A Quick Review

• Functions are the basic building blocks. They may be passed as arguments, may return functions, and can be composed together.

• While one issues commands in an imperative language, in functional programming we specify values, and computers try to reduce the values to their normal forms.

• Formal reasoning: reasoning with the form (syntax) rather than the semantics. Let the symbols do the work!

• ‘Wholemeal’ programming: think of aggregate data as a whole, and process them as a whole.

• Lazy evaluation (an implementation of normal order reduction) enhances modularity — more functions can be reused.

• Once you describe the values as algebraic datatypes, most programs write themselves through structural recursion.

• Programs and their proofs are closely related. They share similar structure, by induction over input data.

• Properties of programs can be reasoned about in equations, just like high school algebra.

1 Some Comments on Efficiency

Constant-Time v.s. Linear-Time Operations

• So far we have (surprisingly) been talking about mathematics without much concern regarding efficiency. Time for a change.
• Our representation of lists is biased: (·), head, and tail are constant-time operations, while init and last takes linear-time.

\[
\begin{align*}
\text{init } [x] & = [] \\
\text{init } (x : xs) & = x : \text{init } xs
\end{align*}
\]

• Consider init [1,2,3,4]:

\[
\begin{align*}
\text{init } (1 : 2 : 3 : 4 : []) & = 1 : \text{init } (2 : 3 : 4 : []) \\
& = 1 : 2 : \text{init } (3 : 4 : []) \\
& = 1 : 2 : 3 : \text{init } (4 : []) \\
& = 1 : 2 : 3 : []
\end{align*}
\]

List Concatenation Takes Linear Time

• Recall (·):

\[
\begin{align*}
\&[] + ys = ys \\
\&(x : xs) + ys = x : (xs + ys)
\end{align*}
\]

• Consider [1,2,3] + [4,5]:

\[
\begin{align*}
(1 : 2 : 3 : []) + (4 : 5 : []) & = 1 : ((2 : 3 : []) + (4 : 5 : [])) \\
& = 1 : 2 : ((3 : []) + (4 : 5 : [])) \\
& = 1 : 2 : 3 : ([] + (4 : 5 : [])) \\
& = 1 : 2 : 3 : 4 : 5 : []
\end{align*}
\]

• (·) runs in time proportional to the length of its left argument.

Sum, Map, etc

• Functions like sum, maximum, etc. needs to traverse through the list once to produce a result. So their running time is definitely \(O(n)\), where \(n\) is the length of the list.

• If \(f\) takes time \(O(t)\), map \(f\) takes time \(O(n \times t)\) to complete. Similarly with filter \(p\).
  
  – In a lazy setting, map \(f\) produces its first result in \(O(t)\) time. We won’t need lazy features for now, however.
2 Accumulating Parameters

Reversing a List

- The function \textit{reverse} is defined by:

\[
\begin{align*}
\text{reverse } \[] &= [], \\
\text{reverse } (x : xs) &= \text{reverse } xs \ast [x].
\end{align*}
\]

- E.g. \textit{reverse} [1, 2, 3, 4] = ((([] \ast [4]) \ast [3]) \ast [2]) \ast [1] = [4, 3, 2, 1].

- But how about its time complexity? Since (\ast) is \(O(n)\), it takes \(O(n^2)\) time to revert a list this way.

- Can we make it faster?

2.1 Fast List Reversal

Introducing an Accumulating Parameter

- Let us consider a generalisation of \textit{reverse}. Define:

\[
\begin{align*}
\text{revcat} &: [a] \to [a] \to [a] \\
\text{revcat } xs ys &= \text{reverse } xs \ast ys.
\end{align*}
\]

- If we can construct a fast implementation of \textit{revcat}, we can implement \textit{reverse} by:

\[
\text{reverse } xs = \text{revcat } xs [].
\]

Reversing a List, Base Case

Let us use our old trick. Consider the case when \(xs\) is \([]\):

\[
\begin{align*}
\text{revcat } [] \ ys \\
= & \quad \{ \text{definition of } \text{revcat} \} \\
\text{reverse } [] \ast ys \\
= & \quad \{ \text{definition of } \text{reverse} \} \\
[] \ast ys \\
= & \quad \{ \text{definition of } (\ast) \} \\
y s.
\end{align*}
\]
Reversing a List, Inductive Case

Case $x: xs$:

\[
\begin{align*}
\text{revcat} (x:xs) \; ys &= \{ \text{definition of } \text{revcat} \} \newline
\text{reverse} (x:xs) + ys &= \{ \text{definition of } \text{reverse} \} \\
&= (\text{reverse} \; xs + [x]) + ys \\
&= \{ \text{since } (xs + ys) + zs = xs + (ys + zs) \} \\
&= \text{reverse} \; xs + ([x] + ys) \\
&= \{ \text{definition of } \text{revcat} \} \\
&= \text{revcat} \; xs \; (x:ys).
\end{align*}
\]

Linear-Time List Reversal

- We have therefore constructed an implementation of $\text{revcat}$ which runs in linear time!

\[
\begin{align*}
\text{revcat} \; [] \; ys &= ys \\
\text{revcat} \; (x:xs) \; ys &= \text{revcat} \; xs \; (x:ys).
\end{align*}
\]

- A generalisation of $\text{reverse}$ is easier to implement than $\text{reverse}$ itself? How come?

- If you try to understand $\text{revcat}$ operationally, it is not difficult to see how it works.
  - The partially reverted list is accumulated in $ys$.
  - The initial value of $ys$ is set by $\text{reverse} \; xs = \text{revcat} \; xs \; []$.
  - Hmm... it is like a loop, isn’t it?

2.2 Tail Recursion and Loops

Tracing Reverse

\[
\begin{align*}
\text{reverse} \; [1,2,3,4] &= \text{revcat} \; [1,2,3,4] \; [] \\
&= \text{revcat} \; [2,3,4] \; [1] \\
&= \text{revcat} \; [3,4] \; [2,1] \\
&= \text{revcat} \; [] \; [4,3,2,1] \\
&= [4,3,2,1]
\end{align*}
\]

\[
\begin{align*}
\text{reverse} \; xs &= \text{revcat} \; xs \; [] \\
\text{revcat} \; [] \; ys &= ys \\
\text{revcat} \; (x:xs) \; ys &= \text{revcat} \; xs \; (x:ys)
\end{align*}
\]
xs, ys ← XS, [];  
while xs ≠ [] do  
    xs, ys ← (tail xs), (head xs : ys);  
return ys  

Tail Recursion

- Tail recursion: a special case of recursion in which the last operation is the recursive call.

\[
\begin{align*}
& f \ x_1 \ldots \ x_n = \text{(base case)} \\
& f \ x_1 \ldots \ x_n = f \ x'_1 \ldots \ x'_n
\end{align*}
\]

- To implement general recursion, we need to keep a stack of return addresses. For tail recursion, we do not need such a stack.

- Tail recursive definitions are like loops. Each \(x_i\) is updated to \(x'_i\) in the next iteration of the loop.

- The first call to \(f\) sets up the initial values of each \(x_i\).

Accumulating Parameters

- To efficiently perform a computation (e.g. \(\text{reverse } xs\)), we introduce a generalisation with an extra parameter, e.g.:

\[
\text{revcat } xs \ ys = \text{reverse } xs \star ys.
\]

- Try to derive an efficient implementation of the generalised function. The extra parameter is usually used to “accumulate” some results, hence the name.

  - To make the accumulation work, we usually need some kind of associativity.

- A technique useful for, but not limited to, constructing tail-recursive definition of functions.

Accumulating Parameter: Another Example

- Recall the “sum of squares” problem:

\[
\begin{align*}
\text{sumsq } [] &= 0 \\
\text{sumsq } (x : xs) &= \text{square } x + \text{sumsq } xs.
\end{align*}
\]
• The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.

• Introduce \( ssp \; xs \; n = sumsq \; xs + n \).

• Initialisation: \( sumsq \; xs = ssp \; xs \; 0 \).

• Construct \( ssp \):

\[
ssp \; [] \; n = 0 + n = n \\
ssp \; (x:xs) \; n = (square \; x + sumsq \; xs) + n \\
\quad = sumsq \; xs + (square \; x + n) \\
\quad = ssp \; xs \; (square \; x + n).
\]

2.3 Being Quicker by Doing More!

Being Quicker by Doing More?

• A more generalised program can be implemented more efficiently?

  – A common phenomena! Sometimes the less general function cannot be implemented inductively at all!

  – It also often happens that a theorem needs to be generalised to be proved. We will see that later.

• An obvious question: how do we know what generalisation to pick?

• There is no easy answer — finding the right generalisation one of the most difficulty act in programming!

• For the past few examples, we choose the generalisation to exploit associativity.

• Sometimes we simply generalise by examining the form of the formula.

Labelling a List

• Consider the task of labelling elements in a list with its index.

\[
index :: \texttt{[a]} \rightarrow \texttt{[(Int, a)]} \\
index = \texttt{zip} \; [0..]
\]

• To construct an inductive definition, the case for \( [] \) is easy. For the \( x : xs \) case:

\[
\begin{align*}
index \; (x : xs) & = \texttt{zip} \; [0..] \; (x : xs) \\
& = (0, x) : \texttt{zip} \; [1..] \; xs
\end{align*}
\]

• Alas, \( \texttt{zip} \; [1..] \) cannot be fold back to \( index \)!

• What if we turn the varying part into...a variable?
Labelling a List, Second Attempt

- Introduce \( idxFrom : [a] \rightarrow \text{Int} \rightarrow ([\text{Int}, a]) \):
  \[
  idxFrom \; xs \; n = \text{zip} \; [n..] \; xs
  \]

- Initialisation: \( index \; xs = idxFrom \; xs \; 0 \).

- We reason:
  \[
  idxFrom \; (x : xs) \; n
  = \text{zip} \; [n..] \; (x : xs)
  = (n, x) : \text{zip} \; [n + 1..] \; xs
  = (n, x) : idxFrom \; xs \; (n + 1)
  \]

3 Proof by Strengthening

Summing Up a List in Reverse

- Prove: \( \text{sum} \cdot \text{reverse} = \text{sum} \), using the definition \( \text{reverse} \; xs = \text{revcat} \; xs \; [] \). That is, proving \( \text{sum} \; (\text{revcat} \; xs \; []) = \text{sum} \; xs \).

- Base case trivial. For the case \( x : xs \):
  \[
  \text{sum} \; (\text{reverse} \; (x : xs))
  = \text{sum} \; (\text{revcat} \; (x : xs) \; [])
  = \text{sum} \; (\text{revcat} \; xs \; [x])
  \]

- Then we are stuck, since we cannot use the induction hypothesis \( \text{sum} \; (\text{revcat} \; xs \; []) = \text{sum} \; xs \).

- Again, generalise \([x]\) to a variable.

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:
  \[
  \text{sum} \; (\text{revcat} \; xs \; ys) = \text{sum} \; xs + \text{sum} \; ys
  \]

- By letting \( ys = [] \) we get the previous property.

- For the case \( x : xs \) we reason:
  \[
  \text{sum} \; (\text{revcat} \; (x : xs) \; ys)
  = \text{sum} \; (\text{revcat} \; xs \; (x : ys))
  = \{ \; \text{induction hypothesis} \; \}
  \text{sum} \; xs + \text{sum} \; (x : ys)
  = \text{sum} \; xs + x + \text{sum} \; ys
  = \text{sum} \; (x : xs) + \text{sum} \; ys
  \]
Work Less by Proving More

- A stronger theorem is easier to prove! Why is that?

- By strengthening the theorem, we also have a stronger induction hypothesis, which makes an inductive proof possible.
  - Finding the right generalisation is an art — it’s got to be strong enough to help the proof, yet not too strong to be provable.

- The same with programming. By generalising a function with additional arguments, it is passed more information it may use, thus making an inductive definition possible.
  - The speeding up of \textit{revcat}, in retrospect, is an accidental “side effect” — \textit{revcat}, being inductive, goes through the list only once, and is therefore quicker.

A Real Case

- A property I actually had to prove for a paper:
  \[
  \text{smsp} \ (\text{trim} \ (x \cdot xs)) = \text{smsp} \ (\text{trim} \ (x \cdot \text{win} \ xs)) \\
  \iff \text{smsp} \ (\text{trim} \ (x \cdot xs)) >_d \text{mds} \ xs
  \]

- It took me a week to construct the right generalisation:
  \[
  \text{smsp} \ (\text{trim} \ (zs + xs)) = \text{smsp} \ (\text{trim} \ (zs + \text{win} \ xs)) \\
  \iff \text{smsp} \ (\text{trim} \ (zs + xs)) >_d \text{mds} \ xs
  \]

- Once the right property is found, the actual proof was done in about 20 minutes.

- “Someone once described research as ‘finding out something to find out, then finding it out’; the first part is often harder than the second.”

Remark

- The \textit{sum} \cdot \textit{reverse} example is superficial — the same property is much easier to prove using the $O(n^2)$-time definition of \textit{reverse}.

- That’s one of the reason we defer the discussion about efficiency — to prove properties of a function we sometimes prefer to roll back to a slower version.

- In our exercises there is an example where you need \textit{revcat} to prove a property about \textit{reverse}.
  - Show that \textit{reverse} \cdot \textit{reverse} = \textit{id}
4 Tupling

Steep Lists

• A steep list is a list in which every element is larger than the sum of those to its right:

\[
\begin{align*}
steep & :: [\text{Int}] \rightarrow \text{Bool} \\
steep \ [\ ] & = \text{True} \\
steep \ (x : xs) & = \text{steep} \ xs \land x > \text{sum} \ xs.
\end{align*}
\]

• The definition above, if executed directly, is an \(O(n^2)\) program. Can we do better?

• Just now we learned to construct a generalised function which takes more input. This time, we try the dual technique: to construct a function returning more results.

Generalise by Returning More

• Recall that \textit{fst} \ (a, b) = a and \textit{snd} \ (a, b) = b.

• It is hard to quickly compute \textit{steep} alone. But if we define

\[
\text{steepsum} \ xs = (\text{steep} \ xs, \text{sum} \ xs),
\]

• and manage to synthesise a quick definition of \textit{steepsum}, we can implement \textit{steep} by \textit{steep} = \textit{fst} \cdot \textit{steepsum}.

• We again proceed by case analysis. Trivially,

\[
\text{steepsum} \ [\ ] = (\text{True}, 0).
\]

Deriving for the Non-Empty Case

For the case for non-empty inputs:

\[
\begin{align*}
\text{steepsum} \ (x : xs) & = \{ \text{definition of steepsum} \} \\
& \quad (\text{steep} \ (x : xs), \text{sum} \ (x : xs)) \nonumber \\
& = \{ \text{definitions of steep and sum} \} \\
& \quad (\text{steep} \ xs \land x > \text{sum} \ xs, x + \text{sum} \ xs) \nonumber \\
& = \{ \text{extracting sub-expressions involving xs} \} \\
& \quad \text{let} \ (b, y) = (\text{steep} \ xs, \text{sum} \ xs) \\
& \quad \text{in} \ (b \land x > y, x + y) \nonumber \\
& = \{ \text{definition of steepsum} \} \\
& \quad \text{let} \ (b, y) = \text{steepsum} \ xs \\
& \quad \text{in} \ (b \land x > y, x + y).
\end{align*}
\]
Synthesised Program

- We have thus come up with a $O(n)$ time program:

\[
\begin{align*}
\text{steep} & \quad = \text{fst} \cdot \text{steepsum} \\
\text{steepsum} [\_] & \quad = (\text{True}, 0) \\
\text{steepsum} (x \mathbin{:} xs) & \quad = \text{let } (b, y) = \text{steepsum } xs \\
& \quad \text{in } (b \mathbin{\&\&} x > y, x + y),
\end{align*}
\]

- Again we observe the phenomena that a more general function is easier to implement.

A Maximum Segment Sum

Recall that the maximum segment sum problem ($mss$) can be specified by

\[
mss = \text{maximum} \cdot \text{map sum} \cdot \text{segments},
\]

where $\text{segments} = \text{concat} \cdot \text{map inits} \cdot \text{tails}$. That is, the specification enumerates all segments of the input list, computes the sum of each of the segment, and pick the maximum. Also recall the definitions of $\text{inits}$ and $\text{tails}$ (not $\text{init}$ and $\text{tail}$!):

\[
\begin{align*}
\text{inits} [\_] & \quad = [[]] \\
\text{inits} (x \mathbin{:} xs) & \quad = [] : \text{map } (x :) \text{ (inits } xs),
\end{align*}
\]

\[
\begin{align*}
\text{tails} [\_] & \quad = [[]] \\
\text{tails} (x \mathbin{:} xs) & \quad = (x : xs) : \text{tails } xs,
\end{align*}
\]

and let $\text{maximum}$ be defined on non-empty lists:

\[
\begin{align*}
\text{maximum} [x] & \quad = x \\
\text{maximum} (x : xs) & \quad = x \mathbin{\ast max} \text{ maximum } xs.
\end{align*}
\]

We start with considering a simpler problem: given a list, compute the maximum sum among its prefixes. Denote this problem by $mps$ (maximum prefix sum):

\[
mps = \text{maximum} \cdot \text{map sum} \cdot \text{inits}.
\]

Can we come up with an inductive definition of $mps$? Yes, you can already do that using
what you have learned. The base case for \([\,]\) is easy. For the inductive case:

\[
\text{mps} \ (x : xs) \\
= \text{maximum} \ (\text{map} \ \text{sum} \ (\text{inits} \ (x : xs))) \\
= \text{maximum} \ (\text{map} \ \text{sum} \ ([\,] : \text{map} \ (x) \ (\text{inits} \ xs))) \\
= \text{maximum} \ (0 : \text{map} \ \text{sum} \ (\text{map} \ (x) \ (\text{inits} \ xs))) \\
= \{ \text{map} \ f \cdot \text{map} \ g = \text{map} \ (f \cdot g) \} \\
\text{maximum} \ (0 : \text{map} \ (\text{sum} \cdot (x)) \ (\text{inits} \ xs)) \\
= \{ \text{sum} \ (x : ys) = x + \text{sum} \ ys \} \\
\text{maximum} \ (0 : \text{map} \ ((x+) \cdot \text{sum}) \ (\text{inits} \ xs)) \\
= \{ \text{defn. of maximum } \} \\
0 \, '\text{max}' \ \text{maximum} \ (\text{map} \ ((x+) \cdot \text{sum}) \ (\text{inits} \ xs)) \\
= \{ \text{maximum} \ (\text{map} \ (x+) \ ys) = x + \text{maximum} \ ys \} \\
0 \, '\text{max}' \ (x + \text{maximum} \ (\text{map} \ \text{sum} \ (\text{inits} \ xs))) \\
= 0 \, '\text{max}' \ (x + \text{mps} \ xs)
\]

Thus we have an inductive definition for \(\text{mps}\):

\[
\text{mps} \ [\,] = 0 \\
\text{mps} \ (x : xs) = 0 \, '\text{max}' \ (x + \text{mps} \ xs),
\]

which runs in linear time. The key step is the one using the lemma that \(\text{maximum} \ (\text{map} \ (x+) \ ys) = x + \text{maximum} \ ys\). It needs a separate proof using the fact:

\((x + y) \, '\text{max}' \ (x + z) = x + (y \, '\text{max}' \ z)\),

that is, addition distributes over maximum. This is the key property that makes an efficient implementation of \(\text{mps}\) (and thus \(\text{mss}\)) possible.

How is \(\text{mps}\) related to \(\text{mss}\)? In fact, solutions of many segment problems start with factoring the problem into the form computing “optimal prefix for each suffix”. Here is how it works for \(\text{mss}\):

\[
\text{maximum} \cdot \text{map} \ \text{sum} \cdot \text{segments} \\
= \text{maximum} \cdot \text{map} \ \text{sum} \cdot \text{concat} \cdot \text{map} \ \text{inits} \cdot \text{tails} \\
= \{ \text{map} \ \text{sum} \cdot \text{concat} = \text{concat} \cdot \text{map} \ (\text{map} \ \text{sum}) \} \\
\text{maximum} \cdot \text{concat} \cdot \text{map} \ (\text{map} \ \text{sum}) \cdot \text{map} \ \text{inits} \cdot \text{tails} \\
= \{ \text{maximum} \cdot \text{concat} = \text{maximum} \cdot \text{map} \ \text{maximum} \} \\
\text{maximum} \cdot \text{map} \ \text{maximum} \cdot \text{map} \ (\text{map} \ \text{sum}) \cdot \text{map} \ \text{inits} \cdot \text{tails} \\
= \{ \text{map} \ f \cdot \text{map} \ g = \text{map} \ (f \cdot g) \} \\
\text{maximum} \cdot \text{map} \ (\text{maximum} \cdot \text{map} \ \text{sum} \cdot \text{inits}) \cdot \text{tails}.
\]

Thus we have

\[
\text{mss} = \text{maximum} \cdot \text{map} \ \text{mps} \cdot \text{tails}.
\]
To compute the best segment-sum, we compute the best prefix-sum for each suffix.

Since \(\text{mps}\) runs in linear time, the definition of \(\text{mss}\) above still runs in \(O(n^2)\) time. However, there is a useful “scan lemma” saying that \(\text{map} \, f \cdot \text{tails}\) can be compute efficiently, if \(f\) has the form:

\[
\begin{align*}
  f [ ] &= e \\
  f (x:xs) &= g \, x \, (f \, xs)
\end{align*}
\]

(that is, if \(f\) is an instance of a \textit{foldr}, an important concept we unfortunately cannot cover yet). The function \(\text{mps}\) fits the pattern if we let \(e = 0\) and \(g \, x \, y = 0 \cdot \text{max} \cdot (x + y)\).

Let \(\text{scan} = \text{map} \, f \cdot \text{tails}\). To derive the scan lemma we will need a property that

\[
\text{head} \,(\text{tails} \, xs) = xs,
\]

whose proof is easy. We try to construct an inductive definition of \(\text{scan}\). The base case \(\text{scan} \, [] = [e]\) is easy. For the inductive case:

\[
\begin{align*}
  \text{scan} \, (x:xs) &= \text{map} \, f \, (\text{tails} \, (x:xs)) \\
                     &= \text{map} \, f \, ((x:xs):\text{tails} \, xs) \\
                     &= f \, (x:xs):\text{map} \, f \, (\text{tails} \, xs) \\
                     &= g \, x \, (f \, xs):\text{map} \, f \, (\text{tails} \, xs) \\
                     &= \{ \, \text{let } ys = \text{map} \, f \, (\text{tails} \, xs) \, \text{ in } \, g \, x \, (\text{head} \, ys):ys \, \}
\end{align*}
\]

Thus we have shown that

\[
\text{mss} = \text{maximum} \cdot \text{scan},
\]

where \(\text{scan}\) is given by

\[
\begin{align*}
  \text{scan} \, [] &= [0] \\
  \text{scan} \, (x:xs) &= \text{let } ys = \text{scan} \, xs \\
                      &\quad \text{in } 0 \cdot \text{max} \cdot (x + \text{head} \, ys):ys.
\end{align*}
\]

You may compare that with the imperative algorithm you may know.