Tries

Section 3.4.4 of the Isabelle/HOL tutorial is a case study about so-called tries, a data structure for fast indexing with strings. Read that section.

The data type of tries over the alphabet type 'a and the value type 'v is defined as follows:

\[
\text{datatype } ('a, 'v) \text{ trie } = \text{Trie } 'v \text{ option } ('a * ('a, 'v) \text{ trie}) \text{ list}
\]

A trie consists of an optional value and an association list that maps letters of the alphabet to subtrees. Type 'a option is defined in Section 2.5.3 of the Isabelle/HOL tutorial.

There are also two selector functions value and alist:

\[
\text{primrec } \text{value} :: ('a, 'v) \text{ trie } \Rightarrow 'v \text{ option}
\text{where }
\text{value } (\text{Trie } ov \text{ al}) = ov
\]

\[
\text{primrec } \text{alist} :: ('a, 'v) \text{ trie } \Rightarrow ('a * ('a, 'v) \text{ trie}) \text{ list}
\text{where }
\text{alist } (\text{Trie } ov \text{ al}) = al
\]

Furthermore there is a function lookup on tries defined with the help of the generic search function assoc on association lists:

\[
\text{primrec } \text{assoc} :: ('key * 'val)\text{list } \Rightarrow 'key \Rightarrow 'val \text{ option}
\text{where }
\text{assoc } [] x = \text{None}
\text{| assoc } (p#ps) x = \\
\text{ (let } (a, b) = p \text{ in if } a = x \text{ then Some } b \text{ else assoc ps x)}
\]

\[
\text{primrec } \text{lookup} :: ('a, 'v) \text{ trie } \Rightarrow 'a \text{ list } \Rightarrow 'v \text{ option}
\text{where }
\text{lookup } t [] = \text{value } t
\text{| lookup } t (a#as) = \text{(case } \text{assoc (alist } t) \text{ a of \}
\text{ None } \Rightarrow \text{None}
\text{| Some } at \Rightarrow \text{lookup at as)}
\]

Finally, update updates the value associated with some string with a new value, overwriting the old one:

\[
\text{primrec } \text{update} :: ('a, 'v) \text{ trie } \Rightarrow 'a \text{ list } \Rightarrow 'v \Rightarrow ('a, 'v) \text{ trie}
\text{where }
\text{update } t [] v = \text{Trie } (\text{Some } v) \text{ (alist } t)
\text{| update } t (a#as) v =
\]
The following theorem tells us that `update` behaves as expected:

**Theorem** 

\[ \forall t v bs. \text{lookup} (\text{update} t as v) bs = \begin{cases} \text{Some} v & \text{if } as = bs \\ \text{lookup} t bs & \end{cases} \]

As a warm-up exercise, define a function

**consts** `modify :: (\'a, \'v) trie \Rightarrow \'a list \Rightarrow \'v option \Rightarrow (\'a, \'v) trie`

for inserting as well as deleting elements from a trie. Show that `modify` satisfies a suitably modified version of the correctness theorem for `update`.

The above definition of `update` has the disadvantage that it often creates junk: each association list it passes through is extended at the left end with a new (letter,value) pair without removing any old association of that letter which may already be present. Logically, such cleaning up is unnecessary because `assoc` always searches the list from the left. However, it constitutes a space leak: the old associations cannot be garbage collected because physically they are still reachable. This problem can be solved by means of a function

**consts** `overwrite :: \'a \Rightarrow \'b \Rightarrow (\'a * \'b) list \Rightarrow (\'a * \'b) list`

that does not just add new pairs at the front but replaces old associations by new pairs if possible.

Define `overwrite`, `modify` to employ `overwrite`, and show the same relationship between `modify` and `lookup` as before.

Instead of association lists we can also use partial functions that map letters to subtrees. Partiality can be modelled with the help of type `\'a option`: if `f` is a function of type `\'a \Rightarrow \'b option`, let `f a = \text{Some} b` if `a` should be mapped to some `b`, and let `f a = \text{None}` otherwise.

**datatype** `(\'a, \'v) trieP = Trie \"\'v option\" (\'a \Rightarrow (\'a,\'v) trieP option)``

Modify the definitions of `lookup` and `modify` accordingly and show the same correctness theorem as above.