Folding Lists and Trees

Some more list functions

Recall the summation function

```plaintext
primrec sum :: "nat list ⇒ nat" where
  "sum [] = 0"
| "sum (x # xs) = x + sum xs"
```

In the Isabelle library, you will find (in the theory `List.thy`) the functions `foldr` and `foldl`, which allow you to define some list functions, among them `sum` and `length`. Show the following:

```plaintext
lemma sum_foldr: "sum xs = foldr (op +) xs 0"
lemma length_foldr: "length xs = foldr (λ x res. 1 + res) xs 0"
```

Repeated application of `foldr` and `map` has the disadvantage that a list is traversed several times. A single traversal is sufficient, as illustrated by the following example:

```
lemma "sum (map (λ x. x + 3) xs) = foldr h xs b"
```

Find terms `h` and `b` which solve this equation.

Generalize this result, i.e. show for appropriate `h` and `b`:

```plaintext
lemma "foldr g (map f xs) a = foldr h xs b"
```

Hint: Isabelle can help you find the solution if you use the equalities arising during a proof attempt.

The following function `rev_acc` reverses a list in linear time:

```plaintext
primrec rev_acc :: "['a list, 'a list] ⇒ 'a list" where
  "rev_acc [] ys = ys"
| "rev_acc (x#xs) ys = (rev_acc xs (x#ys))"
```

Show that `rev_acc` can be defined by means of `foldl`.

```plaintext
lemma rev_acc_foldl: "rev_acc xs a = foldl (λ ys x. x # ys) a xs"
```

Prove the following distributivity property for `sum`:
lemma sum_append [simp]: "sum (xs @ ys) = sum xs + sum ys"

Prove a similar property for foldr, i.e. something like foldr f (xs @ ys) a = f (foldr f xs a) (foldr f ys a). However, you will have to strengthen the premises by taking into account algebraic properties of f and a.
lemma foldr_append: "foldr f (xs @ ys) a = f (foldr f xs a) (foldr f ys a)"

Now, define the function prod, which computes the product of all list elements

prod :: "nat list ⇒ nat"

directly with the aid of a fold and prove the following:
lemma "prod (xs @ ys) = prod xs * prod ys"

Functions on Trees

Consider the following type of binary trees:

datatype 'a tree = Tip | Node ''a tree ''a ''a tree

Define functions which convert a tree into a list by traversing it in pre-, resp. postorder:

preorder :: ''a tree ⇒ ''a list
postorder :: ''a tree ⇒ ''a list

You have certainly realized that computation of postorder traversal can be efficiently realized with an accumulator, in analogy to rev_acc:

consts
postorder_acc :: "['a tree, 'a list] ⇒ 'a list"

Define this function and show:
lemma "postorder_acc t xs = (postorder t) @ xs"

postorder_acc is the instance of a function foldl_tree, which is similar to foldl.

consts
foldl_tree :: "('b ⇒ 'a ⇒ 'b) ⇒ 'b ⇒ 'a tree ⇒ 'b"

Show the following:
lemma "∀ a. postorder_acc t a = foldl_tree (λ xs x. Cons x xs) a t"

Define a function tree_sum that computes the sum of the elements of a tree of natural numbers:

consts
tree_sum :: "nat tree ⇒ nat"
and show that this function satisfies

\textbf{lemma} \ "\texttt{tree\_sum t = sum (preorder t)}"