

The Monad.Reader Issue 14

by Heinrich Apfelmus <apfelmus@quantentunnel.de>
and Jefferson Heard <jefferson.r.heard@gmail.com>
and Leon P Smith <leon@melding-monads.com>

29 July, 2009



Wouter Swierstra, editor.

Contents

Wouter Swierstra	
Editorial	3
Heinrich Apfelmus	
Fun with Morse Code	5
Jefferson Heard	
Hieroglyph 2: Purely Functional Information Graphics Revisited	21
Leon P Smith	
Lloyd Allison's Corecursive Queues: Why Continuations Matter	37

Editorial

by Wouter Swierstra (wouter@chalmers.se)

```
import Data.List
main = do
  let authors = ["Heinrich Apfelmus", "Jeff Heard", "Leon Smith"]
      articles = ["Fun with Morse Code"
                  , "Hieroglyph 2"
                  , "Lloyd Allison's Corecursive Queues"]
      issueNumber = 14
      editorial = makeEditorial issueNumber authors articles
      texheader = "\\documentclass{tmc}\n\\begin{document}\n"
      texfooter = "\n\\end{document}"
      writeFile "Editorial.tex" (texheader ++ editorial ++ texfooter)
makeEditorial :: Int → [String] → [String] → String
makeEditorial issueNumber authors articles =
  let intro = "I'm pleased to announce Issue "
      ++ show issueNumber
      ++ " of the Monad.Reader."
      ++ " This issue consists of the following "
      ++ show (length articles)
      ++ " awesome articles: "
      text = concat (intersperse "; " (zipWith announce authors articles))
      outro = ". Please consider writing something for the next issue!"
  in intro ++ text ++ outro
announce :: String → String → String
announce author article = author
  ++ " has written an awesome article about "
  ++ article
```


Fun with Morse Code

by Heinrich Apfelmus <apfelmus@quantentunnel.de>

In a post [1] to reddit [2], user CharlieDancey presented a challenge to write a short and clever morse code decoder. Of course, what could be more clever than writing it in Haskell? ;-)

In the following, I'll present a series of solutions that gradually include dichotomic search [3] (the straightforward generalization of binary search [4]), stack-based languages [5] or reverse polish notation [6], and finally deforestation [7] (also called fusion [8]), an optimization technique specific to purely functional languages like Haskell.

For your convenience, source code for the individual sections is available [9].

A simple first solution

Morse code [10] was designed to be produced and read, or rather heard by humans, who have to memorize the table shown in Figure 1.

To write a program that decodes sequences of dots and dashes like

```
-- --- .-. ... .    -.-. --- -... .
```

into letters, we will have to store this table in some form. For instance, we could store each letter and corresponding morse code in a big association list [12]

```
type MorseCode = String

dict :: [(MorseCode, Char)]
dict =
  [(".-"    , 'A'),
   ("-..." , 'B'),
   ("-.-." , 'C'),
   ...
```

International Morse Code

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to seven dots

A	● ■■■	U	● ● ■■■
B	■■■ ● ● ●	V	● ● ● ■■■
C	■■■ ● ■■■ ●	W	● ■■■ ■■■
D	■■■ ● ●	X	■■■ ● ● ■■■
E	●	Y	■■■ ● ■■■ ■■■
F	● ● ■■■ ●	Z	■■■ ■■■ ● ●
G	■■■ ■■■ ●		
H	● ● ● ●		
I	● ●		
J	● ■■■ ■■■ ■■■		
K	■■■ ● ■■■	1	● ■■■ ■■■ ■■■ ■■■
L	● ■■■ ● ●	2	● ● ■■■ ■■■ ■■■
M	■■■ ■■■	3	● ● ● ■■■ ■■■
N	■■■ ●	4	● ● ● ● ■■■
O	■■■ ■■■ ■■■	5	● ● ● ● ●
P	● ■■■ ■■■ ●	6	■■■ ● ● ● ●
Q	■■■ ■■■ ● ■■■	7	■■■ ■■■ ● ● ●
R	● ■■■ ●	8	■■■ ■■■ ■■■ ● ●
S	● ● ●	9	■■■ ■■■ ■■■ ■■■ ●
T	■■■	0	■■■ ■■■ ■■■ ■■■ ■■■

Figure 1: International Morse Code [11].

which we name `dict` because it's a dictionary translating between the latin alphabet and morse code. To decode a letter, we simply browse through this list to determine whether it contains a given code of dots and dashes

```
decodeLetter :: MorseCode -> Char
decodeLetter code = maybe ' ' id $ lookup code dict
```

Decoding a whole sequence of letters is done with

```
decode = map decodeLetter . words
```

so that we have for example

```
> decode "-- --- .-. ... . -.-. --- -.. ."
"MORSECODE"
```

Of course, association lists are a rather slow data structure. A binary search tree [13], trie [14], or hash table [15] would be a better choice. But fortunately, it is very easy to change data structures in Haskell. All you have to do is a qualified import of a different module, such as `Data.Map` [16], and change the definition of `dict` to

```
dict :: Data.Map.Map MorseCode Char
dict = Data.Map.fromList $
  [(".-"    , 'A'),
   ("-...." , 'B'),
   ("-.-." , 'C'),
   ...
```

and use `Data.Map.lookup` instead of `Data.List.lookup`.

Dichotomic search

Faithfully replicating the morse code table is clear and preferable, but makes for quite large source code. So, let's make an exception today and think of something as clever as we can [17].

The idea is the following: whenever an encoded letter starts with a dash '-', we know that it can't be for example an A or E, because those start with a dot '.'. And if the next symbol is a dot, then it can't be a G or M because their second symbol is a dash. With each symbol we read, more and more alternatives disappear until we are left with a single alternative. We can depict this with a binary tree as in Figure 2.

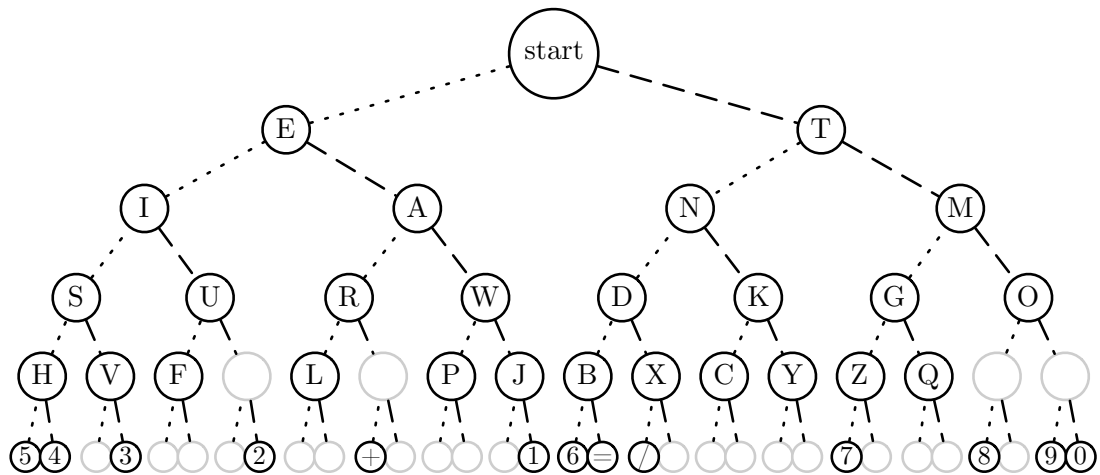


Figure 2: Tree for deciphering morse code [18].

At each node, the left subtree contains all the letters that can be obtained by adding a dot to the current letter, while the right subtree contains those letters that can be obtained with a dash. This is illustrated by means of a dotted line connected to the left subtree and a dashed line connected to the right subtree.

Now, to decode a letter, we simply start at the root of the tree and follow the dotted or dashed lines depending on the symbols read. For instance, to decode the sequence "-...", we have to go right once and then left thrice, ending up at the letter B. This procedure is also called dichotomic search [3] because at each point in our search for the right letter, we ask a yes/no question "Dot or dash?" (a dichotomy [19]) that partitions the remaining search space into two disjoint parts.

Let's implement this in Haskell. First, we assume that our dictionary is given as a tree

```
data Tree a = Leaf
            | Branch { tag    :: a
                    , left  :: Tree a
                    , right :: Tree a
                    }

dict :: Tree Char
dict = Branch ' ' (Branch 'E' ... ) (Branch 'T' ...)
```

Of course, writing out the `dict` tree in full is going to be repetitive and boring, but let's just imagine we have already done that. Then, decoding a letter by walking down the tree is simply a left fold [20] over the code word

```
decodeLetter = tag . foldl (flip step) dict
  where
    step '.' = left
    step '-' = right
```

In this case, the accumulated value of the fold is the current subtree, although the term “accumulate” is a bit misleading in that we don’t make the dictionary bigger; we make it smaller by passing to a subtree.

The curious reader may notice that we have actually implemented a trie [14].

Reverse Polish Notation

Now, let’s reduce the source code needed for the dictionary tree. In Haskell, we’d have to write something like

```
dict = Branch ' ' (Branch 'E' ... ) (Branch 'T' ...)
```

Compared to the association list, we got rid of the dots '.' and dashes '-', they have been made implicit in the structure of our tree. But we still need a lot of parenthesis and applications of the `Branch` function.

A clever way to get rid of parenthesis is reverse polish notation [6], commonly used in stack-based languages [5]. The idea is that a program in such a language is a sequence of instructions that pop and push values from a stack. For example,

```
1 2 +
```

is a program to calculate 1 plus 2. Reading from left to right, the first instruction is 1 which pushes the integer 1 onto the stack. The second instruction is 2, pushing 2 onto the stack. And finally, the instruction + pops the two topmost integers from the stack and pushes their sum onto the stack. Figure 3 visualizes this procedure.

To see how that eliminates the need for parenthesis, take a look the program

```
1 2 + 3 4 + +
```

which calculates the sum $(1+2)+(3+4)$.

To build our dictionary tree, we are going to devise a very similar language. Instead of integers, the stack will store whole trees. In analogy to the numerals, there will be an instruction for pushing a `Leaf` onto the stack; and in analogy to +, there is going to be an instruction for applying the `Branch` constructor to the two topmost subtrees.

Here’s the program for building the tree, stored as a string:

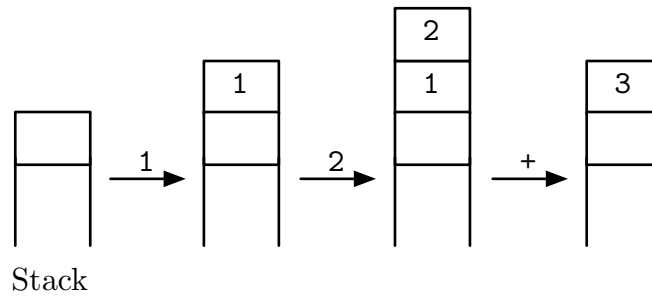


Figure 3: Execution of the stack-based program 1 2 +.

```

program :: String
program = "__5__4H__3VS__F__2 UI__L__+_ R__P__1JWAE"
++ "__6__=B__/_XD__C__YKN__7_Z__QG__8_ __9__0 OMT "

```

Each character represents one instruction. The underscore '_' pushes a **Leaf** onto the stack while all the other character push a **Branch** tagged with the character in question onto the stack. The interpreter for this tiny programming language can be written using a simple left fold [20]:

```

dict = interpret program
where
  interpret          = head . foldl exec []
  exec xs           ' _ ' = Leaf          : xs    -- push Leaf
  exec (x:y:xs)    c   = Branch c y x : xs    -- combine subtrees

```

The stack `xs` is represented as a list of trees.

Deforestation

We have found a concise way to represent the morse code dictionary in source code, but cleverness does not end here. For instance, how about eliminating the tree entirely?

Chopping trees

Call graph

More precisely, it seems wasteful to construct the dictionary tree by creating leaves and connecting branches only to deconstruct them again when decoding letters.

Wouldn't it be more efficient to interpret the morse code tree in Figure 2 as a **call graph** rather than as a data structure `Tree Char`?

In other words, imagine say a function `e` corresponding to the node labeled 'E' in the tree, and working as follows:

```
e :: MorseCode -> Char
e ('.':ds) = i ds
e ('-':ds) = a ds
e []      = 'E'
```

If the next symbol is a dot or dash, we proceed with the functions `i` or `a` corresponding to 'I' and 'A' respectively. And in case we have already reached the end of the code, we know that we have decoded an 'E'. The functions `i` and `a` work just like `e`, except that they proceed with other letters; and the morse code tree becomes their **call graph**.

A combinator...

Now, writing all these functions by hand would be most error-prone and tedious, they all look the same. But abstracting repetitive patterns is exactly where functional programming shines; we are to devise a combinator that automates the boring parts of the code for us.

What does this combinator look like? Well, the non-boring parts are the letter it represents and the two follow-up functions, so these will be parameters:

```
combinator :: Char           -- letter
            -> (MorseCode -> Char) -- function for dot
            -> (MorseCode -> Char) -- function for dash
            -> (MorseCode -> Char) -- result function
```

That's all we need; we can implement

```
combinator c x y = \code -> case code of
  '.':ds -> x ds
  '-':ds -> y ds
  []      -> c
```

which allows us to write

```
e = combinator 'E' i a
i = combinator 'I' s u
... etc ...
```

...revealed!

But wait, what have we done? The type signature of `combinator` looks exactly like that of

```
Branch :: Char          -- letter
        -> Tree Char    -- tree for dot
        -> Tree Char    -- tree for dash
        -> Tree Char    -- result tree
```

but with the type `Tree Char` replaced by `(MorseCode -> Char)`! In fact, let's rename the combinator to lowercase `branch`

```
branch c x y = \code -> case code of
  '.':ds -> x ds
  '-':ds -> y ds
  []      -> c
```

and observe that the tree of functions now reads

```
e = branch 'E' i a
i = branch 'I' s u
... etc ...
```

or

```
e = branch 'E' (branch 'I' ... ) (branch 'A' ...)
```

if we inline the function definitions. But this is of course just the tree from the section on dichotomic search on page 7 with each `Branch` replaced by `branch`!

In other words, `branch` is like a drop-in replacement for the constructor `Branch`. To implement the morse code tree directly as functions instead of as an algebraic data type, we simply replace every occurrence of `Branch` with `branch` in our previous code. Thus, we have a new implementation

```
dict :: MorseCode -> Char
dict = interpret program
  where
    interpret          = head . foldl exec []
    exec xs            ' _ ' = leaf          : xs    -- push leaf
    exec (x:y:xs) c    = branch c y x : xs    -- combine subtrees
```

```
decodeLetter = dict
```

where

```
leaf = undefined
```

is a trivial replacement for the `Leaf` constructor.

A chainsaw called 'catamorphism'

Derivation

To further understand how and why the replacement of `Branch` by `branch` works, it is instructive to derive it systematically from just the program text.

In particular, consider the implementation of `decodeLetter` from the earlier section on dichotomic search on page 7:

```
decodeLetter :: MorseCode -> Char
decodeLetter = tag . foldl (flip step) dict
  where
    step '.' = left
    step '-' = right
```

In this formulation, the focus is on the recursion over the input string performed by `foldl`, neglecting the recursive descent into the dictionary tree. Let's systematically rewrite this code to highlight the latter.

First, we interpret it as converting the dictionary tree into a function that decodes letters

```
decodeLetter = decodeWith dict

decodeWith :: Tree Char -> (MorseCode -> Char)
decodeWith dict = tag . foldl (flip step) dict
  where
    step '.' = left
    step '-' = right
```

Then, we turn the `foldl` into explicit recursion

```
decodeWith dict []      = tag dict
decodeWith dict (d:ds) = decodeWith (step d dict) ds
  where
    step '.' = left
    step '-' = right
```

and dissolve the `step` function into the pattern matching

```
decodeWith dict []      = tag dict
decodeWith dict ('.':ds) = decodeWith (left dict) ds
decodeWith dict ('-':ds) = decodeWith (right dict) ds
```

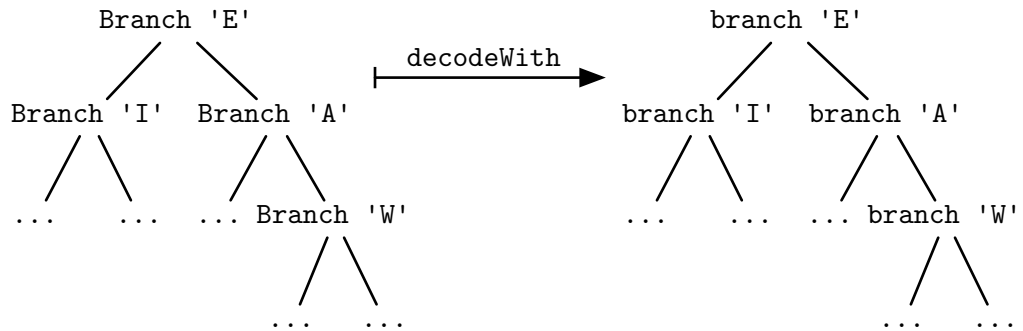


Figure 4: Substituting constructors `Branch` with functions `branch`.

We group the pattern matching on the input code into a case expression

```
decodeWith dict = \code -> case code of
  []          -> tag dict
  ('.':ds)    -> decodeWith (left dict) ds
  ('-':ds)    -> decodeWith (right dict) ds
```

and finally, we replace the field selectors by pattern matching on the tree, also noting the case of `Leaf`:

```
decodeWith Leaf          = undefined
decodeWith (Branch c x y) = \code -> case code of
  []          -> c
  ('.':ds)    -> (decodeWith x) ds
  ('-':ds)    -> (decodeWith y) ds
```

The recursion on the tree is apparent now, `decodeWith` simply traverses both subtrees and combines the results. We can make this even more evident by appealing to the combinator we defined in the previous section:

```
decodeWith Leaf          = leaf
decodeWith (Branch c x y) = branch c (decodeWith x) (decodeWith y)
```

In other words, `decodeWith` takes a tree and simply substitutes each `Leaf` constructor with `leaf` and each `Branch` constructor with `branch`, as shown in Figure 4.

Of course, first using `Branch` and then replacing it with `branch` is a waste; we should rather use `branch` from the start and thus cut away the intermediate tree. That's exactly what we've done in the previous section on page 12!

General pattern

Replacing constructors is a very general pattern, not restricted to binary trees. For instance, you probably know following function:

```
foldr f z []      = z
foldr f z (x:xs) = x 'f' foldr f z xs
```

It's the good old fold [20]! And at its heart, it just replaces the constructors of the list data type: `z` is the substitute for the empty list `[]`, and `f` is put in lieu of the `(:)` constructor.

In its honor, any function that substitutes constructors in such fashion is known as a “generalized fold” [21]. A more exotic but commonly used alternative name is “catamorphism” [21]. In other words, `decodeWith` is a catamorphism.

In this generality, the idea of deforestation [7] is that instead of first constructing a data structure and then chopping it up with a catamorphism, it is more efficient to saw the pieces properly at the time of creation. For example, creating a list and then adding all elements

```
sum [1..100] = foldr (+) 0 (enumFromTo 1 100)
```

is less efficient than adding the elements as they are created

```
sumFromTo a b      -- changes to enumFromTo
  | a > b          = 0      -- 0 replaces []
  | otherwise      = a + sumFromTo (a+1) b -- + replaces (:)
```

Performing deforestation manually, as we did, sacrifices reusability: the morse code tree only exists as a call graph and cannot be printed out anymore, and ‘`sumFromTo`’ is not nearly as useful as are ‘`sum`’ and ‘`enumFromTo`’ were.

Hence, the goal is to teach the compiler to **automatically** fuse catamorphisms with their structure creating counterparts, the so called anamorphisms [22]. That's what efforts like a short-cut to deforestation [7] and the recent stream fusion [23] are set out to do, yielding dramatic gains in efficiency while preserving the compositional style.

Where to go from here

I hope you had fun doodling around with morse code; I certainly did. If you long for more, here a few suggestions:

Cutting words

While we're at it, the intermediate lists created by `words` in

```
decode = map decodeLetter . words
```

can be deforested as well and fused into the letter decoding. Try it yourself, or see the example source code [9].

Polish notation and parsing

Polish notation [24], the converse of reverse polish notation, puts function symbols before their arguments and is thus closer to the Haskell syntax. When the arities of the functions are known in advance, like 2 for `Branch` and 0 for `Leaf`, this notation doesn't require parenthesis either. Write a `program` in polish notation and a corresponding tiny interpreter to create the morse code tree.

Comparing to C

Since deforestation is supposed to make things faster, how about comparing our deforested morse code tree to say an implementation in C? But instead of doing benchmarks, let me give two general remarks on the machine representation.

First, one might think that the call graph we've built is compiled to function calls, with `e` calling `i` or `a` and each function having different machine code, just as we originally intended. But this is actually not the case. When defined with `branch`, the functions are represented as closures [25], sharing the same machine code but carrying different records that store their free variables `c`, `x` and `y`. This is not very different from storing `c`, `x` and `y` in a `Branch` constructor. Template Haskell [26] or some kind of partial evaluation [27] would be needed to hard-code the free variables into the executable. For more on the Haskell execution model, take a look at the GHC commentary [28].

Second, all our implementations decipher a letter by follow a chain of pointers: descending a tree means following pointers to deeper nodes, and making procedure calls means repeatedly jumping to different code parts. This of course raises questions of cache locality [29], branch prediction [30] or memory requirements for storing all these pointers.

In C, however, there is another technique available: instead of following a chain of pointers to get to the destination, the address of the final stop is calculated directly with pointer arithmetic. The example in Listing 1 illustrates this.

There, the tree is represented as an array and paths to nodes are encoded as (zero-less) binary numbers. The program calculates the path `t` to the proper letter

```
#include <stdio.h>
#include <string.h>
char buf[99], tree[] = " ETIANMSURWDKGOHVF L PJBXCYZQ ";
int main() {
    while (scanf("%s", buf)) {
        int n, t=0;
        for(n=0; buf[n]!=0; n++)
            t = 2*t + 1 + (buf[n]&1); /* compute destination */
        putchar(tree[t]);           /* fetch letter */
    }
}
```

Listing 1: Calculating the address of the final destination.

and then fetches it with an $\mathcal{O}(1)$ memory lookup. Compare also the morse code translator by reddit user kayamon [31].

Since the number of memory accesses is kept to a minimum, this technique is the most efficient; but it is impossible to reproduce with algebraic data types alone. Fortunately, arrays libraries [32] are readily available in Haskell as well.

The main drawback of this approach, and of arrays in general, is the lack of clarity; indices are notorious for being non-descript and messy. The raw index arithmetic in the example C code above sure seems like magic!

Such magic is best hidden behind a descriptive abstract data type. How about rewriting the example in Haskell such that `decodeLetter` looks exactly like the one in the section on dichotomic search on page 7? In other words, the abstract data type is to support the functions `left`, `right` and `tag`. One possible solution can be found in the source code accompanying this article [9].

References

- [1] CharlieDancey. Who can write the smallest/tidiest/cleverest morse code translator? challenge! http://www.reddit.com/r/programming/comments/7xjqb/who_can_write_the_smallesttidiestcleverest_morse/.
- [2] Programming reddit. <http://www.reddit.com/r/programming>.
- [3] http://en.wikipedia.org/wiki/Dichotomic_search.
- [4] http://en.wikipedia.org/wiki/Binary_search.
- [5] <http://en.wikipedia.org/wiki/Stack-based>.

- [6] http://en.wikipedia.org/wiki/Reverse_Polish_notation.
- [7] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. pages 223–232. ACM Press (1993). <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.646>.
- [8] Don Stewart. Fusion. <http://donsbot.wordpress.com/tag/fusion/>.
- [9] Heinrich Apfeldmus. Accompanying source code for ‘Fun with Morse Code’. <http://sneezy.cs.nott.ac.uk/darcs/TMR/Issue14/morse-code.zip>.
- [10] http://en.wikipedia.org/wiki/Morse_Code.
- [11] International morse code, public domain image. http://en.wikipedia.org/wiki/File:International_Morse_Code.svg.
- [12] http://en.wikipedia.org/wiki/Association_list.
- [13] http://en.wikipedia.org/wiki/Binary_search_tree.
- [14] <http://en.wikipedia.org/wiki/Trie>.
- [15] http://en.wikipedia.org/wiki/Hash_table.
- [16] The Data.Map module. <http://www.haskell.org/ghc/docs/latest/html/libraries/containers/Data-Map.html>.
- [17] Quotes by Brian Kernighan. http://en.wikiquote.org/wiki/Brian_Kernighan.
- [18] Tree for deciphering morse code, licensed under cc-by-sa-2.5. http://en.wikipedia.org/wiki/File:Morse_code_tree3.png.
- [19] <http://en.wikipedia.org/wiki/Dichotomy>.
- [20] [http://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](http://en.wikipedia.org/wiki/Fold_(higher-order_function)).
- [21] <http://en.wikipedia.org/wiki/Catamorphism>.
- [22] <http://en.wikipedia.org/wiki/Anamorphism>.
- [23] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. **SIGPLAN Not.**, 42(9):pages 315–326 (September 2007). <http://www.cse.unsw.edu.au/~dons/papers/CLS07.html>.
- [24] http://en.wikipedia.org/wiki/Polish_notation.
- [25] [http://en.wikipedia.org/wiki/Closure_\(computer_science\)](http://en.wikipedia.org/wiki/Closure_(computer_science)).
- [26] Template Haskell. http://www.haskell.org/haskellwiki/Template_Haskell.
- [27] http://en.wikipedia.org/wiki/Partial_evaluation.

- [28] The Haskell execution model. <http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/HaskellExecution>.
- [29] http://en.wikipedia.org/wiki/Locality_of_reference.
- [30] http://en.wikipedia.org/wiki/Branch_predictor.
- [31] Morse code translator in C. http://www.reddit.com/r/programming/comments/7xjqb/who_can_write_the_smallestidiestcleverest_morse/c07p22i.
- [32] Arrays in Haskell. <http://www.haskell.org/haskellwiki/Arrays>.

Hieroglyph 2: Purely Functional Information Graphics Revisited

by Jefferson Heard (jefferson.r.heard@gmail.com)

After my tutorial last year at CUFPP [1], I encountered both a lot of interest in functional programming for visualization and a lot of criticism of the current offerings in graphics libraries for functional languages like Haskell. In particular, these graphics libraries, such as the bindings for OpenGL and Cairo, encourage traditional imperative programming. Both use the `do` syntax for everything, do not separate the execution of the code from the semantics of the geometry, and do not, in fact, include any semantic element to them at all.

*To begin to answer this, I have written a toolkit called **Hieroglyph** which aims to tackle the problem of using a single code base to generate both interactive and high-quality static graphics for information visualization and visual analytics in a purely functional manner. The purpose of this paper is to present the toolkit, giving a clear rationale for its existence and style, and provide a tutorial for using it.*

Information Visualization and Visual Analytics

Traditional scientific visualization is primarily concerned with showing structured data about events or phenomena in the physical world: earthquakes, supernovae, ocean currents, air quality, wind-tunnel tests, hydrodynamics. The models generated by scientific simulation or studies of our environment are the concern of scientific visualization. There exists at least one functional toolkit for scientific visualization [2] in Haskell that is reasonably complete in this regard, and many other toolkits [3] for other languages that can be said to follow a more or less functional style.

Information visualization and visual analytics are a bit different [4]. Information visualization concerns itself with representing knowledge, facts, and the structures

we use to order these facts. Graphs, charts, maps, diagrams, and infographics aim to illustrate clearly the relationships in a manner more efficient and more elucidating than, or complimentary to lengthy articles or papers.

Visual analytics extends this notion by adding the visual element to the process of data mining. Visual analytics applications take raw data and apply transforms iteratively (or recursively), at the user's direction, allowing the user to see intermediate results of mining operations and direct the further application of mining techniques based on the results.

Functional programming as realized in data visualization

Functional programming fits both of these problems well for two reasons. Firstly, it can separate the process of drawing from the semantics of geometry. Secondly, functional programming can tie the structure of data to the structure of the geometry. These two properties allow a programmer to view his or her code in a way that is similar to how the data being visualized is structured.

With functional graphics, the programmer can reason about the code and data in a more natural way than traditional graphics programming. Debugging becomes not a problem of wrangling with a state machine or sequencing operations so that they layer onto each other properly, but rather of mapping data points to geometry and color and relating the resultant objects to each other (to solve constraints like occlusion).

Consider the (fairly reasonable) code for displaying a simple rectangle both in HOpenGL [5] in Listing 2, and then in Gtk2Hs Cairo [6] in Listing 3. Admittedly, the OpenGL code is much more obtuse. At its very simplest, you have to make calls to `GL.vertex`, setup an environment, call `GL.color`, call `GL.flush`, and swap buffers. Still, the Cairo code isn't all that much better. Here, although we have a primitive for a rectangle, there is no sense of where the data goes in this code – even though the data is actually the most important part of our work.

What shift of mindset do we need to go from this to a more functional approach? Well, here is the general procedural approach that resulted in the above code:

- ▶ Load the data.
- ▶ Process the data.
- ▶ For each bit of data, call some drawing procedure that realizes it in the correct order.
- ▶ Rinse hands, repeat.

Why is this sequence important? Why does the visualization programmer need to be concerned with how to draw a particular object? Why is the visualization

```
... setup state machine ...
GL.textureBinding GL.Texture2D $= Nothing
GL.color $ GL.Color4 0 0 1 (1::Double)
GL.withName (GL.Name iid)
  . GL.renderPrimitive GL.Quads
  . mapM_ GL.vertex
  . verticesFrom $ vdata

GL.color $ GL.Color4 0 0 1 (1::Double)
GL.withName (GL.Name iid)
  . GL.renderPrimitive GL.LineStrip
  . mapM_ GL.vertex
  . verticesFrom $ vdata
GL.vertex . head . verticesFrom $ vdata
  where verticesFrom (x:y:vs) = GL.Vertex2 x y : verticesFrom vs
        verticesFrom [] = []
GL.flush
Gtk.drawableSwapBuffers drawable
```

Listing 2: OpenGL code for rendering a rectangle, not including setup

```
Cairo.save
Cairo.setSourceRGBA 0 0 1 1
Cairo.rectangle x y w h
Cairo.fill
Cairo.rectangle x y w h
Cairo.rectangle 0 0 0 1
Cairo.stroke
Cairo.restore
```

Listing 3: Cairo code for rendering the same rectangle, also without setup

code coupled so tightly the medium in which it is visualized? (OpenGL is notorious for being difficult to transfer to print, and Cairo isn't hardware accelerated, making it difficult to do interactive graphics.) Why does the visualization programmer care about the underlying state machine? Why do we throw laziness out the window just because we want graphics?

Maybe we need to rethink what data visualisation means in the context of functional programming. To do that, we start with the question, "What is a visualization?" The only good answer to a simple question is equally simple, "A visualization is a collection of data mapped to a visual representation." With that answer in mind, a more functional approach to the problem might look like:

- ▶ Load the data (as needed)
- ▶ Map the data into some form that reflects its semantic and relational structure.
- ▶ Map this structure and the individual data points contained therein to a symmetrical structure of geometric primitives (our basic units of visual representation).
- ▶ Let the underlying drawing system take care of what to display and when.

This approach gives us no notion of sequence to drawing, or even really of the *procedure* of drawing. It abstracts the visual representation of the data from the medium, allowing us to use the same code for both interactive and print media assuming a sufficiently powerful back-end. The geometry can be just as lazy as the data.

There are other nifty things that we gain from approaching the problem this way, like mapping, filtering, reducing, and constraining the geometry with functions after defining it, and like being able to treat visualizations and their underlying data as composable components. I will make it clear how to do this later.

Formalizing the approach

Let us skip step one for now – loading data is important, but how to load it and into what structure is application dependent. First, let's consider the implication of the statement: "Map the data into some form that reflects its semantic and relational structure." The data is the most important part of a visualization. Consider that visualizations are not just beautiful pictures, but also include spreadsheets, document layouts. . . this article itself is a visualization of its \LaTeX source.

Imagine if inside a spreadsheet application, the programmer had to make a copy of the data in a specific and convoluted manner just to get a data value into each cell and then interpret and copy back any edits the user made to the spreadsheet. VisiCalc would have required a mainframe in its day! As tedious as that sounds, an analogue to that procedure is what is commonly used today in programming

data visualizations [7]. It takes up the bulk of the time in writing any visualization application, as opposed to the design of the data.

Rather, then, than forcing our data into the format required by the graphics toolkit, it would be better to simply make the graphics toolkit accept the format of our data. On the face of it, that seems like a daunting task, but Haskell has the notion of typeclasses and quite a few pre-packaged data structures, and we use these to our advantage. Here is how Hieroglyph [8] does just this.

```

type BaseVisual = [Primitive]

class Visual t where
  primitives :: t -> BaseVisual

instance Visual Primitive where
  primitives a = [a]

instance Visual a => Visual [a] where
  primitives x = concat $ primitives `map` x

instance Visual b => Visual (M.Map a b) where
  primitives = primitives . M.elems

instance Visual b => Visual (IM.IntMap b) where
  primitives = primitives . IM.elems

instance Visual t => Visual (S.Set t) where
  primitives = primitives . S.toList

```

Listing 4: Defining the class `Visual`

In Listing 4 we define a class `Visual t` that makes any of its members capable of being visualized directly. The function `primitives` takes any `Visual` and maps it onto geometry, known here as a `BaseVisual`. We define `BaseVisual` as a list for convenience, but in reality, we ignore its property of sequencing and concentrate instead on the things that are important to any data structure that might be visualized: a visual must be foldable, traversable, composable, and capable of holding duplicate elements. Lists give us all these. Properly, a list minus its sequencing is called a *bag*, and this is the effective type of `BaseVisual`.

We go on to define recursive instances of `Visual` for some of the most common data structures: maps, sets, lists, and singleton primitives. A list of `Visuals` is

reducible to a `Visual`. Taken collectively, the elements of a map or set of `Visuals` are also `Visual`. One could define `Visual` instances for other data structures as well or define `Visual` instances for data elements themselves. For instance, if we have an `Floating` type `Magnitude`, an alias for `Double`, we could define:

```
instance Visual Magnitude where
  primitives m = primitives arc{ radius=m }
```

The primitive `arc` will be explained later, but essentially, this definition says that `Magnitudes` are realized as circles with a radius equal to the magnitude. Thus the data maps directly to its visual representation and we could directly visualize, say, a `Data.Map String Magnitude` instead of having to traverse the map ourselves and draw a circle for each magnitude.

Now that we have a definition for `Visual` and `BaseVisual`, we can define composability. A `Visual` essentially describes how a data structure maps to a `BaseVisual`. The `BaseVisual` is the basic unit of composing visualizations together. As `BaseVisual` is a list, it is automatically an instance of `Monad`, `MonadPlus`, `Traversable`, and `Foldable`, as well as `Applicative`. As a result, we can use all Haskell's combinators and operators on these classes to work with `BaseVisuals`. We only need to define a few more to deal specifically with the nature of graphics.

For all graphics, the notion of *occlusion* is important. That is, which objects occlude other objects if they overlap. When composing `Visuals`, we can define this relation by a binary operator (`#/#`), or “occludes”, which we can define as follows:

```
occludes :: (Visual t, Visual u) => t -> u -> BaseVisual
(##) = occludes
```

We also define “beside” to be the same as `mappend` or `(++)`. `Beside` simply signifies that the two visuals do not overlap and therefore are defined to draw *concurrently*. There are other operators, unary combinators, that apply to `Visuals` which we will cover in the next section, where we cover primitives and their attributes.

Geometric primitives

So far we've discussed graphics and what can be done with them, but we've not seen any. That is about to change. To represent geometry, `Hieroglyph` defines a datatype, `Primitive`, which represents geometry. Unlike the class `Visual`, which follows directly from our notion of what a visualization is, the notion of what constitutes a `Primitive` is a little more nebulous.

We could say that a Primitive is nothing more than function on x and y defined piecewise. That could cover all the bases: line, arc, rectangle, polygon, splines, lines, and points; but that would be less than useful for visualization programmers, who once again are more concerned with representing data than with graphics programming per se. A visualization programmer doesn't care about the mathematical representation of a circle, only a circle. So rather than force the programmer to do the math, Hieroglyph defines the following primitives, admittedly somewhat arbitrarily:

- ▶ **Dots** - A collection of points at arbitrary (x,y)
- ▶ **Arc** - A length of the standard parametric function for arcs.
- ▶ **Path** - A (possibly disconnected) string of line segments and splines.
- ▶ **Rectangle** - A rectangle
- ▶ **Text** - A string of formatted text.
- ▶ **Image** - An image loaded from disc.

```
data Primitive = Dots { at :: [Point] }
  | Arc { center :: Point, radius :: Double, angle1 :: Double,
        , angle2 :: Double, negative :: Bool }
  | Path { begin :: Point, segments :: [LineSegment]
        , closed :: Bool }
  | Rectangle { topleft :: Point, width :: Double
        , height :: Double }
  | String { str :: Text.PrettyPrint.Doc, bottomleft :: Point
        , align :: LayoutAlignment, wrapwidth :: Maybe Double
        , wrapmode :: LayoutWrapMode, justify :: Bool
        , indent :: Double, spacing :: Double}
  | Image { filename :: String, dimensions :: Either Point Rect
        , preserveaspect :: Bool }
```

Listing 5: Definition of primitives

Additionally, points and rectangles for dimension and line segment specification are defined by the code in Listing 6.

Rather than specify geometry directly using data constructor syntax, Hieroglyph tries to supply sensible defaults for most primitives. The default primitive is the same name as the data constructor, but without the capital letter. The primitive is then modified to suit the application by using the record modifier syntax. The result is something like a function with optional arguments, or labeled function [9].

These serve to specify geometry, but what of attributes? How thick are lines? Are the shapes filled and/or outlined and if so what color? What of scale, transla-

```

data Point = Point Double Double
data Rect = Plane
  | Singularity
  | Rect { x1 :: Double, y1 :: Double
        , x2 :: Double, y2 :: Double }
data LineSegment = Line Point
  | Spline Point Point Point
  | EndPoint Point

```

Listing 6: Defining dimensions

tion, rotation? Because these kinds of attributes are common to all primitives, we can modify them collectively for `BaseVisuals` rather than primitives alone. We do this using unary combinators. The documentation describes them in more detail, but combinators are defined for the fill and stroke color, linecap style, line stippling, scale, translation, rotation, and setting the object's fill and outline status. Colors are specified using the `Data.Colour [10]` library.

Listing 7, for example, defines an opaque blue circle with a black outline centered at (10,10), with a radius of 10. The result of rendering this code is given in Figure 1.

**Figure 1:** Results of `circle10`

```

circle10 = fillcolour (opaque blue) . strokecolour (opaque black) $
  arc{ center=Point 10 10, radius 10 }

```

Listing 7: A simple example

This more or less covers the outline of the library itself. More detail is available in the Haddock documentation for each primitive and each combinator, but this should be enough to get an idea of how things work. Now to continue on to a couple of examples.

Static visualizations

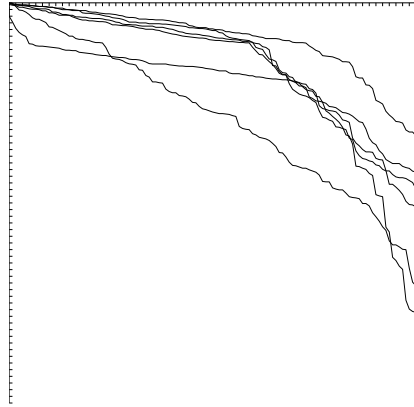


Figure 2: The results of running the linegraph program.

First, let us start with perhaps the simplest example possible, a static linegraph. In Listing 8, we will take the data from a tab delimited file and assign values to several `Path` primitives, draw axis lines, and then axis tics. The function `readFileToMap` actually reads in line-oriented series of data into `Data.Map.Map String DL`. The function `linegraph` defines geometry for axes and applies a color to the map that was read in. The `main` function only needs to take care of reading in some arguments from the command line regarding file and which fields to use, and then calls `renderToPNG` on the generated scatterplot. This is one of several transfer functions from `BaseVisual` to actual visualization in the static visualization context.

As we mentioned before, we can make our data visual by declaring an instance of the `Visual` class for the list of doubles, `newtype DL`. Note this gives visual semantics to the data, rather than forcing us to draw it. After all, that was our point in using functional programming to do visualization in the first place. Instead of drawing data, we define a list of doubles' visual representation to be a path that connects the values along the y-axis, spaced every 10 points away. Figure 2 shows our resultant line graph.

Computing a simple black and white linegraph, though, is trivial. A slightly more extended example (that can't be computed by Excel) is a tag cloud. To be interesting, we're going to color positive words orange and negative words blue.

Note that in Listing 9, a lot more is happening in the instance declaration. We

```

newtype DL = DL [Double]

instance Visual DL where
  primitives (DL (y:ys)) =
    primitives
      path{ begin=Point 0 0
            , segments = Line <$> (zipWith Point [0,5..] ys) }
  primitives (DL []) = primitives hidden

readFileToMap name =
  (map (BS.split '\t') . BS.lines)
  <$> BS.readFile name -> return . foldl' go Map.empty
  where go m (x:xs) =
    Map.insert (BS.unpack x)
              (DL $ map (read . BS.unpack) xs) m

linegraph c m w h = dataseries #/# axes
  where
    dataseries = scale (w/n) (h/mxval) $ strokecolour c m
    xaxis = primitives path{ begin = Point 0 0
                              , segments=[Line (Point w 0)] }
    yaxis = primitives path{ begin = Point 0 0
                              , segments=[Line (Point 0 h)] }
    xtics xs = linewidth 0.5 $
      map (\x -> path{ begin = Point x 0
                       , segments=[Line (Point x 5)] }) xs
    ytics ys = linewidth 0.5 $
      map (\y -> path{ begin = Point 0 y
                       , segments=[Line (Point 5 y)] }) ys
    axes = strokecolour (opaque black) $
      xaxis ## yaxis ##
      (scale (w/n) (h/mxval) $
       xtics [0,1..n] ## ytics [0,mxval/100..mxval])
    mxval = maximum . concat . (\DL a -> a) -> Map.elems $ m

```

Listing 8: Code to create a simple linegraph from a spreadsheet

```

mood :: String -> AlphaColour Double
mood x = ... should determine which words are what color ...

pol2cart (Point r theta) = Point x y
  where x = r * cos theta
        y = r * sin theta

type TagLists = B.ByteString

-- need TypeSynonymInstances for this
instance Visual TagLists where
  primitives s = renderedtag moods counts weights tags
    where
      illustration s = primitives hidden
      tagdata = B.words <$> B.lines s
      counts = length <$> tagdata
      tags = B.unpack . head <$> tagdata
      weights = length <$> tags
      moods = mood <$> tags
      rscale = 10
      tscale = 10
      renderedtag (m:ms) (c:cs) (w:ws) (t:ts) =
        strokecolour m $
          string{ str=tspan ("size=" ++ (max 24 (c+6)))
                        (text t)
                        , bottomleft=pol2cart (Point
                                              (w*logbase 2 c*rscale)
                                              (c*tscale))}

        : renderedtag ms cs ws ts
      renderedtag _ _ _ _ = []

main = renderToPostscript "tagcloud.ps" 100 100
      =<< B.readFile . (!!0) =<< getArgs

```

Listing 9: Creating a tag cloud



Figure 3: A tag cloud produced by the tagcloud program

first declare a type synonym for `B.ByteString`. To have an instance declaration work for a type synonym, we have to enable the GHC language extension `Type-SynonymInstances`. These and `newtype` declarations are very useful in Hieroglyph, since they can be used to distinguish between otherwise identical types whose semantics inside the program (and thus the visualization as well) are different.

We then declare an instance of `Visual` for `TagString` which assumes a series of words on each line that corresponds to the tags for a single document. These are broken up and then counted (which is assigned to font size), their mood is determined, and each tag's length is determined. Length and count are factored into radius, to give a good spread of tags across the page.

Most static visualizations can fit into this mould: declare data to be an instance of `Visual`; read in the data; render the data in the main function. Unlike in most graphics programming, the functional approach of Hieroglyph allows the programmer to spend the majority of the effort on the design of the visual rather than the process of drawing of the design.

```

-- at least initialize the bus

widgets = [initializeBus "window title", ...]

-- setup some arbitrary program behaviour
programBehaviour =
    renderBehaviour
  <~< userDefinedSelectionReaction
  <~< selectionBehaviour
  <~< userDefinedSelectionGenerator
  <~< ... -- define some geometry producing behaviour(s) here...

main = boilerplateHieroglyphMain widgets programBehaviour

```

Listing 10: The typical main function of a Hieroglyph interactive application

Dynamic visualizations

So far we have discussed the meaning of visualization, what a primitive is, and how to construct a static visualization using Hieroglyph, but what about interactivity? What additional concerns are there when dealing with an interactive visualization?

First of all, there's orchestrating it. Doing that is the job of Buster [11], an FRP [12]-like library that has been integrated closely with a hardware accelerated Hieroglyph. A cursory understanding of Buster is required to program interactive Hieroglyph applications, but the learning curve is fairly shallow.

Hieroglyph's interactive implementation defines the following three public functions: `initializeBus`, `selectionBehaviour`, and `renderBehaviour`. The function `initializeBus` does the job of setting up the interactive rendering environment and making sure that the display gets updated when it's necessary. This function is called once at the very beginning of `main`.

The other two functions are Buster *Behaviours*, reactive functions that consume and produce events as time evolves for an application. The `renderBehaviour` looks for an event that indicates that the scene has changed and needs to be rerendered in the future. Upon rerendering, `selectionBehaviour` looks for mouse events (presses) and tests the area underlying the mouse to see if an object was clicked.

The `selectionBehaviour` brings up the other concern we have in an interactive application, namely interaction. Different actions could cause parts of visualizations to be selected, and reacting to these being selected is a large part of the job of an interactive application. How do we know which objects are selected? We introduce another combinator, `name`, that designate's a `BaseVisual`'s name.

Whenever a selection event occurs, the event contains the name of the object that was selected. The general flow for selection looks like this:

- ▶ Event consumed by `selectionBehaviour`
- ▶ Event produced by `selectionBehaviour`.
- ▶ Some user-defined behaviour reacts to the selection

The general flow for rendering looks like this:

- ▶ Event produced by user-defined behaviour.
- ▶ Event consumed by `renderBehaviour`.
- ▶ Events polled by `renderBehaviour`.
- ▶ The `renderBehaviour` collects and renders `Visible`s' associated `BaseVisual` data.

If that seems a little obtuse, the example in Listing 10 will probably help. Now, let's modify our code from earlier to be more... dynamic. When the user clicks on a line, we want to display the actual min and max values. This is not particularly difficult. We need a user defined selection reaction, to wrap our linegraph into a widget, and to define some visual semantics for the min/max data. Note that we need to define `FlexibleInstances` in the GHC "LANGUAGE" header for this code to compile. The code we have to add can be seen in Listing 11.

Conclusion

The Hieroglyph library provides a much more functional approach to programming information visualizations and interactive applications for visual analytics. It is currently being used in several projects and is continuously being improved and refined. My sincere hope is that my readers will grasp the workings of the library well enough to write their own Hieroglyph applications from this, and the body of such visualizations and of functional visualization programmers will grow.

My sincere thanks are extended to Conal Elliott for his help in getting me to think long and hard about what visualization *means* and how that meaning can translate directly into implementation. My thanks to Don Stewart for staying excited about the library despite the long time waiting for the second release. Finally, I also wish to acknowledge my research institute for indulging me in my non-standard "functional programming" ways, and for being willing to critique and discuss my approach with me.

About the author Jeff Heard is a senior visualization and data researcher with the Renaissance Computing Institute at UNC Chapel Hill. His interests include visualization, functional programming, and data mining scalable to large datasets and large text collections. Outside of his professional interests, he is also an amateur nature photographer and avid cyclist.

```

instance Visual (Map String DL) where
  primitives m = map namedprims . Map.toList $ m
  where namedprims (k,v) = name k v

type MiniMax = (Double,Double)
instance Visual MiniMax where
  primitives (mn,mx) = strokecolour (opaque black)
    string{ str=parens (text "min" <> text "max") <+>
      parens (double mn <> comma <> double mx)
      , bottomleft = Point 0 100 }

fileReaderWidget = do
  geo <- (!!0) <$> getArgs >>= readFileToMap
  produce' "Visible" "Hieroglyph" "Line graph" Persistent
    [EOther (Geometry . strokecolour (opaque black) $ geo)]
  let axes = strokecolour (opaque black) $
      xaxis ### yaxis ###
      xtics [0,5..800] ### ytics [0,10..600]

reactToSelection b =
  consumeEventsInGroupWith "Selection" $ \event -> do
    let [EOther (Geometry geo)] =
        eventdata . Set.findMin $ eventsByName "Line Graph"
        mx = maximum $ geo Map.! ename event
        mn = minimum $ geo Map.! ename event
        future b . listM $ produce "Visible" "UserReaction" "Minimax"
            [EOther (Geometry (mn,mx))]

reactToMouseRelease b = consumeNamedEventsWith "ReleaseClick" $
  \event -> future b . maybeToList . Deletion <$>
    eventForQName "Visible" "UserReaction" "Minimax"

programBehaviour =
  renderBehaviour
  <~< (reactToSelection |~| reactToMouseRelease)
  <~< selectionBehaviour
  <~< mouseSelectionBehaviour

```

Listing 11: Getting the min and max values on clicking a linegraph line

References

- [1] Beautiful code, compelling evidence. (2008). <http://vis.renci.org/jeff/wp-content/uploads/2009/01/beautifulcode.pdf>.
- [2] D J Duke, M Wallace, R Borgo, and C Runciman. Fine-grained visualization pipelines and lazy functional languages. **Transactions on Visualization and Computer Graphics**, 12(5):pages 973–980 (2006).
- [3] W Schroeder, K Martin, and B Lorensen. **The Visualization Toolkit, an object oriented approach to 3d graphics**. Prentice Hall (1996).
- [4] Edward Tufte. **Visual Explanations**. Graphics Press (1997).
- [5] Sven Panne. HOpenGL (2009). <http://haskell.org/HOpenGL>.
- [6] Duncan Coutts et. al (2009). <http://www.haskell.org/gtk2hs>.
- [7] Ben Fry. **Visualizing Data**. O’Reilly and associates (2007).
- [8] The hieroglyph toolkit. (2009). <http://vis.renci.org/jeff/Hieroglyph>.
- [9] Simon Peyton Jones and Greg Morrisett. A proposal for records in haskell (2003). <http://research.microsoft.com/en-us/um/people/simonpj/haskell/records.html>.
- [10] Russell O’Connor. colour: A model for human colour/color perception. (2009). <http://ctan.org/tex-archive/macros/latex/required/amslatex/math/>.
- [11] Jefferson Heard. Buster, the not quite entirely unlike FRP toolkit (2009). <http://vis.renci.org/jeff/Buster>.
- [12] Conal Elliott and Paul Hudak. Functional reactive animation. In **International Conference on Functional Programming** (1997). <http://conal.net/papers/icfp97/>.

Lloyd Allison's Corecursive Queues: Why Continuations Matter

by Leon P Smith <leon@melding-monads.com>

Abstract

In a purely functional setting, real-time queues are traditionally thought to be much harder to implement than either real-time stacks or amortized $\mathcal{O}(1)$ queues. In “Circular Programs and Self-Referential Structures,” [1] Lloyd Allison uses **corecursion** and **self-reference** to implement a queue by defining a lazy list in terms of itself. This provides a simple, efficient, and attractive implementation of real-time queues.

While Allison's queue is general, in the sense it is straightforward to adapt his technique to a new algorithm, a problem has been the lack of a reusable library implementation. This paper solves this problem by structuring the corecursion using a monadic interface and continuations.

Because Allison's queue is not fully persistent, it cannot be a first class value. Rather, it is encoded inside particular algorithms written in an extended continuation passing style. In direct style, this extension corresponds to *mapCont*, a control operator found in *Control.Monad.Cont*, part of the Monad Template Library for Haskell. [2] This paper argues that *mapCont* cannot be expressed in terms of *callCC*, *return*, and $(\gg=)$.

The essence of this paper is *mfixish*, a novel fixpoint operator for continuations. It cooperates with *mapCont* to create value recursion. Although this paper tends to avoid explicit use of *mfixish*, it can be used to introduce the self-reference inherent in corecursive queues.

Introduction

Richard Bird is well known for popularizing “circular programming,” [3] which in modern terminology is included under the term “corecursion.” [4] One of the best known examples defines an infinite list of Fibonacci numbers. However, as this paper is about queues, our running example is breadth-first traversals of binary trees. Thus, for our first example in Figure 1, we corecursively define the Fibonacci trees instead.

```

data Tree a b
  = Leaf    a
  | Branch b (Tree a b) (Tree a b)
  deriving (Eq, Show)

labelDisj :: (a → c) → (b → c) → Tree a b → c
labelDisj leaf branch (Leaf    a    ) = leaf    a
labelDisj leaf branch (Branch b _ _) = branch b

childrenOf :: Tree a b → [Tree a b]
childrenOf (Leaf    _    ) = []
childrenOf (Branch _ l r) = [l, r]

```

Listing 12: Binary Trees and useful helper functions

The indexing was chosen so that the number of leaves in the n^{th} Fibonacci tree is equal to the n^{th} Fibonacci number. The branches are labeled with the the depth of the tree. This definition uses self-reference and sharing to efficiently represent each additional tree with a constant amount of extra space. Of course, fully traversing such a tree would take an exponential amount of time.

The second example defines the Stern-Brocot tree, shown in Figure 2. Despite that this definition does not employ self reference, this is a corecursive definition because it is infinite and thus requires lazy evaluation. The Stern-Brocot tree is interesting because every positive rational number is generated in reduced form at exactly one branch. Not only does this prove that the rationals are countable, it can be computed more efficiently than the standard Cantor diagonalization.

These examples were chosen such that any two subtrees in this family are equal if and only if their labels are equal. This is true even for the Fibonacci trees: even though labels are repeated, the subtrees are still equal. This property can be exploited to efficiently and accurately test whether two breadth-first traversals might be equivalent.

Having separate types for the labels of branches and leaves enables one to better exploit Haskell’s type system. An example is the polymorphic leaf type

```

fib :: Int -> Tree Int Int
fib n = fibs !! (n - 1)
  where
    fibs = Leaf 0 : Leaf 0 : zipWith3 Branch [1..] fibs (tail fibs)

```

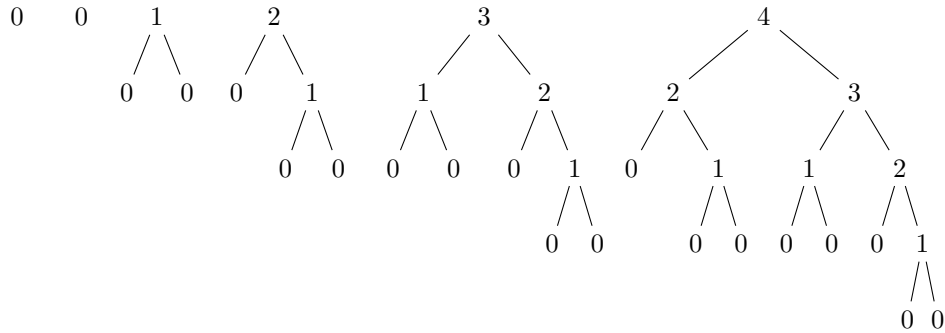


Figure 1: Fibonacci Trees

```

sternBrocot :: Tree a (Ratio Integer)
sternBrocot = loop 0 1 1 0
  where loop a b x y
    = Branch (m % n) (loop a b m n) (loop m n x y)
    where m = a + x
          n = b + y

```

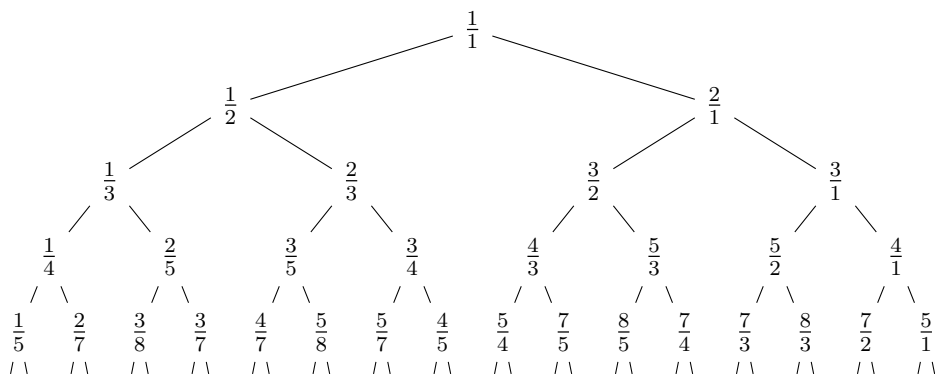


Figure 2: The Stern-Brocot Tree

for *sternBrocot*. Along the lines of Philip Wadler’s classic paper “Theorems for Free”, [5] this almost proves that the Stern-Brocot tree is both complete and infinite, in the sense that every counterexample to this line of reasoning involves **partial data**. Common examples of partial data are **infinite nonproductive loops** and the use of Haskell’s *error* function, such as this definition of \perp :

```
 $\perp :: \forall a. a$ 
 $\perp = error "bottom"$ 
```

However, not every corecursive definition produces a conceptually infinite data structure. Lloyd Allison’s queue is a good example: it is self-referential, and thus depends on lazy evaluation. Allison’s queues can and often do produce a finite object.

A simple breadth-first traversal using Allison’s queue is given along with a sample execution in Figure 3. The execution abuses notation slightly, as necessary for readability: the elements of the queue are trees, not labels. However, as the labels are unique for the examples given, this does not lead to ambiguity.

A corecursive queue is represented by a single lazy list. The end of the queue is represented by a thunk, which can produce the next element on demand. This thunk contains a pointer back to the first element in the queue and the number of elements currently in the queue. When an element is enqueued, call-by-need evaluation **implicitly mutates** the end of the list.

The Fibonacci example uses a lazy list sort of like a queue. New elements are “enqueued” when they are produced by *zipWith*, which occurs in sync with elements being “dequeued” when they are consumed by pattern matching inside *zipWith*.

Of course, most queue-based algorithms don’t have this level of synchronization. During a level-order traversal of a Fibonacci tree, the queue will grow and shrink frequently. We must be careful not to run off the end of the queue and pattern match against elements that aren’t there. The easiest approach is to track the number of elements in the queue.

Pattern matching creates demand for computation, thus pattern matching on the empty queue causes this thunk to reenter itself, creating an infinite nonproductive loop. In effect, in order to compute the answer, the answer must have already been computed. Explicitly tracking length breaks this cycle. By knowing via other means that the queue is empty, we can avoid pattern matching and continue or terminate the queue as needed, as illustrated in the last step of the sample execution.

Note that the type of the counter is explicitly given. Otherwise, Haskell would typically default to arbitrary precision integers. This leads to a noticeable, though modest, slowdown and increase in memory allocation in certain micro-benchmarks, such as a complete traversal of a Fibonacci trees.

If one doesn’t care about leaves or their contents, one might prefer a variant of

```

levelOrder :: Tree a b → [Tree a b]
levelOrder tree = queue
  where
    queue = tree : explore 1 queue
    explore :: Int → [Tree a b] → [Tree a b]
    explore 0 q = []
    explore (n + 1) (Branch _ l r : q') = l : r : explore (n + 2) q'
    explore (n + 1) (Leaf _ : q') = explore n q'

```

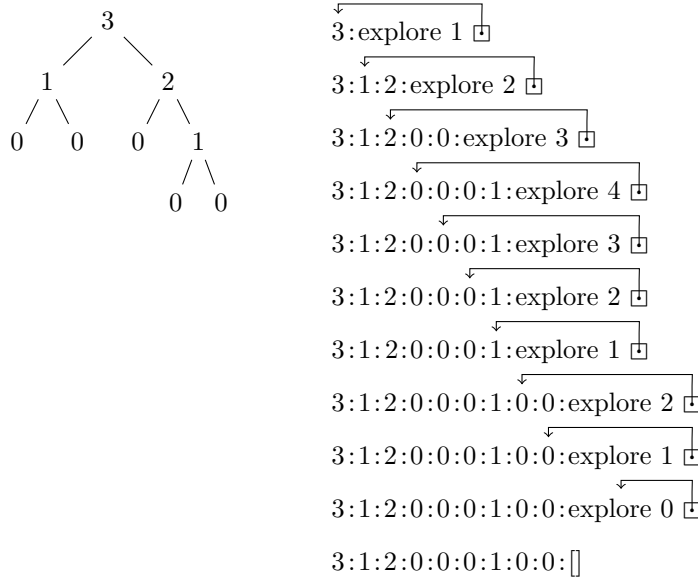


Figure 3: A Corecursive Queue, and a trace of `levelOrder` (*fib 5*)

`levelOrder` that does not enqueue the leaves. Listing 13 presents Lloyd Allison's original code, translated from Lazy ML to Haskell. It contains repeated code in the inner conditional branches. This approach is not unreasonable in this specific case, however, it does not scale. The easiest way to eliminate the redundant branch is introduce a helper function that processes a list of trees to possibly enqueue.

Because the code in Listing 14 makes use of `childrenOf`, it easily generalizes to arbitrary trees. This code is very similar to a breath-first graph searching algorithm I wrote in December 2000, simplified to trees. This is notable because it was the day after I first read Richard Bird's classic paper, which is also prominently cited by Allison.

```

isBranch = labelDisj (const False) (const True)
levelOrder' :: Tree a b → [Tree a b]
levelOrder' tree = queue
  where
    queue | isBranch tree = tree : explore 1 queue
          | otherwise     = explore 0 queue
    explore :: Int → [Tree a b] → [Tree a b]
    explore 0 _ = []
    explore (n + 1) (Branch _ l r : q')
      = if (isBranch l)
        then if (isBranch r)
              then l : r : explore (n + 2) q'
              else l : explore (n + 1) q'
        else if (isBranch r)
              then r : explore (n + 1) q'
              else explore n q'

```

Listing 13: Avoiding leaves

```

levelOrder'2 :: Tree a b → [Tree a b]
levelOrder'2 tree = queue
  where
    queue = enqs [tree] 0 queue
    enqs :: [Tree a b] → Int → [Tree a b] → [Tree a b]
    enqs [] n q = deq n q
    enqs (t : ts) n q
      | isBranch t = t : enqs ts (n + 1) q
      | otherwise  = enqs ts n q
    deq 0 _ = []
    deq (n + 1) (t : q') = enqs ts' n q'
  where ts' = childrenOf t

```

Listing 14: Removing the repeated branch

Although I didn't become aware of Lloyd Allison's work until six years later, at the time I had guessed that somebody had come up with it before. It is a testament to Bird's writing that he has inspired at least two people, and probably more, to independently arrive at the same idea.

The code was rather difficult to write the first time, but I immediately felt that I had a reasonable grasp of what was going on. In fact, I remember part of the thought process behind my endeavor: I was trying to implement a queue using naive list concatenation, employing Richard Bird's technique to eliminate a quadratic number of passes.

Reusable Corecursive Queues

We have now written three corecursive queues. A natural question to ask is how to implement a reusable library for this technique, so that we don't have to start from scratch every time we would like to use it. This subsection informally derives such an implementation.

What kind of interface would this library have? The traditional interface treats queues as first class values, as given by *Queue* in Figure 15. Because Haskell is pure, all first class values are persistent. This gives us the freedom to enqueue an element, back up, enqueue a different element, and use both results in subsequent computations, as demonstrated by the *fork a b q = (enqueue a q, enqueue b q)*.

Due to implicit mutation, Allison's queues cannot be used persistently. This implies that they cannot be first class values in a pure language! Thus looking for an implementation of *enqueue* and *dequeue* is futile, as any interface must enforce linearity upon enqueue. Monadic interfaces offer a well-known solution to this problem, so it seems plausible that we could find an implementation of a monadic interface.

In the examples so far, the logic that traverses the binary trees is entangled with the logic that defines the queue operations. Our goal is to separate these concerns.

```
class Queue q where
  empty :: q e
  enqueue :: e → q e → q e
  dequeue :: q e → (Maybe e, q e)
class Monad m ⇒ MonadQueue e m | m → e where
  enQ :: e → m ()
  deQ :: m (Maybe e)
```

Listing 15: First-class versus monadic interfaces

As the names suggest, *explore* in Listing 13 has no separation of concerns, while *enqs* and *deqs* in Listing 14 isolate the basic operations into two mutually recursive functions.

If we had an implementation of *enQ*, it would be easy to write a helper function that takes a single tree, and enqueues it iff it is a *Branch*. Perhaps if we had this helper, it would be easier to factor out the queue operations. We will work backwards from this guess, and forwards from *levelOrder'2* to write this single-element filter.

Currently, *enqs* loops over candidates to possibly enqueue. By inlining *childrenOf* and then unrolling *enqs*, we get two helper functions, *enq1* and *enq2*, that differ only in their **continuation**. By parameterizing the continuation, we can refactor these into a single function. This process is demonstrated in Listing 16.

Continuations are **required** because every queue operation must return the resulting queue from the rest of the computation. Thus every enqueue or dequeue must be aware of what operation comes next.

The resulting definition of *levelOrder'4* is easily the best means of expression thus far: unlike *levelOrder'* and *levelOrder'3* it does not repeat any logic, and unlike *levelOrder'2*, the queue operations are expressed much more directly, and the logic is all but disentangled. The last step should be easy, all we have to do is pull the branch out of *enq* and define a separate helper function, recovering the original *explore* and fully reusable queue operations, as demonstrated in Listing 17.

In a sense, we have re-invented the traditional continuation passing style [6] (CPS) with a slight twist, namely, that the tail of a list is considered to be a tail call. This is not a problem relative to the current understanding of the CPS transform, where any part of the program that is guaranteed to terminate, such as lazy cons [7], may optionally be left untouched by the transform.

Now, the only difference between *deq* and the monadic *deQ* is that the *deq* breaks out of it's loop and leaves the computation, whereas *deQ* returns *Nothing* in this case. Although the breaking out of the computation is superficially pleasing in this particular case, there are good reasons to prefer the latter in most cases. So we are ready to split Listing 17 into two parts: reusable corecursive queue operations versus the breadth-first tree traversal in Listings 18 and 19 respectively.

I first wrote something like *CorecQ* in August 2005, although it took me several years to understand my own code. Of course, this raises the question of how one can write code that one doesn't understand.

The answer is simple: type theory. Following the reasoning at the beginning of this subsection, I had been growing rather suspicious that corecursive queues could be abstracted via a monadic interface. After reading Wouter Swierstra's "Why Attribute Grammars Matter" [8] and coming to opinion that the examples contained therein are kind of lame, I was motivated to produce better, more compelling examples. Corecursive queues naturally came to mind.

```

type CorecQSt e = Int → [e] → [e]
newtype CorecQ e a
  = CorecQ { unCorecQ :: (a → CorecQSt e) → CorecQSt e }
instance Monad (CorecQ e) where
  return a = CorecQ (λk → k a)
  m ≫= f = CorecQ (λk → unCorecQ m (λa → unCorecQ (f a) k))
instance MonadQueue e (CorecQ e) where
  enQ e = CorecQ (λk n q → e : (k () $! n + 1) q)
  deQ = CorecQ deq
  where
    deq k 0 q = k Nothing 0 q
    deq k (n + 1) (e : q') = k (Just e) n q'
runCorecQ :: CorecQ element result → [element]
runCorecQ m = q
  where q = unCorecQ m (λa n' q' → []) 0 q

```

Listing 18: A reusable implementation of Allison's queue

```

levelOrder'' :: MonadQueue (Tree a b) q ⇒ Tree a b → q ()
levelOrder'' t = handle t ≫= explore
  where
    handle t | isBranch t = enQ t
              | otherwise = return ()
    explore = deQ ≫= maybe (return ())
                  (λ(Branch _ l r) →
                    do
                      handle l
                      handle r
                      explore      )

```

Listing 19: A binary tree traversal using generic queues

For ten hours I struggled, lost and confused, starting over several times. Eventually I came to realize that the state monad was not a suitable vehicle for Allison’s technique, and started thinking towards continuations. Soon enough the types worked out and everything felt “right.” I was rather confident that it would work before I tried it. And as if by magic, it worked.

The problem had to be simplified before I got anything working at all. In the first three failed attempts, I tried to implement a full-blown *CorecQW*. However, fourth and first successful implementation could not even track the length of the queue internally, rather, it was the client’s responsibility to avoid the infinite nonproductive loop.

Observing Monadic Computations

Monadic computations must somehow be **observed**, otherwise they are useless. Note the type of $runCorecQ :: CorecQ\ e\ a \rightarrow [e]$. The only observable aspect of the *CorecQ* monad is the list of enqueued elements. In particular, the final return value a cannot be observed.

Listing 20 gives two generic combinators for level-order traversals. They parameterize *childrenOf*, allowing arbitrary trees to be traversed. In fact, one need not even explicitly construct a tree: the Fibonacci trees could be traversed using this definition of *fibChildren*, for example.

$$\begin{aligned} fibChildren\ 0 &= [] \\ fibChildren\ 1 &= [0, 0] \\ fibChildren\ n &= [n - 2, n - 1] \end{aligned}$$

They also return a generic *MonadQueue*, allowing not only corecursive implementations, but also alternative implementations. For example, more obvious implementations of *MonadQueue* include wrapping the traditional two-stack queue in a state monad, or using explicitly mutable state as provided by *Control.Monad.ST*. As we will see in the next section, although these have the same semantics for *enQ* and *deQ*, they observe only the final return value.

Thus, these combinators produce level-order traversals when observed by *runCorecQ*, but are not particularly useful when observed with implementations that observe only return values.

There are two points worth noting about the definition of *byLevel* and *byLevel'*: the top level definitions are not recursive, and the recursive *explore* does not pass *childrenOf* to itself repeatedly. This combination plays nice with the Glasgow Haskell Compiler as current implemented: inlining *byLevel* opens up the possibility of inlining *childrenOf* as well, or at least eliminating an indirect jump.

While this idiom can be worthwhile; it is far more important performance-wise that *enQ* and *deQ* be inlined. Because we are using typeclasses to abstract over

```

byLevel :: (MonadQueue a m) => (a -> [a]) -> [a] -> m ()
byLevel childrenOf as = mapM_ enQ as >> explore
  where
    explore = deQ >> maybe (return ())
              (\a -> do
                mapM_ enQ (childrenOf a)
                explore
              )
byLevel' :: (MonadQueue a m) => (a -> [a]) -> [a] -> m ()
byLevel' childrenOf as = mapM_ handle as >> explore
  where
    handle a = when (hasChildren a) (enQ a)
    explore = deQ >> maybe (return ())
              (\a -> do
                mapM_ handle (childrenOf a)
                explore
              )
    hasChildren = \neg o null o childrenOf

```

Listing 20: Generic traversals over generic trees

the queue operations, inlining these operations is a clumsy thing to do in GHC. Indeed, ML-style functors are a superior choice for this type of abstraction.

Queues via explicit mutation

Being able to efficiently implement real-time queues using monads is not particularly newsworthy, as a knowledgeable programmer could always make use of *Control.Monad.ST*, which provides genuinely mutable state. However, the point is that the corecursive implementation based on implicit mutation is **shorter and safer** than an imperative linked list based on explicit mutation. Moreover, *CorecQ* is **faster** than *STQ* on current versions of GHC.

In some cases, arrays offer worthwhile constant-factor performance advantages over linked lists. Thus, barring other concerns such as concurrency, the only explicitly mutable implementations of queues truly worth considering in Haskell are those that employ mutable arrays.

Note the type *runSTQ* :: *STQ e a* -> *a*. This observes the return result but does not observe anything about the queue elements. With this in mind, even though *byLevel* visits leaves and *byLevel'* does not, these combinators are observationally equivalent. They both return () if the forest is a finite number of finite trees, and

```

data List    st a = Null | Cons a ! (ListPtr st a)
type ListPtr st a = STRef st (List st a)
type    STQSt st r e = ListPtr st e → ListPtr st e → ST st r
newtype STQ e a
    = STQ { unSTQ :: ∀ r st. ((a → STQSt st r e) → STQSt st r e) }
instance Monad (STQ e) where
    return a = STQ (λk → k a)
    m ≫= f = STQ (λk → unSTQ m (λa → unSTQ (f a) k))
instance MonadQueue e (STQ e) where
    enQ e = STQ $ λk a z → do
        z' ← newSTRef Null
        writeSTRef z (Cons e z')
        k () a z'
    deQ   = STQ $ λk a z → do
        list ← readSTRef a
        case list of
            Null          → k Nothing a z
            (Cons e a') → k (Just e) a' z
runSTQ :: STQ element result → result
runSTQ m = runST $ do
    ref ← newSTRef Null
    unSTQ m (λr _a _z → return r) ref ref

```

Listing 21: Queues via Imperative Linked Lists

diverge otherwise.

If one is interested in things other than thermal output and timing information, a more conventional alternative to changing the monad would be to thread a value through *byLevel*. Listing 22 defines *foldrByLevel*, a function for computing right folds over breadth-first traversals.

Listing 24 demonstrates how we can use *foldrByLevel* to concisely define various level order traversals over a forest of binary trees. For example, we can visit the labels of only leaves, or only branches, or both, and these properties are reflected in the resulting types.

There is no need for *foldrByLevel* to enqueue leaf nodes. Instead of folding elements after they are dequeued, we could instead fold elements before they are enqueued. Listing 23 computes the same fold, even though it enqueues branches only.

```

foldrByLevel :: (MonadQueue a m)
               => (a -> [a]) -> (a -> b -> b) -> b -> [a] -> m b
foldrByLevel childrenOf f b as = mapM_enQ as >> explore
  where
    explore = deQ >>= maybe (return b)
              (\a -> do
                mapM_enQ (childrenOf a)
                b <- explore
                return (f a b)
              )
prop_foldrByLevel childrenOf f b as =
  foldr f b (runCorecQ (byLevel childrenOf as))
  ≡ runSTQ (foldrByLevel childrenOf f b as)

```

Listing 22: Right folds over level-order traversals

```

foldrByLevel' :: (MonadQueue a m)
                => (a -> [a]) -> (a -> b -> b) -> b -> [a] -> m b
foldrByLevel' childrenOf f b as = handleMany as
  where
    handleMany [] = explore
    handleMany (a : as) = do
      when (hasChildren a) (enQ a)
      b <- handleMany as
      return (f a b)
    explore = deQ >>= maybe (return b) (handleMany o childrenOf)
    hasChildren = ¬ o null o childrenOf

```

Listing 23: A slightly lazier fold that does not enqueue leaves

```

cid = const id
getUnion  f = f childrenOf (labelDisj (:) (:)) []
getLeaves f = f childrenOf (labelDisj (:) cid) []
getBranches f = f childrenOf (labelDisj cid (:)) []
runSTQ o getUnion  foldrByLevel :: [Tree a a] -> [a]
runSTQ o getLeaves foldrByLevel :: [Tree a b] -> [a]
runSTQ o getBranches foldrByLevel :: [Tree a b] -> [b]

```

Listing 24: Handy functions and example use cases

An orthogonal change allows *foldrByLevel'* to be more lazy. If the initial forest of trees is infinite, *foldrByLevel* will get stuck in a nonproductive loop. This is because Listing 22 enqueues the entire forest before doing any folding. The enhanced version interleaves folding with enqueue operations. Thus *foldrByLevel'* is a true generalization of *foldr*, whereas the original is not.

Of course, this property depends on the semantics of the monad: the final result must be observed in a sufficiently lazy fashion. This is not true of *runSTQ* even if the lazier variant of *ST* is used, and so *foldrByLevel* is semantically equivalent to *foldrByLevel'* relative to this monad. This equivalency will be relaxed relative to *StateQ* in the next section.

We now have four generic combinators and two ways to run each, for a total of eight possibilities. There is a pleasing combination of symmetry and anti-symmetry to this configuration: if we use *runCorecQ* to observe the elements enqueued, then *byLevel* is equivalent to *foldrByLevel*, which differ from *byLevel'* and *foldrByLevel'*. However, if we use *runSTQ* to observe the result, then *byLevel* is equivalent to *byLevel'*, which differ from *foldrByLevel* and *foldrByLevel'*.

Monad Transformers

This section briefly reviews the traditional two-stack queue, and explores how they can be encapsulated inside a variety of different monads. The first wraps the two-stack queue in a state monad, thus guaranteeing amortized $\mathcal{O}(1)$ performance. The second implementation employs a state transformer and a writer monad to observe both the elements enqueued and the final result. The last introduces the continuation passing state monad, which makes explicit an idiom already used in *CorecQ* and *STQ* of the previous sections.

We explore the guarantees that various types of monads provide, and demonstrate how most of these guarantees are lost when the monad transformers are used. We argue that monad transformers are not robust abstractions, culminating in the rather fragile corecursive queue transformer of the next section.

Two-Stack Queues

Purely functional stacks are easy because the simplest solution works well. Due to sharing, persistent linked lists make reasonably efficient stacks. When pushing an element onto the stack, all that is necessary is to allocate and initialize a single new *cons* cell. Removing an element is a simple pointer dereference, and involves no allocation.

However, the same naive usage of lists leads to quadratic behavior. Concatenating a single element onto the end of the list involves copying the entire list, leading

```

data TwoStackQ e = TwoStackQ [e] [e]
instance Queue TwoStackQ where
  empty = TwoStackQ [] []
  enqueue z (TwoStackQ [] []) = TwoStackQ [z] []
  enqueue z (TwoStackQ (a : as) zs) = TwoStackQ (a : as) (z : zs)
  dequeue (TwoStackQ [] []) = (Nothing, TwoStackQ [] [])
  dequeue (TwoStackQ (a : as) zs)
    | null as = (Just a , TwoStackQ as' [])
    | otherwise = (Just a , TwoStackQ as zs)
  where as' = reverse zs

```

Listing 25: Two-Stack Queues

to $\mathcal{O}(n)$ enqueues. Alternately, one could store the queue in reverse, which makes enqueue an $\mathcal{O}(1)$ operation, but then peeking at or removing the front element then becomes a $\mathcal{O}(n)$ operation.

Traditionally, purely functional queues combine these approaches. [9] The queue is represented by two stacks: the front stack and the back stack. The front stack holds the beginning of the queue, and the back stack holds the remainder of the queue in reverse. To enqueue something, push it on the back stack. To dequeue something, pull it off the front stack. If the front stack is subsequently empty, reverse the back stack onto the front.

Of course, because two-stack queues are first-class values in Haskell, they are automatically persistent. Unlike an imperative language, operations on the queue preserve older versions of the queue. While *dequeue* is still $\mathcal{O}(n)$ in the worst case, it works very well in practice because on average, *dequeue* is actually $\mathcal{O}(1)$, provided that the queue is not used persistently. Under this assumption, every element is moved at most once from the back to the front.

There are implementations that guarantee $\mathcal{O}(1)$ worst-case operations, even with persistent usage, such as Chris Okasaki's incremental reversals of lazy lists. [10] However this solution has a relatively high constant factor, in practice is often slower than other options, sometimes significantly so.

The State Monad

Monadic interfaces can enforce linear, non-persistent usage of data structures, but do not necessarily do so. The *StateQ* monad guarantees linearity by wrapping the queue in a *State* monad and hiding *get* and *put* operations, ensuring amortized $\mathcal{O}(1)$ operations. However, the state transformer monad, *StateT*, cannot make

```
newtype StateQ e a = StateQ (State (TwoStackQ e) a) deriving (Monad)
instance MonadQueue e (StateQ e) where
  enQ = StateQ o modify o enqueue
  deQ = StateQ (State deque)
runStateQ :: StateQ element result → result
runStateQ (StateQ m) = let (result, finalQ) = runState m empty
in result
```

Listing 26: First-class Queues inside *Control.Monad.State.Lazy*

this guarantee! The ability to use state persistently can be recovered by choosing the nondeterministic list monad and lifting *MonadPlus* operations, for example.

The Writer Monad

So far, we have only been able to observe either the enqueued elements or the final result, but not both. We can employ the state transformer in conjunction with the Writer monad to observe both aspects of the computation. The amortized $\mathcal{O}(1)$ guarantee is unaffected by the use of *Writer*.

Writer monads partially enforce the efficient use of list concatenation. In the MTL, *Writer* $[e]$ a is just a newtype isomorphism for $([e], a)$. It provides a function *tell* that takes a list and concatenates the remainder of the computation onto the end of the list. This naturally associates to the right, and thus avoids quadratic behavior. Of course, *tell* accepts arbitrary length lists, and one could

```
newtype WriterQ e a
  = WriterQ (StateT (TwoStackQ e) (Writer [e]) a)
  deriving (Monad)
instance MonadQueue e (WriterQ e) where
  enQ e = WriterQ (tell [e] >> modify (enqueue e))
  deQ   = WriterQ (StateT (return o deque))
runWriterQ :: WriterQ element result → (result, [element])
runWriterQ (WriterQ m)
  = let ((result, final_queue), queue) = runWriter (runStateT m empty)
in (result, queue)
```

Listing 27: Observing the list of elements enqueued

inefficiently produce a long argument to *tell*.

Note the duplication of functionality present in *WriterQ*. Essentially, the *Writer* monad re-creates the queue in a parallel data structure. This observation was another motivation behind the creation of my first corecursive queue, analogous to Listing 14. I started writing a graph traversal using first-class queues, producing a list of nodes as they were visited. Although I was not using monads, I noticed the duplication and saw an opportunity to eliminate it using circular programming.

The Continuation Passing State Monad

```

newtype CpSt st a
  = CpSt { unCpSt ::  $\forall res.(a \rightarrow st \rightarrow res) \rightarrow st \rightarrow res$  }
instance Monad (CpSt st) where
  return a = CpSt ( $\lambda k \rightarrow k a$ )
  m  $\gg$  f = CpSt ( $\lambda k \rightarrow unCpSt m (\lambda a \rightarrow unCpSt (f a) k)$ )
instance MonadState st (CpSt st) where
  get      = CpSt ( $\lambda k st \rightarrow k st st$ )
  put st'  = CpSt ( $\lambda k _ \rightarrow k () st'$ )
  runCpSt :: CpSt st a  $\rightarrow st \rightarrow (a, st)$ 
  runCpSt m st = unCpSt m ( $\lambda a st' \rightarrow (a, st')$ ) st

```

Listing 28: A Continuation Passing State Monad

```

newtype CpStQ r e a
  = CpStQ { unCpStQ :: StateT (TwoStackQ e) (Cont r) a }
  deriving (Monad)
instance MonadQueue e (CpStQ r e) where
  enQ = CpStQ  $\circ$  modify  $\circ$  enqueue
  deQ = CpStQ (StateT (return  $\circ$  deque))
  runCpStQ :: CpStQ r e r  $\rightarrow r$ 
  runCpStQ (CpStQ m) = runCont (runStateT m empty) ( $\lambda(r, q) \rightarrow r$ )

```

Listing 29: Queues via another continuation passing state monad

Less well known than the regular state monad is the continuation passing state monad, as shown in Listing 28. This paper has already used this idiom twice. It

```

class MonadMapCC a m | m → a where
  mapCC :: (a → a) → m b → m b

instance MonadMapCC r (CpStQ r e) where
  mapCC f = CpStQ ∘ mapStateT (mapCont f) ∘ unCpStQ
  foldrByLevel'' :: (MonadQueue a m
                    , MonadMapCC b m)
                  ⇒ (a → [a]) → (a → b → b) → b → [a] → m b
  foldrByLevel'' childrenOf f b as = handleMany as
where
  handleMany [] = explore
  handleMany (a : as) = do
    when (hasChildren a) (enQ a)
    mapCC (f a) (handleMany as)
  explore = deQ ≫ maybe (return b) (handleMany ∘ childrenOf)
  hasChildren = ¬ ∘ null ∘ childrenOf

```

Listing 30: Restoring laziness to *foldrByLevel'* using *mapCont*

has been used to track the length and head of the queue inside *CorecQ*, and to track the references to the start and end of the mutable list inside *STQ*.

The lazy state monad is notoriously inefficient on current implementations of Haskell; one is much better off using the strict state monad. The continuation-passing state monad is even faster. Compared to the lazy state monad, the biggest advantage is that we aren't returning many pairs of lazy tuples, at the cost of sacrificing some laziness.

In the case of *CpSt*, a rank-2 type is used to hide the final result. While this has little effect on the generated code, it has profound consequences. Specifically, we cannot implement the control operators *callCC* and *mapCont*, nor can we break out of the computation early. However we can implement an *mfix* operator! This last observation is due to Matt Morrow and will be demonstrated in Listing 36. As long as the computation terminates, the final continuation that *runCpSt* initially passes to the computation is guaranteed to be called exactly once.

A nearly identical monad can be obtained by passing *Cont* as a parameter to *StateT*. Other than the fact that the final result type is exposed, the effect is the same as implementing *CpSt* in Listing 28. In fact, when compiled `ghc -O2`, the given operations compile into almost the same code. The only significant, though minor, difference is that the continuation $a \rightarrow s \rightarrow r$ is tupled and not curried.

Listing 29 gives an implementation of the queue interface in terms of *StateT*

and *Cont*. However, *StateT* is **lazy** but *CpStQ* is **strict**! Not only have we added a continuation semantics to *StateT*, we have also changed part of the existing semantics. This is in contrast to the use of *Writer* in *WriterQ*, which left the state semantics undisturbed. Not only are monad transformers not robust abstractions, they are not robust in any sense of the word!

More precisely, *StateQ* returns its result incrementally while *CpStQ* doesn't return anything until the entire computation terminates. This is easily observed on the Stern-Brocot tree: *runStateQ (getBranches foldrByLevel [sternBrocot])* returns useful data, while *runCpStQ (getBranches foldrByLevel [sternBrocot])* gets stuck in an infinite nonproductive loop.

Fortunately, incremental results can be recovered through the use of *mapCont*, as illustrated in Listing 29. By tweaking *foldrByLevel* to use this control operator, we can traverse the Stern-Brocot tree. This would not be possible had we used the strict state monad, or hidden the final result type.

Allison's Queues in Direct Style

There are two styles for programming with continuations. The first is by explicitly by writing functions in the continuation passing style. (CPS) In this style, all calls to non-primitive functions are tail calls, and functions have an extra continuation parameter, which this paper has called *k*. By contrast, the direct style does not manipulate continuations explicitly, but rather uses them implicitly or via control operators such as *callCC*, or *shift* and *reset*.

This paper uses an extended CPS that allows the tail of a lazy cons to be considered a "tail call", even though it is not a proper tail call. In fact, the first four *levelOrder* variants are already written in this extended CPS, albeit in a static form that does not parameterize the continuations. They move towards a more direct style, with only *enQ* and *deQ* written in an explicit, parameterized CPS.

Completing this process by writing *enQ* and *deQ* in direct style as well is a natural theoretical endeavor. Of course, *enQ* uses the lazy cons extension to CPS, and in direct style this corresponds to *mapCont*.

$$\begin{aligned} \text{mapCont} &:: (r \rightarrow r) \rightarrow \text{Cont } r \ a \rightarrow \text{Cont } r \ a \\ \text{mapCont } f \ m &= \text{Cont } (\lambda k \rightarrow f (\text{runCont } m \ k)) \end{aligned}$$

The type of our *CpSt* idiom is $(a \rightarrow s \rightarrow r) \rightarrow s \rightarrow r$, which is isomorphic to *ContT r (Reader s) a*, so this would be a plausible place to start. Listing 31 demonstrates a way to implement state operation in terms of continuations and readers. The function *ask* is defined in *Control.Monad.Reader* and retrieves the value from the reader, while *local* takes a function and a monad, and modifies the value in the reader during the execution of it's second argument. The reader maintains it's original value otherwise.

```

getReader :: MonadReader a m => m a
getReader    = ask

setReader :: (MonadReader a m, MonadCont m) => a -> m ()
setReader    = modReader o const

modReader :: (MonadReader a m, MonadCont m) => (a -> a) -> m ()
modReader f = callCC (\k -> local f (k ()))

stepReader :: (MonadReader a m, MonadCont m) => (a -> (b, a)) -> m b
stepReader f = do
  st <- getReader
  let (r, st') = f st
  setReader st'
  return r

```

Listing 31: State effects with readers and *callCC*

```

data Len a = Len ! Int a

deQ_len (Len 0      q ) = (Nothing, Len 0 q )
deQ_len (Len (n + 1) (e : q')) = (Just e  , Len n q')

inc_len (Len n head) = Len (n + 1) head

```

Listing 32: Utility functions for tracking length

```

newtype CorecQ' e a
  = CorecQ' { unCorecQ' :: ContT [e] (Reader (Len [e])) a }
  deriving (Monad)

instance MonadQueue e (CorecQ' e) where
  enQ e = CorecQ' (mapContT (liftM (e:)) (modReader inc_len))
  deQ   = CorecQ' (stepReader deQ_len)

runCorecQ' :: CorecQ' e a -> [e]
runCorecQ' (CorecQ' m) = q
  where q = runReader (runContT m endpoint) (Len 0 q)
        endpoint _ = return []

```

Listing 33: Enqueue and dequeue in direct style

By using *callCC* to grab the entire remainder of the computation, we can use *local* to mutate the reader. With the addition of a few helper functions to manage the counter, we are set up for concise definitions of *enQ* and *deQ* in direct style.

Independence of *mapCont*

The use of *mapCont* is notable because I am confident that *enQ* cannot be expressed in terms of *callCC*, $(\gg=)$, and *return*. The MTL’s continuations are partially delimited, as seems necessary for the general utility of *mapCont*. However, the analogous conjecture in terms of *shift* and *reset* is certainly not true.

It may not be obvious why *mapCont* is independent of the rest, but it turns out to be fairly trivial: *callCC*, $(\gg=)$, and *return* simply offer no way to add to the control context by introducing something that is not a proper tail call. More formally, we can modify the type of *CpSt*, which uses higher-ranked types to hide the result type, into a form that admits *callCC*, but prohibits a useful form of *mapCont*.

```

newtype CpSt' s a
  = CpSt' { runCpSt' ::  $\forall r. (\forall r'. a \rightarrow s \rightarrow r') \rightarrow s \rightarrow r$  }
instance Monad (CpSt' s) where
  return a = CpSt' ( $\lambda k \rightarrow k a$ )
  m  $\gg=$  f = CpSt' ( $\lambda k \rightarrow runCpSt' m (\lambda a \rightarrow runCpSt' (f a) k)$ )
instance MonadCont (CpSt' s) where
  callCC f = CpSt' ( $\lambda k \rightarrow runCpSt' (f (\lambda a \rightarrow CpSt' (\lambda _ \rightarrow k a))) k$ )
  mapCpSt' ::  $(\forall a. a \rightarrow a) \rightarrow CpSt' s b \rightarrow CpSt' s b$ 
  mapCpSt' f m = CpSt' ( $\lambda k \rightarrow f (runCpSt' m k)$ )

```

Listing 34: A “proof” of the independence of *mapCont*

Note that the code in Listing 34 is purely theoretical, not useful, because computations inside of *CpSt'* cannot be observed without cheating. The definitions are the same as the corresponding definitions of *Cont*. Since the use of typeclasses are not essential, these definitions have meaning independent of their types.

The types of *callCC*, *return*, and $(\gg=)$ are unchanged, however the definition of *mapCont* gives a type error without providing an explicit higher-ranked type. Thanks to free theorems, there are only three inhabitants of type $\forall a \circ a \rightarrow a$: *id*, *const* \perp and \perp , none of which are useful. This argument may not be complete, but I believe it can be the basis for a formal proof of independence.

Corecursive Queue Transformers

Now that *CorecQ* is in the direct style, it is somewhat easier to come up with a plausible monad transformer. Unfortunately, *runCorecQT* is mostly broken. For example, as noted previously, the corecursive queue implementation makes use of implicit mutation, and thus depends on enforced linearity. The non-deterministic list monad enables us to regain non-linear, persistent use. Not surprisingly, the list monad is incompatible with this transformer.

Those interested should experiment with which monads work and which don’t. Of particular interest is the *IO* monad. Getting a simple variant of Unix’s *tail* command to work properly around this transformer is an interesting exercise that presents some difficulty.

```

newtype CorecQT e m a
  = CorecQT (ContT (m [e]) (Reader (Len [e])) a)
    deriving (Monad)
instance Monad m  $\Rightarrow$  MonadQueue e (CorecQT e m) where
  enQ z = CorecQT (mapContT (liftM (liftM (z:))) (modReader inc_len))
  deQ   = CorecQT (stepReader deQ_len)
runCorecQT :: (MonadFix m)  $\Rightarrow$  CorecQT e m a  $\rightarrow$  m [e]
runCorecQT m = mfix ( $\lambda$ q  $\rightarrow$  run m end_point (Len 0 q))
where
  end_point _ = return (return [])
  run (CorecQT m) k st = runReader (runContT m k) st

```

Listing 35: A rather fragile queue transformer

The *mfix* :: *a* \rightarrow *m a* used here is the topic of Levent Erkök’s Ph.D. thesis, “Value Recursion in Monadic Computations”. [11] The thesis argues that there is no *mfix* on continuations. Note that this transformer does not contradict this conjecture, as we are using *mfix* to define the run operation, not defining an *mfix* for *CorecQT*.

Value recursion is also a primary topic of this paper, however, our application requires the use of continuations. Thus it would appear that we are discussing an alternate form of value recursion, and that CPS enables some varieties of value recursion while disabling others.

The *StateQ* monad is implemented via *State*, which has an *mfix* operator. Intuitively, it would seem as though this *mfix* semantics makes sense for any *MonadQueue* implementation. Whether or not a corecursive implementation can actually compute this semantics for *mfix* is a very interesting question. Perhaps *CorecQ* would be a good avenue for research regarding Remark 5.2.1 on page 61

of Erkök’s thesis, speculating on the existence of special cases when continuations happen to have an *mfix*.

Matt Morrow has observed that by hiding the result type of a continuation via higher-ranked types, a *mfix* operator can in fact be implemented. Of course, this technique also prohibits implementations of *callCC* and *mapCont*. An *mfix* for *CpSt* is given in Listing 36. This does not directly contradict Erkök’s conjecture, because the type of *CpSt* differs from *Cont*. Currently, this observation is a bit of a mystery, so this paper will not attempt to expound further.

Also included in Listing 36 is the alternate fixpoint operator *mfixish* that is at the heart of this paper. It does not have the same type as *mfix*; so neither does it contradict Erkök’s conjecture.

```

instance MonadFix (CpSt st) where
  mfix f = CpSt (\k st → let (a, st') = unCpSt (f a) (,) st in k a st')
  mfixish :: (r → Cont r a) → Cont r a
  mfixish f = Cont (\k → fix (\r → runCont (f r) k))
  mfixishT :: (MonadFix m) ⇒ (r → ContT r m a) → ContT r m a
  mfixishT f = ContT (\k → mfix (\r → runContT (f r) k))

```

Listing 36: Fixpoints for Value Recursion on Continuations

One might assume, as this paper tacitly does, that there is no *mfix* over a monad implemented using continuations with an exposed return type. This would imply that *CorecQ* cannot be used in conjunction with *CorecQT*, ruling out an way that one might intuitively try to implement multiple queues.

Returning Results from Corecursive Queues

Thankfully, *Control.Monad.Writer* is compatible with the queue transformer of the last section. This enables us to observe results other than the queue itself. The benefit is that we can now usefully execute *foldrByLevel* and its variants using corecursive queues.

Unfortunately, because the writer monad expects monoids, this approach isn’t really suitable for preserving the result semantics of *STQ* and other implementations given in this paper. The type *Writer e a* is just a newtype alias for (a, e) , so instead of using our monad transformer directly, we will simply use lazy pairs and start over.

Because *CorecQW* can return results other than the queue, it makes sense to implement *mapCont* and *mfixish* for this monad. For a demonstration of *mfixishQW*,

we implement Chris Okasaki’s breadth-first renumbering algorithm [12] in Listing 38.

```

newtype CorecQW w e a
  = CorecQW {unCorecQW :: ContT ([e], w) (Reader (Len [e])) a}
  deriving (Monad)

instance MonadQueue e (CorecQW w e) where
  enQ e = CorecQW (mapContT (liftM ((e:) *** id)) (modReader inc_len))
  deQ   = CorecQW (stepReader deQ_len)

instance MonadMapCC w (CorecQW w e) where
  mapCC f = CorecQW ◦ mapContT (liftM (id *** f)) ◦ unCorecQW
  runCorecQW :: CorecQW w e w → ([e], w)
  runCorecQW m = (q, w)
  where (q, w) = run m (λw → return ([], w)) (Len 0 q)
        run m k st = runReader (runContT (unCorecQW m) k) st

mfixishQW :: (w → CorecQW w e a) → CorecQW w e a
mfixishQW f = CorecQW (mfixishT (λ~(q', w) → unCorecQW (f w)))

```

Listing 37: Corecursive queues with return values

Chris Okasaki’s algorithm uses two queues and a stack to relabel a tree with increasing integers. Our implementation makes use of two separate, corecursive queues in place of the first-class queues used in the original paper. In both Chris Okasaki’s original implementation and ours, the stack is represented implicitly using the program stack. As the stack guards the second queue from falling off the end and entering a nonproductive loop, there is no need to track the length of this second queue explicitly.

Although our rendering of Chris Okasaki’s solution is **implemented** using corecursion; the function itself is **not** corecursive. It cannot renumber the Stern-Brocot tree, for example. Instead it gets stuck in an infinite nonproductive loop. For a truly corecursive implementation of breadth-first renumbering, we recall Jones and Gibbons’ solution [12][13] in Listing 39.

Performance of CorecQW

CorecQW exhibits a subtle performance discrepancy; due to the fact that we are returning lazy pairs, there are two paths of execution through the computation. Which path is followed depends on whether the consumer is demanding elements of the queue, or part of the result. This concept is fairly well known among logic programmers, but may be surprising to many functional programmers.

```

renum :: Integral int => Tree a b -> Tree int int
renum t = last q2
  where
    (-, q2) = runCorecQW (mfixishQW (\q2 -> trav 0 q2 t >> return []))
    trav n q t@(Leaf _)
      = do
        q' ← mtrav (n + 1) q
        mapCC ((Leaf n):) (return q')
    trav n q t@(Branch _ l r)
      = do
        enQ l >> enQ r
        mtrav (n + 1) q >>= \λ(r' : l' : q') ->
          mapCC ((Branch n l' r'):) (return q')
    mtrav n q = deQ >>= maybe (return q) (trav n q)

```

Listing 38: Chris Okasaki's Breadth-First Renumbering Algorithm

```

lazyRenum :: Integral int => Tree a b -> Tree int int
lazyRenum t = t'
  where
    (ns, t') = loop (0 : ns, t)
    loop (n : ns, Leaf _ _) = (n + 1 : ns', Leaf n _)
    loop (n : ns, Branch _ l r) = (n + 1 : ns'', Branch n l' r')
      where
        (ns', l') = loop (ns, l)
        (ns'', r') = loop (ns', r)

```

Listing 39: Jones and Gibbons' Corecursive Renumbering Algorithm

When applied to the running example of breadth first search, returning lazy pairs incurs either about 25% or 63% abstraction penalty compared to the original *CorecQ*, even if the extra result is (). Eager programmers accustomed to quality implementations of ML and Scheme are used to returning multiple values with little or no undue overhead. To be fair, GHC performs similar optimizations on strict pairs [14], and neither Scheme nor ML offer lazy tuples natively. Supporting lazy tuples efficiently is a significantly harder problem.

As a thought experiment, I attempted to implement my own value return mechanism, by starting with the original *CorecQ* and using *unsafePerformIO* and *IORefs* to open up a “side channel.” In the process, I broke the full laziness optimization, [15] which must be turned off in order for this code to terminate. It was instructive, as I'm suspicious I ended up creating something similar to what GHC is already doing.

The basic idea is that if we demand the result, we enter a thunk which forces a small bit of queue computation, and then re-reads itself. This process repeats until the queue computation terminates: then the thunk gets replaced with a concrete value which gets returned the next time the thunk re-reads itself. By enabling the trace output and running this code, you can see it in action.

The downside to this naive approach is that it exhibits an **inversion of demand**. The queue should be smart enough to realize that if a result is demanded, then it should demand it's own computation until a result (or part thereof) is returned, saving a number of indirect jumps.

Let me emphasize I am not advocating this style of programming, nor the use of this code! In fact, GHC's native tuples are faster! This code is simply to demonstrate the two code paths, and as such will produce different output depending on whether or not one demands the result.

This experiment appears to be a constant factor slower than GHC's tuples. It exhibits the same performance dissimilarity between the two code paths. It appears to work in the presence of *callCC*, but only implements *mapCont* for the queue, not the result. Thus an incremental *foldrByLevel* is not possible with this monad.

Performance Measurements

This was tested on an Intel Core 2 Duo T9550, and both GHC 6.10.3 and 6.8.3. The code that was used to produce these benchmarks is available on [hackage](#) as `control-monad-queue`. [16] The results of the tests can be found in Figures 4 and 5.

Each of the variants in the table were run on the 34th fibonacci tree, which has 5.7 million branches. The functions were run 20 times, and the first few trials

```

type QSt r e = IORef r → IORef [e] → Int → [e] → [e]
newtype Q r e a = Q { unQ :: ((a → QSt r e) → QSt r e) }
instance Monad (Q r e) where
  return a = Q ($a)
  m >>= f = Q (λk → unQ m (λa → unQ (f a) k))
unsafeRead ref = unsafePerformIO (readIORef ref)
unsafeWrite ref a = unsafePerformIO (writeIORef ref a)
unsafeNew a = unsafePerformIO (newIORef a)
instance Show e ⇒ MonadQueue e (Q r e) where
  enQ x = Q (λk r e ! n xs → let xs' = (k () r e $! n + 1) xs
    in trace ("enQ $ " ++ show x)
    (unsafeWrite e xs' 'seq' (x : xs')))

  deQ = Q delta
  where
    delta k r e 0 xs = k Nothing r e 0 xs
    delta k r e (n + 1) (x : xs) = trace ("deQ " ++ show x)
      (k (Just x) r e n xs)

runQ m = (trace "reading return value" 'seq' unsafeRead r (), queue)
where
  r = unsafeNew init
  init () = unsafePerformIO $ do
    trace "forcing computation\n" (return ())
    xs ← readIORef e
    force xs
    trace "reading return value\n" (return ())
    f ← readIORef r
    return (f ())
  e = unsafeNew queue
  queue = unQ m breakK r e 0 queue

  force [] = return ()
  force (_: _) = return ()
  breakK a r e n xs = trace ("setting return value: " ++ show a)
    (unsafeWrite r (λ() → a) 'seq' [])

```

Listing 40: Side channel thought experiment

Description	Time		-H500M		Bytes
	mean	σ	mean	σ	
levelOrder'	446	5	172	15	44.0
CorecQ	555	5	619	4	133.5
CorecQW _	696	5	1128	6	213.6
CorecQW ()	907	56	2235	11	213.6
Side Channel _	959	3	1171	7	228.7
Side Channel ()	1500	56	2171	7	276.4
STQ	1140	8	1087	14	371.2
TwoStack	1158	4	778	10	185.8
Okasaki	1553	7	1574	12	209.0
Data.Sequence	962	5	1308	5	348.1

Figure 4: Performance using GHC 6.10.3

Description	Time		-H500M		Bytes
	mean	σ	mean	σ	
levelOrder'	461	2	173	15	44.1
CorecQ	458	4	267	13	67.5
CorecQW _	526	5	713	5	141.2
CorecQW ()	781	62	1775	62	141.3

Figure 5: Performance using GHC 6.8.3

were discarded. The remaining trials were averaged, and the standard deviation σ was computed. Timing information, presented in milliseconds, was gathered using *System.getCPUTime*, which on the test system had a resolution of 10 milliseconds. The final column of the two tables gives the average number of bytes allocated for every *Branch*.

Note that the code in this paper was not benchmarked directly for a variety of reasons. Each description is essentially equivalent to *levelOrder''* (Listing 19) run with the appropriate monad. This means that the bottom four variants don't return anything useful. While this isn't fair for implementing a drop-in replacement for *levelOrder' :: Tree a b → [Tree a b]*, it is more fair for comparing the relative performance of the queues themselves.

The tests were also attempted using the `-Hsize` option to set a suggested heap size and reduce the frequency of garbage collection; this did indeed reduce the percentage of time spent in the garbage collector, but this was usually more than offset in increased time spent in the mutator.

Related Work

The Glasgow Haskell Compiler provides *Data.Sequence*, which is based on 2-3 finger trees. [17] This offers amortized, asymptotically efficient operations to many kinds of operations on persistent sequences, and is much more general data structure than a queue.

Chris Okasaki [10] implements first-class real-time queues, even under persistent usage. It is interesting that this solution also makes essential use of laziness, and is based around the incremental reversal of lazy lists.

Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan [18] have a clever way of implementing a queue using delimited continuations. This employs the dynamic extent of *control* and *prompt*, as opposed to the static extent of *shift* and *reset*. This solution does not employ the use of circular programming.

Conclusions

For whatever reason, Lloyd Allison's queue is not widely appreciated within the modern functional programming community. This deserves to change, as corecursive queues are both academically interesting and practical. They are not as general as other queues, but when they fit a problem, they are an excellent choice. Thus they occupy an interesting place in the functional programmer's toolbox.

Acknowledgements

I'd like to thank Amr Sabry and Olivier Danvy for particularly useful comments, Matt Hellige for a fun discussion that lead to the *unsafePerformIO* thought experiment, Matt Morrow the insight that *mapCont* could not be implemented on *CpSt*, Andres Löh for some assistance with LaTeX, Stefan Ljungstrand for some criticism, and others including Dan Friedman, Will Byrd, Aziz Ghuloum, Ron Garcia, Roshan James, and Michael Adams, who enthusiastically endured my often inept attempts at explaining this work before I really understood it.

I'd also like to acknowledge the giants whose shoulders made this work possible, including Richard Bird, Philip Wadler, Daniel Friedman and David Wise, the designers and implementors of Haskell and the Monad Template Library, and of course, Robin Milner and J. Roger Hindley.

References

- [1] Lloyd Allison. Circular programs and self-referential structures. **Software Practice and Experience**, 19(2) (Feb 1989).
<http://www.csse.monash.edu.au/~lloyd/tildeFP/1989SPE/>.
- [2] Andy Gill et al. The monad template library.
<http://hackage.haskell.org/package/mtl>.
- [3] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. **Acta Informatica**, 21(3):pages 239–250 (Oct 1984).
- [4] Kees Doets and Jan van Eijck. **The Haskell Road to Logic, Maths, and Programming**. King's College Publications (2004).
- [5] Philip Wadler. Theorems for free! In **FPCA '89: Proceedings of the fourth international conference on functional programming languages and computer architecture**, pages 347–359. ACM, New York, NY, USA (1989).
<http://homepages.inf.ed.ac.uk/wadler/topics/parametricity.html>.
- [6] Daniel P Friedman, Mitchell Wand, and Christopher T Haynes. **Essentials of Programming Languages**. MIT Press, 2 edition (2001).
- [7] Daniel P Friedman and David S Wise. Cons should not evaluate it's arguments. **Automata, Languages, and Programming**, pages 257–284 (1976).
<http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR44>.
- [8] Wouter Swietstra. Why attribute grammars matter. **The Monad Reader**, 4 (Jul 2005). <http://www.haskell.org/sitewiki/images/8/85/TMR-Issue13.pdf>.

- [9] Chris Okasaki. **Purely Functional Data Structures**. Cambridge University Press (1998). <http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#cup98>.
- [10] Chris Okasaki. Simple and efficient purely functional queues and dequeues. **Journal of Functional Programming**, 5(4):pages 583–592 (Oct 1995). <http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#jfp95>.
- [11] Levent Erkök. **Value Recursion in Monadic Computations**. Ph.D. thesis, OGI School of Engineering, OHSU, Portland, Oregon (2002). <http://leventerkok.googlepages.com/erkok-thesis.pdf>.
- [12] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In **ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming**, pages 131–136. ACM, New York, NY, USA (2000). <http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#icfp00>.
- [13] Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical report, Dept of Computer Science, University of Auckland (1993). <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.4052>.
- [14] Clem Baker-Finch, Kevin Glynn, and Simon Peyton-Jones. Constructed product result analysis for haskell. **Journal of Functional Programming**, 14(2):pages 211–245 (Mar 2004). <http://research.microsoft.com/en-us/um/people/simonpj/Papers/cpr/>.
- [15] André L. M. Santos. **Compilation by transformation in non-strict functional languages**. Ph.D. thesis, University of Glasgow (Jul 1995). <http://www.di.ufpe.br/~alms/ps/thesis.ps.gz>.
- [16] Leon P Smith. control-monad-queue. <http://hackage.haskell.org/package/control-monad-queue>.
- [17] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. **Journal of Functional Programming**, 16(2):pages 197–217 (2006). <http://www.soi.city.ac.uk/~ross/papers/FingerTree.pdf>.
- [18] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. **Science of Computer Programming**, 60(3):pages 274–297 (2006). <http://www.brics.dk/RS/05/36/>.