15-150 Fall 2012
Lecture 4
Recursion and induction (part 2)

Stephen Brookes

Thursday, September 5

1 Overview

In the previous lecture we introduced "simple" or "mathematical" induction, and progressed to "complete" (or "strong") induction. These methods are designed to prove statements that can be expressed in the form "For all non-negative integers n, some property holds". Now we will illustrate how to use inductive techniques to reason about functions that manipulate or return lists. For simplicity, we choose examples involving lists of integers. We will see later that all of these ideas and techniques work just as well for lists built from other types of value. And we will later introduce a more general formulation of *structural induction* applicable to more general datatypes in ML, and this will be a nice generalization of the list techniques in this chapter.

2 Lists

Suppose we want to prove:

> For all integer lists L, $P(L)$ holds.

Here $P(L)$ is some property of lists.

How might we go about this? The key is to realize that every list value is *constructible* from the empty list using a finite number of "cons" operations. For example, the list `[1,2,3]` is expressible as `1::(2::(3::nil))`. Actually, since the ML cons operator is right-associative, we can suppress the parentheses and say `[1,2,3] = 1::2::3::nil`. And every integer list value is either empty (equal to `nil`), or non-empty and of the form `x::R` for some integer value `x` and integer list value `R`.

List induction is the following proof strategy:

(a) Show that $P($`nil`$)$ holds.

(b) Let `L=x::R` be an arbitrary integer list value, and assume as induction hypothesis that $P($`R`$)$ holds. Show that $P($`L`$)$ also holds, under these assumptions.

We call `nil` the "base case" here; we also refer to `x::R` as the "inductive case" here, and $P($`R`$)$ as the "induction hypothesis".

If you can prove (a) and (b), it follows that for all integer lists L, $P(L)$ holds. This is because (a) tells us that $P($`nil`$)$ holds; so by (b) we get

that $P(\texttt{x} :: \texttt{nil})$ holds, for all integer values `x`; by (b) again we get that $P(\texttt{y} :: \texttt{x} :: \texttt{nil})$ holds for all integers `x` and `y`; and so on. For each non-negative integer k, after k steps of this proof process we get that $P(L)$ holds *for all integer lists L with k elements.* (This paragraph is not part of the proof strategy, but should serve to convince you why the strategy works!)

Depending on your choice of property P, this simple form of list induction may or may not work as a proof strategy. Once you get familiar with using this technique you will learn to identify which kinds of problems are susceptible to this strategy.

An example using list induction

Consider the ML function `length:int list -> int` defined by:

```
fun length (L:int list):int =
   case L of
         [ ]  => 0
     | (_::R) => 1 + length R;
```

Your task: prove that for all integer list values `L`, `length(L)` evaluates to a non-negative integer. (Informally, "every list value has a finite length.")

(i) State the theorem to be proven:

> (THM): For all `L` : `int list`, `length`(`L`) evaluates to a non-negative integer.

Let us write P(`L`) for this property, i.e. "`length(L)` evaluates to a non-negative integer".

(ii) State what the proof method is going to be:

> Proof: By list induction on `L`.

(iii) State and prove the base case:

> Base case: `L` = `nil`.
> Need to show: P(`nil`), i.e. `length(nil)` terminates.
> Proof: This is obvious from the function definition, since `length(nil) =>* 0`, and `0` is a non-negative integer.

(iv) State the inductive case and the induction hypothesis, then prove the inductive step:

> Inductive case: `L=x::R`.
> Induction hypothesis: P(`R`), "there is a non-negative integer `n` such that `length R =>* n`".
> Inductive step: We need to show, with these assumptions, P(`x` :: `R`) holds, i.e. that `length(x::R)` evaluates to a non-negative integer.
> Proof: Again this is easy.
>
> ```
> length(x::R)
>    =>* 1 + length(R)
>    =>* 1 + n     for some integer value n>=0, by P(R)
>    =>* v,        where v is the numeral for n+1
> ```
>
> Since we assumed `n>=0`, we also have `n+1>=0`. So `P(x::R)` holds, as required.

(v) By (iii) and (iv) it follows that (THM) holds.

Although this example is rather obvious, it does mean we have proven the fundamental fact that every ML list value is "finite", i.e. has a finite length, where length is defined as above. This notion of length coincides with the number of items in the list. Also notice that for all `L`, `length`(`L`) ≥ 0; and `length`(`x` :: `R`) $>$ `length`(`R`). So it should be fairly obvious that every theorem that has a proof using *list induction* also has a proof by *mathematical induction on the length of lists*.

Simple list functions

List induction can be useful for reasoning about recursive functions with a simple structural property. For example, suppose we have a function `f` of type `int list -> int`, and we want to prove that for all integer lists `L`, `f(L)` evaluates to an integer value, say `v`, with a certain property, say `Q(v)`. For example, if `Q(v)` is trivial (i.e. `true`), we're trying to prove that `f(n)` terminates. Or `Q(v)` may say that the value of `v` is equal to some quantity that depends on the value of `n`.

Suppose that the function definition for `f` has a clause for `f(nil)` giving the result directly, and a clause for `f(x::R)` that makes a recursive call to

`f(R)`. We can try using *list induction* to prove that for all integer lists `L`, `P(L)` holds, where we let `P(L)` be the property: `f(L)` evaluates to an integer value `v` such that `Q(v)` holds.

A proof of this result, using list induction on `L`, has the following form:

(i) State the theorem to be proven:

> (THM): For all `L : int list`, `f(L)` evaluates to a value `v` such that `Q(v)` holds.

(ii) State what the proof method is going to be:

> Proof: By list induction on `L`.

(iii) State and prove the base case:

> Base case: `L=nil`.
> To show: `P(nil)`, i.e. `f(nil)` evaluates to a value v_0 such that `Q(`v_0`)` holds.
> Proof: (Use the function definition for this!)

(iv) State the inductive case and the induction hypothesis, then prove the inductive step:

> Inductive case: `L=x::R`.
> Induction Hypothesis: `P(R)`, i.e.
> `f(R)` evaluates to a value v' such that `Q(`v'`)` holds.
> To show: `P(L)`, i.e. `f(x::R)` evaluates to a value `v` such that `Q(v)` holds.
> Proof: show this, using `P(R)`, the assumption that `n>0`, and the function definition.

(v) By (iii) and (iv) it follows that (THM) holds.

Comment: we deliberately used different "meta-variables" `v`, v_0, v' in the different subtasks above, to avoid any confusion; if we had written the same `v` everywhere you might have gotten mixed up. You don't have to label the theorem or the induction hypothesis, but doing so makes it easy to refer to them in your proof.

In practice, there may be more than one reasonable way to write down a theorem statement, or to state an induction hypothesis. For example, it may be convenient to write an equation like `f(L)=v` instead of saying "`f(L)` evaluates to `v`". This is especially common when we are trying to prove that `f(L)` evaluates to a specific value (that is expressible explicitly and usually

depends on L). Recall that since the return type of f here is int, saying that f(L) evaluates to v means the same as saying that "the value of f(L) is v", or just f(L) = v.

And if the function definition is not as simple as the assumptions we made above about f, for example if f(L) makes more than one recursive call, or has more than one explicit "base" clauses in which no recursive calls are needed, we'll need to adapt the proof strategy and use a more sophisticated form of induction (coming soon!).

A simple list example

```
(* sum : int list -> int *)
fun sum (L:int list):int =
  case L of
       [ ] => 0
    | (x::R) => x + sum(R)
```

Your task: Prove that for all L, sum L returns the sum of the integers in L.

Notice that the definition of sum gives the value sum(nil) directly, without making any recursive calls; so L=nil is a "base" clause. Otherwise it is clear that sum(x::R) makes a recursive call to sum(R). So this function definition fits the pattern above and we can adapt the proof template from above for list induction.

(i) The theorem to be proven:

(SUM): For all L, sum(L) returns the sum of the integers in L.

(ii) Proof: By list induction on L.

(iii) Base case: L=nil.
To show: sum(nil) = 0.
Proof: This is clear from the function definition and the fact that there are no integers in the empty list, so its sum is 0.

(iv) Inductive case: L = x :: R.
Induction Hypothesis:

(IH): sum R returns the sum of the integers in R.

To show: with these assumptions, `sum(x :: R)` returns the sum of the integers in `x::R`.

Proof: Using the function definition for `sum`, plus obvious facts about addition, and the induction hypothesis (IH), we have

```
sum (x::R)
  = x + sum R  by definition of sum
  = x + k      where k is the sum of the integers in R,
               by IH
  = the sum of the integers in x::R
```

as required.

(v) By (iii) and (iv) it follows that (SUM) holds.

Incrementing along a list

Consider the function

```
addtoeach : int * int list -> int list
```

defined by

```
fun addtoeach (x:int, L:int list) =
   case L of
         [ ]  => [ ]
     | (y::R) => (x+y)::addtoeach(x,R);
```

Intuitively, we might say that `addtoeach(x, L)` returns a list obtained by incrementing each item in `L` by `x`. Even though the word "increment" is sometimes used with imperative connotations, we are *not* talking about an imperative or destructive operation here – we are doing functional programming. No list is being destroyed or modified. Rather, a new list is constructed and returned by the function call.

Prove that for all integers `x` and integer lists `L`,

`sum(addtoeach(x, L)) = sum L + x * length L.`

Prove that for all integers `x` and `y`, and all integer lists `L`,

`addtoeach(x, addtoeach(y, L)) = addtoeach(x+y, L).`

Inserting into a sorted list

We all know what it means to say that a list of integers is *sorted* (into non-decreasing order). For example, `[1,1,2,4,4,10]` is a sorted list, but `[3,2,1]` is not. Technically, a list is sorted if each item is less-than-or-equal-to the items that appear later in the list. Note the key facts that the empty list is sorted; a singleton list `[x]` is sorted; and if `y::L` is sorted and $\texttt{x} \leq \texttt{y}$, then `x::(y::L)` is sorted.

Let's assume we know what it means to say that one list is a *permutation* of another. For example, `[1,3,2]` is a permutation of `[3,2,1]`, but not of `[1,3,3,2]`. Technically a permutation of a list is another list that contains the same items possibly in a different arrangement.

Consider the function `ins:int*int list -> int list` defined by:

```
fun ins (x, [ ]) = [x]
 |  ins (x, y::R) = if x>y then y::ins(x, R) else x::y::R;
```

We show first that for all integers `x` and all lists `L`, `ins(x, L)` evaluates to a permutation of `x::L`.

(i) The theorem to be proven:

> For all integer lists `L`, and all integers `x`,
> `ins`(`x`, `L`) evaluates to a permutation of `x::L`.

(ii) Proof: By list induction on `L`.

(iii) Base case: `L=nil`.
To show: for all integers `x`, `ins`(`x`, `nil`) evaluates to a permutation of `x::nil`.
Proof: This is clear from the function definition, because
`ins(x, nil) =>* [x]` and this is obviously a permutation of `x::nil` (remember that `x::nil = [x]`).

(iv) Inductive case: `L` = `y` :: `R`.
Induction Hypothesis (IH):

> For all integers `x'`,
> `ins(x', R)` evaluates to a permutation of `x'::R`.

To show: with these assumptions, for all integers `x`, `ins(x, y::R)` evaluates to a permutation of `x::(y::R)`.

Proof: Using the function definition for `ins`, and (IH), we have

```
ins (x, y::R)
  = if x>y then y::ins(x,R) else x::(y::R)
```

Now we do a case analysis.

- Either `x>y`, in which case `ins(x,y::R)=y::ins(x,R)`. By IH, `ins(x,R)` evaluates to a permutation of `x::R`, say `P`. So `ins(x,y::R)` evaluates to `y::P`, which is a permutation of `y::(x::R)`. This is also a permutation of `x::(y::R)`, as required.
- Or $\texttt{x} \leq \texttt{y}$, in which case `ins(x, y::R) = x::(y::R)` and this is clearly a permutation of `x::(y::R)`, as required.

We show next, *using the now established result that* `ins(x,L)` *returns a permutation of* `x::L`, that for all integers `x` and all sorted lists `L`, `ins(x, L)` evaluates to a sorted list.

(i) The theorem to be proven:

For all sorted integer lists `L`, for all integers `x`, `ins(x,L)` returns a sorted list.

(ii) Proof: By list induction on `L`.

(iii) Base case: `L=nil`.
To show: for all integers `x`, `ins(x,nil)` evaluates to a sorted list.
Proof: This is clear from the function definition, because
`ins(x, nil) =>* [x]` and this is a sorted list.

(iv) Inductive case: $\texttt{L} = \texttt{y} :: \texttt{R}$, a sorted list. Note that this implies that `R` is also a sorted list, and that `y` is less-than-or-equal-to every integer in `R`.
Induction Hypothesis:

(IH): For all integers `x`, `ins(x, R)` returns a sorted list.

To show: with these assumptions, for all `x`, `ins(x, y::R)` returns a sorted list.

Proof: Using the function definition for `ins`, and (IH), we have

```
ins (x, y::R)
  = if x>y then y::ins(x,R) else x::y::R
```

Now we do a case analysis.

- Either `x>y`, in which case `ins(x,y::R)=y::ins(x,R)`. By IH, `ins(x,R)` evaluates to a sorted list, say `S`. By the previous proof, `S` is a permutation of `x::R`. Since `x>y` and `y` is less-than-or-equal-to every integer in `R`, we see that `y::S` is sorted. So we have shown that `ins(x,y::R)` evaluates to `y::S`, a sorted list.
- Or $\texttt{x} \leq \texttt{y}$, in which case `ins(x, y::R)=x::(y::R)`. And this list is sorted, because (by assumption) `y::R` is sorted and `x` is less-than-or-equal-to `y`, which is $\leq$ every integer in `R`. (We're appealing here to the fact that the $\leq$ ordering on integers is *transitive*, i.e. if $\texttt{x} \leq \texttt{y}$ and $\texttt{y} \leq \texttt{z}$, then $\texttt{x} \leq \texttt{z}$).

It should now be an obvious consequence of these two results that for all integers `x` and all sorted lists `L`, `ins(x, L)` evaluates to a sorted permutation of `L`.

(Actually for an integer list `L` there is exactly one permutation of `L` that is sorted, so it's not just "a" sorted permutation of `L`, but "the".)

3 Exercises

1. What goes wrong if you try to prove that inserting into a sorted list always produces a sorted list, but without using the permutation property? Again we refer to the `ins` function from above.

2. Write the combined specification for `ins` (about producing a sorted permutation) using the REQUIRES and ENSURES format.

3. For each of the following specifications, which (if any) are satisfied by the `ins` function? No need to prove each case, but in any case where you think the answer is NO please explain briefly.

(i)
```
(* REQUIRES true *)
(* ENSURES ins(x, L) returns a list value S                    *)
(* such that there are lists A and B for which S = A@(x::B).  *)
```

(ii)
```
(* REQUIRES true *)
(* ENSURES ins(x, L) returns a list value S                    *)
(* such that there are lists A and B for which S = A@(x::B)   *)
(* and every item in A is less than x.                         *)
```

(iii)
```
(* REQUIRES true *)
(* ENSURES ins(x, L) returns a list value that contains x. *)
```

(iv)
```
(* REQUIRES L is sorted *)
(* ENSURES ins(x, L) returns a list value S                    *)
(* such that there are lists A and B for which S = A@(x::B).  *)
```

4. Consider the function

```
append:int list * int list -> int list
```

defined by

```
fun append(A:int list, B:int list):int list =
  case A of
       [ ]    => B
   |  (x::A') => x:: append(A', B);
```

Prove that for all integer lists `A` and `B`, there is a list `L` such that

```
append(A, B) =>* L and length(L) = length(A)+length(B).
```

Using equational reasoning, show that for all integer lists `A` and `B`,

```
length(append(A, B)) = length(A) + length(B).
```

5. The built-in infix append operator of ML is `@`. For all integers `x` and all integer lists `L` and `R`, the following equations hold:

```
   [ ]@R = R
(x::L)@R = x::(L@R)
```

(Remember that an equation like this means that the two expressions evaluate to the same value.) Using these facts, prove that for all integer lists `A` and `B`, `append(A, B) = A@B`.

6. Consider the function `foo : int list -> int list` given by

```
fun foo (L:int list) : int list =
   case L of
        [ ]   => [ ]
    |  (x::R) => x :: foo(rev R);
```

Assume that `rev` is a given function of type `int list -> int list` such that for all integer lists `A`, `rev(A)` evaluates to the reverse of list `A`. The reverse of `A` has the same length as `A`.

You cannot use structural induction on lists to prove that for all integer lists `L`, `foo(L)` terminates, because `foo(x::L)` makes a recursive call to `foo(rev L)`, and the value of `rev L` is *not* the tail of the original list! Nevertheless we do know that the value of `rev L` is a list with shorter length than the length of `x::L`. So you should be able to use induction on list length.

Use induction *on the length of lists* to prove that for all integer lists `L`, `foo(L)` terminates.

Use induction to prove that for all integer lists `L`, `foo(L)` evaluates to a permutation of `L`. (You can assume here, without proof, that for all integer lists `A`, `rev A` evaluates to a permutation of `A`.)

7. Consider the function `upto:int * int -> int list` given by

```
fun upto(i:int, j:int):int list = if i>j then [ ] else i::upto(i+1,j);
```

Show that for all integer values `i` and `j`, `upto(i,j)` terminates.
(Hint: it's obvious when `i>j`. For $\texttt{i} \leq \texttt{j}$ use induction on the value of `j-i`.)

Prove that for all integers `i` and `j` such that $\texttt{i} \leq \texttt{j}$, `length(upto(i,j)) = (j-i)+1`.