Lesson 3
INDUCTION AND RECURSION

May 23, 2023

1. Recursion
2. Induction
3. Proving Correctness
4. More Language Features
5. Case Study: Fast Exponentiation

In the last lecture, we learned about some of the more complex types in SML, like tuples and functions.

We learned about how to declare variables and functions. We learned that declaring a variable **binds** it, which is different than assignment.

We also learned how we can mathematically prove things about SML code using a notion of **extensional equivalence**.

1 - Recursion

In the first lecture, we mentioned how we will be focusing on **recursion** in this class.

Def We call a function **recursive** if it calls itself. More generally, something which refers to itself is recursive.

Ex. This sentence is recursive.

SML doesn't have for loops, so recursion will be our preferred method for repeating some computation. We will find that, in conjunction with purity and binding, this will greatly help us reason about our code.

While recursion is a concept which is usually taught in programming classes, it's just as much a mathematical concept.

The factorial function is usually specified as:

$$\mathsf{fact}(n) = \left\{ \begin{array}{ll} 1, & \text{if } n = 0 \\ n * \mathsf{fact}(n-1), & \text{if } n > 0 \end{array} \right\} \tag{1}$$

which references the factorial function in its own definition. In other words, it's recursive.

We see that the SML code for the factorial function closely resembles the mathematical definition.

There is a simple four-step formula to writing any recursive function:

- Identify and write the **base case**,
- Identify the recursive case,
- Assume that the function **already works** on a "smaller" input,
- Write the recursive case under that assumption.

Def We say that the **base case** for a recursive function is the branch of the function which does not recurse.

Let's try to write out a different function using this formula.

`pow : int * int -> int`
REQUIRES: `k >= 0`
ENSURES: `pow (n, k)` evaluates to n^k

We know that raising any number to the power of 0 will produce 1, so our base case will simply be the case where the second value we receive is 0.

```
fun pow (n : int, 0 : int) : int = 1
  | (* ... *)
```

Since our base case is casing upon the value of the exponent, this is a good indication that the exponent is going to be the value that we recurse on. We call this the **variable of recurrence**.

What should we write for our recursive case? We want to be able to compute an arbitrary `pow(n, k)`, for non-zero `k`.

Thinking recursively, we want to be able to assume that our function already works on a smaller input. Since our variable of recurrence is `k`, then we can assume that `pow (n, k - 1)` already evaluates to the $(k-1)$th power of n.

Given n^{k-1}, we can recover n^k by just multiplying by n, so we get:

```
fun pow (n : int, 0 : int) : int = 1
  | pow (n, k) = n * pow (n, k - 1)
```

This is the magic of recursive thinking!

There is a simple four-step formula to writing any recursive function:

- Identify and write the **base case**,
- Identify the recursive case,
- Assume that the function **already works** on a "smaller" input,
- Write the recursive case under that assumption.

Def We say that the **base case** for a recursive function is the branch of the function which does not recurse.

Note But wait, this formula looks similar to something we've already seen... Let's give the perspective a switch...


RECURSION
15150
INDUCTION
15150

There is a simple four-step formula to proving any simple inductive theorem:

- Identify and write the **base case**,
- Identify the recursive case,
- Assume that the induction hypothesis holds,
- Prove the recursive case under that assumption.

Def We say that the **base case** for an inductive proof is the case of the proof which does not require an inductive hypothesis. But wait, this formula looks similar to something we've already seen...

Note Perfect.

2 - Induction

Let's take a brief excursion into mathematical induction.

Def The principle of mathematical **induction** (or **simple induction**) is a proof technique for theorems on the natural numbers. In particular, it has the logical form:

$$P(0) \land (\forall n.P(n) \implies P(n+1)) \implies \forall n.P(n)$$

In other words: "if you can prove $P(0)$, and that any step follows from the previous, then the statement is true for all numbers".

There is a simple four-step formula to proving any simple inductive theorem:

- Identify and write the **base case**,
- Identify the recursive case,
- Assume that the induction hypothesis holds,
- Prove the recursive case under that assumption.

Def We say that the **base case** for an inductive proof is the case(s) of the proof which do not require an inductive hypothesis.

Let S_n be the sum of the first n odd natural numbers.

Thm. Prove that S_n is n^2

We proceed by mathematical induction on n.

BC $n = 1$ Then, $1 = 1^2$.

IH Let k be arbitrary and fixed. Assume that $S_k = k^2$.

IS Then, we want to show that $S_{k+1} = (k+1)^2$.

We know the $k+1$th odd natural number is $2k+1$, so:

$$\begin{aligned} S_{k+1} &= S_k + 2k + 1 \\ &= k^2 + 2k + 1 \\ &= (k+1)^2 \end{aligned}$$

Therefore, by the principle of mathematical induction, the theorem holds for all natural numbers n.

We see that **recursion** and **induction** are really just two sides of the same coin.

Recursion is about **using the answers to sub-problems to solve a bigger one.**

Induction is about **using the inductive assumption to prove the $n+1$th case.**

The key thing to remember is the **recursive leap of faith**, which entails assuming that the function already works on a smaller input.

We have seen now that induction can be used as a tool which can help us to *write* recursive functions.

But not all recursive functions are obvious in their logic! We said earlier that we are not just interested in writing code, we are interested in writing *correct* code.

The best way to be certain of a function's correctness is to *prove* that it is correct. We will see now that induction helps us with recursion in another way, because we can prove that SML functions are correct using induction.

3 - Proving Correctness

Let's look at the `pow` function we just wrote.

```
fun pow (n : int, 0 : int) : int = 1
  | pow (n, k) = n * pow (n, k - 1)
```

Let's try to write an inductive proof of correctness.

When we do induction, we usually want a quantity to induct on, however. What should we pick for this function?

In this case, the **variable of recurrence** is usually a prime candidate for our induction!

Thm. Prove that `pow (n, k)` $\hookrightarrow n^k$, for all `k >= 0`

We proceed by mathematical induction on `k`.

BC `k = 0`

$$\texttt{pow(n, 0)} \Longrightarrow \texttt{1} \qquad \text{(clause 1 of } \texttt{pow}\text{)}$$

IH Assume that `pow (n, k)` $\hookrightarrow n^k$, for some `k`

IS Then, we want to show that `pow (n, k + 1)` $\hookrightarrow n^{k+1}$

$$\begin{aligned} \texttt{pow (n, k + 1)} &\Longrightarrow \texttt{n * pow (n, k)} && \text{(clause 2 of } \texttt{pow}\text{)} \\ &\Longrightarrow \texttt{n * } n^k && \text{(induction hypothesis)} \\ &\Longrightarrow n^{k+1} && \text{(math)} \end{aligned}$$

We see that in an inductive proof on a recursive function, the isomorphism between recursion and induction is made even more clear.

Program	Proof
Base case (`pow (n, 0)`)	**Base case** (n^0)
Recursive call (`pow (n, k)`)	**Inductive hypothesis** (n^k)
Variable of recurrence (`k`)	**Induction variable** (k)

4 - More Language Features

An alternative to matching on an input in a function clause is to use a *case expression*.

```
case <expr> of
  at1> => <expr1>
| at2> => <expr2>
...
| atn> => <exprn>
```

Note The first "arm" of a case expression has no bar!

Note `case` expressions follow similar typing rules as function clauses, where each case's expression must share the same type.

So for instance, we could rewrite the `fact` function as:

```
fun fact (n : int) : int =
  case n of
    0 => 1
  | _ => n * fact (n - 1)
```

Note We could have written this with `n` instead of the wildcard pattern, which would have shadowed the original `n` with a new binding with exactly the same value.

It's still an expression, though, so it's totally OK to write something like

```
(case 2 of
  2 => 3
| _ => 5) + 1
```

To write more interesting functions, we will also introduce **lists**.

Def For any type `t`, there is a type `t list`, which describes an ordered collection of 0 or more elements of type `t`.

Here are some examples of lists:

- `[1, 2, 3] : int list`
- `["hi", "there"] : string list`
- `[] : int list`
- `[] : bool list`
- `[[1, 2]] : int list list`

What if we want to add to an existing list, though? We can also construct lists out of other lists, using a **constructor**!

A list is characterized by two **constructors**, which are used to construct values of some list type. These constructors are

- `[]`[1], which is the empty list of type `t list`
- `::` (pronounced "cons"), an infix operator such that `x :: xs : t list` if `x : t` and `xs : t list`.

While the bracket notation is simple, it's actually just syntactic sugar! The list `[1, 2, 3]` is really just `1::2::3::[]`

Note The `::` operator is *right-associative*, meaning that `1::2::3::[]` is implicitly understood to be `1::(2::(3::[]))`

[1] If you're viewing this presentation online, after the lecture, it's important to me that you know this is pronounced "nil". Hard to convey digitally.

Constructors are patterns too!

This means that we can `case` upon a list to figure out what kind of **variant** it is. For instance, we can write a function to check whether a list is empty as:

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

We might say that these constructors are all that characterize a list. A list *must be* either empty (`[]`) or have a first element, and the rest of the list (`x::xs`). Pattern matching just lets us handle either case.

Let's apply the recursion formula on a simple length function on lists.

The base case must be the empty list `[]`, in which case we just return `0`.

In the recursive case `x::xs`, we also assume that `length` works on a smaller input.

In this case, `xs` happens to be exactly a smaller input than the entire list `x::xs`, so we assume `length xs` works, and we get:

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

5 - Case Study: Fast Exponentiation

Suppose that we are interested in solving a slightly more involved problem.

The `pow` function we just implemented is fine, but it's not necessarily as fast as it could be! From eyeing the function, we can see that it would take us k many multiplications to compute the kth power of n. This is more than it needs to be, because of a nice fact that we can take advantage of:

If k is odd, then

$$n^k = n * \left(n^{\lfloor k/2 \rfloor}\right)^2$$

If k is even, then

$$n^k = \left(n^{k/2}\right)^2$$

Can we turn this mathematical definition into a program?

Let's try it out!

```
fun fast_pow (n : int, 0 : int) : int = 1
  | fast_pow (n, k) =
      if k mod 2 = 0 then
        fast_pow (n, k div 2) * fast_pow (n, k div 2)
      else
        n * fast_pow (n, k - 1)
```

How's this look?[2]

[2] I actually adjusted the definition slightly from the mathematical definition, because it makes the proof easier. It can be done either way, though.

There's a problem with this function, however. It's not actually necessarily saving us that many multiplications, because we make two recursive calls!

We wanted to save on effort by reusing our answers from the recursive call, but we see here that we are just recomputing each time!

`fast_pow` is a pure function, meaning that we should be able to do some extensionally equivalent refactoring, using the power of referential transparency...

To do this, however, we need to be able to bind a variable within an expression. How shall we do this?

The syntax for binding a variable for use within another expression is as follows:

```
let
  <declarations>
in
  <expr>
end
```

where the declarations are much the same as the `val` and `fun` declarations that we saw earlier. This means there can be multiple of them!

Note `let` expressions are expressions in their own right, so it is valid to write

```
(let
  val x = 2
in
  x + x
end) + 3
```

SML is a **lexically scoped** language, which means that the scope in which variable bindings are visible depends on its context in the text of the program.

This means, for instance, that anything declared within a `let` expression is only potentially visible within the expression included with it. This means, for instance, that this is invalid code:

```
(* INVALID CODE! What is `y`? *)
val x =
  (let
    val y = 3
  in
    4
  end) + y
```

So let's modify our original `fast_pow`:

```
fun fast_pow (n : int, 0 : int) : int = 1
  | fast_pow (n, k) =
      if k mod 2 = 0 then
        let
          val half_ans = fast_pow (n, k div 2)
        in
          half_ans * half_ans
        end
      else
        n * fast_pow (n, k - 1)
```

Now, we only ever have at maximum one recursive call to `fast_pow`!

How do we know that behavior is preserved, however? We want to be able to prove this.

However, we see that the recursive call is to `fast_pow (n, k div 2)`.

This is not as straightforward as what we usually saw, with the recursive call being on one less than the current value.

If the recursive call corresponds to our inductive hypothesis, then will a change to our recursive call change what our inductive hypothesis should be?

Note Yes.

Since we need to know something about $k/2$ rather than $k - 1$, we will proceed by **strong induction** instead of **simple induction**.

Def **Strong induction** is a variant of inductive proof where instead of assuming the inductive hypothesis for just $n-1$, the hypothesis is assumed for all numbers between 0 and n.

You can think of it as, if we have gone to all the trouble of proving our theorem incrementally by adding larger and larger numbers to our collection, we should still be able to make use of all of the intermediary theorems we proved (i.e. $P(0), P(1), ..., P(n)$), not just the last one.

Now, let's do the proof.

Thm. Prove that `fast_pow` $\cong$ `pow`

We proceed by strong induction on k.

BC $k = 0$

$$\texttt{fast\_pow (n, 0)} \cong 1 \qquad \text{(clause 1 of } \texttt{fast\_pow}\text{)}$$
$$\cong \texttt{pow (n, 0)} \qquad \text{(clause 1 of } \texttt{pow}\text{)}$$

IH Assume `fast_pow (n, k)` $\cong$ `pow (n, i)`, for all `0 <= i < k`, for some `k`

IS Then, we want to show that `fast_pow (n, k)` $\cong$ `pow (n, k)` But what do we do in this case?

Here, we have a dilemma, because we have two cases, the case where `k` is even, and the case where `k` is odd.

The way we deal with this in a proof is that we must prove that the theorem holds *no matter which case we are in*.

So we must prove the theorem for both cases. To do this, we will make use of a **lemma**:

Lemma 1 When `k mod 2 = 0`, then

```
pow (n, k div 2) * pow (n, k div 2) ≅ pow (n, k)
```

IS Then, we want to show that `fast_pow (n, k)` ≅ `pow (n, k)`

Case 1: `k mod 2 = 0`

`fast_pow (n, k)` ≅ `half_ans * half_ans` (clause 2 of `fast_pow`)

≅ `fast_pow (n, k div 2) * fast_pow (n, k div 2)` (def of `half_ans`)

≅ `pow (n, k div 2) * pow (n, k div 2)` (induction hypothesis)

≅ `pow (n, k)` (lemma 1, case assumption)

Case 2: `k mod 2 <> 0`

$$
\begin{aligned}
\texttt{fast\_pow (n, k)} &\cong \texttt{n * fast\_pow (n, k - 1)} && \text{(clause 2 of } \texttt{fast\_pow}\text{)} \\
&\cong \texttt{n * pow (n, k - 1)} && \text{(induction hypothesis)} \\
&\cong \texttt{pow (n, k)} && \text{(clause 2 of } \texttt{pow}\text{)}
\end{aligned}
$$

Now, we have finished proving our theorem!

Program	Proof
Base case (`pow (n, 0)`)	**Base case** (n^0)
Recursive call (`pow (n, k)`)	**Inductive hypothesis** (n^k)
Variable of recurrence (`k`)	**Induction variable** (k)
Simple recursive call (`k - 1`)	**Simple induction**
Complex recursive call (`k div 2`)	**Strong induction**
Branching behavior (`if`)	**Proof casing**

Thank you!