Yesterday we talked about structural recursion and induction on the natural numbers. Today we will discuss structural recursion and structural recursion on lists, which will allow us to illustrate the differences between structural induction and induction on the natural numbers.

First, we have to talk about what lists are. Lists are another type built in to SML (like ints, strings, bools, etc.). Like tuples, lists hold a number of elements of other types (like ints), but unlike tuples, all the elements of a list have to have the same type, and there can be an unspecified number of them.

Lists are a really natural and important data structure for functional programming, and we’ll use them a lot. Lists are so important to functional programming in fact, that Lisp, one of the first functional programming languages, got its name for being largely a List Processor.1 Believe it or not, Lisp was so important in areas like AI at one point that there were actually specialized computers whose architecture was optimized for running Lisp code.

1 Lists

A list of integers (a value of type int list) is either

[] , or

x :: xs where x : int and xs : int list.

And that’s it!

[] is pronounced “nil” or “the empty list”; :: is pronounced “cons”. Therefore, the values of type int list are lists like these:

1 :: (2 :: (3 :: (4 :: [])))

This can also be written without the parens as

1 :: 2 :: 3 :: 4 :: []

*Based on material by Brandon Bohrer, Dilsun Kaynar, Mike Erdmann and others
1The list processing functions of Lisp actually come from an even older language, IPL, by Newell, Shaw and Simon.
because :: is right-associative. For a particular list with a fixed number of elements, you can also write the elements inside of square-brackets separated by commas, as in

\[ [1,2,3,4] \]

This is just a convenient notation from SML; it’s short hand for the above form.

Notice that :: combines one element with a list\(^2\), as opposed to appending two lists together. So we cannot write: \([1,2] :: [3,4]\) to get the list \([1,2,3,4]\). If we wanted to do that, we would use the append operator \([1,2]@[3,4]\) (which we will do later in lecture).

The primary operation on lists is case analysis (and recursion):

\[
\text{case } l \text{ of } \\
\quad [] \Rightarrow \langle \text{branch1} \rangle \\
\quad \mid x :: xs \Rightarrow \langle \text{branch2, with } (x : \text{int}) \text{ and } (xs : \text{int list}) \text{ in scope} \rangle
\]

That is, giving a branch for [] and a branch for ::. In the body of the :: branch, the variable \(x\) stands for the first element of the list, and the variable \(xs\) stands for the rest of the list.

Note that :: is being used both to create lists, in value declarations, and take lists apart, in the part of the :: branch to the left of =>.

2 Structurally Recursive Functions on Lists

2.1 sum

Let’s write a function to compute the sum of the elements in a list. We’ll follow the structure of the definition of a list and see how far that gets us.

(* REQUIRES: true 
  * ENSURES: sum l ==> the sum of the numbers in l 
  *)
fun sum (l : int list) : int = 
  case l
     of [] => 0 
     | x :: xs => x + sum xs

val 15 = sum [1,2,3,4,5]

sum transforms a list into an integer by walking down the list, leaving each element alone, replacing each :: with a +, and evaluating up all of the additions once the recursion finishes. We transform a list of integers into an integer by replacing every :: with a +.

So, for example, in the trace

\^2\]We can also have lists of lists, in which case the first element will happen to be a list as well
we transform the int list expression

\[ 1 \cdot 2 \cdot [] \]

into the int expression

\[ 1 + 2 + 0 \]

and then evaluate \( 1 + 2 + 0 = 3 \).

Observe that the specification for \( \text{sum} \) looks a bit different from the specs we wrote for functions on the natural numbers. Whenever we have a function on the natural numbers, we write \( n \geq 0 \) in the spec, but we didn’t have to write anything like “L is a list” in the spec for \( \text{sum} \). The reason for this is that we only specify things that aren’t already guaranteed by the type system. Our code wouldn’t even compile if \( L \) wasn’t a list, so why bother worrying about it? On the other hand, when we write functions with natural numbers, we are actually representing natural numbers as \( \text{int} \), which is a slightly different type. To make up for the difference between the \( \text{int} \) and the mathematical concept of natural numbers, we add a spec \( n \geq 0 \). If the type already says everything we wanted to know, then there’s no need for an additional spec.

2.2 Replace

Let’s write another (relatively silly) function that follows this pattern of recursion on a list. In particular, \( \text{replace} \) replaces every element in a list of integers with the integer 7.

(* REQUIRES: true
 * ENSURES: replace l ==> a list with the same length as l,
   * all of whose elements are 7
 *)

fun replace (l : int list) : int list =
  case l
    of [] => []
    | x :: xs => 7 :: (replace xs)

val [] = replace []
val [7] = replace [-50]
val [7,7,7] = replace [1,2,3]
2.3 General Form

These two functions suggest a general template for a function on lists:

(* operate on every element of a list *)
fun f (l : int list) : A =
  case l of
   [] => < expression of type A >
   | x :: xs => < expression of type A in terms of x : int,
               xs : int list,
               and (f xs) : A >

In the cons case, you can use the first element of the list x, the rest xs, and a recursive call to f on the smaller list xs. Note that any call to f on any expression equivalent to l that gets evaluated in the recursive case causes non-termination.

2.4 Reverse

Now, let’s write a function to reverse a list. We start with the template, assuming we’ll do structural recursion:

(* REQUIRES: true
   * ENSURES: reverse [x1,x2,...,xn] ==> [xn,xn-1,...,x2,x1]
   *)
fun reverse (l : int list) : int list =
  case l of
   [] => ... (reverse xs) ...
   | x :: xs => ... (reverse xs) ...

At this point, I’m stuck, so let’s do some examples: what should the reverse of [] be? According to the spec, it needs to be all the elements of [] in the opposite order, which is []. So we can fill in the base case:

fun reverse (l : int list) : int list =
  case l of
   [] => []
   | x :: xs => ... (reverse xs) ...

Now if you’re stuck on the cons case, a good technique is to do an example where you assume the recursive call works:

reverse [1,2,3,4] should evaluate to [4,3,2,1] by the spec
reverse [2,3,4] should evaluate to [4,3,2] if the recursive call works

So the question is how to get from [4,3,2] to [4,3,2,1]. And the answer is: put 1 at the back! We can’t do this using ::, which only adds something to the front of a list. But we can using the operation @ (append), which combines two lists in order; e.g. [1,2,3]@[4,5,6] == [1,2,3,4,5,6]. Thus, the final code is:
fun reverse (l : int list) : int list =  
    case l of  
        [] => []  
        | x :: xs => (reverse xs) @ [x]

2.4.1 Append
The way we wrote reverse calls @, so let’s take a detour and talk about @ first. How do you define append? It’s built-in, but it would be defined like this:

infix @
fun (l1 : int list) @ (l2 : int list) : int list =  
    case l1 of  
        [] => l2  
        | x::xs => x :: (xs @ l2)

Append recurs down l1 and creates a new list where the [] from l1 is replaced with l2.3. Some facts about @ that might be useful later:

• @ is associative (i.e. for all lists l1, l2 and l3, (l1 @ l2) @ l3 @ l1 @ (l2 @ l3)).
• @ is total (i.e. for all lists l1 and l2, there exists a value l12 such that l1 @ l2 ⇒ l12).

2.5 Fast reverse
The reverse function we wrote before is really slow for reasons you’ll be able to explain formally soon (hint: think about what we’re doing when we call @ every time). And that makes sense. For example, if you just gave someone a stack of papers and asked them to reverse it, that’s not what they would do: picking up the whole stack is heavy and annoying. Instead, it’s more natural to pull off the first paper, put it in a new stack, and iterate that process until the first stack is empty.

To implement this, we generalize the function so that it takes a second argument, representing the second pile. This second argument is called an accumulator. Adding an accumulator is often a useful technique to make a function easier to express or (in this case) more efficient.

(* REQUIRES: true  
* ENSURES: revA ([x1,x2,...,xn], [y1, ..., ym])  
* ==> [xn,xn-1,...,x2,x1,y1,...,ym]  
*)
fun revA (l : int list, acc : int list) : int list =  
    case l of  
        [] => acc  
        | x::xs => revA(xs , x :: acc)

3 The keyword infix that appears just above the function definition there allows us to use the name @ between two arguments rather than in front of them. We’ll see this a lot later on, but we’ve actually already seen it with integer addition: + has type int * int -> int but by making it infix lets us write 9 + 10 instead of +(9,10). Lisp, which we mentioned earlier, doesn’t have infix operators and forces you to write things like (+ 9 10). As a result, Lisp is very easy for computers to read, but very difficult for humans. That’s one of the many reasons we are not teaching this class using Lisp.
fun fastReverse (l : int list) : int list = revA (l , [])

In the \texttt{x::xs} case of \texttt{revA}, we take one element from the first list, put it on the second, and then continue the process by recurring.

\texttt{fastReverse} calls \texttt{revA} with the initial right pile empty, since we start out by putting the first paper on an empty new pile.

As you probably should have guessed by now, our next goal is to prove that \texttt{fastReverse} and \texttt{reverse} are extensionally equivalent.

**Theorem 1 (Reverse (attempt)).** For all values \( l : \text{int list}, acc : \text{int list} \),

\[
\text{fastReverse} \; l \; \cong \; \text{reverse} \; l
\]

We can try to prove this theorem, but the induction hypothesis we’d get isn’t strong enough to prove the inductive case. So, we need to prove a stronger theorem directly about \texttt{revA} (this will then let you show the above theorem pretty easily).

**Theorem 2 (Reverse (stronger)).** For all values \( l : \text{int list} \),

\[
\text{revA} \; (l, acc) \; \cong \; (\text{reverse} \; l) \; @ \; acc
\]

**Proof.** The proof is by structural induction on \( l \). In all of the cases below, let \( acc \) be any value of type \text{int list}.

**Case for \([]\)** To show:

\[
\text{revA} \; ([], acc) \; \cong \; (\text{reverse} \; []) \; @ \; acc
\]

Proof:

\[
\begin{align*}
\text{revA} \; ([], acc) \\
\cong & \; acc & \text{step} \\
\cong & \; [] \; @ \; acc & \text{step, sym} \\
\cong & \; (\text{reverse} \; []) \; @ \; acc & \text{step, sym}
\end{align*}
\]

By transitivity of \( \cong \), this concludes this case.

**Case for \(x::xs\)** Inductive hypothesis:

\[
\text{revA} \; (xs, acc) \; \cong \; (\text{reverse} \; xs) \; @ \; acc
\]

To show:

\[
\text{revA} \; (x :: xs, acc) \; \cong \; (\text{reverse} \; x :: xs) \; @ \; acc
\]
Proof:

\[
\text{revA } (x :: xs, acc) \\
\cong \text{revA } (xs, x :: acc) \quad \text{step} \\
\cong (\text{reverse } xs) @ (x :: acc) \quad \text{IH} \\
\cong (\text{reverse } xs) @ (x :: (\emptyset @ acc)) \quad \text{step, sym} \\
\cong (\text{reverse } xs) @ ([x] @ acc) \quad \text{step, sym} \\
\cong ((\text{reverse } xs) @ [x]) @ acc \quad \text{***} \\
\cong (\text{reverse } x :: xs) @ acc \quad \text{step, sym}
\]

By transitivity of \(\cong\), and taking *** on faith, this concludes this case and the proof.

\[\square\]

2.6 Valuability

To finish the proof of Theorem 2 above, we need to give a justification for why

\[(\text{reverse } xs) @ ([x] @ acc) \cong ((\text{reverse } xs) @ [x]) @ acc\]

Why can’t we just use the fact that \@ is associative? Because we only know that fact about lists, and reverse \(xs\) is not a list! If we knew that there was some value \(v\) such that

\[\text{reverse } xs \Rightarrow v\]

then we could conclude

\[ (\text{reverse } xs) @ ([x] @ acc) \cong v @ ([x] @ acc) \cong (v @ [x]) @ acc \]

This is basically a more precise definition of valuable from Lecture 2.

1. **Definition.** An expression \(e\) is valuable if and only if there exists some value \(v\) such that \(e \cong v\). Specifically to this proof,
   
   - **(cons)** If \(e = e_1 :: e_2\) then \(e\) is valuable iff \(e_1\) is valuable and \(e_2\) is valuable.
   - **(app)** If \(e = (f e_1)\) then \(e\) is valuable iff \(f\) is total and \(e_1\) is valuable

2. **Definition.** A function \(f : \alpha \rightarrow \beta\) is total if and only if for all values \(v : \alpha\), \((f \ v)\) is valuable.

Note that this also introduces a new definition stating when a function is total so that we can argue about the valuability of an expression that is an application of a function to some arguments of the right type.

So, to complete the proof, since \(xs\) is a value, we need to know that reverse is total.

**Theorem 3** (Reverse is total). For all values \(l : \text{int list}\), there exists a value \(v\) such that \(\text{reverse } l \cong v\).
Proof. The proof is by structural induction on \( l \).

Case for \([\ ]\) To show: \( \text{reverse} \ [\ ] \cong v \).
Proof:

\[
\text{reverse} \ [\ ] \cong [\ ] \quad \text{step}
\]

Case for \( x :: xs \) Inductive hypothesis: \( \text{reverse} \ x :: xs \cong v \)
To show: \( \text{reverse} \ x :: xs \cong v' \)
Proof:

\[
\begin{align*}
\text{reverse} \ x :: xs & \cong (\text{reverse} \ xs) @ [x] \quad \text{step} \\
& \cong v @ [x] \quad \text{IH} \\
& \cong v' \quad \text{totality of} \ @
\end{align*}
\]

This also lets us give the following rule.

Given any function

\[
\text{fun} \ f \ (x : A) : B = e_1
\]

and any expression \( e_2 : A \), if \( e_2 \) is valuable then

\[
(f \ e_2) \cong [e_2/x]e_1
\]

This rule is useful when we’re trying to step inside a function, but the argument isn’t a value. For example, if we’re trying to show \( \text{sum} \ x :: (\text{repeat} \ xs) \cong 7 + \text{sum} \ (\text{repeat} \ xs) \), we might be tempted to simply apply a step, but we can’t because \( x :: (\text{repeat} \ xs) \) isn’t a value and SML is call-by-value. But by our rule above, as long as we can show that \( \text{repeat} \ xs \) is valuable (i.e. \( \text{repeat} \) is total), we can pretend that SML is call-by-name and conclude this equivalence.

2.7 Template for Structural Induction on Lists

Induction is applicable if you’re trying to prove a theorem of the form

“for all \( l : \text{int list} \), [some statement about \( l \) is true]”

Here’s the format that any proof by structural induction on lists should have:
Proof. The proof is by structural induction on \( l \).

Case for \( [] \). To show: [substitute \( [] \) into the statement for every instance of \( l \)]

Proof: \ldots

Case for \( x :: xs \). Let \( x \) and \( xs \) be any values of their respective types.

Inductive hypothesis: [substitute \( xs \) into the statement for every instance of \( l \)].

To show: [substitute \( x :: xs \) into the statement for every instance of \( l \)].

Proof: \ldots

\[ \square \]

2.8 Wait, What About Length?

It’s critical to note that following this template does not produce a proof by induction on the length of the list: we’re arguing about the structure of the list directly and length is never involved. This may be troubling to some of you, since we are often taught that induction is all about the natural numbers.

Why can we get away with this? It’s because we gave a rigorous inductive mathematical definition of what the structure of a list is. That definition itself is already strong enough to prove theorems about lists; we don’t need to know how long they are. Or to put it another way: Any rigorous inductive definition gives us a rigorous induction principle.

We could easily compute the length of a list—it’s just like \texttt{sum} but instead of keeping the elements around, we replace them by \texttt{1}’s:

\[
\begin{align*}
(* & \texttt{REQUIRES: } \texttt{true} \\
& \texttt{* ENSURES: } \texttt{length } l \implies \texttt{the number of elements in } l \\
& \texttt{*)} \\
\texttt{fun } \texttt{length } (l : \texttt{int list}) : \texttt{int} \texttt{=} \\
& \quad \texttt{case } l \\
& \quad \quad \texttt{of } [] \Rightarrow \texttt{0} \\
& \quad \quad | x :: xs \Rightarrow 1 + \texttt{length } xs
\end{align*}
\]

\texttt{val 5 = length (1 :: (2 :: (3 :: (4 :: (5 :: [])))))}

But this is a property of lists that can be derived from the definition of lists; it has nothing to do with how lists are formed.

This is not to say that it’s wrong to prove theorems about lists through the property of length; it just often creates many more proof obligations that a proof operating directly on the structure of the list avoids.