this article, and the discovery of some similar work in parsing: the "Parsing Permutation Phrases" (PPP) functional pearl [12].

The PPP work, on which the Parsec.Perm module is based, has a very similar aim to the combinators described in this article, albeit targeted specifically at parsing. More recently, the idea behind the PPP work has been generalised and packaged as a library using the general Applicative class (and therefore not restricted to parsing), called `action-permutations` [13].

There are three main differences to the user between the work described in this article and the `action-permutations` library. The first difference is an API design choice: `action-permutations` has no combinator like endAfter. The permutation in `action-permutations` only terminates once all non-optional actions have completed (and all optional actions have been tried). This has the disadvantage of not being able to terminate the stream of actions early, but has the advantage that it can be guaranteed that certain actions will take place. Specifically, a primary combinator in `action-permutations` is:

```
atom :: Alternative p => p a -> Perms p a
```

The InterleaveT library presented in this article does not have that specific function; our similar once function has a type with a Maybe return:

```
once :: Alternative f => f a -> InterleaveT f (Maybe a)
```

This difference – having the Maybe wrapper – is because the possibility of early termination means that we cannot guarantee that the action wrapped by once will ever execute before the interleaved behaviour finishes.

The second difference is also an API design choice: `action-permutations` has no combinator like inOrder, which allows limited ordering among the otherwise "anything goes" ordering of InterleaveT and `action-permutations`. As with endAfter, there is no technical reason why such a combinator could not be added.

The third, and most significant, difference between `action-permutations` and InterleaveT is that InterleaveT uses Monad where `action-permutations` uses Applicative (in fact, the Applicative work was partly inspired by the original PPP work [14]). Many of the basic combinators in this article, such as unlimited, alongside, inOrder and endAfter, can be implemented in either setting. However, the endWhen, unlimitedRecurse and unlimitedRecurse_ combinators cannot be sup-

ported without Monad, and neither can the approaches seen earlier using error monad transformers with InterleaveT.

The use of Monad restricts what types can be used with the library (strictly fewer types are instances of Monad than Applicative), but allows more powerful combinators to be implemented. The use of Monad also makes the code for InterleaveT simpler, and does not require the use of rank-N types or GADTs as an Applicative-based implementation does.

Summary

This article explained the InterleaveT abstraction for repeated choices between actions. We have seen the API, implementation and various examples in which the abstraction has been found to apply. This code has been packaged and appears on Hackage as the `interleave` package [1].

We end with one final example of using InterleaveT: parsing anagrams. To parse a (case-insensitive) anagram of a word, we can use this parser:

```
anagram :: String -> Parsec String () ()
anagram s = engage $ sequenceA_ $ endAfter (once_ eof) : map charAnyCase s
  where
    charAnyCase :: Char -> InterleaveT (Parsec String ()) ()
    charAnyCase c = once_ ( satisfy (((==) `on` toLower) c) <?> show [toLower c])
```

We can then test this function as follows:

```
> runP (anagram "Alternative") () "" "InterleaveT"
Left (line 1, column 10):
unexpected 'e'
expecting end of input, "a" or "t"
```

The parsing fails on the second to last letter of InterleaveT because Alternative has one fewer 'e'. Hmmmm – in the finest tradition of coding, we can fix that with a quick ugly hack:

```
> runP (anagram "Alternative") () "" "IntarleaveT"
Right ()
```

There you go. We've shown that InterleaveT really *is* Alternative with a flexible ordering (but also an extra 'e' instead of an 'a', because life is never quite as neat as you'd like it to be).

References

[1] Neil Brown. Interleave library. `http://hackage.haskell.org/package/interleave`.

[2] Jeffrey Friedl. Source of the famous "now you have two problems" quote. `http://regex.info/blog/2006-09-15/247`.

[3] Daan Leijen and Paolo Martini. Parsec library. `http://hackage.haskell.org/package/parsec`.

[4] Neil Brown. Communicating Haskell Processes library. `http://hackage.haskell.org/package/chp`.

[5] Neil Brown. Sticky platelets in a pipeline (part 1). `http://chplib.wordpress.com/2009/12/21/sticky-platelets-in-a-pipeline-part-1/`.

[6] Bryan O'Sullivan. Attoparsec library. `http://hackage.haskell.org/package/attoparsec`.

[7] Neil Brown. Game-probability library. `http://hackage.haskell.org/package/game-probability`.

[8] Neil Brown. Sharp cards in Haskell: Drawing cards. `http://chplib.wordpress.com/2010/08/23/sharp-cards-in-haskell-drawing-cards/`.

[9] Donald X. Vaccarino. Dominion card game. Rio Grande Games (2008).

[10] Andy Gill. Type-safe observable sharing in Haskell. In **Haskell '09: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell**, pages 117–128 (2009).

[11] Neil Brown. Concisely expressing behaviours with process combinators. `http://chplib.wordpress.com/2009/12/06/concisely-expressing-behaviours-with-process-combinators/`.

[12] Arthur Baars, Andres Löh, and Doaitse Swierstra. Functional pearl: Parsing permutation phrases. **Journal of Functional Programming**, 14(06):pages 635–646 (2004).

[13] Ross Paterson. Action-permutations library. `http://hackage.haskell.org/package/action-permutations`.

[14] Conor McBride and Ross Paterson. Applicative programming with effects. **Journal of Functional Programming**, 18(01):pages 1–13 (2008).

```
    engage (unlimited_ m)
=           {apply unlimited_}
    engage (Continue () (unlimited_ m <$ m))
=           {apply engage}
    (unlimited_ m <$ m) >>= engage
=           {using: x <$ m = m >> return x}
    (m >> return (unlimited_ m)) >>= engage
=           {monad laws}
    m >> engage (unlimited_ m)
=           {coinduction}
    m >> forever m
=           {definition of forever}
    forever m
```

Figure 2: Equational reasoning to prove the law: engage . unlimited_ = forever

```
    engage (unlimited_ x `alongside` unlimited_ y)
=           {apply unlimited_ in two places}
    engage (Continue ()
              (unlimited_ x <$ x) `alongside` Continue () (unlimited_ y <$ y))
=           {apply alongside}
    engage (Continue ((), ())
              ((flip alongside (unlimited_ y) <$> (unlimited_ x <$ x))
                <|> (alongside (unlimited_ x) <$> (unlimited_ y <$ y))))
=           {using: f <$> (x <$ m) = f x <$ m twice}
    engage (Continue ((), ())
              ((alongside (unlimited_ x) (unlimited_ y) <$ x)
                <|> (alongside (unlimited_ x) (unlimited_ y) <$ y)))
=           {using: (x <$ m) <|> (x <$ n) = x <$ (m <|> n)}
    engage (Continue ((), ())
              ((alongside (unlimited_ x) (unlimited_ y)) <$ (x <|> y)))
=           {apply engage}
    ((alongside (unlimited_ x) (unlimited_ y)) <$ (x <|> y)) >>= engage
=           {using: x <$ m = m >> return x}
    (x <|> y) >> return (alongside (unlimited_ x) (unlimited_ y)) >>= engage
=           {monad laws}
    (x <|> y) >> engage (alongside (unlimited_ x) (unlimited_ y))
=           {coinduction}
    (x <|> y) >> forever (x <|> y)
=           {definition of forever}
    forever (x <|> y)
```

Figure 3: Equational reasoning to prove a law

The Reader Monad and Abstraction Elimination

by Petr Pudlák ⟨petr.mvd *at* gmail *dot* com⟩

When I first saw the types of the return and ≫= functions of the reader monad, a remarkable similarity between them and the types of the well known combinators K and S caught my attention. It took me a while to realize that (and why) this is not a coincidence. In this article I would like to share my thoughts and show how the reader monad relates to these combinators and to the abstraction elimination process. I will describe the standard process of abstraction elimination, but in the context of the reader monad. I will also show how various well known combinators correspond to standard Haskell functions.

Introduction

In the early 1920s, Moses Ilyich Schönfinkel aimed at creating a logic system without bound variables. To do this, he invented a set of special operators (which we now call **combinators**) which allowed him to construct such a system. Some of these operators correspond to the combinators now known as I, K and S, which we will discuss later. His ideas were later rediscovered and developed by Haskell Curry, and gave rise to a whole field of logic called combinatory logic. See History of Lambda-calculus and Combinatory Logic [1] for details.

Combinatory logic is closely related to the lambda calculus, which was invented by Alonzo Church and his students in the 1930s. As it turned out, a lambda calculus term can be converted into an equivalent combinatory logic term through a process called **abstraction elimination**. And since any combinatory logic term can be converted into a lambda calculus term, the whole process can be carried out within the lambda calculus: first the required combinators are expressed as lambda terms, and then any lambda term can be converted into a term that does

not use lambda abstractions except for the encoded combinators. This will be one of the main topics of this article.

Abstraction elimination is not only a theoretical mathematical concept. For example, when implementing a reduction machine for a functional language it is much easier to first perform abstraction elimination and then to work with combinators instead of lambda abstractions and variables. (See Chapter 16 in Peyton Jones's book [2].)

Also, coding functional programs without variables is in many cases very natural and improves readability. Instead of thinking about how functions apply to data one thinks about how functions compose together. For example, instead of writing

$$f\ x = g\ (h\ (i\ (j\ x)))$$

one writes

$$f = g \circ h \circ i \circ j$$

This coding style, called **point-free style** (also pointless style), is generally considered much cleaner and good practise. (See the Pointfree page on the Haskell wiki [3].)

Converting Haskell expressions into point-free expressions has been incorporated by Thomas Jäger into the famous Lambdabot program as the `@pl` plugin. The plugin indeed uses the reader monad in the process. Quoting the Pointfree Haskell wiki page [3]:

> The @pl (point-less) plugin is rather infamous for using the $(\rightarrow)$ monad to obtain concise code.

Hopefully this article will help the reader understand (among other things) how Lambdabot actually uses the reader monad for this purpose.

Preliminaries

I will assume that the reader is acquainted with the Haskell language and in particular with monads, and with the very basics of the lambda calculus. There are many monad tutorials available on the Internet. For introduction to the lambda calculus I recommend Barendregt's monographs ([4, 5]).

As this article is intended for Haskell programmers, I will express lambda calculus terms using a subset of the Haskell language restricted to the basic syntax of the lambda calculus: lambda abstraction, variables, constants, function application and parentheses. We will use Haskell's standard type system to assign types to these terms.

Combinators and combinatory logic

The main idea of combinatory logic is to present a formal logic system without the need for variables. It is similar to the lambda calculus, but lambda abstractions are replaced by a fixed set of primitive (higher-order) functions, called combinators. By choosing a suitable set of combinators one can get a system that has the same expressive power as the lambda calculus. For a short introduction, see Wikipedia [6].

In combinatory logic, each primitive combinator comes with a reduction rule which describes how the combinator is evaluated. The term on the right hand side of such a rule may contain only variables that are present on its left hand side. For example,

$$k\ x\ y = x$$

defines a combinator K that takes two arguments and evaluates to the first one, ignoring the second.

It is evident that any such primitive combinator can be expressed as a **closed** lambda term (one containing no free variables) that performs the same computation as the corresponding reduction rule. Often in the lambda calculus (and functional languages) any closed lambda term is called a combinator.

Remark 1. In the literature, combinators are usually denoted by uppercase letters. However, Haskell reserves identifiers starting with an uppercase letter for constructors. Therefore, when referring to combinators in general, I will use uppercase letters, such as K, and when referring to actual Haskell functions I will use lowercase letters, such as k.

The combinators K, I and S

Two of the most widely used combinators are K and S (they will be defined later). Their importance comes from the fact that any (untyped) lambda calculus term can be converted to another term which:

- performs the same computation, and
- uses only function application, the two combinators K and S, and no lambda abstraction.

For more information, see Barendregt [4]. The actual conversion process is called **abstraction elimination**, and we will discuss it in detail later.

Remark 2. From the point of view of the lambda calculus, the converted term is not the same object as the original term. For us, however, it is important that these two terms perform the same computation. We will call (any) two terms that perform the same computation **extensionally equal** or just **equivalent**.

The two combinators K and S can be defined (in Haskell notation) as follows:

$$k\ x\ y\ \ = x$$
$$s\ x\ y\ z = (x\ z)\ (y\ z)$$

Note that k is just Haskell's *const* function. It is also convenient to define one more combinator, called I:

$$i\ x = x$$

which corresponds to Haskell's *id* function.

In Haskell, the types assigned to these combinators are:

$$k :: a \to b \to a$$
$$s :: (a \to b \to c) \to (a \to b) \to a \to c$$
$$i :: a \to a$$

Exercise 3. The I combinator is redundant. Try to create I using just the combinators S and K and function application. (You will find the answer on page 41.)

Example 4. The term $\lambda x\ y \to y\ x$ is equivalent to $s\ (k\ (s\ i))\ (s\ (k\ k)\ i)$.

The reader monad

The reader monad (also called the environment monad) encapsulates computations which read values from a shared environment. For example, suppose that you read a configuration for your program from a file and you wish to pass the configuration to several different functions. Instead of passing the configuration as another argument of your functions, you can define them using the reader monad and use monadic combinators to bind them together.

There are actually two variants of the reader monad in the Haskell library. The first one, called *Reader* (see [7]), is generally more useful and better known. It is defined as follows:

```
newtype Reader r a = Reader {
  runReader :: r → a
}
instance Monad (Reader r) where
  return a = Reader $ λ_ → a
  m >>= k  = Reader $ λr → runReader (k (runReader m r)) r
```

The other one is $((\to)\ r)$. I always found this type very obscure and cryptic, until I realized that this is just the reader monad. It is a type function that maps a

type a to $r \to a$, just like the Reader monad, but without wrapping it into the constructor. Its monad instance is defined as:

$$\begin{aligned}&\textbf{instance } \mathit{Monad}\ ((\to)\ r)\ \textbf{where}\\ &\quad \mathit{return}\ a = \lambda\_ \to a\\ &\quad m \gg\!= k\ = \lambda r \to k\ (m\ r)\ r\end{aligned}$$

Indeed it is almost the same as the instance of *Monad* (*Reader* r); only the constructor *Reader* and its unwrapping counterpart *runReader* are missing.

We could use both these monads for our purpose, but $((\to)\ r)$ will make the presentation much clearer, since we will be spared from constant wrapping and unwrapping of the constructor. Therefore, from now on, I will refer just to the $((\to)\ r)$ monad.

The types of the reader monad functions

Let us take a closer look at the types of the reader monad functions:

$$\begin{aligned}&\mathit{return} :: a \to r \to a\\ &(\gg\!=)\ \ :: (r \to a) \to (a \to r \to b) \to r \to b\end{aligned}$$

The similarity to the types of the combinators k and s is immediate. But we can do even better. Every monad gives rise to an applicative functor. The two essential functions of every applicative functor f are:

$$\begin{aligned}&\mathit{pure} :: a \to f\ a\\ &(<\!\!*\!\!>) :: f\ (a \to b) \to f\ a \to f\ b\end{aligned}$$

The function *pure* is the same as *return* and $(<\!\!*\!\!>)$ is the same as the derived monadic function *ap*.

Let us call the applicative functor induced by the reader monad the **reader applicative functor**. Instances of the respective functions for the reader applicative functor have these types:

$$\begin{aligned}&\mathit{pure} :: a \to r \to a\\ &(<\!\!*\!\!>) :: (r \to a \to b) \to (r \to a) \to r \to b\end{aligned}$$

The types of *pure* and $(<\!\!*\!\!>)$ are the same as the types of K and S. Moreover, *pure* must be equivalent (perform the same computation) to K and $(<\!\!*\!\!>)$ must be equivalent to S. We may prove the equivalence from their definition, but in this case, we can deduce it just from their types. Since these functions are fully polymorphic, they have no way of constructing their return values other than combining their arguments. The function *pure* has no way of obtaining a result of

type a other than returning its first argument. The function $(<\!*\!>)$ can compute a result of type b only by first applying its second argument of type $r \to a$ to its third argument of type r, thereby computing an intermediate value of type a, and then applying its first argument of type $r \to a \to b$ to its third argument and the intermediate value.

(Many things can be deduced just from the type of a function. See Wadler's article [8] for more information.)

Abstraction elimination

In this section we will take a closer look at the process of abstraction elimination. We will show how to use *pure* and $(<\!*\!>)$ to convert Haskell terms containing lambda abstractions into abstraction-free terms.

Suppose we have a term of the form $\lambda x \to s$. We want to convert this term into an equivalent one which is free of the lambda abstraction. First, we recursively apply the procedure to s and convert it into a term t which is abstraction-free and equivalent to s. Now the simplified task is to eliminate just the single abstraction from $\lambda x \to t$.

Let the type of $\lambda x \to t$ be $r \to a$, where $x :: r$ and $t :: a$. We will define a transformation $[\![\cdot]\!]_x$ which will map the term t with (possible) free occurrences of x to an abstraction-free term $[\![t]\!]_x :: r \to a$, such that $[\![t]\!]_x$ is equivalent to $\lambda x \to t$.

Observe that since $[\![t]\!]_x$ is of the type $r \to a$, it is actually a value in the reader monad $((\to)\ r)$. Therefore, we can construct and combine various $[\![\cdot]\!]_x$ expressions using the monad (or applicative functor) functions.

Because t is already abstraction-free, it must be one of the following terms:

A term that does not contain x as a free variable. Note that this includes the case when t is just a variable different from x. We simply set $[\![t]\!]_x = \mathit{pure}\ t$.

The term t is equal to the variable x. In this case, we set $[\![x]\!]_x = \mathit{id}$.

The term t is a function application $t = u\ v$. There must be types a and b such that $t :: a$, $v :: b$ and $u :: b \to a$. If we recursively transform u and v, we get two terms $[\![u]\!]_x :: r \to (b \to a)$ and $[\![v]\!]_x :: r \to b$.

Recall the type of $(<\!*\!>) :: (r \to (b \to a)) \to (r \to b) \to (r \to a)$. This function does just what we need. It applies a function within the reader applicative functor to a value within the reader applicative functor. Therefore we simply combine the two terms with $(<\!*\!>)$ and we set $[\![u\ v]\!]_x = [\![u]\!]_x <\!*\!> [\![v]\!]_x$.

In essence, we are recursively lifting ordinary terms into applicative functor computations within the reader applicative functor. Simple terms are lifted using *pure* and *id* and function applications are lifted using $(<*>)$.

Remark 5 (Answer to Exercise 3)**.** The I combinator can be expressed as SKK. Therefore, if in the second step we want to be perfectionists and if we do not mind less efficient code, instead of $[\![x]\!]_x = id$ we can write

$$[\![x]\!]_x = (<*>)\ pure\ pure = pure <*> pure$$

Example 6. Let us recall Example 4 and transform $\lambda x\ y \to y\ x$ step by step. First, we will compute a term equivalent to $\lambda y \to y\ x$:

$$\begin{aligned}[\![y\ x]\!]_y &= (<*>)\ [\![y]\!]_y\ [\![x]\!]_y \\ &= ((<*>)\ id)\ (pure\ x)\end{aligned}$$

(We keep the prefix notation in order to simplify the next step.) Second, the term equivalent to $\lambda x \to (\lambda y \to y\ x)$ is:

$$\begin{aligned}[\![[\![y\ x]\!]_y]\!]_x &= [\![((<*>)\ id)\ (pure\ x)]\!]_x \\ &= [\![(<*>)\ id]\!]_x <*> [\![pure\ x]\!]_x \\ &= (pure\ ((<*>)\ id)) <*> ([\![pure]\!]_x <*> [\![x]\!]_x) \\ &= (pure\ ((<*>)\ id)) <*> ((pure\ pure) <*> id)\end{aligned}$$

which, converted to prefix notation, gives

$$= (<*>)\ (pure\ ((<*>)\ id))\ ((<*>)\ (pure\ pure)\ id)$$

In the language of the SKI combinators, it is $s\ (k\ (s\ i))\ (s\ (k\ k)\ i)$.

More examples – converting Haskell programs

I could not resist the temptation to write a very simple converter for Haskell programs. It converts only explicit lambda abstractions and copies everything else as it is. Therefore it actually works only on very special Haskell programs. At the end, I thought that the program was not very illustrative. So I used it only for creating a few examples and I decided not to include it in this article. Nevertheless, if you want to play around with the ideas, the program is bundled with the sources for this issue of The Monad.Reader, available from `http://code.haskell.org/~byorgey/TMR/Issue17/Convert.lhs`.

Since the functions *pure* and $(<*>)$ are applicable to many possible applicative functors, the compiler fails when trying to infer the most general form of our terms.

It is therefore necessary to define aliases for these functions which are explicitly typed for the reader applicative functor:

$$
\begin{aligned}
&\mathit{pure}' = \mathit{pure} :: r \to (a \to r) \\
&(\mathbin{<\circledast>}) = (\mathbin{<*>}) :: (r \to (a \to b)) \to (r \to a) \to (r \to b)
\end{aligned}
$$

Example 7 (Fibonacci sequence)**.** Let us convert some slightly more complicated code. I chose a recursive function for computing the Fibonacci sequence. Since the translation program does not translate pattern matching nor the **if-then-else** construct, one must define **if-then-else** as a combinator:

$$
\begin{aligned}
&\mathit{ite} :: \mathit{Bool} \to a \to a \to a \\
&\mathit{ite}\ c\ x\ y = \mathbf{if}\ c\ \mathbf{then}\ x\ \mathbf{else}\ y
\end{aligned}
$$

Then we define a function that computes **n**-th Fibonacci number:

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib} = \lambda x \to \mathit{ite}\ (x \leqslant 1)\ 1\ (\mathit{fib}\ (x-1) + \mathit{fib}\ (x-2))
\end{aligned}
$$

The translation program converts *fib* to:

$$
\begin{aligned}
\mathit{fib} = {} & ((((\mathit{pure}'\ \mathit{ite}) \mathbin{<\circledast>} \\
& \quad ((((\mathit{pure}'\ (\leqslant)) \mathbin{<\circledast>} \mathit{id}) \mathbin{<\circledast>} (\mathit{pure}'\ 1)))) \\
& \quad \mathbin{<\circledast>} (\mathit{pure}'\ 1)) \\
& \mathbin{<\circledast>} \\
& ((((\mathit{pure}'\ (+)) \mathbin{<\circledast>} \\
& \quad ((\mathit{pure}'\ \mathit{fib}) \mathbin{<\circledast>} \\
& \quad \quad ((((\mathit{pure}'\ (-)) \mathbin{<\circledast>} \mathit{id}) \mathbin{<\circledast>} (\mathit{pure}'\ 1))))) \\
& \quad \mathbin{<\circledast>} \\
& \quad ((\mathit{pure}'\ \mathit{fib}) \mathbin{<\circledast>} \\
& \quad \quad ((((\mathit{pure}'\ (-)) \mathbin{<\circledast>} \mathit{id}) \mathbin{<\circledast>} (\mathit{pure}'\ 2)))))))
\end{aligned}
$$

While this definition is far from efficient nor comprehensible, it uses just function application, parentheses, combinators and numeric primitive functions.

Reducing the length of the output

It is obvious from the preceding examples that the output term can be much larger than the input term. Indeed, the length of the output can be as large as $\Theta(3n)$ where n is the length of the input term. There are various strategies for improving conversion. For example, see Chapter 16.2 in Peyton Jones's book [2], or Chapter 5.2 in Barendregt's [4], which also references several papers on the subject.

In this article I will discuss just one optimization, which is adding two more specialized combinators. Recall the step that eliminated a variable x of type $x :: r$ from a term $(u\ v) :: a$, where u and v have some types $u :: b \to a$ and $v :: b$. If x was free in $(u\ v)$, we defined $[\![u\ v]\!]_x = [\![u]\!]_x \mathbin{<*>} [\![v]\!]_x$. But we did not check whether x is free in u, in v or in both. Let us split this step and specialize for these cases:

If x is free in both u and v, there is not much room for improvement. We do what we did before and put $[\![u\ v]\!]_x = [\![u]\!]_x \mathbin{<*>} [\![v]\!]_x$.

If x is free in just u, we do not have to translate v. By translating u we get $[\![u]\!]_x :: r \to (b \to a)$. If we "insert" somehow v into u's second argument, we get the desired result. We can achieve this by applying Haskell's function $\mathit{flip} :: (a \to b \to c) \to (b \to a \to c)$ to u. This function takes a function and returns a new function that performs the same computation, but with its two arguments flipped. Now $\mathit{flip}\ [\![u]\!]_x$, with type $\mathit{flip}\ [\![u]\!]_x :: b \to (r \to a)$, can be applied directly to v, so we can define $[\![u\ v]\!]_x = \mathit{flip}\ [\![u]\!]_x\ v$. Interestingly, *flip* is a combinator well known in the literature and is called C.

If x is free in just v, we do not have to translate u. Speaking in monad language, we have an impure value $[\![v]\!]_x$ and a pure function u. At this point you might be already thinking of the higher-order lifting functions *liftA* or (<$>). The function (<$>), which in the case of applicative functors is the same as *fmap* or *liftA*, takes a pure function and lifts it into an applicative functor:

$$(\mathbin{<\$>}) :: \mathit{Applicative}\ f \Rightarrow (a \to b) \to (f\ a \to f\ b)$$

The instance of (<$>) for the reader applicative functor is of the type

$$(\mathbin{<\$>}) :: (a \to b) \to (r \to a) \to (r \to b)$$

This is the same type as function composition:

$$(\circ) : (a \to b) \to (r \to a) \to (r \to b)$$

Again, just from the types we can deduce that the instance of (<$>) for the reader applicative functor is equivalent to $(\circ)$. Therefore we may set

$$[\![u\ v]\!]_x = u \mathbin{<\$>} [\![v]\!]_x$$

or more simply

$$[\![u\ v]\!]_x = u \circ [\![v]\!]_x$$

The function $(\circ)$ (or the instance of (<$>) for the reader applicative functor) is also known in literature as the combinator B.

Examples revised

Let us apply this optimization to our previous examples. The term from Example 4 will be converted as:

$$[\![\lambda x\ y \to y\ x]\!]_x = \mathit{flip}\ \mathit{id}$$

And the function for computing Fibonacci sequence

$$\mathit{fib} = \lambda x \to \mathit{ite}\ (x \leqslant 1)\ 1\ (\mathit{fib}\ (x-1) + \mathit{fib}\ (x-2))$$

from Example 7 will be converted to

$$\begin{aligned}[\![\mathit{fib}]\!]_x = ((\mathit{ite} \circ ((\leqslant)\ `\mathit{flip}`\ 1))\ `\mathit{flip}`\ 1) \mathbin{\langle\circledast\rangle} \\ (((+) \circ (\mathit{fib} \circ ((-)\ `\mathit{flip}`\ 1))) \mathbin{\langle\circledast\rangle} (\mathit{fib} \circ ((-)\ `\mathit{flip}`\ 2)))\end{aligned}$$

Certainly, this is a significant improvement.

Conclusion

We described a process of abstraction elimination by means of the reader applicative functor. The process can be viewed as a gradual lifting of terms into instances of the reader applicative functor using *pure* or *id* and then combining them with $(\langle\circledast\rangle)$.

The process is exactly the same as the classical replacement of lambda terms by the SKI combinators. However, I feel that visualising the process as operations on the reader applicative functor gives a new and inspiring insight into the process and reveals the underlying idea. It certainly did for me.

We have also discovered that several functions from the standard library and from the library of applicative functors correspond to well known combinators. They are summarized in Table 4.1.

References

[1] Felice Cardone and J. Roger Hindley. Lambda-calculus and combinators in the 20th century. In John Woods and Dov M. Gabbay (editors), **Handbook of the History of Logic, Volume 5: Logic from Russell to Church**. Elsevier/North Holland (2009). `http://lambda-the-ultimate.org/node/2679`.

[2] Simon L. Peyton Jones. **The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987). `http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/`.

Combinator	Instance of an applicative functor function for the reader app. functor	Function from the standard Haskell library
I		*id*
K	*pure*	*const*
S	(<*>)	
B	(<$>) and *liftA*	(∘)
C		*flip*

Table 4.1: Summary of combinators and corresponding Haskell functions

[3] Pointfree – HaskellWiki (2010). `http://www.haskell.org/haskellwiki/Pointfree`. [Online; accessed 18-Dec-2010].

[4] Hendrik Pieter Barendregt. Functional programming and lambda calculus. In **Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)**, pages 321–363. Elsevier and MIT Press (1990).

[5] Hendrik Pieter Barendregt. Lambda calculi with types. pages 117–309. Oxford University Press, Inc., New York, NY, USA (1992). `ftp://ftp.cs.ru.nl/pub/CompMath.Found/HBK.ps`.

[6] Wikipedia. Combinatory logic – Wikipedia, the Free Encyclopedia (2010). `http://en.wikipedia.org/w/index.php?title=Combinatory_logic&oldid=385632685`. [Online; accessed 27-September-2010].

[7] Control.Monad.Reader. `http://hackage.haskell.org/packages/archive/mtl/1.1.0.2/doc/html/Control-Monad-Reader.html`.

[8] Philip Wadler. Theorems for free! In **FPCA**, pages 347–359 (1989). `http://homepages.inf.ed.ac.uk/wadler/topics/parametricity.html`.