Lesson 4

STRUCTURAL INDUCTION AND TAIL RECURSION

May 25, 2023

1. Deconstructing Data
2. Structural Induction
3. Tail Recursion
4. More List Functions

In the last lecture, we learned about the relationship between recursion and induction.

We learned about some more fundamental constructs to the Standard ML language, such **case** and `let` expressions.

We then used these concepts to write an implementation of the `pow` function using naive recursion, as well as a faster variant `fast_pow`, using repeated squaring to do less work.

1 - Deconstructing Data

In the last lecture, we saw a brief treatment of lists and `case` expressions, which allow us to **deconstruct** lists, or dispatch on the kind of data that is present within a list.

We had an idea about how to think about lists, which is worth repeating now:

Mantra Lists can be either `[]` or `x::xs`, and *nothing more*.

But what does that really mean? It means that doing anything that involves examining a list involves *at least* two cases:

```
case L of
   [] => (* ... *)
| x::xs => (* ... *)
```

We can view this another way, equivalently:

Mantra Lists can be either empty or not, and *nothing more*.

But this perspective actually is significantly less useful for us. There is a slight distinction here.

Suppose we are interested in writing the following list function:

`take : int * int list -> int list`
REQUIRES: `n >= 0`
ENSURES: `take (n, L)` evaluates to the first `n` elements of `L`, in order. If there are not enough elements, then `L` should just be however many elements are left.

Recall our `isEmpty` function from before:

```
fun isEmpty (L : int list) : bool =
  case L of
    [] => true
  | _ => false
```

We could also define a `hd` function, which takes off the first element of a list:

`hd : int list -> int`
REQUIRES: `L` is nonempty
ENSURES: `hd L` evaluates to the first element of `L`

```
fun hd (x::xs : int list) : int = x
  | hd [] = raise Fail "impossible"
```

Furthermore, suppose we had a `tl` function, which takes every element but the first of a list:

`tl : int list -> int list`
REQUIRES: `L` is nonempty
ENSURES: `tl L` evaluates to `L`, excluding the first element

```
fun tl (x::xs : int list) : int list = xs
  | tl [] = raise Fail "impossible"
```

We'll use these to write our `take` function.

```
fun take (n : int, L : int list) : int list =
  if isEmpty L orelse n = 0 then
    []
  else
    let
      val x = hd L
      val xs = tl L
    in
      x :: take (n - 1, xs)
    end
```

What's wrong here?

We see in this implementation of `take` that we are too reliant upon preconditions.

`hd` and `tl` are functions which can possibly raise exceptions, so we have to be very careful about using them! In this case, we know that it is safe because of the `isEmpty` call, but this can quickly lead us to error if we are not careful.

Moreover, it's wasteful! We pattern match once to get out a value of type `bool`, about whether or not the list is empty or not, and then we pattern match once more to get the head and tail out. **This is duplicated work!**

The issue is that we are adhering too much to the mantra that "a list is either empty or not empty".

This is a true statement, but it's strictly less powerful than the statement that a list is `[]` or `x::xs`. The former is a statement that only gives two kinds of information – true or false. The second one sums up *exactly what data is present in a list*.

This is another way of saying that pattern matching is strictly more powerful than `if` expressions.

This general idea is an instance of a popular idea in functional programming called "parse, don't validate".

Def We say that a function `p : t -> bool`, for some type `t`, is a **validator**. It ascertains some property P of the input, and returns a boolean.

The idea is to avoid simply **validating** input by only producing a single value of truth because, as we saw with the `hd` and `tl` example, this doesn't stop us from maybe having to query that property many times down the road, producing wasted effort, and uglier code.

Def We say that a function `f : t -> t2` is, in a sense, a **parser**[1], as opposed to a validator `p`, if it produces output data such that the property `p` validates is present in the type of `t2`.

[1] This idea comes from this excellent article. I should also disclaim that this is a nontraditional usage of the word "parse", so while you can apply this thinking, be careful in using this wording, as people might be confused what you mean.

Let's try rewriting `take` again, using this logic.

```
fun take (n : int, L : int list) : int list =
  case (n, L) of
    (0, _)      => []
  | (_, [])     => []
  | (_, x::xs) => x :: take (n - 1, xs)
```

This code is much cleaner. It avoids needing to use the `hd` and `tl` functions, which are possibly error-producing, and acts as a **parser**, because instead of using an intermediary `bool` as the signal for whether `L` is nonempty, in the nonempty case, it produces a value of type `int * int list`.

The basic idea is that a value of type `int * int list` is more useful than a boolean, because **it itself is proof that the list is nonempty**. A list which contains an `int` cannot be empty, and having access to the elements that are inside the list is strictly more powerful/useful.

This is why pattern matching is more powerful than conditionals. It lets you see what values *really are*, rather than simply querying them. It actually **produces the goods**, as opposed to just making claims.

Viewed another way, "parse, don't validate" is about lifting preconditions to types instead of checks, wherever possible.

For example, let's take an example of some code that exhibits significant branching behavior:

```
if isEmpty L then
  (* 1 *)
else if List.length L >= 2 then
  (* 2 *)
else if List.length L = 1 then
  if hd L = 2 then
    (* 3 *)
  else
    (* 4 *)
```

```
case L of
   [] => (* 1 *)
| x::y::xs => (* 2 *)
| 2::xs => (* 3 *)
| x::xs => (* 4 *)
```

Much better.

2 - Structural Induction

The induction principle on natural numbers lets us prove things about numbers by viewing them as *built up from other numbers*. In essence, we are expanding our store of numbers for which we know the theorem to be true.

By way of analogy, we might imagine a bucket.

Figure 1: The "Bucket of Mathematical Truth"[2]

This bucket is meant to contain all the things that we know satisfy our theorem-to-prove.

[2] Patent pending.

By default, the bucket is empty. Our first action as theorem provers is to throw our base case into the bucket. In the case of induction on the natural numbers, this is zero.

The bucket still has a ways to go, however! To prove our statement for the next number, 1, we will apply our *inductive step*.

This tells us that, from $P(0)$, we can achieve $P(1)$.

By analogy, you should be able to convince yourself that eventually, we can throw every single natural number into the bucket.

This means that, for any natural number n, we can *eventually*, in finitely many applications of this logic, prove $P(n)$.

This is the "bucket" view of mathematical induction.

In the last lecture, we learned about the `list` type constructor, which allows us to talk about types like `int list` and `string list`.

Lists are not just good for storing data, but they admit a simple structure which allows us to easily prove things about lists. For instance, consider the following function:

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length xs
```

How might we convince ourselves that the `length` function is total?

Recall that a function `f : t1 -> t2` is total if, for all values `v : t1`, there exists a `v' : t2` such that `f v` $\hookrightarrow$ `v'`. In other words, the function reduces to a value for each valuable input.

Intuitively, it seems like we could reason about the function like so:

Clearly, the function `length` must terminate, because when given any list, it must be either empty or have a first element. If it's the first case, then we terminate, because `length` will return `0`. If not, then we will recurse and enter a shorter case, which is always guaranteed to enter a smaller case, which will eventually reach `[]`.

This is what is called a **paragraph proof**.

An Informal Proof

We would prefer to see formal proofs of correctness! Intuitive reasoning and colloquial wording can mask errors, and precision is necessary when reasoning about complex programs.

Similarly to how we will avoid "dot dot dot" reasoning when proving claims on the natural numbers, by using the technique of **mathematical induction**, we will employ the technique of **structural induction** when proving claims about lists.

Def The principle of **structural induction on lists** is as follows:

Let P be a theorem on values `v : t list`, for some type `t`. We would like to show that, for all values `v : t list`, $P(\texttt{v})$ holds.

It suffices to show that:

- $P(\texttt{[]})$ holds
- Assuming that $P(\texttt{xs})$ holds, for some `xs : int list`, show that $P(\texttt{x::xs})$ holds, for an arbitrary `x : int`.

We call this **proof by structural induction**.

Thm. `length` is total

We proceed by **structural induction** on `L : int list`.

BC `L = []`

$$\texttt{length []} \cong 0 \qquad \text{(def. of } \texttt{length}\text{)}$$

IH Case: `L = xs`, for some `xs : int list`. Assume that `length xs` $\hookrightarrow$ `v`.

IS Case: `L = x::xs`, for some `x : int`. Let's show that `length (x::xs)` $\hookrightarrow$ `v`.

$$\begin{aligned} \texttt{length (x::xs)} &\Longrightarrow \texttt{1 + length xs} && \text{(clause 2 of } \texttt{length}\text{)} \\ &\Longrightarrow^{*} \texttt{1 + v} && \text{(inductive hypothesis)} \\ &\Longrightarrow \texttt{v'} && \text{(totality of +)} \end{aligned}$$

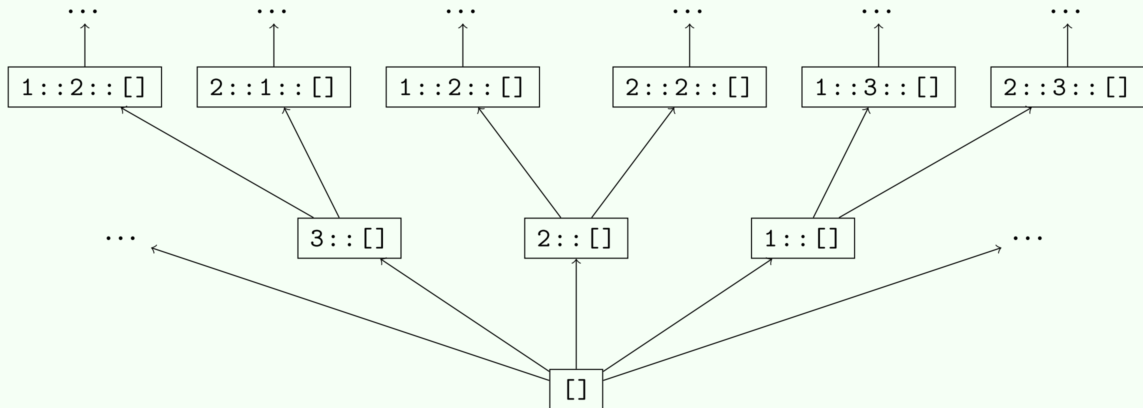
The principle of structural induction on lists looks very similar to that of mathematical induction on the natural numbers!

It's key to remember that the principle is just the same, in terms of what is "really happening" with the proof. We are showing that, through finite applications of the same inductive step, we can prove the claim for any list, from the empty list.

To visualize this, our bucket for structural induction would start with just the empty list in it:

Upon the first application of the inductive step, we would then throw in `x::[]`, for any `x : int`. Our bucket would then contain the empty list, along with every singleton list.

Repeated application of this technique will eventually produce every single list of integers.



Every node represents a particular value at which our theorem needs to hold, and the edges depict applications of the inductive step `x::xs`, at different values of `x`.

One thing to be wary of is to be sure you have correct quantification in your inductive proof.

A surefire way to lose points on a homework is to write the following statement: "Assume that, for all `xs : int list`, $P(\texttt{xs})$ holds"
Why? This is assuming the theorem!

Technically, the structural induction principle for lists looks like:

$$P(\texttt{[]}) \wedge (\forall \texttt{xs}, \texttt{x} : [P(\texttt{xs}) \implies P(\texttt{x::xs})]) \implies (\forall \texttt{L} : [P(\texttt{L})])^3$$

We do not assume that $P(\texttt{xs})$ holds for all `xs`, we show that for all `xs`, if $P(\texttt{xs})$ holds, then we also have $P(\texttt{x::xs})$!

It is best to be safe and explicit in your proofs by writing the case out explicitly, as well as what variables you are introducing for it.

[3] Where `x, xs, L` range over values such that `x : int`, `xs : int list`, and `L : int list`

We proceed by structural induction on `L : int list`.

BC Case: `L = []`

⟨ proof that $P($`[]`$)$ holds ⟩

IH Case: `L = xs`. Assume that the theorem holds for `xs`.

IS Case: `L = x::xs`. We would like to show that $P($`x::xs`$)$ holds.

⟨ proof of $P($`x::xs`$)$, under hypothesis that $P($`xs`$)$ holds ⟩

Thus, by the principle of structural induction, the theorem holds for all `L`.

3 - Tail Recursion

So now, after proving that `length` is total, we might be a little more assured about its behavior.

What do we say about its correctness, though? Can we prove that `length` is correct?

Usually, when trying to prove a function correct, we will start with a single, correct implementation (called the **reference implementation**), then attempt to prove that they are equivalent. We will take `length` as our reference implementation, on faith.

We can try it out on a few values, to see what happens (click me!)

Woah! Trying out `length` on a very long list ends up producing an extremely large trace. What gives?

Recall that the body of the `length` function is written as `1 + length xs`.

Because SML is an **eagerly evaluated** language, this means that both `1` and `length xs` must be independently evaluated to values, before they can be summed, in that order.

This means that we end up on a very large rabbit hole of computing `length L` on successively smaller lists `L`, before we ever get to add the first `1`!

We say that `length` is not **tail recursive**.

For recursive functions like `length`, we can write a version that, instead of making a recursive call and then doing some work with it, first does some work and then computes the answer by making a recursive call.

This sounds like a small distinction, but it makes a big difference! Such functions never have to **remember what they have to do next** after the recursive call, meaning they use less memory.

Def A function is **tail recursive** if it makes a singular recursive call as the *last thing that it does*, in the recursive case.

Let's look at some functions we already implemented:

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length xs
```

We already saw this was not tail recursive.

```
fun fact (0 : int) : int = 1
  | fact n = n * fact (n - 1)
```

Similarly, this one ends up not being tail recursive.

```
fun isEmpty ([] : int list) : bool = true
  | isEmpty (x :: xs) = false
```

Let's call this one "vacuously" tail recursive.

Most of the functions we've seen so far are not tail recursive! Let's write one.

Let's write a tail-recursive version of `length`!

We will use an idea which will come up multiple times this semester, of an **accumulator**.

Def An **accumulator** is an additional argument to a function, which is meant to store the final answer, carrying it forward into future recursive calls.

In this case, we will make `length` take in an `int` as an argument.

```
fun tlength ([] : int list, acc : int) : int = acc
  | tlength (x::xs, acc) = tlength (xs, 1 + acc)
```

```
fun tlength ([] : int list, acc : int) : int = acc
  | tlength (x::xs, acc) = tlength (xs, 1 + acc)
```

We see that in the recursive case, we first compute the sum of `1` and `acc`, and then make a tail-recursive call.

Then, we can define our original `length` function as simply:

```
fun length (L : int list) : int = tlength (L, 0)
```

`tlength` in Action

We can see our new `length` function put to the test!

4 - More List Functions

We've seen that we can use `::` to add a single element to the beginning of a list, but what about multiple?

For concatenating two lists together, we have the `@`[4] function, which can be defined as follows:

```
infix @

fun ([] : int list) @ (R : int list) : int list = R
  | (x::xs) @ R = x :: (xs @ R)
```

[4] Pronounced "append".

Sometimes we're interested in reversing a list.

The rev[5] function can be implemented as follows:

```
fun rev ([] : int list) : int list = []
  | rev (x::xs) = rev xs @ [x]
```

[5] Pronounced "rev".

`rev` is Not Tail Recursive

We will take both of these as our reference implementations for the `@` and `rev` functions.

There's a catch, though. While relatively simple to define, `rev` leaves something to be desired, because it makes a non tail-recursive call to itself!

```
fun rev ([] : int list) : int list = []
  | rev (x::xs) = rev xs @ [x]
```

Looking at the recursive call for `rev`, it makes a call to the `@` function. What's the time complexity of `@`?

```
infix @

fun ([] : int list) @ (R : int list) : int list = R
  | (x::xs) @ R = x :: (xs @ R)
```

From the definition of `@`, we see that it never inspects the second argument that it is given, namely the right list.

However, it deconstructs the first list by one element each time, and then adds it to the resulting recursive call. This ends up being a linear amount of operations, in the length of the first list.

This means that any operation like `L @ [x]` is very costly, because it is spending `length L` operations to add to a singleton list! In general, we call `L @ [x]` an **anti-pattern** that should hopefully be avoided, if possible.

The `rev` function uses this very anti-pattern. Can we avoid it, using an accumulator and tail recursion?

We will use an accumulator of type `int list` to achieve this:

```
fun trev ([] : int list, acc : int list) : int list = acc
  | trev (x::xs, acc) = trev (xs, x :: acc)

fun rev (L : int list) : int list = trev (L, [])
```

This `trev` function runs in less space, and as we will see in two lectures, less time!

Thank you!