Deriving Pure, Naturally-Recursive Operations for Processing Tail-Aligned Lists

Jason Hemann Daniel P. Friedman
Indiana University
{jhemann,dfried}@indiana.edu

Abstract

We present a pattern of programming useful for creating naturally-recursive solutions to a class of problems over linked lists of possibly different lengths, and perhaps even of unknown relative size. Our problems all require these lists be viewed as tail-aligned, rather than head aligned as is usual. These problems are simply-stated operations over familiar, benign data structures but they appear to require sophisticated programming techniques in order to achieve reasonable and elegant solutions.

Categories and Subject Descriptors F.2.2 [*Computations on discrete structures*]

General Terms Languages

Keywords continuations, lists, suffix, Scheme

It's a poor sort of memory that only works backwards
— Lewis Carroll, *Alice in Wonderland*

Dear Reader:

Before proceeding any further, could you spend a few minutes thinking about the following programming exercises?

Lopsided Longest Common Suffix

Given two lists of lengths m and n respectively ($m \leq n$), write a function that finds the *longest common suffix* using n recursive calls, and at most m returns from those calls.

Longest Common Suffix

Given two lists of lengths m and n respectively, find the *longest common suffix* using $max(m, n)$ serious function calls, and at most $min(m, n)$ returns from those calls.

Thank you.

1. Introduction

We present a pattern of programming for solving problems over tail-aligned linked lists. These are interesting in part because while the problems are simply stated and the data structures are common and unassuming we appear to require sophisticated programming techniques in order to achieve elegant, naturally-recursive solutions. Moreover, while left-facing linked lists are inefficient structures for these sorts of problems, they may nonetheless show up in practice. Chief aspects of the problems we solve are are:

- No sharing among the lists
- Lists are finite (non-circular) but of arbitrary length
- Lists do not know their length
- Lists must be viewed as tail-aligned (right-justified)
- cannot precompute the locations or number of pairs of elements to compare

As a chief motivating example we consider the problem of finding the *longest common suffix* of two lists of unknown and possibly different (finite) lengths. There is a perfectly natural recursive solution for finding longest common *prefix* of two such lists: we proceed using a fold-like recursive descent down both lists in tandem until either (a) one of the lists terminates, or (b) the elements l_{1_i} and l_{2_i} differ.

```
(define (lcp l₁ l₂)
  (cond
    ((or (null? l₁)
         (null? l₂)
         (not (eqv? (car l₁) (car l₂))))
     '())
    (else (cons (car l₁) (lcp (cdr l₁) (cdr l₂))))))
```

We assume without loss of generality that elements of l_1 and l_2 can be compared with Scheme's `eqv?` predicate [9]. This is the kind of ordinary list processing functions that might be assigned in an introductory programming course [11]. A naïve solution to the longest common suffix problem relies on `lcp`. If we reverse both

inputs, find the longest common prefix, and then reverse it, the result is the longest common suffix.

```
(define (lcs l₁ l₂)
  (reverse (lcp (reverse l₁) (reverse l₂))))
```

While intuitive, this implementation is inelegant in certain respects. We draw attention to several features of this. Unlike the definition of `lcp`, we do not traverse the lists in parallel. Nor do we make this single, fold-like traversal. We instead make multiple traversals over the lists, and over the result. We aim to create a definition of `lcs` that also shares this behavior. We improve our implementation starting with a simple change to `lcp`. We will now cons onto an accumulator as we recur down the lists until we reach the termination condition. This builds the largest common prefix in reverse, and so removes the need for the waiting call to `reverse` in `lcs`.

```
(define (rlcp l₁ l₂ a)
  (cond
    ((or (null? l₁)
         (null? l₂)
         (not (eqv? (car l₁) (car l₂))))
     a)
    (else (rlcp (cdr l₁) (cdr l₂) (cons (car l₁) a)))))

(define (lcs l₁ l₂)
  (rlcp (reverse l₁) (reverse l₂) '()))
```

Can we do better still? Indeed, but further improvements are not as straightforward. A still more elegant solution to the longest common suffix problem is also a more complicated one. We can however derive it via a sequence of correctness-preserving transformations. We first solve the simpler problem of *lopsided longest common suffix*: the longest common suffix when $|\texttt{l}_1| \leq |\texttt{l}_2|$.

2. Lopsided Longest Common Suffix

We begin with an elegant solution for finding the longest common suffix of two lists with the *same* length.

```
(define (lcs l₁ l₂)
  (call/cc
    (λ (j)
      (letrec
        ((lcs
          (λ (l₁ l₂)
            (cond
              ((null? l₁) '())
              ((eqv? (car l₁) (car l₂))
               (cons (car l₁) (lcs (cdr l₁) (cdr l₂))))
              (else (j (lcs (cdr l₁) (cdr l₂))))))))
        (lcs l₁ l₂)))))
```

In this definition, we first grab a "jumpout" continuation `j` and define the recursive function `lcs`. We walk down both lists simultaneously until reaching the `null?` case. Because the problem specification requires both lists have the same length, this check is sufficient in the base case. When returning from the recursion, we cons elements onto the result until reaching to the rightmost position i for which the elements l_{1_i} and l_{2_i} differ. This is the first position, starting from the final tails of the lists, where the elements at that position differ. When we reach this position, we invoke the continuation and abandon all waiting `cons`es and continuation invocations. This requires at worst n steps and n stack frames. We make only one traversal into the lists, and only return as far as the rightmost difference.

```
> (lcs '(1 2 3) '(1 2 3))
(1 2 3)
```

Since both lists are required to have the same length, at each step as we walk down the lists we know how to compute the result of the recursive call. The lists have the same length, so the elements we wish to compare are always in lock-step.

From the solution to this problem, we will derive a solution to the more general problem of finding the longest common suffix of two lists when we know that $|\texttt{l}_1| \leq |\texttt{l}_2|$. For this more general version of the problem we cannot set up the correct element-wise comparisons as we recur into the lists. We must compare the lists element-wise, aligning on the *ends* of the lists. We cannot if or by how much `l₂` is longer than `l₁` until we reach the end of `l₂` (see Figure 1).

We first transform the internal definition of `lcs` into continuation-passing style, still just invoking the captured jumpout continuation upon encountering the rightmost difference. This correctness-preserving transformation yields a program that still correctly computes the longest common prefix for lists of the same length.

```
(define (lcs l₁ l₂)
  (call/cc
    (λ (j)
      (letrec
        ((lcs
          (λ (l₁ l₂ k)
            (cond
              ((null? l₁) (k '()))
              (else
                (lcs (cdr l₁)
                     (cdr l₂)
                     (λ (r)
                       (if (eqv? (car l₁) (car l₂))
                           (k (cons (car l₁) r))
                           (j r)))))))))
        (lcs l₁ l₂ j)))))
```

Since we do not know the length of the second list, we cannot possibly know which of it's elements to compare to those of `l₁` until reaching the end of `l₂`. However, we *do* know as we traverse the lists where and in what

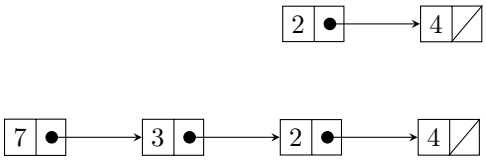

Figure 1: Lists with longest common suffix `(2 4)`.

order we will compare the elements of l_1. We can exploit this by setting up the computation on the shorter list in advance, and figuring out later against which elements of the longer list to compare.

```
(define (lcs l₁ l₂)
  (call/cc
    (λ (j)
      (letrec
        ((lcs
          (λ (l₁ l₂ k)
            (cond
              ((null? l₁) (k '()))
              (else
                ((lcs (cdr l₁)
                      (cdr l₂)
                      (λ (r)
                        (λ (a₂)
                          (if (eqv? (car l₁) a₂)
                              (k (cons a₂ r))
                              (j r)))))
                 (car l₂)))))))
        (lcs l₁ l₂ j)))))
```

We make this change in the definition above. We no longer carry in the "continuation" both elements to compare. Instead, the body is now a function expecting the second element to compare, and we set up the elements of l_2 as the arguments to these functions that are now the results of the recursion. This too is a correctness-preserving transformation; this function still correctly computes the largest common suffix of two lists of equal length. We put "continuation" in quotes here because in making this change our solution is no longer in continuation-passing style, or even tail recursive.

This change is important for our purpose because it enables us to sever the connection between the "into" and "out of" the recursion. We are no longer forced to compare the lists' elements based on their positions in our traversal into the lists.

We can think of this sequence of non-tail recursive functions as a way to "slot" a sequence of element-wise comparisons. To correctly solve this problem for a potentially-longer second list, we need to slot these comparisons in the right places. Rather than immediately begin comparing elements when we reach the end of the first list we will move the continuation, as if by some occult hand, to the end of the longer list. We recur into the remainder of the longer list, and only when reaching the end of the second list do we begin a tail-aligned element-wise comparison against the elements of l_1.

```
(define (lcs l₁ l₂)
  (call/cc
    (λ (j)
      (letrec
        ((lcs
          (λ (l₁ l₂ k)
            (cond
              ((null? l₁)
               (fold-right (λ (a₂ f) (f a₂))
                           (k '())
                           l₂))
              (else
                ((lcs (cdr l₁)
                      (cdr l₂)
                      (λ (r)
                        (λ (a₂)
                          (if (eqv? (car l₁) a₂)
                              (k (cons a₂ r))
                              (j r)))))
                 (car l₂)))))))
        (lcs l₁ l₂ j)))))
```

This completes the definition of `lcs` for lopsided lists. We rely on the `fold-right` operator of SRFI-1 [10]. Instead of `'()`, the base case of the fold is `(k '())`, which is the function awaiting the rightmost element of l_2.

```
> (lcs '(3 4 5) '(3 4 5 4 5))
(4 5)
```

3. Longest Common Suffix

We can modify this solution to solve the still more general problem of finding the longest common prefix for two lists of unknown and possibly different lengths. In the previous statement of the problem we knew that $|l_1| \leq |l_2|$ – we can no longer rely on that. Now, for both l_1 and l_2, we test if the list is empty and if so, continue to the end of the other before comparing elements. We now also build two "slottable continuations", even though we will only use one. We can only know which list is the shortest when we reach it's end. So as we traverse the lists we build one for each list, and when we reach an empty list we discover which one to use. We abstract both the creation and placement of these "slottable continuations" into help functions that we can reuse in the definition of `lcs`. For the sake of brevity in the latter definition we rely on the curried `define` syntax common to many Scheme implementations [1, 6].

```
(define (slot-f f l)
  (fold-right (λ (a f) (f a)) f l))

(define (((build-k j aₛ k) r) aₗ)
  (if (eqv? aₛ aₗ) (k (cons aₛ r)) (j r)))
```

One additional complication arises when we generalize our solution to lists of unknown relative lengths. Because we cannot know which continuation will be used as we recur into the lists, we provide the car of both lists as arguments to the slotted continuation. This creates an arity mismatch between these functions and the context in which they are invoked. One possible solution is to make these functions binary, and pass "dummy arguments" along with the elements of `l` in `slot-f`. We decline this option in favor of what we deem a more elegant solution. The `f` functions remain unary, and we eliminate the unneeded arguments by composing uncurried versions of Church booleans `tt` and `ff` with our `f` functions on the way out of the recursion.

```
(define (tt a₁ a₂) a₁)

(define (ff a₁ a₂) a₂)
```

We perform this operation repeatedly via an operator `2→1`. We take as primitive a generalized definition of `compose` common to many Scheme implementations [1, 6]. We use our uncurried Church Boolean to select the desired argument, which is then passed to the appropriate function `f`. If invoking `f` on that argument returns another such `f` awaiting yet another argument, that `f` is again passed to `(2→1 s)`, to again select the appropriate argument.

```
(define ((2→1 s) f) (compose (2→1 s) f s))
```

We now complete a definition of `lcs` correct for lists of unknown relative lengths with the help of these functions.

```
(define (lcs l₁ l₂)
  (call/cc
    (λ (j)
      (letrec
        ((lcs
           (λ (l₁ l₂ k₁ k₂)
             (cond
               ((null? l₁)
                ((2→1 ff) (slot-f (k₁ '()) l₂)))
               ((null? l₂)
                ((2→1 tt) (slot-f (k₂ '()) l₁)))
               (else
                 ((lcs
                    (cdr l₁)
                    (cdr l₂)
                    (build-k j (car l₁) k₁)
                    (build-k j (car l₂) k₂))
                  (car l₁)
                  (car l₂)))))))
          (lcs l₁ l₂ j j)))))
```

We might have curried the two cars and instead used actual Church Booleans as our selector functions, but preferred this solution as there was no *a priori* reason to always take the element of one list before the element of the other. The following examples demonstrate `lcs`'s usage on different lists of varying lengths.

```
> (lcs '() '())
()
> (lcs '(1 2 3) '(1 2 3))
(1 2 3)
> (lcs '(3 4 5) '(3 4 5 4 5))
(4 5)
> (lcs '(3 4 5) '())
()
> (lcs '(3 4 5 4 5) '(3 4 5))
(4 5)
```

The `call/cc` continuation we grab will be invoked at most once, if invoked is used only to abort waiting computation, and is never used as a value. We could rely on more efficient control operators if our Scheme system provides them [8]. Even without a more precise control operator than `call/cc`, we have implemented an elegant solution to the longest common suffix problem. We perform a single, as far as possible simultaneous, traversal of both lists, and continue into the longer list from there. We return from the recursion at most only as many times as the length of the shorter list, and in fact only as far as the position of the lists' rightmost difference, tail-indexed.

To recapitulate, we derive our solution in the following method:

- Solve the problem for lists of the same length
- Expose the control structure of the problem.
- Sever the connection between the lists by breaking tail form.
- Move the "slottable continuation" to the end of the longer list.
- Duplicate continuations to generalize solution to lists of unknown relative lengths.

Other problems can be solved in a similar fashion. In the following section, we give an example of another such problem and derive its solution.

4. Element-wise maxima of tail-aligned lists

Consider also the problem of computing a list of the element-wise maxima of two tail-indexed lists with unknown relative lengths. We wish to return a list that at each tail-indexed position contains the maximum of the two lists' elements at that tail-indexed position, to the beginning of the shorter list. From that point forward, the result contains the longer lists' elements at that position, i.e. the prefix of the longer list.

```
> (maxls '(4 1) '(2 3))
(4 3)
> (maxls '(4 1) '(1 2 3))
(1 4 3)
> (maxls '(0 1 2 3) '(4 1))
(0 1 4 3)
```

We develop our solution via a similar derivation. We begin with a direct implementation of `maxls` correct for lists of the same length.

```
(define (maxls l₁ l₂)
  (cond
    ((null? l₁) '())
    (else
      (cons (max (car l₁) (car l₂))
            (maxls (cdr l₁) (cdr l₂))))))
```

As before, we will construct a version that relies on internal, continuation-passing definition of `maxls`, and then modify this internal definition to break tail form by passing an argument to the result of the recursion. We grab a jumpout continuation as we did in `lcs`. The `maxls` function demands more sophisticated control management than did `lcs`. We no longer simply abort when we return to the end of the shorter list. Instead, we build up additional computation to perform. In order to facilitate this additional complexity we first *doubly* CPS the internal definition of `maxls`, and again use `j` as our initial continuation.

```
(define (maxls l₁ l₂)
  (call/cc
    (λ (j)
      (letrec
        ((maxls
           (λ (l₁ l₂ c k)
             (cond
               ((null? l₁) (c '() k))
               (else
                 (maxls
                   (cdr l₁)
                   (cdr l₂)
                   (λ (r k)
                     (c (cons (max (car l₁) (car l₂)) r)
                        k))
                   k))))))
        (maxls l₁ l₂ (λ (r k) (k r)) j)))))
```

This definition of `maxls` is correct for lists of equal length. We then sever the connection between the "into" and "out of" the recursion, passing `(car l₂)` as an argument, and changing the result of the recursion into a function expecting such an argument. To make `maxls` also correct when the `l₂` is longer, we must "slot" the continuation into the correct location and also extend the continuation to correctly operate on the initial, leftmost elements of the longer list. We accomplish both of these with a new, recursive definition of `slot-f`.

```
(define (slot-f c l k)
  (cond
    ((null? l) (c '() k))
    (else
     ((slot-f c (cdr l) (λ (r)
                          (λ (a)
                            (k (cons a r)))))
      (car l)))))
```

In the case that `l₂` is empty, we proceed as before. Otherwise, we instead extend the continuation to cons the remaining leftmost elements of the longer list onto the result of the recursion. We break tail form here as well, both because we cannot know when recurring into `l₂` which elements `a₂` we want to cons onto the front and because we will likely need to compare the elements of `l₁` against the elements of `l₂` over which we are currently traversing. The following definition of `maxls` is now correct also when the second list is longer than the first.

```
(define (maxls l₁ l₂)
  (call/cc
    (λ (j)
      (letrec
        ((maxls
           (λ (l₁ l₂ c k)
             (cond
               ((null? l₁) (slot-f c l₂ k))
               (else
                 ((maxls
                    (cdr l₁)
                    (cdr l₂)
                    (λ (r k)
                      (λ (a₂)
                        (c (cons (max (car l₁) a₂) r)
                           k)))
                    k)
                  (car l₂)))))))
        (maxls l₁ l₂ (λ (r k) (k r)) j)))))
```

To make `maxls` correct for two lists of unknown relative lengths we make essentially the same changes we did for `lcs`: we add a second base case, construct and pass two continuations `c`, and pass the cars of both lists. We avoid passing continuations `k` as arguments to `maxls` since we only extend this continuation as we return from the recursion. A definition of `maxls` with all of these changes is below.

```
(define (((build-k c aₛ) r k) a₁)
  (c (cons (max aₛ a₁) r) k))
```

We also rely on the definitions of `tt`, `ff`, and `2→1` from before. We use `j` as the initial continuation rather than the identity function because the `call/cc` continuation escapes a waiting call to `2→1` when we compute the final value.

```
(define (maxls l₁ l₂)
  (call/cc
    (λ (j)
      (letrec
        ((maxls
           (λ (l₁ l₂ c₁ c₂)
             (cond
               ((null? l₁)
                ((2→1 ff) (slot-f c₁ l₂ j)))
               ((null? l₂)
                ((2→1 tt) (slot-f c₂ l₁ j)))
               (else
                 ((maxls
                    (cdr l₁)
                    (cdr l₂)
                    (build-k c₁ (car l₁))
                    (build-k c₂ (car l₂)))
                  (car l₁)
                  (car l₂)))))))
         (let ((c (λ (r k) (k r))))
           (maxls l₁ l₂ c c))))))
```

5. Related Work

Programs over lists and other singly-linked data structures have been of interest since the earliest days of non-numeric computation [13]. There is an almost unending supply of fun and interesting puzzles based around lists, continuations, or both [2, 5, 7].

Writing programs of this sort brings obviously to mind the "There and Back Again" (TABA) programming style [3]. The TABA style requires traversing a second data structure at return-time. We impose the additional complications of traversing the two structures when proceeding into the recursion, and further, that we traverse them simultaneously as far as possible (until exhausting the shorter of the two). These additional restrictions required more work on our part – for instance, in the case of `lcs` a TABA definition might have consumed one list in forward order, and at return-time consume the second list (this presumes the second argument was provided in reverse order).

We expect further connections with work on defunctionalization. Wand's "Continuation-based program transformation strategies" [12] suggests transforming programs into continuation-passing style and defunctionalizing the continuations to achieve a different kind of program. Danvy explores defunctionalization in a variety of settings [4]. Although we explicitly and necessarily break tail-form to achieve our solutions, we expect the defunctionalized versions of our solutions should likewise lead to interesting direct-style solutions.

6. Conclusion

In practice, we don't always receive data in the most convenient format. This may be simply the result of unfortunate happenstance or coincidence. It may also be a result of deliberate design. For instance, good design practice suggests keeping data in a format that easily supports common operations. We may have to keep data in a format unfriendly for infrequently-executed operations in order to increase overall efficiency. For instance, if we frequently add elements to the front of lists, but only infrequently search for the longest common suffix, linked lists might actually be the appropriate data structures. We still want infrequent operations to perform no worse than necessary. In truth, though, we found these problems an intellectually challenging exercises and the solutions beautiful. It is heartening to find that simply-stated problems over such familiar data structures can still lead to new and exciting programming patterns.

References

[1] L. Courtès, A. Wingo, N. Jerram, J. Blandy, et al. Guile 2.0.11 Reference Manual, March 2014. *Full text available at* `http://www.gnu.org/software/guile/manual`, page 254.

[2] O. Danvy. On Listing List Prefixes. *ACM SIGPLAN Lisp Pointers*, 2(3-4):42–47, 1989.

[3] O. Danvy and M. Goldberg. There and Back Again. *Fundamenta Informaticae*, 66(4):397–413, 2005.

[4] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd PPDP*, pages 162–174. ACM, 2001.

[5] O. Danvy and M. Spivey. *On Barron and Strachey's cartesian product function*, volume 42. ACM, 2007.

[6] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

[7] D. P. Friedman, C. T. Haynes, and E. Kohlbecker. Programming with continuations. In *Program transformation and programming environments*, pages 263–274. Springer, 1984.

[8] C. T. Haynes and D. P. Friedman. Embedding continuations in procedural objects. *TOPLAS*, 9(4):582–598, 1987.

[9] J. Rees and W. Clinger. Revised(3) Report on the Algorithmic Language Scheme. *SIGPLAN Not.*, 21(12): 37–79, Dec. 1986. ISSN 0362-1340.

[10] O. Shivers. List Library. Scheme Request for Implementation. SRFI-1, 1999. URL `http://srfi.schemers.org/srfi-1/srfi-1.html`.

[11] G. Springer and D. P. Friedman. *Scheme and the Art of Programming*. McGraw-Hill, Inc., 1990.

[12] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM (JACM)*, 27(1):164–180, 1980.

[13] P. Woodward. List programming. *Advances in Programming and Non-numerical Computation*, 3:29, 1966.