The Monad.Reader Issue 16

by Aran Donohue ⟨aran@arandonohue.com⟩
and John W. Lato ⟨jwlato@gmail.com⟩
and Louis Wasserman ⟨lowasser@uchicago.edu⟩

May 12, 2010

Brent Yorgey, editor.

Contents

Editorial

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

It was a dark and stormy night; the rain fell in torrents, much like the billions of bits screaming through underground fiber optic cables (protected from the seeping rainwater by plastic sheathing) every second, carrying compressed encodings of *The Matrix* to be transformed into an uncompressed torrent of photons bombarding the eyes of the residents of Monadriton, who watched in willful ignorance of the strained simile. It did not, however, escape the watchful mind of undercover detective A. J. L. Donolatowasser, who also wondered idly whether the rain and the synaptic firings in the brains of people watching *The Matrix* could be simulated by a sufficiently large finite state—

Suddenly, a shot rang out, misinterpreted at first (due to the rain, even heavier now (if that were possible) and the relative paucity of violent crime in Monadriton) by Donolatowasser as a car backfiring, probably his own; considering the number of moving parts involved and the embedded software written by people who had never even heard of the Curry-Howard Isomorphism, it amazed him things didn't go wrong with it more often, like a floating point error and maybe some wrong units causing the accelerator to become stuck, barrelling down the highway at a hundred twenty miles per hour (except the car would think it was feet per hour) until the highway ran out and he crashed through a reinforced concrete retaining wall into the ocean, a tangled wreck of twisted metal, concrete, rebar, and maybe some fiber optic cables—

A second shot, unmistakable this time (due mostly to the way in which it punched a neat hole through his rear windshield, and then a similar one through the front, both embellished with spiderweb tracings of cracked glass and admitting a steady trickle of water which, due to some weird surface-tension effects that he didn't really understand, separated out into individual drops by the time it hit the newly-cleaned carpets) rang out, jolting him awake from his attempts to escape his quickly sinking car by smashing his laptop against the (curiously intact) driver's side window. He noted with annoyance, as he stomped on the gas pedal (changing the trajectory of the water trickling through the windshield into a neat

parabola ending on his laptop in the front passenger seat – at least he thought it was a parabola, working out derivatives in his head, but couldn't really look at it except out of the corner of his eye while he swerved around a mailbox), that central command probably knew this had been a trap all along, but hadn't told him because he hadn't asked; they really needed to switch to a more efficient push-based intelligence and operations processing method. He began working out a possible design for such a system in his head, and even came up with a cute name for it based on the fact that—

His radio crackled to life: a robbery in progress, or maybe some snobbery, it was hard to make it out above the din of the rain and the continuing gunfire and his racing engine due to the accelerator which was apparently stuck. He tried to remember from his training whether under the present circumstances he was expected to respond to the call, but the training materials had been strangely silent on the proper priorities when being chased in the rain by potential assissins; in any event it also reminded him of his dentist appointment which was in – he checked the clock while careening up an entry ramp to the freeway – twenty minutes. What he really needed was some sort of prioritized system for tracking obligations like dentist appointments and responding to reports of snobbery so that he always knew what the most important thing to do was, and he wondered how to implement such a thing efficiently in a functional programming—

Demand More of Your Automata

by Aran Donohue ⟨aran@arandonohue.com⟩

Most regular expressions only ever parse text. We programmers should demand more of them and their automata cousins!

Theme and Variations

I think most programmers think of regular expressions as light tools for parsing text. We don't hesitate to burp out gobbledygook like `"(?P<area>\d{3})[-.](?P<n1>\d{3})[-.](?P<n2>\d{4})"`. We check that it works and move on, hoping that no one will need to take a closer sniff. If our boss asks us to explain our broken regular expression-based phone number extraction code three months after we wrote it, we instinctively know the correct response is to quit. And maybe steal a stapler on the way out, if the office has one of those nice non-jamming new Swinglines.

In this article, I'll show several twists on the traditional view of regular expressions. We'll rush through the basics, go beyond strings, avoid regular expressions when building matchers, gain new insights into our matching machines and finally optimize our matchers by compiling to machine code via LLVM.

Regular Language Theory in Three Minutes

For those readers that had hangovers in their automata-theory classes: a regular expression defines a boolean function ("language", "set", "predicate") on strings. We say a regular expression might "match" a string. There are regular expressions that match the empty string or any given single character. We can glue two regular expressions one after the other to make a new regular expression. We can make a regular expression that matches what either of two regular expressions match. Finally, we can make a regular expression that matches zero or more times what

```
data RegularExpression
    = Empty
    | Singleton Char
    | Kleene RegularExpression
    | Catenation RegularExpression RegularExpression
    | Alternation RegularExpression RegularExpression
```

Listing 1: A representation of theoretical regular expressions.

another matches, called the "Kleene closure". See Listing 1 for a representation in Haskell.

Regular expressions are commonly written with a syntax like `(ab)*c|d`. Catenation is implicit in the left-to-right reading order of the string. Parentheses are used to group regular expressions. The `*` character is the Kleene closure of whatever is immediately left of it. Finally, `|` is alternation and has the lowest precedence. Thus we have the following example mappings from constructors to syntax:

1. `Singleton 'a'` ↦ `a`
2. `Kleene (...)` ↦ `(...)*`
3. `Catenation (Singleton 'a') (Singleton 'b')` ↦ `ab`
4. `Alternation (Singleton 'a') (Singleton 'b')` ↦ `a|b`

As an example, the regular expression `(ab)*c|d` matches "d", "c", "abc", "ababc", "abababc", and so on.

Many people use the term "Regular Expression" to refer to more powerful machines that are not based on this same theoretical foundation. I'll just call those beasts "frankenspressions" to avoid confusion. Actually, I'll never need to refer to frankenspressions again, but I think we should all agree to stop calling them regular expressions. (*"Hello, Jeffrey Friedl? Theoretical Computer Science here. We want our precisely-defined terminology back." [1])* Compared to frankenspressions, regular expressions offer the pleasant performance guarantee that all processing can take place in exactly one pass over the input.

Many useful patterns can be specified by regular expressions. Those languages that can be defined by a regular expression are called regular languages. Regular expression matchers are usually implemented by non-deterministic finite automata (NFAs) or deterministic finite automata (DFAs). Both DFAs and NFAs are (only) powerful enough to match regular languages.

A DFA is a graph with one "start" state and one or more "accepting states". Each state has a labeled transition to some other state for every possible character. An NFA is like a DFA except for two differences: states can have more than one transition for a character, and states can have transitions called ε-transitions that

don't require consuming any input. That means that at any given place in our processing of the input, we could be in several states because we had several transitions to follow at some point. A string is matched by a machine if and only if there's a path labeled with that string from the start state to an accepting state.

Haskell Representations of NFAs and DFAs

```
data DFA
    = DFA { transition :: State -> Char -> Maybe State
          , start      :: State
          , accepting  :: State -> Bool
          }

data NFA
    = NFA { transition :: State -> Maybe Char -> Set State
          , start      :: State
          , accepting  :: State -> Bool
          }
-- Assume "type State = Int" or something like that.
```

Listing 2: Haskell representation of theoretical DFAs and NFAs.

Listing 2 illustrates this description in code. I made a couple decisions worth explaining. The DFA transition function has a result type of `Maybe State`. This is because we don't care to carefully track every possible transition. Instead, we assume the existence of an implicit error state, which has transitions only to itself. If the DFA transition function leads to this error state, it returns `Nothing`.

The NFA transition function takes a `Maybe Char` parameter. This is because we represent ε-transitions by `Nothing`.

Beyond [Char]

The theoretical terminology for regular expressions talks about "languages" over "strings". Who says they need to be strings of characters? We can define regular languages over numbers, bytes, XML nodes or any other data. This leaves us the problem of specifying regular expressions of non-character inputs.

As a first step, we parameterize `RegularExpression` over a type variable. See Listing 3.

```
data RegularExpression a
    = Empty
    | Singleton a
    | Kleene (RegularExpression a)
    | Catenation (RegularExpression a) (RegularExpression a)
    | Alternation (RegularExpression a) (RegularExpression a)
    deriving (Show)
```

Listing 3: Parameterized RegularExpression type.

Strictly speaking, regular expressions range over a finite alphabet, so `a` should be a finite type. We won't let that trouble us now, though.

Assuming a regular expression has been created by parsing a string such as `"(ab)*c|d"`, we can use a `Functor` instance for `RegularExpression` to build a regular expression over some non-`Char` type. See Listing 4.

```
instance Functor RegularExpression where
    f `fmap` Empty             = Empty
    f `fmap` (Singleton a)     = Singleton (f a)
    f `fmap` (Kleene r)        = Kleene (f `fmap` r)
    f `fmap` (Catenation r s)  = Catenation (f `fmap` r) (f `fmap` s)
    f `fmap` (Alternation r s) = Alternation (f `fmap` r) (f `fmap` s)
```

Listing 4: RegularExpression Functor instance.

Thus, with a simple mapping `Char -> a` we can process regular languages over `a`. For example, imagine we have a parsed log of user activity, as in Listing 5. We can specify patterns of interest for analysis using a regular expression.

Bringing Automata to Life

It's hard to find a good live automaton these days. A live automaton is a data structure which we completely control. It can be introspected, manipulated or recombined after it is made. Alas! Programmers use automata produced stiff by black-box functions like Python's `re.compile` or invocations of generators like Alex, Lex or Flex. It would be much nicer to take control of our automata while they're alive and dynamic, and only kill them at the last moment for efficiency.

First, we need data types that give us more flexibility than the theoretical NFAs and DFAs shown above. Below, we show a `MapNFA a` data type which holds transi-

```
data UserAction = Click
                | Type
                | FollowLink
                deriving (Show, Eq, Ord)

-- Transformation to allow us to make a regular language
-- of UserActions
charToAction 'c' = Click
charToAction 't' = Type
charToAction 'f' = FollowLink

typetype = fmap charToAction (eParse "tt")
typeClicksThenLeave = fmap charToAction (eParse "tc*f")
match r = NMatcher.matches (mapNFAToNFA (regularExpressionToNFA r))

-- In action:
> match typetype [Click, Click]
False
> match typetype [Type]
False
> match typetype [Type, Type]
True
> match typeClicksThenLeave [Type, Type]
False
> match typeClicksThenLeave [Type, Click, Type]
False
> match typeClicksThenLeave [Type, Click, FollowLink]
True
```

Listing 5: Matching things that aren't strings.

tion maps in `Data.Map` and acceptance and transition sets in `Data.Set`. With this data type and some helper functions, we can make NFAs directly. For example, we can give NFA constructors for several of the regular expression constructors. (Unfortunately, the `Ord` constraint taints all our signatures because we use `Set`.)

```
type StateLabel = Int
type MapNFATransitionMap a
    = Map.Map (StateLabel, Maybe a) (Set.Set StateLabel)
data MapNFA a = MapNFA { transitionMap :: MapNFATransitionMap a
                       , startKey :: StateLabel
                       , acceptKeys :: Set.Set StateLabel
                       } deriving (Show)
```

Listing 6: A flexible NFA data type.

- `empty :: Ord a => MapNFA a`
- `singleton :: Ord a => a -> MapNFA a`
- `catenate :: Ord a => MapNFA a -> MapNFA a -> MapNFA a`
- `alternate :: Ord a => MapNFA a -> MapNFA a -> MapNFA a`

Instead of implementing `kleene` as a basis function, we can implement:

- `loop :: Ord a => MapNFA a -> MapNFA a`

Then `kleene` can be implemented as in Listing 7.

```
optional :: Ord a => MapNFA a -> MapNFA a
optional = alternate empty

kleene :: Ord a => MapNFA a -> MapNFA a
kleene = optional . loop
```

Listing 7: Kleene closure as a derived combinator.

This leads naturally to our next variation on traditional ways of using automata. Why should we restrict ourselves to automata built through regular expressions? We can construct many interesting automata using derived combinators. See Listing 8 for combinators that compose multiple NFAs, Listing 9 for combinators that build NFAs out of elements, and Listing 10 for some advanced combinators.

We could imagine a wonderful new world of programming ecstasy. In this world, we would make a new regular expression syntax that exposed combinators like `set` or anything else we desire. We would build automata at run-time, such as a dead-

```
-- | Matches the language of each provided NFA in turn.
chain :: Ord a => [MapNFA a] -> MapNFA a
chain = foldr1 catenate

-- | Matches the language of any of the provided NFAs.
anyOf :: Ord a => [MapNFA a] -> MapNFA a
anyOf = foldr1 alternate

-- | Matches the languages of the provided NFAs in any order,
-- with no repetition.
allOf :: Ord a => [MapNFA a] -> MapNFA a
allOf nfas = anyOf $ map chain (List.permutations nfas)

-- | Matches the language of the provided NFA 'n' times.
count :: Ord a => Int -> MapNFA a -> MapNFA a
count n dfa = chain $ replicate n dfa
```

Listing 8: Derived NFA combinators.

simple trie for an autocompletion function. We would be giants, in a land of plentiful coffee and fast, obviously correct code.

Automata Visualization

Whether we build our automata using regular expressions or combinators, sometimes we'll get it wrong. Well, I know *you* never get them wrong, but sometimes *I* make mistakes. In these cases, I like to be able to introspect the automaton to rapidly track down what's going on. *You* might just like to look at pretty pictures of your automata that have *nothing* to do with finding (non-existent) bugs.

Unfortunately, most regular expression implementations have nothing to offer. When we have live automata objects, though, we can compile them to Graphviz or other visualization formats. These drawings are easy to inspect and make some kinds of mistakes very obvious.

Haskell's `Data.Graph.Inductive.Graph` and `Data.GraphViz` modules give us excellent support for Graphviz output. We'll exploit that support by rendering our `MapDFA` and `MapNFA` objects to `Gr` objects. See Listing 12.

Given a `Gr` object, we can easily customize our final Graphviz output. In the example code in Listing 13, we render our visualization in the center of a North American letter format page, with accepting states distinguished in bold.

```
lift :: Ord a => [a] -> [MapNFA a]
lift = map singleton

-- | NFA that matches the provided sequence in order.
string :: Ord a => [a] -> MapNFA a
string = chain . lift

-- useful for e.g. regex character classes
-- | NFA that matches any item in the provided list.
oneOf :: Ord a => [a] -> MapNFA a
oneOf = anyOf . lift

-- | NFA that matches the provided items in any order,
-- with no repetition.
set :: Ord a => [a] -> MapNFA a
set = allOf . lift

-- | NFA that matches a sequence of 'n' repetitions of
-- the provided item.
nTimes :: Ord a => Int -> a -> MapNFA a
nTimes n ch = string (replicate n ch)
```

Listing 9: Derived NFA constructors on elements.

```
-- | Efficiently matches any of the provided sequences.
trie :: Ord a => [[a]] -> MapNFA a
trie strings = anyOf $ map string strings

-- | Efficiently matches the sequence of 'nfas', each separated by
-- the language of 'middle'.
intersperse :: Ord a => MapNFA a -> [MapNFA a] -> MapNFA a
intersperse middle nfas = chain $ List.intersperse middle nfas

-- | Matches the language of the provided 'nfa', optionally preceded
-- by language of 'optionalNFA'.
optionallyThen :: Ord a => MapNFA a -> MapNFA a -> MapNFA a
optionallyThen optionalNFA nfa = optional optionalNFA `catenate` nfa
```

Listing 10: Advanced NFA constructors.

```
-- | NFA that matches any single item.
-- Much like the regex . operator
any :: (Ord a, Enum a, Bounded a) => MapNFA a
any = oneOf [minBound .. maxBound]

-- | NFA that matches any item except those provided.
-- Much like a regex negated character class.
noneOf :: (Bounded a, Enum a, Ord a) => [a] -> MapNFA a
noneOf xs = oneOf $ [minBound .. maxBound] List.\\ xs
```

Listing 11: NFA constructors for Bounded alphabets.

```
makeGraph :: (Show a, Ord a) => MapNFA a -> Gr StateLabel String
--fix type to fix ambiguity
makeGraph nfa = mkGraph (nodes nfa) (edges nfa)
  where nodes nfa = [(a, a) | a <- Set.toList $ states nfa]
        edges nfa = Map.foldWithKey fn [] (transitionMap nfa)
        fn (from, transition) tos accum
           = let label = maybe "ε" show transition
             in [(from, to, label) | to <- Set.toList tos] ++ accum
```

Listing 12: Conversion of an NFA to a generic graph.

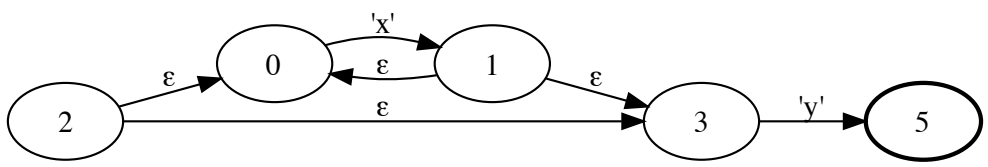

Figure 1: NFA of x*y

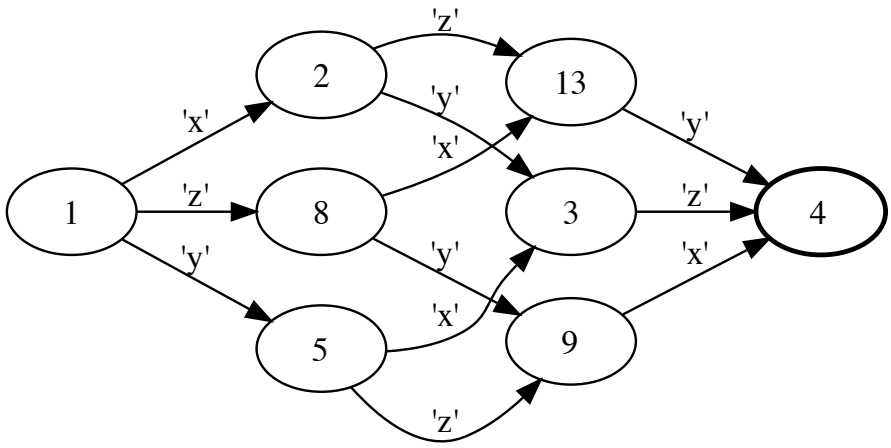

Figure 2: DFA of set “xyz”

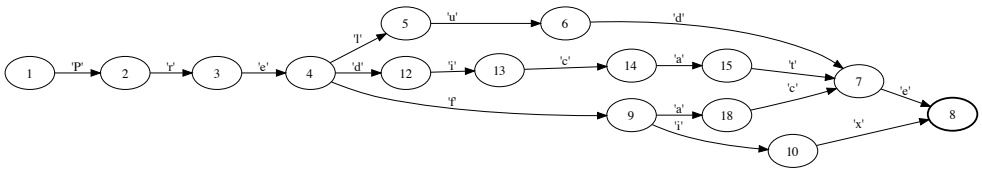

Figure 3: DFA of trie [“Prelude”, “Prefix”, “Predicate”, “Preface”]

```
toGraphviz :: (Show a, Ord a) => MapNFA a -> String
toGraphviz = printDotGraph . toDotGraph

toDotGraph nfa = graphToDot True
                            (makeGraph nfa)
                            [GraphAttrs [Page (PointD 11 8.5),
                                         RankDir FromLeft,
                                         Center True]]
                            nodeToAttr
                            edgeToAttr
    where nodeToAttr (n, a) | n `Set.member` acceptKeys nfa
                                = [Style [SItem Bold []]]
                            | otherwise = []
          edgeToAttr (from, to, label) = [Label (StrLabel label)]
```

Listing 13: Rendering a graph to Graphviz

When in a Rush

With data structures built on log-time maps and sets in a high-level lazy language, one might worry about the performance of our automata. Using good algorithms gives us a guarantee that we'll match a string of length n in $O(n)$ time (in one pass, in fact). But we might care about how fast that pass can go.

Since we can compile both regular expressions and NFAs to DFAs, we can go one step further and compile DFAs to machine code. The code to do this doesn't lend itself to succinct presentation in an article, so instead I'll just describe the high-level approach.

We take advantage of the LLVM compiler infrastructure and its Haskell bindings to handle most of the work. When we compile a DFA to LLVM, we can feel confident in our automata knowing that we match strings with just a few machine instructions per character.

For simplicity, we only compile automata over ASCII strings. Our target, LLVM IR, is a static single-assignment (SSA) intermediate representation with excellent support for compiling and executing straight to memory. LLVM code is divided into Modules, which are divided into Functions, which are divided into Basic Blocks. For more, see the LLVM documentation or a reference on optimizing compilers.

Our compilation strategy for a boolean-result DFA is as follows: each DFA state maps to a basic block. We keep track of a single integer representing the current offset into the null-terminated input string. Each DFA state basic block

increments the integer. Each DFA state basic block has a Phi instruction for the value of the integer, with possibilities coming from each inbound DFA transition. Each DFA state examines the value of the string at the offset and jumps to the appropriate next state with a "switch" instruction. (The switch instruction can be implemented *very* efficiently, at least on x86.) If there is no appropriate next state for the "switch", we jump to a failure state that returns false. Each accepting DFA state also has a transition that recognizes the terminating null character and jumps to a success state that returns true.

To make Just In Time (JIT) compilation easy to use, we wrap our LLVM code generator such that it performs a dangerous cast from `String` to an ASCII string. The final function signature is

```
compile :: Ord stateLabel =>
             MapDFA stateLabel Word8 -> IO (String -> Bool)
```

To get a `MapDFA` over `Word8` out of a `RegularExpression Char`, we take advantage of the `Functor` instance of `RegularExpression` and a `Char -> Word8` function such as `Data.ByteString.Internal.c2w`.

Future Work

There is a working library backing this code, called *Automata* [2]. I'm just a beginner to Haskell, so the code might cause bleeding of the eyes. It's a new library, with much low-hanging fruit for future improvement. Some bigger future improvements could include standards-compliant parsers and backends, capture groups (implicitly: conflict detection and the ability to make lexers), support for actions on transitions like Ragel [3], transducer support, better Unicode support (via transducers, maybe), and output to object code. I did my best—help wanted.

My main algorithm reference was the Dragon Book [4]. Russ Cox's Regular Expression Matching Can Be Simple and Fast [5] was part of the inspiration.

Thanks to Zachary Kincaid for numerous insights, significant help with content and editing, and for cudgelling me to give Haskell a second look. Thanks to Brent Yorgey for encouragement and fine editing.

References

[1] `http://regex.info/blog/2006-09-15/248`.

[2] `http://workshop.arandonohue.com/Automata/`.

[3] `http://www.complang.org/ragel/`.

[4] M.S. Lam, R. Sethi, J.D. Ullman, and A. Aho. **Compilers: Principles, Techniques and Tools**. Addison-Wesley (2007).

[5] `http://swtch.com/~rsc/regexp/regexp1.html`.

Iteratee: Teaching an Old Fold New Tricks

by John W. Lato ⟨jwlato@gmail.com⟩

*The programming technique of **enumeration-based I/O**, introduced by Oleg Kiselyov [1], has recently gained some attention within the Haskell community. Despite growing interest in the topic in general and the **iteratee** package, documentation has been lacking; people frequently cannot understand what it is, how it works, what it offers, or anything else. This document aims to provide an introduction to enumeration-based I/O by constructing an iteratee library from first principles, comparing the result with the iteratee-0.3.x package and several of Oleg's implementations.*

Haskell I/O

The main intent of enumeration-based I/O is to provide an alternative to the current widespread I/O techniques based upon low-level IO code or laziness. The predominant paradigm for I/O is the **cursor**. A cursor is a data structure that holds the current state of an I/O resource and provides a function to retrieve the next data available from that resource. Most Haskellers are familiar with `Handle`, based upon the venerable C handle and its distinguished history in imperative programming.

While handles and other cursors are generally available in functional languages, their position is slightly less illustrious. Handles in Haskell suffer from the following shortcomings:

1. Lack of composability: Handle-based code usually means writing explicit I/O loops. Haskellers are typically advised to avoid this practice because it can severely limit the composability of the resulting functions.

2. Not functional: Handle-based code is not an idiomatic functional programming style. Ideally, we would write functions that transform input to output, but low-level I/O loops require that we manually process individual chunks of data.

3. It's ugly: this is a consequence of the second point. As far as I'm concerned, if I'm writing code like this I may as well use C.

Due to these severe shortcomings, Haskell provides many functions and libraries to minimize programmer exposure to the details of cursor-based I/O. These range from built-in Prelude functions such as `getContents` and `hGetContents` to sophisticated libraries like **ByteString**. These libraries provide a very nice functional interface by using lazy I/O to manage the details behind the scenes. Unfortunately lazy I/O has shortcomings of its own.

Lazy I/O works by creating a lazy data structure in which chunks of data are filled in on demand. When processing a file, a function will operate on a small chunk, then release that memory and request the next chunk from the lazy data structure as needed. Since the underlying resource must be kept available until the computation has finished, and the compiler is free to re-order computations, it can be very difficult to manage resources with lazy I/O. If the programmer waits for garbage collection to free the resource, she doesn't know when it will be available again. This is unacceptable with scarce resources like network sockets on a high-performance web server. The programmer must perform manual resource management by tracking resources and ensuring that data is fully evaluated by liberally applying `seq` and `deepseq`, adding clutter and more opportunity for errors.

The desired API

Enumeration-based I/O in general aims to overcome these problems. The features we would like to see are:

- Composable
- High-level (the user shouldn't ever need to open a `Handle` or deal with manual chunking)
- Robust
- Predictable automatic resource management
- Performance on par with current lazy I/O implementations

It so happens we can achieve all of these goals by turning a cursor inside-out. A cursor works by giving the user a function to pull more data on demand. In contrast to this, an enumerator pushes data to a data structure that processes data as it's received. By taking the pull-based cursor and converting it into a

push-based enumerator, we can describe an interface that reclaims composability and good functional style along with the other goals above. Perhaps surprisingly, the basic functionality of this interface has been available in Haskell for some time in the form of the versatile `foldl`.

Enter Iteratees

The Haskell Prelude function `foldl` has the familiar type:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Generally, a fold is a higher-order function that takes three arguments: a combining function, a starting accumulator value, and a list of data elements. It successively feeds each data element into the accumulator using the combining function, producing the final accumulator as output. In other words, a fold creates an enumeration by iterating over input data with an accumulator, which we call the **iteratee**. The iteratee is responsible for maintaining the running state of the computation. This is pretty close to the interface we would like for I/O. If we consider the iteratee as a computation that is fed input by an enumerator, we will ideally be able to compose iteratees to generate larger computations to be run within a single enumerator.

Behavior

I find it useful to begin with an informal description of the desired behavior for our code. An iteratee is a **stream processor** – it takes a number of elements from an input stream, performs a calculation with them, and returns the value of the calculation. After completion, the elements used by the iteratee should be removed from the stream. As a stream processor, an iteratee must take all elements in order; skipping an element is not permitted. If an iteratee does not use an element for its calculation but still requires input, that skipped element must be removed from the stream.

We would also like to support composition of both multiple iteratees and multiple enumerators. By composing two iteratees, the first iteratee should run its calculation and upon completion, the second iteratee should perform its calculation with the remaining stream elements. Composition of enumerators should create a new enumerator that creates a stream of all the elements of the first enumerator followed by all the elements in the second enumerator.

Another important question is the matter of equality. What does it mean for two iteratees to be equal? Informally, we can say that iteratees are equal if, for all input streams sufficient to run both iteratees to completion, both iteratees consume the

same number of elements from the stream and calculate equal outputs. Although informal, this definition is simple to reason with and proves very useful.

First Implementation

Let's begin with a few simple types that will let us capture this behavior. We'll need a type to represent the notion of a data stream, as well as the iteratee itself.

```
data StreamG el = Empty | El el | EOF
data IterV el a = Done a (StreamG el)
                | Cont (StreamG el -> IterV el a)
```

Listing 14: iteratee definition

Remark 1. Our `StreamG` differs from the usual `List` and `Stream` definitions in that it does not attempt to to capture the entire stream. Rather, it holds only a single element. The full stream could be properly represented as `[StreamG el]`. Our `StreamG` is used to thread stream information such as end-of-file through composed iteratees.

The `IterV` computation can be finished, in which case it has a result and a stream element, or it can be a continuation that takes an input element to yield a new computation. In the done case it is necessary to keep track of the current stream in case a computation either didn't use the final element or to represent end-of-file.

Along with `IterV`, we also need an `enum` function to feed data to an `IterV` and a `run` function to retrieve the final result, shown in Listing 15.

It is helpful to think of three categories of iteratees. The first are iteratees that take a finite amount of input before they are in a done state. The iteratee analogs of functions like `head` and `drop` are in this category. The second category are iteratees that will consume an entire stream of input and only return a value on EOF. Functions like `sum` and other running totals would be in this category. The third category are iteratees which never enter the done state, not even upon receiving EOF. We refer to these as divergent. The function `run` must handle each of these conditions by sending EOF to incomplete iteratees, checking the result and returning Nothing if the iteratee still has not reached a final calculation.

The `enum` function is essentially a specialized fold, albeit with some important differences. The most obvious is the type. While the first argument to `foldl` is a function to combine each new element with the accumulator, our `IterV` incorporates this step automatically. The other major difference is the iteratee's ability

```
enum :: IterV el a -> [el] -> IterV el a
enum i [] = i
enum i@(Done _ _) _ = i
enum (Cont k) (x:xs) = enum (k (El x)) xs

run :: IterV el a -> Maybe a
run (Done x _) = Just x
run (Cont k) = run' (k EOF)
  where
    run' (Done x _) = Just x
    run' _ = Nothing
```

Listing 15: iteratee input and output

to signal to the enumerator when it has finished. This gives the desirable property that iteratees are able to stop consuming input on completion in the same manner as `foldr` or lazy I/O.

Some iteratees

Let's write a few iteratees. Analogs of many standard list operations are directly expressible. Some basic ones can be defined as shown in Listing 16.

Our iteratee `head` will return the first element it receives as input and is then complete. Upon receiving `Empty`, it returns a continuation to the same function. This means that upon receiving `Empty`, the iteratee will continue to replicate itself until it receives either an element or `EOF`. We will see this pattern of behavior on `Empty` repeated throughout many iteratees.

Composition

Now that we have some iteratees, we can describe a mechanism of composition. Fortunately, our `IterV` forms a monad. The `Monad`, `Applicative`, and `Functor` instances can be defined as shown in Listing 17.

Remark 2. Proving the monad laws for iteratees is a tedious exercise due to the multiple case matches, so I will omit it. However, the process leads to some important observations. One result is that it is much harder (perhaps impossible?) to prove the laws for divergent iteratees. If we limit ourselves to iteratees that produce a `Done` result, we can use induction from the base case.

```
head :: IterV el (Maybe el)
head = Cont step
  where
    step (El el) = Done (Just el) Empty
    step Empty = Cont step
    step EOF  = Done Nothing EOF

-- Return the first element of the stream
peek :: IterV el (Maybe el)
peek = Cont step
  where
    step c@(El el) = Done (Just el) c
    step Empty = Cont step
    step EOF = Done Nothing EOF

-- Some iteratees are in the Done state before taking input
drop :: Int -> IterV el ()
drop 0 = Done () Empty
drop n = Cont step
  where
    step (El _) = drop (n-1)
    step Empty = Cont step
    step EOF = Done () EOF

-- Iteratees frequently need to keep an accumulator
length :: IterV el Int
length = Cont (step 0)
  where
    step acc (El _) = Cont (step (acc+1))
    step acc Empty = Cont (step acc)
    step acc EOF = Done acc EOF
```

Listing 16: three simple iteratees

```
instance Monad (IterV el) where
  return x = Done x Empty
  m >>= f = case m of
    Done x str -> case f x of
      Done x' _ -> Done x' str
      Cont k    -> k str
    Cont k -> Cont (\str -> k str >>= f)

instance Functor (IterV el) where
  fmap f (Done x str) = Done (f x) str
  fmap f (Cont k) = Cont (fmap f . k)

instance Applicative (IterV el) where
  pure x = Done x Empty
  (Done f str) <*> i2 = fmap f i2
  (Cont k) <*> i2 = Cont (\str -> k str <*> i2)
```

Listing 17: Monad and related instances

An important feature of these instances is that the stream state is threaded through the iteratees as they are combined. The definition of `(>>=)` shows that when the first iteratee has finished, any extra data is passed to the second iteratee, replacing stream data in the second iteratee as necessary. Generally, an iteratee which is already in the done state, such as `drop 0`, doesn't have valid stream data so it can be safely discarded.

Exercise 3. Using the `EOF` constructor, trace stream threading through two iteratees combined with `(>>=)`.

Now we can compose iteratees using standard functions from `Control.Monad` and `Control.Applicative`. As an example, Listing 18 shows an iteratee to drop alternate elements from a list.

In order to use an iteratee, first we enumerate a stream over it with `enum`, then we `run` the result, as shown in Listing 19.

In our definition of `alternates`, we sequence a **bounded** number of `drop1keep1` functions. If instead we had written `alternates` with `repeat`, we would have created an infinite iteratee by chaining together an infinite number of base functions. Such an iteratee results in an infinite loop when run.

Exercise 4. Although we cannot combine iteratees with `repeat`, it is possible to make a function with the type `IterV el a -> IterV el [a]` that functions like

```
drop1keep1 :: IterV el (Maybe el)
drop1keep1 = drop 1 >> head

alternates :: IterV el [el]
alternates = fmap catMaybes . sequence . replicate 5 $ drop1keep1
```

Listing 18: Iteratee composition

```
*Main> let alts = enum alternates [1..10]
*Main> run alts
Just [2,4,6,8,10]
*Main> let alts2 = enum alternates [1..]
*Main> run alts2
Just [2,4,6,8,10]
*Main>
```

Listing 19: Running an iteratee

`sequence . repeat`. This function will pattern match on the input stream and end processing on receiving `EOF`. Try to implement it.

Enter Monads

At this point we've looked at iteratees and enumerators, but we have yet to examine I/O. Other than our instance for composing iteratees, we don't have any monads! That will have to change.

Monadic enumerators

Interestingly, although most of the I/O difficulties described earlier involve input, adapting our iteratees to process input with `IO` is actually simpler than output. We can use our current implementation by adding a new type, a **monadic enumerator**, and a few simple functions to create our first monadic enumerator (Listing 20).

The function `enumHandle` enumerates over a `Handle`, feeding input to an iteratee until the result is calculated. Like our previous enumerator, `enumHandle` takes an `IterV` and returns an `IterV`, except its return value is within `IO`. Note that when the iteratee is finished, `enumHandle` returns a value immediately rather than continuing to enumerate over the handle.

```
type EnumeratorM el m a = IterV el a -> m (IterV el a)

enumHandle :: Handle -> EnumeratorM Char IO a
enumHandle h iter = loop iter
  where
    loop i@(Done _ _) = return i
    loop i@(Cont k) = do
      isEOF <- hIsEOF h
      if isEOF then return i else hGetChar h >>= loop . k . El

enumFile :: FilePath -> EnumeratorM Char IO a
enumFile fp i = bracket
                  (openFile fp ReadMode)
                  (hClose)
                  (flip enumHandle i)
```

Listing 20: A monadic enumerator

Our two monadic functions together have several important safety features. First of all, the `Handle` cannot leak from within `enumFile`. It is never given to the iteratee, so it cannot escape as part of the return value. Due to `bracket`, it is guaranteed to be closed even in the presence of exceptions. Also, the iteratee is fully evaluated (or has consumed all input and returned a continuation) and the handle is closed immediately upon completion of the IO operation produced by `enumFile`, greatly simplifying reasoning about resource usage.

Note that `enumHandle` does **not** send `EOF` to the iteratee at the end of the file. Since most iteratees will complete upon receipt of `EOF`, doing so would make composition of enumerators impossible. By manually sending EOF as necessary via `run`, we can compose enumerators and feed data from multiple inputs to a single iteratee.

```
lengthOfTwoFiles :: FilePath -> FilePath -> IO (Maybe Int)
lengthOfTwoFiles fp1 fp2 =
  fmap run $ ((enumFile fp1) >=> (enumFile fp2)) length
```

Listing 21: Composition of monadic enumerators

Exercise 5. Prove that each `Handle` will be closed upon the completion of each `enumFile`. You may find it helpful to trace the recursion through `enumHandle`.

Monadic Iteratees

We now have a working model for input, but what about output? As I hinted earlier, this topic is more complicated to implement, although the result is an incredibly powerful tool.

Let's return to the original conception of I/O based enumerations and iteratees as the interior of a fold. We have examined how input is a data stream that can be enumerated over. In this model, what does it mean for a calculation to produce output? If the calculation is represented within the iteratee, then we can conclude that the iteratee itself must enumerate data over some structure to produce output.

Our only primitive for output is a `Handle`, so let's try that. We can enumerate data over a handle by repeatedly calling a function to put data to it, such as `hPutChar :: Handle -> Char -> IO ()`. This function provides exactly the desired operation! The output enumerator, *i.e.* the iteratee, will repeatedly call `hPutChar` with every element of output. The `Handle` serves as a "real-world iteratee", collecting input provided by our enumerator and "doing something" with it.

Remark 6. It's interesting that, from a functional perspective, handles provide the wrong interface for input but exactly the right interface for output. I would conjecture that the primary attribute of a "good" interface is that it gets push-vs-pull correct. That is, push-driven evaluation is favored.

Unfortunately our current implementation does not allow us to embed monadic actions within an iteratee. To do so, we need a new definition for `IterV`, as well as another new type.

```
data IterVM el m a = DoneM a (StreamG el)
                   | ContM (StreamG el -> Iteratee el m a)

newtype Iteratee el m a = Iteratee
                          { runIter :: m (IterVM el m a) }
```

Listing 22: Monadic Iteratees

Whereas our previous implementation used only the `IterV` type, our monadic implementation requires an `Iteratee` type and the `IterVM` helper. The intention is that we would create data with type `Iteratee` and only use `IterVM` internally and for certain enumeration constructs.

Most of our previous code needs only minor changes to be compatible with the new formulation. The `Monad` instance for `Iteratee` is very similar to the `IterV`

instance. Since none of `head`, `peek`, `drop`, or `length` use monadic effects, they simply need a few `return`s added. We could also write a function to automatically lift them to monadic iteratees.

```
liftIter :: Monad m => IterV el a -> Iteratee el m a
liftIter (Done x str) = Iteratee . return $ DoneM x str
liftIter (Cont k)     = Iteratee . return $ ContM (liftIter . k)
```

Even `enumHandle` can be adapted in a straightforward manner.

It is very useful to write `MonadTrans` and `MonadIO` instances for our monadic iteratees:

```
instance MonadTrans (Iteratee el) where
  lift m = Iteratee $ m >>= \x -> return (DoneM x Empty)

instance MonadIO m => MonadIO (Iteratee el m) where
  liftIO = lift . liftIO
```

An unfortunate consequence of monadic iteratees is that the monad parameter of an iteratee cannot be generally independent from that of an enumerator. This is apparent when we examine how a monadic enumerator, *e.g.* enumHandle, recursively interleaves its own monadic actions with evaluation of the iteratee state. It is possible to write enumerators of the form

```
(MonadTrans t, Monad m) =>
    Iteratee el m a -> t m (Iteratee el m a)
```

However, in practice this is only rarely more useful than requiring iteratees and enumerators to share a common monad.

Equipped with monadic iteratees and a `MonadTrans` instance, we can now accomplish some pretty fancy tricks. We'll first define a simple output function, and then a function that writes a message every 100 elements.

The `throbber` function (Listing 23) demonstrates an exciting feature of iteratees. They give meaning to interleaving monadic actions with reading chunks of data. This makes it very easy to write iteratee-based code that updates a progress bar, incremental logs, or any other action that you wish to embed within a long-running computation. The design of most lazy I/O systems makes this impossible, at least without digging into low-level implementation details. With iteratees it's both simple and high-level. With one function hinted at earlier, our `throbber` could be defined using composition (Listing 24).

```
-- Write all data in a stream to a file.
streamToFile :: FilePath -> Iteratee Char IO ()
streamToFile fp = Iteratee (openFile fp WriteMode >>= go)
  where
    go h = return $ ContM (step h)
    step h (El el) = Iteratee (hPutChar h el >> go h)
    step h Empty   = Iteratee (go h)
    step h EOF     = Iteratee (return $ DoneM () EOF)

-- Create a running count of input.
throbber :: Iteratee el IO ()
throbber = Iteratee (cont $ step (100 :: Int) 99)
  where
    step acc 0 (El _)   = Iteratee
                            (printf "Read %d chars\n" acc >>
                             cont (step (acc+100) 99))
    step acc cnt (El _) = Iteratee (cont $ step acc (cnt-1))
    step acc cnt Empty  = Iteratee (cont $ step acc cnt)
    step _   _   EOF    = Iteratee (return $ DoneM () EOF)
    cont                = return . ContM
```

Listing 23: output with iteratees

```
sequenceI_ :: [Iteratee el m a] -> Iteratee el m ()
-- implementation left as exercise

throbber2 :: Iteratee el IO ()
throbber2 = sequenceI_ $ map i ([100,200..] :: [Int])
 where
  i n = liftIter (drop 100) >> lift (printf "Read %d elems\n" n)
```

Listing 24: No low-level work required

Nested enumerators

There remains one major implementation detail to examine. We have seen how to represent functions such as `head`, `drop`, and `length`, and could generally work out fold-like functions. But is it possible to represent a function like `filter` or `map`?

To address these, we need to introduce a new concept, the **nested enumerator**. A nested enumerator, or "enumeratee", is a type of stream transformer. It takes as input elements of the outer stream, converts those to an inner stream, and finally processes the inner stream using a provided iteratee. An enumeratee thus functions as both an iteratee and enumerator, from the perspective of the outer stream and inner stream respectively.

We define the most general enumeratee as:

```
type EnumerateeM elOuter elInner m a =
   Iteratee elInner m a
   -> Iteratee elOuter m (Iteratee elInner m a)
```

To see an enumeratee in action, we first examine `filter`, shown in Listing 25. In order to simplify the operation, we'll examine a non-monadic version.

A filter is a function that takes a predicate and a stream, and discards all stream elements for which the predicate is false. As a nested enumerator, the outer stream is the complete data stream and the inner stream is the filtered result. An iteratee-filter takes one extra argument, an iteratee to process the inner stream. Since the elements themselves are not altered, the inner and outer element types are identical.

The iteratee `mapIter` is implemented similarly.

A common feature of enumeratees is that the inner iteratee needs to be run and pattern matched upon to determine if more elements need to be taken from the outer stream. This functionality can be captured in a small helper function, greatly easing the creation of nested iteratees.

Nested iteratees are necessary for many useful functions. They are used for several forms of `take` in addition to `filter` and `mapIter`. The very general function `convStream` is also a nested iteratee:

```
convStream :: Iteratee elOuter m [elInner]
              -> EnumerateeM elOuter elInner m a
```

The first argument to `convStream` converts from outer elements to inner elements. Unlike `mapIter`, there need not be a one-to-one correspondence between elements. This function can create many-to-one, many-to-none, one-to-many, and even none-to-many relationships between the inner and outer streams. One example use would be combining each pair of `Word8`s into a single `Int16` to convert a stream of bytes into a stream of 16-bit integers.

```
filter :: (el -> Bool) -> IterV el a -> IterV el (IterV el a)
filter pred i@(Done _ _) = return i
filter pred (Cont k) = Cont step
  where
    step e@(El el) | pred el = filter pred (k e)
    step EOF = Done (k EOF) EOF
    step _ = Cont step

mapIter :: (elOuter -> elInner)
           -> IterV elInner a
           -> IterV elOuter (IterV elInner a)
mapIter f i@(Done _ _) = return i
mapIter f (Cont k) = Cont step
  where
    step (El el) = mapIter f (k (El $ f el))
    step Empty = Cont step
    step EOF = Done (k EOF) EOF
```

Listing 25: Enumeratees

Alternate implementations

One confusing aspect of enumeration-based I/O is the multiplicity of implementations. The available packages and writings mention pure and monadic variants, CPS-style, and other implementations. With just a small bit of familiarity it becomes easy to navigate.

Thus far, all available implementations can exist in both pure and monadic forms. A pure implementation does not permit sequencing of monadic effects within an iteratee, whereas a monadic form does. This article presents a pure iteratee as `IterV` and a monadic version as `Iteratee` (with `IterVM`). As previously noted, pure iteratees are sufficient for processing input as the `IO` operations can take place within the enumerator.

The implementation in this article is substantially the same as the implementations given in **iteratee-0.1.0** [2], as well as in Oleg Kiselyov's **IterateeM.hs** [3]. A related implementation can be formulated as shown in Listing 26. This implementation is used in **iteratee-0.2** [2] and higher versions, and is also in the comments of Oleg's implementation.

The difference is when data is fed to the iteratee. In the first implementation, an iteratee can be in the done state before receiving any data. All enumerators must check the iteratee first, then provide data if necessary. In the second implemen-

```
data IterVM el m a = DoneM a (StreamG el)
                   | ContM (Iteratee el m a)

newtype Iteratee el m a =
  Iteratee { runIter :: StreamG el -> m (IterVM el m a) }
```

Listing 26: An alternative implementation of iteratees

tation, all iteratees require at least one datum to produce an output that can be checked by the enumerator. In other respects they are identical, although effects of this difference can have consequences in other parts of the library design.

A final implementation to consider is a continuation-passing style, as given by Oleg Kiselyov [4] and to be implemented in **iteratee-0.4.0**. CPS-style iteratees use an implementation like the following, which requires `RankNTypes`.

```
newtype Iteratee el m a =
  Iteratee {
    runIter :: forall r.
      (a -> Stream el -> m r) ->
      ((Stream el -> Iteratee el m a) -> Maybe ErrMsg -> m r) ->
      m r
  }
```

CPS-style iteratees have some advantages over other representations. They only introduce one type compared to the two used for our other monadic iteratees. They can be composed with standard composition operators such as `>>=` and `(.)`, whereas other implementations fare better with custom combinators. Another welcome benefit is that low-level iteratees tend to have simpler definitions.

Performance

The last of our design goals was to achieve performance comparable to the current state-of-the-art lazy I/O structures. The implementation in this article predictably fails miserably at this test. Reading and writing one character at a time is slow.

The solution is to change our `StreamG` to hold a chunk of data rather than a single element. This allows I/O to work with a large buffer of data. As our iteratees are already designed to thread elements through `>>=`, this ends up being a small adaptation. Each iteratee is responsible for reading as much data as required from a chunk, removing that data from the chunk, and passing the remaining data along to the next iteratee. We can also remove the `Empty` constructor, as nearly every suitable data type also has an empty representation.

The choice of buffer has a large impact on performance. One obvious choice is a list. An advantage of a list is that it is polymorphic over elements, however it suffers from well-known performance problems, particularly with larger amounts of data. Strict, packed data types such as `ByteString`s offer excellent performance, but with a loss of generality.

In the **iteratee** package, the decision was to use a polymorphic container with the `ListLike` type class [5]. By allowing the user to choose the implementation, the correct container for a given application can easily be selected. Different containers can even be mixed in the same function, for example using a `[ByteString]`. Although the `ListLike` class is quite comprehensive, only a few functions are needed for the iteratee implementation. Aside from `empty` and `null`, `take`, `drop`, and `splitAt` are the most widely used functions. When used with suitable data structures, the performance of enumeration-based code is highly competitive.

Further explorations

I hope that, having read this far, the reader will be able to pick up an iteratee implementation and begin to make use of it right away. Nevertheless there are a few remaining items to highlight.

Most iteratee implementations will also thread error conditions through the iteratee. This adds several levels of complexity to the implementation, but also provides a rich set of error handling operations.

The interested reader is strongly encouraged to read all of Oleg Kiselyov's work on iteratees. His comments within source files offer insightful discussion of various consequences of certain design decisions, particularly regarding the order of effects.

Iteratee-based code tends to be less prone to space leaks than the equivalent lazy-I/O code. It is generally difficult for the programmer to unintentionally hold references to old data. Of course if the user explicitly retains old data iteratees offer no protection.

I have frequently heard reports from Haskellers (including highly-talented and experienced users) that the only way they could understand enumeration-based I/O was by re-implementing it themselves. I hope that this document will make a contribution towards greater understanding of iteratees, and allow readers the opportunity to explore this exciting new development without rebuilding everything themselves.

About the author

A musician by training, John fell down the Haskell-hole while searching for less-verbose alternatives to Python and Java. He is still a Haskell journeyman as he has yet to write a monad tutorial.

References

[1] Oleg Kiselyov. Iteratee io: safe, practical, declarative input processing. `http://okmij.org/ftp/Haskell/Iteratee/IterateeIO-talk.pdf`.

[2] John W. Lato. Package 'iteratee'. `http://hackage.haskell.org/package/iteratee`.

[3] Oleg Kiselyov. IterateeM.hs. `http://okmij.org/ftp/Haskell/Iteratee/IterateeM.hs`.

[4] Oleg Kiselyov. IterateeMCPS.hs. `http://okmij.org/ftp/Haskell/Iteratee/IterateeMCPS.hs`.

[5] John Goerzen. Package 'ListLike'. `http://hackage.haskell.org/package/ListLike`.

Playing with Priority Queues

by Louis Wasserman ⟨lowasser@uchicago.edu⟩

We explore Haskell implementations of priority queues, ranging from the very simple, suitable for beginner Haskell programmers, to a rich implementation of binomial queues with interesting mathematical connections, which will be entertaining for more experienced Haskellers. Many of these approaches were explored in the process of building a priority queue implementation intended for the Haskell Platform.

Preliminaries

Priority queues are one of the most ubiquitous data structures, taught in every introductory data structures class and used for a tremendous variety of applications. Briefly, priority queues are collections of elements that efficiently support inserting elements, and extracting the minimum element. Many priority queue implementations offer a number of other operations, as well, such as union or unordered traversal. Several Haskell libraries make use of priority queues, including `fgl` in Dijkstra's algorithm, `primes` in efficient primality sieving, and `huffman` in the efficient generation of Huffman codes.

Recently, I worked on designing a priority queue implementation to submit to the Haskell Platform. Such an implementation needed to be reliable, efficient, and effective at as many different tasks as possible. This article discusses some of the priority queue implementations we considered, and our adventures with priority queues.

There are many different ways of implementing the priority queue abstraction, so let's start by picking one. We'll define a type class for generic, portable priority queues as shown in Listing 27.

Note that q has kind $(* \to *)$, that is, it takes a type argument, a, representing an element type. Furthermore, our priority queues will support extracting the

```
class PriorityQueue q where
  extractMin :: Ord a ⇒ q a → Maybe (a, q a)
  (⊎) :: Ord a ⇒ q a → q a → q a   -- the union operation
  singleton :: Ord a ⇒ a → q a
  insert :: Ord a ⇒ a → q a → q a
```

Listing 27: The *PriorityQueue* class

minimum element. If the queue is empty, *extractMin* returns *Nothing*, otherwise it returns *Just* the minimum element, and the queue with that element deleted.

These are all the key priority queue operations. We might want other operations – *size*, *deleteMin*, *peekMin*, or *null*, for instance – but they're all trivial to implement.

Typically, we will implement

$$insert\ x\ q = singleton\ x \uplus q$$

but on occasion, there might be a more efficient implementation.

Remark 7. One of our goals for the `pqueue` package was to also be able to provide some of the functions common to other `containers` data structures, including *filter*, *partition*, and an unordered *toList*. As we work through different priority queue implementations, experienced users might find it entertaining to try to implement these methods for each data structure in turn.

Heaps

Priority queues are typically based on labeled trees that have the following property:

Definition 8 (Heap property)**.** A tree has the **heap property** if the value of every node is less than or equal to the value of each of its children. Equivalently, the value of each node is less than or equal to the value of all of its descendants.

Remark 9. In our discussion, we will frequently conflate nodes in the tree with their values. When we talk about nodes as if they are ordered, for instance, we are actually referring to the values of those nodes.

Corollary 10. *The root of any tree with the heap property is the minimum node. Moreover, the second-smallest node in any tree with the heap property is one of the children of the root.*

This is why trees satisfying the heap property are typically the basis for priority queues: because the minimum element is the root, and we don't have far to look to find the next smallest element.

Amortization

Each of the implementations that we'll be investigating make use of a concept called amortization.

Definition 11. Essentially, the **amortized runtime** of a function is the average runtime over a worst-case *sequence* of operations. As an example, if we have n operations which must run in a total of $O(n)$ time, it doesn't necessarily follow that each operation must run in $O(1)$ time. Indeed, it would be perfectly acceptable if $O(1)$ of those operations ran in $O(n)$ time! Amortized analysis is a technique that allows us to analyze data structures that behave in this way.

While there are a number of ways to prove amortized runtime bounds, the most common, and the only one we will use, is called the potential method. In the potential method, we imagine that we're "paying for" future computations in advance, storing unused computation time in a "savings account" of runtime. Specifically, we define a function Φ on every possible state of our program or data structure, representing accumulated savings. Suppose S_0 is the initial state of the program, that $\Phi(S_0) = 0$, and that $\Phi(S) \geq 0$ for all states S. Intuitively, this says we start out without any savings, and that we never go into "debt."

Suppose that S_i is the state of the program at time step i, and that t_i is the actual time taken to get from S_i to S_{i+1}. Define the amortized time of operation i, a_i, as

$$a_i := t_i + \Phi(S_{i+1}) - \Phi(S_i)$$

Essentially, if Φ decreases during operation i, then we use the decrease in potential to "pay for" that much computation. If Φ increases, then we're paying for more computation time now, but putting the extra time into "savings" to be used on later computations.

Intuitively, since we guarantee that our "savings" never go into the red, we have always paid for all the computation we do. The math verifies this intuition:

$$\sum_{0 \leq i < n} a_i = \Phi(S_n) - \Phi(S_0) + \sum_i t_i = \Phi(S_n) + \sum_i t_i \geq \sum_i t_i$$

so that a bound on $\sum_i a_i$ is an upper bound on the total runtime. Therefore, if we can successfully bound the amortized runtime of each operation, then that gives us bounds on the total runtime of any sequence of operations.

Remark 12. Given the amortized performance of a collection of operations, we can accurately bound the total runtime of any sequence of those operations. In some cases, however, programmers may be concerned about real-time performance, that is, that each individual operation takes a reasonable amount of time. In these cases, amortized bounds may be inadequate.

Let's do an example. Consider the following code, which takes a binary number implemented as a $[Bool]$ (a list of bits, least significant first) and increments it.

$$\begin{aligned}
&incr :: [Bool] \to [Bool] \\
&incr\ (False : bs) = True : bs \\
&incr\ (True : bs) = False : incr\ bs \\
&incr\ [\,] = [True]
\end{aligned}$$

Theorem 13. *Suppose that we start with an empty list. Then incr runs in amortized time $O(1)$, even though its worst-case performance is $O(\log n)$.*

Proof. Let Φ map a $[Bool]$ to the number of *True* elements of the list. Clearly, starting at the empty list, this value starts out at 0 and remains nonnegative.

Suppose we view k elements of the list during any one increment operation. A little thought shows that the list must have looked like one of these:

$$[\overbrace{True, True, \ldots, True}^{k-1}, False, \ldots] \qquad \text{or} \qquad [\overbrace{True, True, \ldots, True}^{k-1}]$$

which will be replaced by

$$[\overbrace{False, False, \ldots, False}^{k-1}, True, \ldots] \qquad \text{resp.} \qquad [\overbrace{False, False, \ldots, False}^{k-1}, True]$$

In each case, the net change in the number of *True* elements of the list is $2 - k$, since we replaced $k - 1$ *True*s with *False*s and we added one *True*. Therefore, the amortized cost of each call to *incr* is $k + (2 - k) = 2 = O(1)$.

Since the length of the list is $O(\log n)$, we won't need any longer than that for any one operation, so $O(\log n)$ is the worst-case run time. □

Now that we have the preliminaries out of the way, let's get started with a priority queue implementation.

Skew heaps

Skew heaps were invented by Sleator and Tarjan [1], as a self-adjusting, but notably unstructured, variety of heap. Skew heaps are binary trees, but unlike many heap implementations, they don't have any other constraints. Binary heaps, for instance, are generally required to be completely balanced. The structure of a skew heap is just a vanilla binary tree:

$$\textbf{data}\ \mathit{SkewHeap}\ a = \mathit{Empty} \mid \mathit{SkewNode}\ a\ (\mathit{SkewHeap}\ a)\ (\mathit{SkewHeap}\ a)$$

What makes the skew heap special is the implementation – specifically, how you take the union of two skew heaps. To take the union of two skew heaps, here's what we do:

- Let $t_{\leqslant}$ be the tree with the smaller root, and $t_{>}$ be the tree with the bigger root.
- Take the union of $t_{>}$ with the **right** child of $t_{\leqslant}$, and store this as the right child of $t_{\leqslant}$.
- Swap the children of $t_{\leqslant}$.

Theorem 14 (Sleator and Tarjan [1])**.** *The union operation on skew heaps takes $O(\log n)$ amortized time. We won't give the proof here, but intuitively, swapping the children in this unusual fashion keeps the tree from getting excessively unbalanced. For a particularly readable proof, consult the York University lecture notes [2].*

The union operation might be a bit tricky to implement in an imperative context, but in Haskell, it's difficult to imagine a simpler algorithm! Let's try it:

$$\begin{aligned}
&\mathit{heap}_1@(\mathit{SkewNode}\ x_1\ l_1\ r_1) \uplus \mathit{heap}_2@(\mathit{SkewNode}\ x_2\ l_2\ r_2)\\
&\quad \mid x_1 \leqslant x_2 \quad = \mathit{SkewNode}\ x_1\ (\mathit{heap}_2 \uplus r_1)\ l_1\\
&\quad \mid \mathit{otherwise} = \mathit{SkewNode}\ x_2\ (\mathit{heap}_1 \uplus r_2)\ l_2\\
&\mathit{Empty} \uplus \mathit{heap} = \mathit{heap}\\
&\mathit{heap} \uplus \mathit{Empty} = \mathit{heap}
\end{aligned}$$

That was easy! Even more astonishingly, it turns out that every operation in a skew heap is implemented with $(\uplus)$, for example:

$$\begin{aligned}
&\mathit{extractMin}\ \mathit{Empty} = \mathit{Nothing}\\
&\mathit{extractMin}\ (\mathit{SkewNode}\ x\ l\ r) = \mathit{Just}\ (x, l \uplus r)
\end{aligned}$$

Corollary 15. *Every operation on skew heaps takes amortized time $O(\log n)$.*

Proof. Every priority queue operation makes at most one call to $(\uplus)$ and otherwise does $O(1)$ work. □

Shockingly, we're done – this is a complete, reasonably well-performing priority queue implementation. Once we're willing to grant the claim that $(\uplus)$ takes amortized $O(\log n)$, the implementation is exceedingly simple. Wasn't that easier than the first time your data structures professor asked you to implement a binary heap?

Pairing heaps

Pairing heaps, invented by Fredman *et al.* [3], are very nearly as simple as skew heaps, and yet are – in many contexts – one of the fastest priority queue implementations known [4]. While their time bounds are amortized, making pairing heaps a poor choice for users who need real-time performance, they offer fantastic performance for applications that don't extensively use persistence.

To begin: a pairing heap will be implemented simply as a tree where each node may have arbitrarily many children.

$$\textbf{data}\ \mathit{PairingHeap}\ a = \mathit{Empty} \mid \mathit{PairNode}\ a\ [\mathit{PairingHeap}\ a]$$

Taking the union of two pairing heaps is possibly even simpler than the skew heap implementation: we just add the bigger root to the children of the smaller root. Clearly, this will take $O(1)$ time.

$$\begin{aligned}
&\mathit{heap}_1@(\mathit{PairNode}\ x_1\ ts_1) \uplus \mathit{heap}_2@(\mathit{PairNode}\ x_2\ ts_2)\\
&\quad \mid x_1 \leqslant x_2 \quad = \mathit{PairNode}\ x_1\ (\mathit{heap}_2 : ts_1)\\
&\quad \mid \mathit{otherwise} = \mathit{PairNode}\ x_2\ (\mathit{heap}_1 : ts_2)\\
&\mathit{Empty} \uplus \mathit{heap} = \mathit{heap}\\
&\mathit{heap} \uplus \mathit{Empty} = \mathit{heap}
\end{aligned}$$

Remark 16. Note that we never store an empty heap as a child. Therefore, every child of every node will be a *PairNode*.

Exercise 17. Recall

$$\mathit{insert}\ x\ q = \mathit{singleton}\ x \uplus q$$

Observe that

$$\mathit{fromList}\ xs = \mathit{foldr}\ \mathit{insert}\ \mathit{Empty}\ xs$$

What does *fromList xs* look like if *xs* is in ascending order? In descending order?

We implement *extractMin* as follows. As usual, we need to delete the root of the heap, and merge the resulting forest into a single heap. Additionally, we need to do this in a way that keeps the tree reasonably balanced. Suppose the root has k children. The idea is to take each two successive children and "partner" them, merging them together. This crucial piece of the algorithm is the origin of the name "pairing heap." This results in $\lceil k/2 \rceil$ subtrees. Then, we do a single merging pass, unioning each one into the full tree. The whole process takes $O(k)$ time.

$$
\begin{aligned}
&\mathit{extractMin} :: \mathit{Ord}\ a \Rightarrow \mathit{PairingHeap}\ a \to \mathit{Maybe}\ (a, \mathit{PairingHeap}\ a)\\
&\mathit{extractMin}\ (\mathit{PairNode}\ x\ ts) = \mathit{Just}\ (x, \mathit{meldChildren}\ ts)\ \mathbf{where}\\
&\quad \mathit{meldChildren}\ (t_1 : t_2 : ts) = (t_1 \uplus t_2) \uplus \mathit{meldChildren}\ ts\\
&\quad \mathit{meldChildren}\ [t] = t\\
&\quad \mathit{meldChildren}\ [\,] = \mathit{Empty}\\
&\mathit{extractMin}\ \mathit{Empty} = \mathit{Nothing}
\end{aligned}
$$

Theorem 18 (Fredman *et al.* [3]). *extractMin takes amortized* $O(\log n)$ *time.*

Theorem 19 (Iacono [5]). *insert and* $(\uplus)$ *take amortized* $O(1)$ *time.*

Remark 20. With the potential function approach, it is possible that the amortized runtime of an operation could be worse than the worst-case runtime, if the operation always increases the value of the potential function. This is why the amortized runtimes of *insert* and $(\uplus)$ are not obvious – indeed, it took almost 14 years to prove.

Exercise 21. What does *extractMin* (*fromList xs*) look like when *xs* is in descending order? After k *extractMin*s? When *xs* is in ascending order?

Exercise 22. In practice, it is more efficient to deal solely with nonempty pairing heaps, and then to construct a queue type (separate from the heap type) that may be empty. It'll look something like this:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{PairQueue}\ a = \mathit{Empty} \mid \mathit{NonEmpty}\ (\mathit{PairHeap}\ a)\\
&\mathbf{data}\ \mathit{PairHeap}\ a = \mathit{PairHeap}\ a\ [\mathit{PairHeap}\ a]
\end{aligned}
$$

Modify the implementation for this approach.

Again, that's essentially the whole implementation. The analysis, of course, was difficult, but the reader can feel free to leave that to the theorists, because the implementation is simple.

Persistence with performance

Perhaps the biggest problem with this implementation is that the amortized time bounds are not persistent. Persistence is when we keep an old version of a structure present. Suppose $h = \mathit{PairNode}\ x\ ts$, and we call *extractMin* on h. The result is just *meldChildren ts*. Suppose that a and b are two more pairing heaps with roots $> x$, and consider $h \uplus a \uplus b$. The result is $h' = \mathit{PairNode}\ x\ (a : b : ts)$. Calling *extractMin* h' will duplicate the work of calling *extractMin* h, which could have been $O(n)$. This violates the idea of amortization, where we store up "computational savings" in order to pay for an expensive operation later, but here the expensive operation was performed twice.

Okasaki was the first to identify this problem, and was the first to realize that amortization in a persistent context could be solved through laziness [6]. In particular, he invented a version of pairing heaps that used this approach [7]. However, amortized bounds have only been proven in the single-threaded context, and not for Okasaki's persistent variant [8]. Interested users should examine Okasaki's lazy pairing heap implementation, although the laziness overhead makes this version significantly less competitive.

Binomial queues

More than any of the previously discussed data structures, a binomial queue is seriously difficult to implement in an imperative language. In a functional language, however, it's not very difficult. We'll be taking a somewhat different approach, though, than most Haskell implementations. I came up with this approach in the process of designing an implementation for the Haskell Platform, but though I didn't know it, Hinze had developed an equivalent approach a decade earlier [9].

Binomial trees in the type system

Definition 23. A **binomial tree** of rank k is defined inductively as follows:

- A binomial tree of rank 0 is a node with no children.
- A binomial tree of rank $k+1$ is a node with one child of each rank i, $0 \leq i \leq k$.

A **binomial forest** is a list of binomial trees, with at most one of each rank. We will assume that the trees are in ascending order of rank. A **binomial heap** is a binomial forest in which every tree satisfies the heap property.

Exercise 24. Prove that a binomial tree of rank k has size 2^k. Conclude that if a binomial forest has total size n, then it has a tree of rank k iff n has a 1 in the kth bit of its binary representation. Note that this implies that the number of trees in a binomial forest is at most $\log n + 1$.

Definition 25. Define a **straight** of rank k to be a collection of binomial trees, $t_0, t_1, \ldots, t_{k-1}$, such that each t_i has rank i. (The term "straight" is used in analogy to poker, where a straight is an sequence of cards of consecutive ranks.) Note that the children of a binomial tree of rank k form a straight of rank k.

If you have experience with Peano numbers and set theory, you may recall the construction of the natural numbers from set theory.

$$\begin{aligned} 0 &:= \emptyset & 3 &:= \{0, 1, 2\} \\ 1 &:= \{0\} & &\vdots \\ 2 &:= \{0, 1\} & k + 1 &:= k \cup \{k\} = \{0, \ldots, k\} \end{aligned}$$

When designing a binomial heap implementation, I hoped to encode the binomial tree structure directly into the type system, rather than keeping track of ranks and maintaining invariants explicitly. Since binomial tree ranks are natural numbers, we'll have to end up constructing the natural numbers in the type system in some form. It will actually turn out, however, that the most useful construction for our purposes will be very similar to the *set*-theoretic construction. Cool!

The type *rk* corresponding to a rank $k \in \mathbb{N}$ will be the type of a straight of rank k, and a binomial tree with rank *rk* with elements of type a will contain one value of type a, the label of the root, and a value of type *rk a*, the collection of its children.

```
data Zero a = Zero
    -- We want BinomTree Zero a to have no children, so this is
    -- just a unit constructor.
data Succ rk a = BinomTree rk a ◁ rk a
    -- To get the successor of rk, we add a binomial tree of rank k
    -- to the collection.
    -- This is analogous to k + 1 := k ∪ {k}.
data BinomTree rk a = BinomTree a (rk a)
    -- Recall that rk a is the type of a collection of trees of ranks
    -- 0 through k − 1.
```

We get inductively that if $rk = Succ^k\ Zero$, then a value of type *rk a* is a straight of rank k, and that a value of type *BinomTree rk a* is a binomial tree of rank k. Beautiful!

We'll need to define binomial forests in a linked-list like format, except that we'll need nodes even for skipped ranks. We will define *BinomForest rk a* to correspond to a binomial forest whose smallest rank is at least *rk*, and whose elements are of type a:

```
data BinomForest rk a =
  Nil   -- the empty forest
  | Skip (BinomForest (Succ rk) a)
        -- no tree of this rank
  | Cons (BinomTree rk a) (BinomForest (Succ rk) a)
        -- there is a tree of this rank
```

Implementing the binomial heap

A binomial heap is a binomial forest, in which each binomial tree satisfies the heap property. Since it we may have nodes of any rank ≥ 0, we'll define

```
type BinomHeap = BinomForest Zero
```

Observe that the minimum must be one of the roots. Furthermore, we can combine two roots of the same rank into a new tree of the next higher rank as follows. We use the $\wedge$ symbol to allude to the fact that we're melding just two trees, rather than a whole forest.

```
(∧) :: Ord a ⇒ BinomTree rk a → BinomTree rk a
   → BinomTree (Succ rk) a
t1@(BinomTree x1 ts1) ∧ t2@(BinomTree x2 ts2)
   | x1 ⩽ x2     = BinomTree x1 (t2 ◁ ts1)
   | otherwise = BinomTree x2 (t1 ◁ ts2)
```

Now, merging two binomial forests becomes precisely analogous to binary addition. Recalling the correspondence between the ranks of binomial trees and the binary digits of n, the correspondence becomes even more natural. We'll define the following:

```
mergeForest :: Ord a ⇒ BinomForest rk a → BinomForest rk a
   → BinomForest rk a   -- union of two binary forests
carryForest :: Ord a ⇒ BinomTree rk a → BinomForest rk a
   → BinomForest rk a → BinomForest rk a
     -- when we would "carry" a 1 in binary addition, we carry
     -- a binomial tree instead
incrForest :: Ord a ⇒ BinomTree rk a → BinomForest rk a
   → BinomForest rk a
     -- specialized for binary incrementation: instead of
     -- adding 1, we add a binomial tree
```

$$
\begin{aligned}
&\mathit{mergeForest}\ ts_1\ ts_2 = \textbf{case}\ (ts_1, ts_2)\ \textbf{of}\\
&\quad (\mathit{Nil}, \_) && \to ts_2\\
&\quad (\_, \mathit{Nil}) && \to ts_1\\
&\quad (\mathit{Skip}\ ts_1', \mathit{Skip}\ ts_2') && \to \mathit{Skip}\ (\mathit{mergeForest}\ ts_1'\ ts_2')\\
&\quad (\mathit{Skip}\ ts_1', \mathit{Cons}\ t_2\ ts_2') && \to \mathit{Cons}\ t_2\ (\mathit{mergeForest}\ ts_1'\ ts_2')\\
&\quad (\mathit{Cons}\ t_1\ ts_1', \mathit{Skip}\ ts_2') && \to \mathit{Cons}\ t_1\ (\mathit{mergeForest}\ ts_1'\ ts_2')\\
&\quad (\mathit{Cons}\ t_1\ ts_1', \mathit{Cons}\ t_2\ ts_2') && \to \mathit{Skip}\ (\mathit{carryForest}\ (t_1 \wedge t_2)\ ts_1'\ ts_2')\\
&\mathit{carryForest}\ t_0\ ts_1\ ts_2 = \textbf{case}\ (ts_1, ts_2)\ \textbf{of}\\
&\quad (\mathit{Nil}, \_) && \to \mathit{incrForest}\ t_0\ ts_2\\
&\quad (\_, \mathit{Nil}) && \to \mathit{incrForest}\ t_0\ ts_1\\
&\quad (\mathit{Skip}\ ts_1', \_) && \to \mathit{mergeForest}\ (\mathit{Cons}\ t_0\ ts_1')\ ts_2\\
&\quad (\_, \mathit{Skip}\ ts_2') && \to \mathit{mergeForest}\ ts_1\ (\mathit{Cons}\ t_0\ ts_2')\\
&\quad (\mathit{Cons}\ t_1\ ts_1', \mathit{Cons}\ t_2\ ts_2')\\
&\qquad && \to \mathit{Cons}\ t_0\ (\mathit{carryForest}\ (t_1 \wedge t_2)\ ts_1'\ ts_2')\\
&\mathit{incrForest}\ t\ ts = \textbf{case}\ ts\ \textbf{of}\\
&\quad \mathit{Nil} && \to \mathit{Cons}\ t\ \mathit{Nil}\\
&\quad \mathit{Skip}\ ts' && \to \mathit{Cons}\ t\ ts'\\
&\quad \mathit{Cons}\ t'\ ts' && \to \mathit{Skip}\ (\mathit{incrForest}\ (t \wedge t')\ ts')
\end{aligned}
$$

The $(\uplus)$ operation on binomial heaps is essentially *mergeForest* applied to a basic *BinomHeap*, and we can easily define *insert* from *incrForest*.

Theorem 26. *Merging two binomial heaps takes $O(\log n)$. Inserting an element takes amortized time $O(1)$ and worst-case time $O(\log n)$.*

Proof. Merging two binomial forests takes $O(\log n)$, since it can be done in a single traversal of both forests, and each forest has at most $\log n + 1$ trees. Inserting an element is precisely isomorphic to incrementing a binary number, and so an adaptation of that proof, in which the potential function is the number of trees in the heap, suffices. □

Now, of course, we must deal with *extractMin*. The algorithm is as follows:

- Extract the smallest root in the forest.
- Reverse its children to make a binomial forest in ascending rank order.
- Merge these children with the remainder of the binomial forest.

Theorem 27. *extractMin on a binomial heap takes $O(\log n)$.*

Proof. Visiting all roots in the forest takes $O(\log n)$. The smallest root has at most $\log n + 1$ children to reverse, and merging them with the remainder of the forest, as shown above, takes $O(\log n)$. Therefore, the whole operation will take $O(\log n)$. □

The problem, however, is that the type of the smallest root in the forest could be $BinomTree\ (Succ^n\ Zero)\ a$ for an unknown value of n. Therefore, we don't know the type of the smallest tree in the forest in advance, so we will need to invent a polymorphic approach.

The most natural solution is actually to visit the roots backwards, moving to the end of the forest, attempting an extract-min, and moving backwards to see if there is something better. Essentially, we'll be using the *foldr* programming pattern.

$$\textbf{data}\ Extract\ rk\ a = Extract\ a\ (rk\ a)\ (BinomForest\ rk\ a)$$

Let's assume that *minRoot* refers to the smallest root discovered so far – that is, the smallest root with rank $\geq k$ – and *minKey* refers to its value. Then the arguments to *Extract* represent, in order:

- The value of *minKey*.
- The children of *minKey* with rank $< k$. These form a straight of rank k, and therefore are stored as a value of type $rk\ a$.
- The forest containing the nodes in the original forest with rank $\geq k$, minus *minRoot*, unioned with those children of *minRoot* with rank $\geq k$.

Then the algorithm for

$$extractForest :: Ord\ a \Rightarrow BinomForest\ rk\ a \rightarrow Maybe\ (Extract\ rk\ a)$$

is as follows.

- If the forest is empty, return *Nothing*.
- Recurse to the next level of the forest, which we'll name *ts*. We get back a value of type $Maybe\ (Extract\ (Succ\ rk)\ a)$, which we name *succExtract*.
- If this was a *Skip* node, convert *succExtract* to a $Maybe\ (Extract\ rk\ a)$ as follows:
  - If the recursion returned *Nothing*, return *Nothing*.
  - Otherwise, *succExtract* has the form

    $$Just\ (Extract\ minKey\ (mChild \lhd mChildren)\ forest)$$

    Then we return $Just\ (Extract\ minKey\ mChildren\ (Cons\ mChild\ forest))$. Essentially, we were in the process of merging the children of *minRoot* back into the forest, and this piece of the merge did not require a carry.
- If this was a *Cons* node, then let the *Cons*ed tree be t, and let $t = BinomTree\ tKey\ tChildren$. Convert *succExtract* to a $Maybe\ (Extract\ rk\ a)$ as follows:
  - If *succExtract* had the form

    $$Just\ (Extract\ minKey\ (mChild \lhd mChildren)\ forest)$$

where $minKey < tKey$, then return

$$Just\ (Extract\ minKey\ mChildren\ (Skip\ (incrForest\ (mChild \wedge t)\ forest)))$$

Intuitively, we were merging the children of *minRoot* back into the forest, and we needed to perform a carry.

- Otherwise, this node is the new *minRoot*. Return

$$Just\ (Extract\ tKey\ tChildren\ (Skip\ ts))$$

Exercise 28. Verify the correctness of this algorithm. Verify that it runs in $O(\log n)$ time.

Implementing this is a very literal affair.

```
extractForest :: Ord a ⇒ BinomForest rk a → Maybe (Extract rk a)
extractForest Nil = Nothing
extractForest (Skip ts) = case extractForest ts of
  Nothing → Nothing
  Just (Extract minKey (mChild ◁ mChildren) forest)
      → Extract minKey mChildren (Cons mChild forest)
extractForest (Cons t@(BinomTree tKey tChildren) ts) = Just
  (case extractForest ts of
    Just (Extract minKey (mChild ◁ mChildren) forest)
      | minKey < tKey → Extract minKey mChildren
          (Skip (incrForest (mChild ∧ t) forest))
    _ → Extract tKey tChildren (Skip ts))
```

Note that when this is applied to $BinomHeap\ a = BinomForest\ Zero\ a$, that the middle argument to *Extract* will always be *Zero*, and can be discarded. Therefore,

```
extractMin heap = case extractForest heap of
  Nothing → Nothing
  Just (Extract minKey _ heap')
      → Just (minKey, heap')
```

and we're done.

Making the `pqueue` package

All of these data structures were actually considered among the implementations we wrote. Our objectives were to get reliability on data sets of all sizes and patterns, to ensure high real-time performance, and ideally, to be able to support a

wide variety of nonstandard functions common to other `containers` data structures, including *filter*, *partition*, and *foldr*.

The original, strict implementation of pairing heaps was impressively fast, especially on smaller data sets, but an implementation that couldn't deal with persistence was clearly unacceptable. The lazy, persistence-resistant implementation degenerated on larger data sets, and wasn't nearly as fast as the strict implementation – and good bounds on its performance haven't yet been proven. Moreover, users with real-time performance needs wouldn't tolerate $O(n)$ worst-case *deleteMin*, in any event.

For our binomial heap implementation, we borrowed a trick from Brødal and Okasaki [10], which was to keep track of the global minimum separately – essentially, recording the minimum key separately, and then maintaining a binomial heap containing every other value. This allows *findMin* to be implemented in $O(1)$ worst-case time. This technique can be applied to any priority queue, although the other implementations we considered already support *findMin* in $O(1)$, anyway. Implementing this optimization for the above implementation of binomial queues is a straightforward exercise.

This variation on the binomial heap implementation satisfied all our needs. We compared it to a variety of other binomial heap implementations, such as implementations that maintained a linked list of their trees, to handle sparse forests better. Note that the above implementation uses a linked list with $\log n + 1$ spaces for trees, even when only one of those spaces is in use. A size of 2^n yields the worst case, just as it is least efficient to store the binary representation of 2^n as a $[Bool]$. The sparse implementations didn't perform noticeably better, however, which I believe can be ascribed to the strict constructor unpacking that is possible with this implementation.

While *insert* has a worst-case performance of $O(\log n)$, this is no worse than a binary heap, for instance, and doesn't seem likely to be a major issue, given that its amortized performance, which is much more realistic, is $O(1)$. While there are binomial heap variants with worst-case $O(1)$ insertion, they significantly increased the size of what was already a sizable implementation, and didn't seem worth the code bloat and readability cost. Furthermore, the strict constructor unpacking was very space-efficient in the actual storing of binomial trees, which compensated significantly for these issues. On extremely large data sets, this implementation was faster than any of the other implementations, primarily due to the reduced space usage. Finally, this implementation distinctly reflects the Haskell spirit of exploiting the type system For Great Justice.

The current version of the `pqueue` package is available from Hackage [11], and the darcs repository is also available [12]. I hope to be submitting `pqueue` for consideration to the Platform in the near future, and several people have expressed support for such a project. Milan Straka and Jim Apple were particularly involved

in discussing the choice of data structure, and Ross Paterson was extremely helpful in choosing a maximally useful API.

Further reading

There are a wide variety of functional priority queue implementations, of which we have considered some of the most efficient. King [13] was the first to consider implementing binomial queues in a functional context, and wrote his examples in Haskell. Brødal and Okasaki [10] used a sequence of variations on binomial queues to implement a functional priority queue with worst-case asymptotically optimal performance for every operation, though it is slow in practice. Hinze [14] is an excellent summary of the motivation for the binomial heap structure, with examples in Haskell.

References

[1] Daniel Dominic Sleator and Robert Endre Tarjan. Self adjusting heaps. **SIAM J. Comput.**, 15(1):pages 52–69 (1986).

[2] `http://www.cse.yorku.ca/~andy/courses/4101/lecture-notes/LN5.pdf`.

[3] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. **Algorithmica**, 1(1-4):pages 111–129 (November 1986).

[4] Bernard M. E. Moret and Henry D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In **DIMACS Series in Discrete Mathematics and Theoretical Computer Science**, pages 400–411. Springer (1991).

[5] John Iacono. Improved upper bounds for pairing heaps. In **Algorithm Theory - SWAT 2000**, volume 1851 of **Lecture Notes in Computer Science**, pages 63–77. Springer Berlin / Heidelberg (2000).

[6] Chris Okasaki. The role of lazy evaluation in amortized data structures. **SIGPLAN Not.**, 31(6):pages 62–72 (1996).

[7] Chris Okasaki. Functional data structures. Technical report (1996).

[8] Chris Okasaki. Personal communication (2010).

[9] Ralf Hinze. Numerical representations as higher-order nested datatypes (1998).

[10] Gerth Stølting Brødal and Chris Okasaki. Optimal purely functional priority queues. **Journal of Functional Programming**, 6(06):pages 839–857 (1996).

[11] `http://hackage.haskell.org/package/pqueue/`.

[12] `http://code.haskell.org/containers-pqueue/`.

[13] David King. Functional binomial queues. In **In Glasgow Workshop on Functional Programming**, pages 141–150. Springer-Verlag (1994).

[14] Ralf Hinze. Explaining binomial heaps. **Journal of Functional Programming**, 9(01):pages 93–104 (1999).