Accumulators_S11

Accumulators

Accumulation of a result is a very useful way to view algorithms that traverse data structures. All recursive algorithms on recursive data types, e.g. lists and trees, can be viewed as accumulator algorithms. Accumulator algorithms howver, or NOT limited to recursive data structures!

The fundamental notion of an accumulator algorithm is that the data needed for a particular calculation is spread throughout a data structure, thus necessitating a traversal process that "accumulates" the answer as the algorithm "travels" from one data element to the next. Note that the notion of "accumulation" here is at its most general and does not necessarily mean an additive or multiplicative process--it could be one where each element is kept or discarded, as a filtering algorithm would do.

"Reverse Accumulation"

Let's look at an example of what the HTDP book calls a "natural" recursion algorithm:

```
;; sum a list of numbers
(define (sum lon)
   (cond
      [(empty? lon) 0]
      [(cons? lon) (+ (first lon) (sum (rest lon)))]))
```

Look at the role of the recursive result "(sum (rest lon))". The cons? clause can be described as the addition of first with the recursive result. We can say that the recursive result is the accumulation of the sum of the rest of the list. That is, the value returned by the recursive call to "sum" is the accumulated sum so far in the algorithm. All the cons? clause is doing is to create a new value for the accumulated result by adding first to recursive result.

In this sort of algorithm, the final answer is created by sweeping the data from the rear of the list (i.e. from the empty list towards the front of the list) and accumulating a sum as one sweeps through the data. The accumulated result is passed along via the return value of the function.

We will call this style of accumulation a **Reverse Accumulation** algorithm because the result is being accumulated as one "reverses" back out of the data structure.

Here's another example that uses a more generalized notion of "accumulation":

```
;; return the largest positive number in a list of numbers or zero if there are no positive values.
(define (get_largest_pos lon)
   (cond
      [(empty? lon) 0]
      [(cons? lon)
       (local
          [(define acc (get_largest_pos (rest lon)))]
          (if (> (first lon) acc) (first lon) acc))]))
```

Here we can see that the accumulated value is just the largest value from the rest of the list at any given point.

We can extract a template for reverse accumulation algorithms, which is based on the template for a list-of-any:

```
(define (f_rev loa)
   (cond
      [(empty? loa) base]
      [(cons? loa) (... (first loa)...(f_rev (rest loa)))]))
```

But we can take this a step further by noting that the "..."'s in the cons? clause just represent some function applied to first and the recursive result. In addition, the "base" is just some parameter that we could have passed in as an input parameter. Thus, we can rewrite the template as follows, including a name change, which I will justify afterwards:

```
;; foldr: (lambda any1 any2 -> any2) any2 list-of-any1 --> any2
(define (foldr func base loa)
   (cond
      [(empty? loa) base]
      [(cons? loa) (func (first loa)(foldr func base (rest loa))])))
```

The template is gone! The code is now 100% concrete, with nothing left to fill in! We have defined a function, called "foldr" ("fold-right"), which the same foldr defined in the last lab:

```
(foldr f base (list x1 x2 ... xn)) = (f x1 (f x2 ... (f xn base)))
```

Convince yourself that this definition is the generic process of reverse accumulation in a list.

foldr is a higher order function that performs a reverse accumulation algorithmic process on a list.

We can see by this analysis that a reverse accumulation requires 2 parts, the two input parameters to foldr other than the list itself:

- A function that takes first and the accumulator (the recursive result) and calculates the next accumulator value.
- A base value that is the intial accumulator value (that the empty list returns).

The above examples can thus be written in terms of foldr, by simply supplying the appropriate accumulator generating function and the accumulator base value:

```
(define (sum_rev lon)
    (foldr + 0 lon))

(define (get_largest_pos_rev lon)
   (foldr (lambda (x rr)
            (if (> x rr) x rr))
          0
          lon))
```

"Forward Accumulation"

A general mantra that we have about moving data during an algorithm's execution is to say that we are "moving data from where it is, to where it can be processed".

In a reverse accumulation algorithm on a list, we recur all the way to the empty list before we start accumulating our result because that is the only place where the return value is unequivocally defined. The result is thus accumulated in the "reverse" direction as we slowly exit the recursion layer by layer.

If we can accumulate results by moving data from the rear of the list towards the front of the list, can we also accumulate a result by moving data the *other* direction, namely from the front of the list towards the rear? But of course!

Let's look at how we would sum a list of numbers using **"Forward accumulation**":

```
(define (sum_fwd lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon)
     (local
       [(define (helper acc aLon)
          (cond
            [(empty? aLon) acc]
            [(cons? aLon) (helper (+ (first aLon) acc) (rest aLon))]))]
       (helper (first lon) (rest lon)))]))
```

Notice, first of all, that the function requires a helper function? Why?

Because the only way to pass data forward in the list is to use an input parameter, but since forward accumulation is an implementation detail of the function, there are no (and should not be any) provisions in the input parameters of the original function, sum_fwd, for passing the accumulated result. Thus a helper function with an extra input parameter is needed, to handle the accumulating value.

All the "outer" function, "sum_fwd", does is to set up the initial value of the accumulator. It is is *not* a recursive function. *Only the helper is recursive*.

We can write a template for forward accumulation, once again based on the template for a list-of-any:

```
(define (f_fwd loa)
    (cond
        [(empty? loa) base]
        [(cons? loa)
            (local
                 [(define (helper acc loa2)
                      (cond
                          [(empty? loa2) acc]
                          [(cons? loa2) (helper (... (first loa2)...acc...) (rest loa2))]))]
                 (helper (...(first loa)...base...) (rest loa)))]))
```

We can clearly see the non-recursive outer function and the recursive helper function. The accumulator value is calculated and passed forward via an extra input parameter on the helper function. We can also see that the empty cases of the two functions are not the same. In particular, the helper's empty case returns the accumulator value while the outer function's empty case performs some base operation.

Here are the two reverse accumulation examples above written in a forward accumulation style:

```
(define (sum_fwd lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon)
     (local
       [(define (helper acc aLon)
          (cond
            [(empty? aLon) acc]
            [(cons? aLon) (helper (+ (first aLon) acc) (rest aLon))]))]
       (helper (first lon) (rest lon)))]))

(define (get_largest_pos_fwd lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon)
     (local
       [(define (helper acc aLon)
          (cond
            [(empty? aLon) acc]
            [(cons? aLon) (helper (if (> (first aLon) acc) (first aLon) acc) (rest aLon))]))]
       (helper (if (> (first lon) 0) (first lon) 0) (rest lon)))]))
```

A Forward Accumulation Special Case

It is very tempting to say that the above is equivalent to:

```
(define (f_fwd loa)
    (cond
        [(empty? loa) base]
        [(cons? loa)
            (local
                 [(define (helper acc loa2)
                      (cond
                          [(empty? loa2) acc]
                          [(cons? loa2) (helper (... (first loa2)...acc...) (rest loa2))]))]
                 (helper base loa))]))
```

But to make that leap requires 2 conditions to be true:

1. The "..."'s in both the helper and the outer cons? clause must be identical.
2. "base" must represent a value, as opposed to something such as a exception.

It is important to emphasize again that this is a special case where a base value is well-defined. ***The general case template above will always work*** *and should be the template of choice if you are not sure how to write your forward accumulation algorithm.* Once you get it working, you can convert it to the more specialized case here if it warrants it.

For instance, the finding the largest element in a list follows only the general template:

```
(define (get_largest lon)
  (cond
    [(empty? lon) (error "no largest in an empty list")]
    [(cons? lon)
     (local
       [(define (helper acc lon2)
          (cond
            [(empty? lon2) acc]
            [(cons? lon2) (helper (if (> (first lon2) acc) (first lon2) acc) (rest lon2))]))]
       (helper (first lon) (rest lon)))]))
```

That said, let us consider this special case, where the two above conditions do actually hold. In that case, we can collapse our template down further, with a requisite name change as well:

```
;; foldl: (lambda any1 any2 --> any2) any2 list-of-any1 --> any2
(define (foldl func base loa)
   (cond
      [(empty? loa) base]
      [(cons? loa)
       (local
          [(define (helper acc loa2)
              (cond
                 [(empty? loa2) acc]
                 [(cons? loa2) (helper (func (first loa2) acc ) (rest loa2))]))]
          (helper base loa))]))
```

Once again, our template has reduced down to the 100% concrete code of a higher-order function. Here, the function is called "foldl" ("fold-left") and is the same as the foldl defined in the last lab:

```
(foldl f base (list x1 x2 ... xn)) = (f xn ... (f x2 (f x1 base))...)
```

foldl is a higher order function that performs a foward accumulation algorithmic process on a list *when a well-defined base value exists*.

We can see by this analysis that a foldl-type forward accumulation requires 2 parts, the two input parameters to foldl other than the list itself:

- A function that takes first and the accumulator and calculates the next accumulator value.
- A base value that is the intial accumulator value.

Hey! Aren't those exactly the same two parts that we said that reverse accumulation requires? What is this saying about forward and reverse accumulation?

Forward and reverse accumulation are just opposite directions for traversing a data structure as it is being processed and thus are fundamentally the same process.

For instance, the examples we have been using above are independent of the direction in which the list is traversed, We can write them in terms of foldl as well:

```
(define (sum_fwd lon)
    (foldl + 0 lon))

(define (get_largest_pos_fwd lon)
   (foldl (lambda (x rr)
            (if (> x rr) x rr))
          0
          lon))
```

But isn't this exactly the same code as above for the reverse accumulation case, where we just substituted foldl for foldr? Does this make sense?

On the other hand, try the following:

```
(foldr cons empty (list 1 2 3 4 5))
(foldl cons empty (list 1 2 3 4 5))
```

Can you explain the difference?

Tail Recursion

If you look at the templates for forward accumulation, you will see an interesting fact: every return value is the direct, un-processed return value of a function call, specifically the call to the helper function. Comparing to the reverse accumulation template, we see that this fact is not true in that scenario. To directly return the result of a recursive call has a special name, "tail recursion" because the "tail" of the recursive call is returned. For reasons beyond the scope of our current discussion, it turns out that it is possible to perform additional optimizations on tail-recursive algorithms, namely, one can convert them to loops, which execute much faster and with much less memory than recursive calls. In fact, in theoretical computer science, loops are nothing more than the special case of optimized tail-recursive algorithms.

Scheme compilers are defined by the Scheme standards to perform tail-recursion optimizations. We can see this if we run the following code:

```
;; make-ints: int -> list-of-ints
;; makes a list of ascending integers from 1 to n or empty if n = 0.
;; Examples:
(check-expect (make-ints 0) empty)
(check-expect (make-ints 1) (list 1))
(check-expect (make-ints 2) (list 1 2))
(check-expect (make-ints 3) (list 1 2 3))

(define (make-ints n)
  (cond
    [(zero? n) empty]
    [(positive? n)
     (local
       [(define (helper acc i)
          (cond
            [(zero? i) acc]
            [(positive? i) (helper (cons i acc) (sub1 i))]))]
       (helper (list n) (sub1 n)))]))


(define ints (make-ints 1000000))  ;; reduce this value if you run out of memory intially.

;;max: num num --> num
;; returns the larger of the two numbers.

;; The following functions both return the largest value in the given list of positive numbers
(time (foldr max 0 ints))  ;; try increasing the size of ints above by factors of 10 until you run out of
memory.
(time (foldl max 0 ints))  ;; comment out the foldr line and see if you still run out of memory.
```

You should see that the foldl version runs significantly faster than the foldr version. And if you comment out foldr and foldl one at a time, try increasing the size of "ints" by a factor of 10, you will see that foldl runs with much less memory than foldr. Why do you think that I wrote "make-ints" as a forward accumulation algorithm?

Delegation Model Programming

Loking beyond the bounds of functional programming, there is an important viewpoint on the accumulation process that becomes increasingly useful as the size of your programs scale upwards. Consider this comparison between reverse and forward accumulation:

- In reverse accumulation, you delegate to a function that processes the rest of the list. That function then returns a value (the accumulator) that you then use to complete your processing of the current data (first) and thus create the new accumulator value, (the return value, which is the completed result).
- In forward accumulation, you delegate to a function that processes the rest of the list. That function takes the partially processed data (the new accumulator, formed from first and the previous accumulator value) and completes the processing, returning the completed result.

Both processes can be expressed in terms of a delegation to the rest of the list. The only difference is what the function on the rest of list is asked to do.

In "delegation model programming", the notion is that the overall processing of data involves two main factors: the local processing of local data, e.g. the first of a list, and the delegated processing of non-local data, e.g. the rest of the list. Processing never crosses encapsulation barriers, so to the processing of encapsulated data, e.g. the data contained in the rest of the list, is always done via a delegation to another function.

When we move to object-oriented programming, the delegation model viewpoint will become the major mode in which we break down the processing of data between objects, which are amongst other things, encapsulated data. Specifically, the overall processing of a linked structure of objects will involve the delegation from one object to another.