More list processing

Jia Xiaodong

September 13, 2021

Admin matters

- Code style
- Plagiarism
- Mastery check

Built in operations

- `append(xs, ys)`
- `reverse(xs)`
- `for_each(f, xs)`
- `map(f, xs)`
- `filter(pred, xs)`
- `accumulate(f, init, xs)`
- Online reference

This week we will cover more advanced list manipulations and operations after last week's introduction to pairs and lists. Please refer to the online reference for all the built in operations on lists. Of course all of these can be done with recursion and head/tail, but take them as another layer of convenience and abstraction.

What we will mainly focus on are map, filter, and accumulate.

Map

- `map` takes in parameters `(f, xs)`
  - `xs` is a list of type `T`
  - `f` is a function of type `T => any`.
- In short: `map(f, xs)` brings `xs` from `list(e1, e2, ...)` to `list(f(e1), f(e2), ...)`.

Ex: Negating a list

```
map(x => -x, xs)
```

Map is in fact quite simple. Map takes in a function `f` and a list `xs`. `f` must be able to operate on the contents of `xs`. For example if `xs` is a list of numbers then `f` cannot be a function operating on strings. Nothing is stopping you from having a list of both numbers and strings for example, but then again `f` must be able to operate on both numbers and strings.

What map does is simply apply `f` to every single element in the list `xs`, and returns the result. The original list is not changed. This also explains the restriction on `f`. The restriction may seem quite obvious but very often it is a good sanity check on your program.

Accumulate

- `accumulate` takes in 3 parameters, `(f, init, xs)`.
  - `xs` is a list of type `T`.
  - `init` is a variable of type `U`.
  - `f` is a function of type `(T, U) => U`
- Imagine `xs, init` as a flat sequence of elements: `list(n1, n2, ..., nk, init)`. Then `accumulate` returns `f(n1, f(n2, ... f(nk-1, f(nk, init)) ...))`

Accumulate is also called folding or reducing. It takes in a list `xs`, a initial value `init`, and a reducing function `f`. The purpose of the function is to move gobble the list from right to left, 2 elements at a time. Note that accumulate does not necessarily return a list.

Most of the time `init` and the elements of `xs` will be of the same type, i.e. numbers, strings, etc. But there can be a little freedom here. Again this is just a check on if your function `f` makes sense.

The reason why there needs to be an `init` is to provide the rightmost element to be gobbled (remember `f` works on 2 elements at a time). Given `list(n1, n2, ..., nk)`, here is what happens:

- `temp = f(nk, init)`
- `temp = f(nk-1, temp)`
- `f(nk-2, temp)`
- and so on until the list is exhausted, where we return `temp`.

Filter

- `filter` takes in 2 parameters, `(pred, xs)`.
  - `xs` is a list of type `T`.
  - `pred` is a function of type `T => true/false`

Get even elements

```
filter(x => x % 2 === 0, list(1, 2, 3, 4, 5))
```

Filter may be commonly used. It takes a predicate `pred`, and goes through the whole list testing each element with it. Those that evaluate to `false` are excluded from the final return result.

Trees

Definition

A tree is a list of either elements or trees.

Example

Draw `list(list(1, 2), 3, 4)`. Compare with Fig. 2.6 in the textbook.

We have actually seen something like trees before. Back when we were drawing box and pointer diagrams, we encountered lists of lists, and in fact today we know they are in fact trees.

Since trees are in fact lists, it is very convenient to simply utilize recursion together with the list operations on them. See textbook §2.2.2 for more details. Ex. 2.32 is recommended.

S6 Q1

Implement `map` using `accumulate`.

- `accumulate(pair, init, xs)` ~ `identity` [1].

```
function my_map(f, xs) {
    return accumulate((x, y) => pair(f(x), y), null, xs);
}
```

Challenge

Implement `filter` with `accumulate`.

[1] Here again ~ is used to represent something like `equals()` or ≅.

The hint tells us this is a one-liner so it has to do with `f`.

Notice that using `pair` as the accumulating function gives us the identity function.

Once we have this settled then `f = (x, y) => pair(g(x), y)` gets us what we want.

As a challenge try implementing some other list operations in terms of other list operations.

S6 Q2

Use `filter` to write `remove_duplicates`.

```
function remove_duplicates(xs) {
    function pred(v) {
        function f(x, y) {
            return x === v ? y + 1 : y;
        }
        // return accumulate(f, *, xs) ... ?
        return accumulate(f, 0, xs) < 2;
    }
    return filter(pred, xs);
}
```


The idea behind this is to use accumulate to run through the list and check for the number of occurrences of `v`.

(If you have looked through the online reference for LISTS, you would find a particular function `member` that has not been mentioned before. An easy but slow way of checking for uniqueness is `member(v, member(v, xs))`. See if you know how this works. Exactly what is it's time complexity?)

S6 Q2

Alternative

```
function remove_duplicates(xs) {
    return is_null(xs)
        ? null
        : pair(head(xs),
            remove_duplicates(
                filter(x => !equal(x, head(xs)), tail(xs))
            );
}
```

Here is an alternative implementation. The idea behind this is to take one item, and attach it to a "cleaned" version of the remaining items.

S6 Q3

```
function makeup_amount(x, coins) {
    if (x === 0) {
        return list(null);
    } else if (x < 0 || is_null(coins)) {
        return null;
    } else {
        // Combinations that don't use the head coin.
        const combi_A = makeup_amount(x, tail(coins));
        // Combinations after we remove the head coin.
        const combi_B = makeup_amount(x - head(coins), tail(coins));
        // Combinations that use the head coin.
        const combi_C = map(x => pair(head(coins), x), combi_B);
        return append(combi_A, combi_C);
    }
}
```


We have been given a hint to complete `makeup_amount`. This function takes in a list of coins and outputs all the possible ways to makeup `x` with those coins.

This is a recursive function that just considers two cases: you use the head coin, or you don't.

If we use don't use the head coin, we can just discard it.

On the other hand, by using the head coin, the problem is equivalent to swallowing the head coin and making `x` smaller by that amount. However, we have to remember to append the head coin back to our solution list. That is the difference between `combi_B` and `combi_C`.

S6 Q4

Use `accumulate` to write `remove_duplicates`.

```
function remove_duplicates(xs) {
    return
        accumulate(
            (x, ys) => is_null(member(x, ys))
                ? pair(x, ys)
                : ys,
            null, xs);
}
```

This is also quite simple. We just run through the list, and if we can find another `x` in the tail of the list, we ignore it.

S6 Q5

Write a function `subsets(xs)` that returns the set (a list) of all subsets of `xs`.

```
function subsets(xs) {
        if (is_null(xs)) {
            return list(null)
        }
        else {
            const subset_a = subsets(tail(xs));
            const subset_b =
                map(x => pair(head(xs), x), subset_a);
            return append(subset_a, subset_b);
        }
}
```


A good source of inspiration would be coin change. A subset can either not contain the head, or it can contain it.

First let us fill in the base case. This is simple enough. Moving on, not using the head element is also fairly easy. How do we construct the subsets that contain the head element? Note that `subset_a` already contains all the subsets without the head element (wishful thinking). So actually what we have to do is to just add the head element in to every one of these elements.

Finally we just return both of the lists. Do they have conflicting elements? Try to convince yourself if they do or do not.

S6 Q6

Write a function `permute(xs)` that returns a list of all permutations of `xs`.

```
function permutations(xs) {
    if(is_null(xs) {
        return list(xs);
    }
    else {
        return map(x =>
            map(z => pair(x, z),
                permutations(remove(x, xs)) ),
            xs);
    }
}
```


There is one easier way of thinking about this problem. What we do is to hold a single element at the front, and then permute the rest.

The base case is fairly easy to fill out. Next, we can use wishful thinking on the "permute the rest" part. What follows next is then

1. Put `x` at the front of every permutation.
2. Run this for every possible item `x` in the list `xs`.

Q7

Write `accumulate_n` that accumulates a list of lists.

```
function accumulate_n(op, init, seqs) {
    return is_null(head(seqs))
        ? null
        : pair(
            accumulate(op, init, map(head, seqs)),
            accumulate_n(op, init, map(tail, seqs))
        );
}
```

`accumulate_n` works like accumulate but for a list of lists. Imagine it as creating a new list out of all the first elements and applying accumulate on them. Then, do the same thing for the second elements, and so on.

The hint already tells us what is to be done. The check for `is_null(head(seqs))` tell us that we are going to go through the inner lists element by element, recursively calling `accumulate_n`. Therefore it would be a pair of the result of accumulating on all the heads, and the recursive call to `accumulate_n` with all the tails.

How do we get all the heads and all the tails? We can simply map those functions across the list.

Q8

Write `insert(x, xs)` that puts `x` at the correct spot in `xs`.

```
function insert(x, xs) {
    if (is_null(xs)) {
        return list(x);
    }
    else {
        const y = head(xs);
        return x < y
            ? pair(x, pair(y, tail(xs)))
            : pair(y, insert(x, tail(xs)));
    }
}
```

When comparing `x` with the head of the list, if `x` is smaller then we can put it down right now and return the result. Otherwise, we will insert `x` somewhere after `y`.

Q8

Implement insertion sort using `insert`.

```
function sort(xs) {
    return accumulate(
        (x, acc) => insert(x, acc),
        list(),
        xs);
}
```

We just insert each element into an empty list, starting from the first one. What is the complexity of this sorting procedure?