

15–150: Principles of Functional Programming

List Reversal

Michael Erdmann

Spring 2020

This note proves an extensional equivalence for list reversal as discussed in lecture.

SUGGESTION: Read the first page of this note, then try to prove the Theorem. Then look at the second page for a solution. That is a good way to prepare yourself for homework and midterm.

Here is the relevant code:

```
(* rev : int list -> int list
   REQUIRES: true
   ENSURES:  rev(L) evaluates to the list L in reverse order.
*)
fun rev (l : int list) : int list = []
  | rev (x::xs) = (rev xs) @ [x]

(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc)  $\cong$  (rev L) @ acc
*)
fun trev(l : int list, acc : int list) : int list = acc
  | trev(x::xs, acc) = trev(xs, x::acc)

(* reverse : int list -> int list
   REQUIRES: true
   ENSURES:  reverse(L) evaluates to the list L in reverse order.
*)
fun reverse(L : int list) : int list = trev(L, [])
```

Theorem: For all values L and acc of type `int list`, $trev(L, acc) \cong rev(L) @ acc$.

In order to prove the theorem, we will use two lemmas, stated next. The first lemma says that $@$ is an associative operator. The second lemma says that appending a list consisting of one element onto a second list reduces to the process of consing that one element onto the second list.

Lemma 1: For all expressions e_1, e_2, e_3 of type `int list`, $e_1 @ (e_2 @ e_3) \cong (e_1 @ e_2) @ e_3$.

Lemma 1 is proved in the structural induction notes accompanying this lecture.

Lemma 2: For all values $x : int$ and $acc : int list$, $[x] @ acc \implies x::acc$.

Exercise: Prove Lemma 2 from the code for $@$ given here:

```
fun @ (l : int list, R : int list) : int list = R
  | @ (x::xs, R) = x :: @ (xs, R)
infixr @
```

Proof: By structural induction on L .

BASE CASE: $L = []$.

NEED TO SHOW: $\text{trev}([], \text{acc}) \cong \text{rev}([]) @ \text{acc}$ for all values $\text{acc} : \text{int list}$.

SHOWING: Let us evaluate the left and right sides of this “NEED TO SHOW” separately:

$$\begin{aligned} & \text{trev}([], \text{acc}) \\ \Rightarrow & \text{acc} && \text{[first clause of trev]} \\ \\ & \text{rev}([]) @ \text{acc} \\ \Rightarrow & [] @ \text{acc} && \text{[first clause of rev]} \\ \Rightarrow & \text{acc} && \text{[first clause of @]} \end{aligned}$$

Since both expressions reduce to the same value, they are extensionally equivalent. That establishes the base case.

INDUCTION STEP: $L = x::xs$, for some values $x : \text{int}$ and $xs : \text{int list}$.

INDUCTION HYPOTHESIS:

For all values $\text{acc}' : \text{int list}$, $\text{trev}(xs, \text{acc}') \cong \text{rev}(xs) @ \text{acc}'$.

(We wrote acc' rather than acc merely to distinguish it from the acc used below. We emphasize that the quantification in the IH is *for all* accumulator arguments.)

NEED TO SHOW: For all values $\text{acc} : \text{int list}$, $\text{trev}(x::xs, \text{acc}) \cong \text{rev}(x::xs) @ \text{acc}$.

SHOWING: We could again evaluate the left and right sides of the “NEED TO SHOW” separately. However, to illustrate a different proof approach, we will use explicit equivalences at each step:

$$\begin{aligned} & \text{trev}(x::xs, \text{acc}) \\ \cong & \text{trev}(xs, x::\text{acc}) && \text{[second clause of trev]} \\ \cong & \text{rev}(xs) @ (x::\text{acc}) && \text{[IH, with value acc' = x::acc]} \\ \cong & \text{rev}(xs) @ ([x] @ \text{acc}) && \text{[Lemma 2]} \\ \cong & (\text{rev}(xs) @ [x]) @ \text{acc} && \text{[Lemma 1]} \\ \cong & \text{rev}(x::xs) @ \text{acc} && \text{[second clause of rev]} \end{aligned}$$

That establishes the induction step.

The base case and the induction step together establish the Theorem, by a principle of structural induction for int list .

Thought Questions:

- Lemma 2 says $[x] @ \text{acc} \Rightarrow x::\text{acc}$, but above we use $x::\text{acc} \cong [x] @ \text{acc}$. Why is that valid?
- Why is the last step above, which cites the second clause of rev , a valid equivalence?