Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design

(Functional Pearl)

Chris Okasaki
Department of Computer Science
Columbia University
cdo@cs.columbia.edu

ABSTRACT

Every programmer has blind spots. Breadth-first numbering is an interesting toy problem that exposes a blind spot common to many—perhaps most—functional programmers.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Algorithms, Design

Keywords

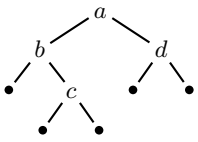
Breadth-first numbering, breadth-first traversal, views

1. INTRODUCTION

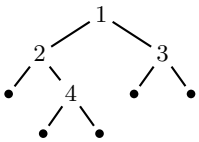
Breadth-first traversal of a tree is easy, but rebuilding the tree afterwards seems to be much harder, at least to functional programmers. At ICFP'98, John Launchbury challenged me with the following problem:

> Given a tree T, create a new tree of the same shape, but with the values at the nodes replaced by the numbers $1 \ldots |T|$ in breadth-first order.

For example, breadth-first numbering of the tree



should yield the tree



Launchbury knew of a solution by Jones and Gibbons [5] that depended on lazy evaluation, but wondered how one would solve the problem in a strict language like Standard ML [6]. I quickly scribbled what seemed to me to be a mostly straightforward answer and showed it to him at the next break.

Over the next year, I presented the problem to many other functional programmers and was continually amazed at the baroque solutions I received in reply. With only a single exception, everyone who came near a workable answer went in a very different direction from my solution right from the very beginning of the design process. I gradually realized that I was witnessing some sort of mass mental block, a communal blind spot, that was steering programmers away from what seemed to be a very natural solution. I make no claims that mine is the *best* solution, but I find it fascinating that something about my solution makes it so difficult for functional programmers to conceive of in the first place.

STOP!

Before reading further, spend ten or fifteen minutes sketching out a solution. For concreteness, assume that you have a type of labeled binary trees

```
datatype 'a Tree = E
                 | T of 'a * 'a Tree * 'a Tree
```

and that you are to produce a function

```
bfnum : 'a Tree -> int Tree
```

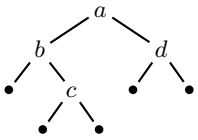
---

2. BREADTH-FIRST TRAVERSAL

When attempting to solve any non-trivial problem, the first step should always be to review solutions to related problems. In algorithm design, as in programming in general, theft of ideas is to be applauded rather than condemned. In this case, the most obvious candidate for plunder is the well-known queue-based algorithm for *breadth-first*

ICFP'00, Montreal, Canada

traversal, that is, producing a list of the labels in a tree, in breadth-first order [5]. For example, breadth-first traversal of the tree



should yield the list $[a, b, d, c]$.

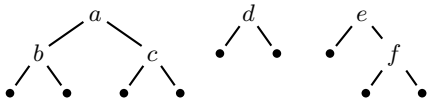
The key step in developing an algorithm for breadth-first traversal is to generalize the problem, illustrating the paradoxical, yet common, phenomenon that a more general problem is often easier to solve. In particular, we generalize the problem from breadth-first traversal of a tree to breadth-first traversal of a forest, that is, from

```
bftrav : 'a Tree -> 'a list
```

to

```
bftrav' : 'a Tree Seq -> 'a list
```

where `Seq` is some as-yet-undetermined type of sequences used to represent forests. For example, breadth-first traversal of the forest



should yield the list $[a, d, e, b, c, f]$.

Then, `bftrav` can be specified by the equation

```
bftrav t = bftrav' ⟨t⟩
```

where $\langle$`t`$\rangle$ denotes the singleton forest containing `t`.

Now, `bftrav'` is easy to specify with the following three equations:

```
bftrav' ⟨ ⟩ = []
bftrav' (E ◁ ts) = bftrav' ts
bftrav' (T (x,a,b) ◁ ts) =
    x :: bftrav' (ts ▷ a ▷ b)
```

The last equation takes the children of the first tree and adds them to the end of the sequence. The empty sequence is denoted $\langle\ \rangle$, and the symbols $\triangleleft$ and $\triangleright$ denote infix "cons" and "snoc", respectively. Since this is a specification rather than an implementation, I feel free to use $\langle\ \rangle$, $\triangleleft$, and $\triangleright$ on both sides of the equations.

The final step before actually producing code is to choose an implementation for the sequence ADT. The main operations we need on these sequences are adding trees to the end of the sequence and removing trees from the beginning of the sequence. Therefore, we choose queues as our sequence representation. Figure 1 gives a concrete implementation in Standard ML.

The use of queues as an ADT makes this code look rather ugly to an eye accustomed to the cleanliness of pattern matching, especially the `if-then-else` and `case` expressions in `bftrav'`. The problem is that pattern matching cannot normally be performed on ADTs. *Views* [10] offer a way around this problem. Figure 2 reimplements breadth-first traversal more cleanly using the syntax for views proposed in [9]. Note that the definition of `bftrav'` is now nearly identical to the specification.

```
signature QUEUE =
sig
  type 'a Queue
  val empty   : 'a Queue
  val isEmpty : 'a Queue -> bool

  val enq : 'a Queue * 'a -> 'a Queue
  val deq : 'a Queue -> 'a * 'a Queue
end

signature BFTRAV =
sig
  val bftrav : 'a Tree -> 'a list
end

functor BreadthFirstTraversal (Q:QUEUE) : BFTRAV =
struct
  open Q

  fun bftrav' q =
    if isEmpty q then []
    else case deq q of
           (E, ts) => bftrav' ts
         | (T (x,a,b), ts) =>
             x :: bftrav' (enq (enq (ts,a),b))

  fun bftrav t = bftrav' (enq (empty, t))
end
```

Figure 1: Breadth-first traversal in SML.

```
signature QUEUE =
sig
  type 'a Queue
  val empty : 'a Queue
  val >>    : 'a Queue * 'a -> 'a Queue

  viewtype 'a Queue = Empty | << of 'a * 'a Queue
end

functor BreadthFirstTraversal (Q:QUEUE) : BFTRAV =
struct
  open Q
  infix >>
  infixr <<

  fun bftrav' Empty = []
    | bftrav' (E << ts) = bftrav' ts
    | bftrav' (T (x,a,b) << ts) =
        x :: bftrav' (ts >> a >> b)

  fun bftrav t = bftrav' (empty >> t)
end
```

Figure 2: Breadth-first traversal using views.

Provided each queue operation runs in $O(1)$ time, this algorithm runs in $O(n)$ time altogether. A good implementation of queues for this application would be the usual implementation as a pair of lists [1, 2, 3]. Since this application does not require persistence, fancier kinds of queues (e.g., [3, 7]) would be overkill.

3. BREADTH-FIRST NUMBERING

We next attempt to extend the solution to breadth-first traversal to get a solution to breadth-first numbering. As in breadth-first traversal, we will begin by generalizing the problem. Instead of breadth-first numbering of a tree, we will consider breadth-first numbering of a forest. In other words, we introduce a helper function that takes a forest and returns a numbered forest of the same shape. It will also be helpful for the helper function to take the current index, so its signature will be

```
bfnum' : int -> 'a Tree Seq -> int Tree Seq
```

Then `bfnum` can be specified in terms of `bfnum'` as

```
bfnum t = t'
  where ⟨t'⟩ = bfnum' 1 ⟨t⟩
```

Extending the equations for `bftrav'` to `bfnum'` is fairly straightforward, remembering that the output forest must always have the same shape as the input forest.

```
bfnum' i ⟨ ⟩ = ⟨ ⟩
bfnum' i (E ◁ ts) = E ◁ ts'
  where ts' = bfnum' i ts
bfnum' i (T (x,a,b) ◁ ts) = T (i,a',b') ◁ ts'
  where ts' ▷ a' ▷ b' = bfnum' (i+1) (ts ▷ a ▷ b)
```

Notice how every equation textually preserves the shape of the forest.

Given these specifications, we need to choose a representation for sequences. The main operations we need on forests are adding and removing trees at both the front and the back. Therefore, we could choose double-ended queues as our sequence representation (perhaps using Hoogerwoord's implementation of double-ended queues [4]). However, a closer inspection reveals that we treat the input forest and the output forest differently. In particular, we add trees to the back of input forests and remove them from the front, whereas we add trees to the front of output forests and remove them from the back. If we remove the artificial constraint that input forests and output forests should be represented with the same kind of sequence, then we can represent input forests as ordinary queues and output forests as backwards queues.

If we want to represent both input forests and output forests as ordinary queues (perhaps because our library doesn't include backwards queues), then we can change the specification of `bfnum'` to return the numbered forest in reverse order. Then, the equations become

```
bfnum' i ⟨ ⟩ = ⟨ ⟩
bfnum' i (E ◁ ts) = ts' ▷ E
  where ts' = bfnum' i ts
bfnum' i (T (x,a,b) ◁ ts) = ts' ▷ T (i,a',b')
  where b' ◁ a' ◁ ts' = bfnum' (i+1) (ts ▷ a ▷ b)
```

Now it is a simple matter to turn this specification into running code, either with views (Figure 4) or without (Figure 3). Either way, assuming each queue operation runs in

```
signature BFNUM =
sig
  val bfnum : 'a Tree -> int Tree
end

functor BreadthFirstNumbering (Q:QUEUE) : BFNUM =
struct
  open Q

  fun bfnum' i q =
    if isEmpty q then empty
    else case deq q of
           (E, ts) => enq (bfnum' i ts, E)
         | (T (x,a,b), ts) =>
             let val q  = enq (enq (ts, a), b)
                 val q' = bfnum' (i+1) q
                 val (b', q'') = deq q'
                 val (a', ts') = deq q''
             in enq (ts', T (i,a',b')) end

  fun bfnum t =
    let val q = enq (empty, t)
        val q' = bfnum' 1 q
        val (t',_) = deq q'
    in t' end
end
```

Figure 3: Breadth-first numbering in SML.

```
functor BreadthFirstNumbering (Q:QUEUE) : BFNUM =
struct
  open Q
  infixr <<
  infix >>

  fun bfnum' i Empty = empty
    | bfnum' i (E << ts) = bfnum' i ts >> E
    | bfnum' i (T (x,a,b) << ts) =
        let val b' << a' << ts' =
                    bfnum' (i+1) (ts >> a >> b)
        in ts' >> T (i,a',b') end

  fun bfnum t =
    let val t' << Empty = bfnum' 1 (empty >> t)
    in t' end
end
```

Figure 4: Breadth-first numbering using views.

$O(1)$ time, the entire algorithm runs in $O(n)$ time. Once again, the usual implementation of queues as a pair of lists would be a good choice for this algorithm.

4. LEVEL-ORIENTED SOLUTIONS

Nearly all the alternative solutions I received from other functional programmers are *level oriented*, meaning that they explicitly process the tree (or forest) level by level. In contrast, my queue-based solutions do not make explicit the transition from one level to the next. The main advantage of the level-oriented approach is that it relies only on lists, not on fancier data structures such as queues or double-ended queues.

I will not attempt to describe all the possible level-oriented solutions. Instead, to provide a fair comparison to my queue-based approach, I will describe only the cleanest of these designs. (For completeness, I also review Jones and Gibbons' algorithm in Appendix A, but their algorithm is not directly comparable to mine since it depends on lazy evaluation.)

Given a list of trees, where the roots of those trees form the current level, we can extract the next level by collecting the subtrees of any non-empty nodes in the current level, as in

```
concat (map children lvl)
```

where

```
fun children E = []
  | children (T (x,a,b)) = [a,b]
```

Later, after a recursive call has numbered all the trees in the next level, we can number the current level by walking down both lists simultaneously, taking two numbered trees from the next level for every non-empty node in the current level.

```
fun rebuild i [] [] = []
  | rebuild i (E :: ts) cs = E :: rebuild i ts cs
  | rebuild i (T (_,_,_) :: ts) (a :: b :: cs) =
      T (i,a,b) :: rebuild (i+1) ts cs
```

The last tricky point is how to compute the starting index for numbering the next level from the starting index for the current level. We cannot simply add the length of the list representing the current level to the current index, because the current level may contain arbitrarily many empty nodes, which should not increase the index. Instead, we need to find the number of non-empty nodes in the current level. Although we could define a custom function to compute that value, we can instead notice that each non-empty node in the current level contributes two nodes to the next level, and therefore merely divide the length of the next level by two. The complete algorithm appears in Figure 5.

This algorithm makes three passes over each level, first computing its length, then collecting its children, and finally rebuilding the level. At the price of slightly messier code, we could easily combine the first two passes, but there seems to be no way to accomplish all three tasks in a single pass without lazy evaluation.

5. DISCUSSION

Comparing my queue-based solution with the level-oriented solution in the previous section, I see no compelling reason

```
structure BreadthFirstNumberingByLevels : BFNUM =
struct
  fun children E = []
    | children (T (x,a,b)) = [a,b]

  fun rebuild i [] [] = []
    | rebuild i (E :: ts) cs = E :: rebuild i ts cs
    | rebuild i (T (_,_,_) :: ts) (a :: b :: cs) =
        T (i,a,b) :: rebuild (i+1) ts cs

  fun bfnum' i [] = []
    | bfnum' i lvl =
        let val nextLvl = concat (map children lvl)
            val j = i + (length nextLvl div 2)
            val nextLvl' = bfnum' j nextLvl
        in rebuild i lvl nextLvl' end

  fun bfnum t = hd (bfnum' 1 [t])
end
```

Figure 5: Level-oriented breadth-first numbering.

to prefer one over the other. The level-oriented solution is perhaps slightly easier to design from scratch, but the queue-based algorithm is only a modest extension of the queue-based algorithm for breadth-first traversal, which is quite well-known (more well-known, in fact, than the level-oriented algorithm for breadth-first traversal). Informal timings indicate that the level-oriented solution to breadth-first numbering is slightly faster than the queue-based one, but the difference is minor and is not in any case an a priori justification for favoring the level-oriented approach.

Why is it then that functional programmers faced with this problem so overwhelmingly commit to a level-oriented approach right from the beginning of the design process? I can only speculate, armed with anecdotal responses from those programmers who have attempted the exercise. I have identified four potential explanations:

- *Unfamiliarity with the underlying traversal algorithm.* A programmer unfamiliar with the queue-based algorithm for breadth-first traversal would be exceedingly unlikely to come up with the queue-based algorithm for breadth-first numbering. However, this accounts for only a small fraction of participants in the exercise.

- *Unfamiliarity with functional queues and double-ended queues.* A programmer unfamiliar with the fact that such data structures can be implemented functionally would be unlikely to design an algorithm that required their use. In this category, I perhaps have an unfair advantage, having invented a variety of new implementations of functional queues and double-ended queues [8]. But most programmers profess an awareness that these data structures are available off-the-shelf, even if they couldn't say offhand how those implementations worked.

- *Premature commitment to a data structure.* Most functional programmers immediately reach for lists, and try something fancier only if they get stuck. Even the programmer who initially chooses queues is likely to

run into trouble because of the opposite orientations of the input and output queues. The queue-based algorithm is easiest to develop if you begin with an abstract notion of sequences and commit to a particular representation of sequences only at the end of the process.

- *Premature commitment to a programming language.* Or, to be more precise, premature commitment to a single programming language feature: *pattern matching.* This ties back into the previous reason. Functional languages such as Standard ML and Haskell do not permit pattern matching on abstract types, thereby encouraging early commitment to a particular concrete type, in particular to a concrete type such as lists that blends nicely with pattern matching. Because of their more complicated internal structure, queues and double-ended queues do not blend nearly as well with pattern matching. *Views* offer a way around this problem, but because Standard ML and Haskell do not support views, they do not help the programmer who commits to writing legal code right from the beginning of the design process. (Again, I perhaps have an unfair advantage, having earlier proposed a notation for adding views to Standard ML [9].)

The last two reasons, if true, are particularly worrisome. We tell our students about the engineering benefits of ADTs, but then fail to use them. We nod at platitudes such as "Program *into* a language, not *in* it", but then ignore or fail to recognize the blinders imposed by our own favorite language.

Of course, one does not generally use a sledgehammer to crack a walnut—when working on a toy problem, we often permit ourselves a degree of sloppiness that we would never tolerate on a large project. Furthermore, ending up with a level-oriented solution is not by itself evidence of any sloppiness whatsoever. Still, if you accept the claim that neither solution is intrinsically easier to design than the other, then you have to wonder what external factor is causing the disparity in proposed solutions.

Acknowledgments

Thanks to John Launchbury for originally proposing the problem and to the many programmers who participated in this experiment.

6. REFERENCES

[1] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.

[2] David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981.

[3] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–53, November 1981.

[4] Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992.

[5] Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report No. 71, University of Auckland, 1993. (Also known as IFIP Working Group 2.1 working paper 705 WIN-2.).

[6] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Massachusetts, 1997.

[7] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[8] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[9] Chris Okasaki. Views for Standard ML. In *Workshop on ML*, pages 14–23, September 1998.

[10] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 307–313, January 1987.

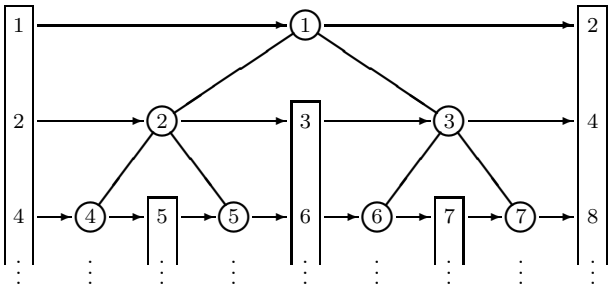


Figure 6: Threading a list of indices through a tree.

APPENDIX

A. BREADTH-FIRST NUMBERING WITH LAZY EVALUATION

Jones and Gibbons' original solution is actually for a slightly different problem known as breadth-first labelling [5]. To make comparisons easier, I adapt their algorithm to the somewhat simpler framework of breadth-first numbering.

Suppose you are magically given a list of integers representing the first available index on each level. The following Haskell function produces a tree where each level is numbered beginning with the given index. It also produces a list containing the next available index at each level. The list of indices acts as state that is threaded through the tree.

```
bfn :: ([Int], Tree a) -> ([Int], Tree Int)
bfn (ks, E) = (ks, E)
bfn (k : ks, T x a b) = (k+1 : ks'', T k a' b')
  where (ks',  a') = bfn (ks,  a)
        (ks'', b') = bfn (ks', b)
```

The effect of this function is illustrated in Figure 6.

But how do we create the initial state? Clearly, the first available index on the first level should be 1, but what about the other levels? The essential trick in Jones and Gibbons' solution is to realize that, when the entire tree has been processed, the next available index at the end of one level is actually the first available index for the next level. In other words, if `ks` is the final state, then we can construct the initial state as `1 : ks`. The overall algorithm can thus be expressed as

```
bfnum t = t'
  where (ks, t') = bfn (1 : ks, t)
```

This trick of feeding the output of a function back into the input, as illustrated in Figure 7, is where lazy evaluation is required. Without lazy evaluation, you could still use their main algorithm, but would need to calculate the initial list of indices in a separate pass.

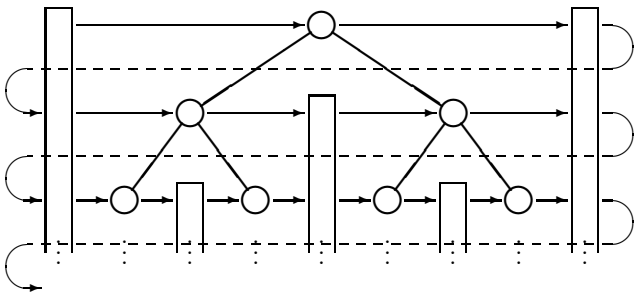


Figure 7: Threading the output of one level into the input of the next level.