Proving Equality of Functional Programs

Lecture Notes Functional Programming SS2007

Stefan Blom

Date: 2007/05/09 20:31:32
Revision: 1.3

Contents

1 Structural Induction **1**

2 Tail Recursion **5**

3 Exercises **7**

1 Structural Induction

In OCaml, the append on two native lists is written as the infix function @. The definition of @ is

```
let rec (@) l y = match l with
  | [] -> y
  | a::x -> a::(x@y)
;;
```

This definition can be translated to Haskell style as follows

$$\begin{aligned} [] \mathbin{@} y &= y \\ (a :: x) \mathbin{@} y &= a :: (x \mathbin{@} y) \end{aligned} \tag{1}$$

As an example, we will prove associativity of @. That is, for all finite lists x, y and z we have that

$$(x \mathbin{@} y) \mathbin{@} z = x \mathbin{@} (y \mathbin{@} z)$$

To do this, we need structural induction on lists. If we want to prove a property ϕ for all lists l by structural induction on l then we need to prove two things. First, we must prove that ϕ holds for the empty list. Second, we must show that if ϕ holds for a list x then ϕ holds for any list $a :: x$. In other words, we have a proof that fits the pattern

> We prove ϕ by structural induction on l.
> We have two cases:
>
> $l = []$ "*proof of* $\phi[l := x]$"

$l = a :: x$ Let a be any element and let x be any list.
Assume that $\phi[l := x]$.
"*proof of* $\phi[l := a :: x]$"

q.e.d.

This text corresponds to the formal proof rule

$$\frac{\phi[l := []] \qquad \forall a : \alpha . \forall x : \alpha \text{ list}.(\phi[l := x]) \rightarrow (\phi[l := a :: x])}{\forall l : \alpha \text{ list}.\phi} \; a, x \text{ fresh}$$

The restriction that a and x are fresh means that a and x should not occur in ϕ or any of our assumptions. (Note that technically it suffices that a and x do not occur free in any of the assumptions used.)

Using structural induction, we prove that the append function on lists is associative. That is, we need to prove that for all lists x, y and z we have that

$$(x \mathbin{@} y) \mathbin{@} z = x \mathbin{@} (y \mathbin{@} z)$$

Proof. By structural induction on x.
We have two cases:

$x = []$ We must prove that $([] \mathbin{@} y) \mathbin{@} z = [] \mathbin{@} (y \mathbin{@} z)$.
By applying Def. 1 once and simplifying, we get

$$([] \mathbin{@} y) \mathbin{@} z = (y) \mathbin{@} z = y \mathbin{@} z$$

and also

$$[] \mathbin{@} (y \mathbin{@} z) = (y \mathbin{@} z) = y \mathbin{@} z$$

$x = a :: u$ Given an element a and a list u, such that $(u \mathbin{@} y) \mathbin{@} z = u \mathbin{@} (y \mathbin{@} z)$ we must prove that $((a :: u) \mathbin{@} y) \mathbin{@} z = (a :: u) \mathbin{@} (y \mathbin{@} z)$.
By applying Def. 1 twice, we get

$$((a :: u) \mathbin{@} y) \mathbin{@} z = (a :: (u \mathbin{@} y)) \mathbin{@} z = a :: ((u \mathbin{@} y) \mathbin{@} z)$$

By applying Def. 1 once, we get

$$(a :: u) \mathbin{@} (y \mathbin{@} z) = a :: (u \mathbin{@} (y \mathbin{@} z))$$

The induction hypothesis is that $(u \mathbin{@} y) \mathbin{@} z = u \mathbin{@} (y \mathbin{@} z)$. Thus, by induction hypothesis we have

$$a :: ((u \mathbin{@} y) \mathbin{@} z) = a :: (u \mathbin{@} (y \mathbin{@} z))$$

□

Our second example involves two function that both compute the length of a list.

```
let rec len1 l = match l with
  | [] -> 0
  | a::x -> 1 + (len1 x)
;;
let rec lena a l = match l with
  | [] -> a
  | x::xs -> lena (a+1) xs
;;
let len2 = lena 0;;
```

We want to prove that for all lists l, we have that

$$len_1\ l = len_2\ l$$

The shortest proof is to first prove a claim and then prove the statement. **Proof.** We claim the lemma

$$\forall n:\ n + (len_1\ l) = (len_a\ n\ l)$$

We prove the claim by structural induction on l. We have two cases:

$l = []$ By unfolding the definition of len_1 we get

$$n + (len_1\ []) = n + 0 = n$$

By unfolding the definition of len_a we get

$$len_a\ n\ [] = n$$

$l = x :: xs$ Assume that $\forall n:\ n + (len_1\ xs) = (len_a\ n\ xs)$ and let k be given.
By unfolding the definition of len_1 and associativity of $+$ we get

$$k + (len_1\ (a :: xs)) = k + (1 + (len_1\ xs)) = (k + 1) + (len_1\ xs)$$

By unfolding the definition of len_a we get

$$len_a\ k\ (a :: xs) = len_a\ (k + 1)\ xs$$

By induction hypothesis $(k + 1) + (len_1\ xs) = len_a\ (k + 1)\ xs$, so

$$\forall k:\ k + (len_1\ (a :: xs)) = len_a\ k\ (a :: xs)$$

This proves the claim.
From the claim we get

$$len_1\ l = 0 + (len_1\ l) = len_a\ 0\ l = len_2\ l$$

□

This short proof depends on being able to correctly guess the lemma. Usually, guessing the lemma right requires attempting a proof and getting stuck in it. So again we claim

$$\forall l:\ len_1\ l = len_2\ l$$

Proof. By structural induction on l. We have two cases:

$l = []$ Evaluating the left and right-hand sides yields

$$len_1\ [] = 0$$

and

$$len_2\ [] = len_a\ 0\ [] = 0$$

$l = x :: xs$ Assuming that $len_1\ xs = len_2\ xs$, we need to show that

$$len_1\ (x :: xs) = len_2\ (x :: xs)$$

Starting from both sides, we can derive

$$\begin{array}{rcll} len_1\ (x :: xs) & = & 1 + (len_1\ xs) & \text{definition of } len_1 \\ & = & 1 + (len_2\ xs) & \text{induction hypothesis} \\ & = & 1 + (len_a\ 0\ xs) & \text{definition of } len_2 \end{array}$$

$$\begin{array}{rcll} len_2\ (x :: xs) & = & len_a\ 0\ (x :: xs) & \text{definition of } len_2 \\ & = & len_a\ 1\ xs & \text{definition of } len_a \end{array}$$

At this point the proof is stuck: $1 + (len_a\ 0\ xs)$ and $len_a\ 1\ xs$ are not the same. There are two possibilities to continue. First, we might be able to see now what the correct lemma is and start a new proof from scratch. Second, we might try to make a claim that allows us to finish the induction proof and then see about proving the claim later. The first try for such a claim would be $1 + (len_a\ 0\ xs) = len_a\ 1\ xs$, but that is too simple. So instead of 0 and 1, we use n and $n + 1$.

We claim that

$$\forall l :\ \forall n :\ 1 + (len_a\ n\ l) = len_a\ (n + 1)\ l$$

Which shows that

$$len_1\ (x :: xs) = len_2\ (x :: xs)$$

This completes the proof by structural induction, but we still need to prove the claim.
We prove the claim by structural induction on l:

$l = []$

$$1 + (len_a\ n\ []) = 1 + n = n + 1$$

$$len_a\ (n + 1)\ l = n + 1$$

$l = x :: xs$ Assuming that $\forall n :\ 1 + (len_a\ n\ xs) = len_a\ (n + 1)\ xs$, we have

$$1 + (len_a\ k\ (x :: xs)) = 1 + (len_a\ (k + 1)\ xs) = (len_a\ (k + 2)\ xs)$$

$$(len_a\ (k + 1)\ (x :: xs)) = (len_a\ (k + 2)\ xs)$$

□

2 Tail Recursion

Consider the two versions of length of lists:

```
let rec len1 l = match l with
  | [] -> 0
  | a::x -> 1 + (len1 x)
;;
let rec lena a l = match l with
  | [] -> a
  | x::xs -> lena (a+1) xs
;;
let len2 = lena 0;;
```

If we execute these functions on the list $[1;2;3;4]$ we get

$$
\begin{aligned}
len_1\ [1;2;3;4] &= 1 + (len_1\ [2;3;4]) \\
&= 1 + (1 + (len_1\ [3;4])) \\
&= 1 + (1 + (1 + (len_1\ [4]))) \\
&= 1 + (1 + (1 + (1 + (len_1\ [])))) \\
&= 1 + (1 + (1 + (1 + 0))) \\
&= 1 + (1 + (1 + 1)) \\
&= 1 + (1 + 2) \\
&= 1 + 3 \\
&= 4
\end{aligned}
\qquad
\begin{aligned}
len_2\ [1;2;3;4] &= len_a\ 0\ [1;2;3;4] \\
&= len_a\ (0 + 1)\ [2;3;4] \\
&= len_a\ 1\ [2;3;4] \\
&= len_a\ (1 + 1)\ [3;4] \\
&= len_a\ 2\ [3;4] \\
&= len_a\ (2 + 1)\ [4] \\
&= len_a\ 3\ [4] \\
&= len_a\ (3 + 1)\ [] \\
&= len_a\ 4\ [] \\
&= 4
\end{aligned}
$$

These sequences have the same length: $\mathcal{O}(|l|)$ where l is a list and $|l|$ is the length of l. Thus their execution time is similar. However, the first version uses $\mathcal{O}(|l|)$ stack space and the second version uses $\mathcal{O}(|1|)$ stack space. This makes the second version preferable.

The trick used to make the first version tail-recursive is to use an *accumulator*. Normally a recursive function will first copy a list to the stack and then build a new list. A function that uses and accumulator builds the result in the accumulator while it is working through the list.

Another good example is the function *rev* that reverse the order of a list. A direct recursive definition of *rev* would be

```
let rec revd l = match l with
  | [] -> []
  | a::x -> (revd x)@[a]
;;
```

This is a bad way of defining *rev*: the time complexity is $\mathcal{O}(|l|^2)$ and the stack space used is $\mathcal{O}(|l|)$. It is much better to define *rev* using an auxiliary function with an accumulator:

```
let rec reva a l = match l with
  | [] -> a
  | x::xs -> reva (x::a) xs
;;
let rev l = reva [] l ;;
```

this version uses $\mathcal{O}(|l|)$ time and $\mathcal{O}(|1|)$ stack space.

The function *map*, whose standard definition is

```
let rec map f l = match l with
  | [] -> []
  | x::xs -> (f x)::(map f xs)
;;
```

is not easy to write with an accumulator. To actually write an efficient version of the map function, one needs to define *maprev* which maps a function over a list and reverses the order of the list at the same time using an accumulator:

```
let rec mapacc f a l = match l with
  | [] -> a
  | x::xs -> mapacc f ((f x)::a) xs
;;
let maprev f l = mapacc f [] l ;;
```

With the function maprev defined, one can then define

```
let compose f g = fun x -> f (g x);;
let mapr f = compose rev (maprev f);;
```

The compose function has a mathematical notation as well: compose f g $\equiv f \circ g$.

With these definitions, we have that

$$\forall f : \ \forall l : \ mapr \ f \ l = map \ f \ l$$

The proof is left as an exercise.

3 Exercises

Exercise 1 Prove that for any list l, we have that

$$l \mathbin{@} [] = l$$

Exercise 2 Prove that $map\ (f \circ g) = (map\ f) \circ (map\ g)$. That is, prove that for any list l, we have that

$$map\ (f \circ g)\ l = ((map\ f) \circ (map\ g))\ l$$

Exercise 3 Consider the function

```
let rec sum l = match l with
  | [] -> 0
  | x::xs -> x + (sum xs)
;;
```

i) Prove that (len1 l)=(sum (map (**fun** x −> 1) l)) for any list l.
(You may use $\lambda x.1$ instead of **fun** x −> 1.)

ii) Give a tail-recursive function sum2 that is equal to sum.

iii) Prove that sum2 l=sum l for any list l.

Exercise 4 Prove that $mapr = map$. That is, prove that for any function f and any list l, we have that

$$mapr\ f\ l = map\ f\ l$$

Exercise 5 Consider the functions

```
let rec foldl f a l = match l with
  | [] -> a
  | x::xs -> foldl (f a x) xs
;;
let rec foldr f l b = match l with
  | [] -> b
  | x::xs -> f x (foldr f xs b)
;;
let swap f = fun x y -> f y x ;;
```

Prove that

```
foldr f l b = foldl (swap f) b (rev l)
```