This document is based on material from the "Interactive Theorem Proving Course" by Thomas Tuerk (https://www.thomas-tuerk.de):
https://github.com/thtuerk/ITP-course

This document includes additions by:

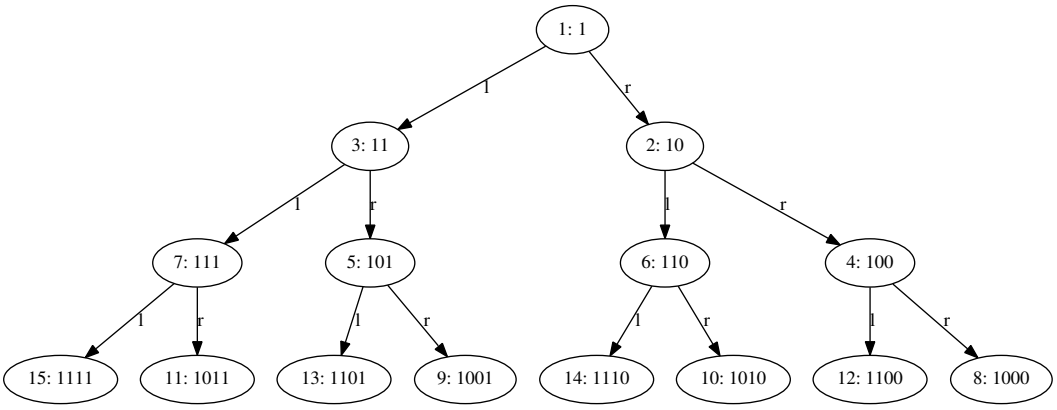
- Pablo Buiras (https://people.kth.se/~buiras/)
- Karl Palmskog (https://setoid.com)

ITPPV Homework 4

due 23:59 CET, Tuesday February 18, 2020

1 Functional Arrays

Arrays are usually not available for use in purely functional programs. However, one can easily encode finite maps with natural numbers as keys. One such finite map implementation is known as *functional arrays*. Functional arrays are binary trees that use the binary representation of the key to determine the position in the tree. As a result, the trees are always balanced. This is illustrated by the following picture:



The nodes in this tree are annotated with the both the decimal and the binary representation of their keys. The root node is for key 1; key 0 is not allowed. All values in the right subtree are even, while all values in the left subtree are odd. This means that the last digit is always 0 or always 1 depending on the subtree. We then continue with this scheme recursively. At level 2, we look at the second bit, level 3 looks at the third bit, and so on.

Navigating to the node for key k can easily be implemented recursively. We check whether k is 1. If so, we look at the root node of our tree. Otherwise, we check whether k is even. If it is, we search for key $k\ DIV\ 2$ in the right subtree, otherwise we look for $k\ DIV\ 2$ in the left subtree. Another way of describing the procedure is that we always look at the last bit. If we see 0, we go to the right subtree, and if we see 1, we go to the left subtree. We then throw the last bit away and continue. Once we reach the number 1, we stop.

Let's implement functional arrays in HOL4. Please use the file `hw4ArraysScript.sml`[1] for this purpose. It contains auxiliary definitions and an outline. Please read all of the homework instructions (except perhaps the hints section and the instructions for the optional visualization task) before you start working on any task.

[1] `https://github.com/kth-step/itppv-course/blob/master/homeworks/hw4-supplementary/hw4ArraysScript.sml`

1.1 Datatype

Define a datatype for functional arrays. This should be a binary tree type with leafs and nodes. Leafs don't store any information. Each node should have a left and a right subtree as well as, sometimes, a value. That is, some keys might have a value stored, and others not. So, please use an option type here.

In C, one would have nodes with a left and right subtree pointer. NULL values for these pointers would indicate that we don't have a subtree. This role of NULL pointers is in our functional implementation taken by leafs. Notice that leafs are not shown in the picture above. Each node in the last row above has implicitly a left and a right subtree, which are leafs.

1.2 Implement Basic Operations

Let's now implement a lookup, an update, a remove operation as well as a constant for empty arrays. Checking the bits of the key directly in the recursive definitions of these operations can be finicky. One has to reason about arithmetic a lot and deal with some awkward termination conditions. Therefore, the most finicky parts are already provided. The function `num2arrayIndex` takes a key and returns an array index. An array index is encoded as a list of booleans. If the list is empty, we should stop at the current node. If it starts with `F`, we should look at the left subtree, and if it is `T`, at the right one. There is also an inverse operation `arrayIndex2num` as well as a few lemmas. Notice that `num2arrayIndex` implicitly adds 1 to the number before looking at the bits. Thus, we can handle 0 and don't need a special case.

1.2.1 `EMPTY_ARRAY`

Define a constant `EMPTY_ARRAY` that represents an array, which has no values stored in it at all.

1.2.2 `UPDATE` and `REMOVE`

Define an update function. Start with defining a function `IUPDATE v a idx` that updates array `a` to contain `v` for index `idx`. It should return the updated array. Then use the definition of `UPDATE` already present in the theory to lift this definition to keys that are natural numbers. Similarly, define a function `REMOVE a idx` that removes the value stored for index `idx` from array `a`. No value should be stored for this index in the resulting array.

The remove and update functions are very similar. It is beneficial to define a generalised update function that takes an optional value argument. If a value is provided, the current value is updated with it. If no value is provided, the current one is removed.

1.3 Test your definition

Use `EVAL` to test whether your definitions work as expected.

1.3.1 `LOOKUP`

Define a lookup function. Similarly to for `UPDATE`, define a function `ILOOKUP` on indexes first and then lift it to numbers. `LOOKUP a k` should return `SOME v` if and only if the value `v` is stored for the key `k` and `NONE` if no value is stored.

1.4 Test your definition

Use `EVAL` to test whether your definition of `LOOKUP` works as expected.

1.5 Basic Properties

Show that you indeed implemented a finite map data structure. For this purpose, fill in the missing proofs in `hw4ArraysScript.sml`. Make sure that the resulting theory compiles properly using `Holmake`.

You should prove the following properties:

1. `!k. LOOKUP EMPTY_ARRAY k = NONE`
2. ```
   !v n n' a. LOOKUP (UPDATE v a n) n' =
         (if (n = n') then SOME v else LOOKUP a n')
   ```
3. `!n n' a. LOOKUP (REMOVE a n) n' = (if (n = n') then NONE else LOOKUP a n')`
4. `!v1 v2 n a. UPDATE v1 (UPDATE v2 a n) n = UPDATE v1 a n`
5. `!v n a. UPDATE v (REMOVE a n) n = UPDATE v a n`
6. `!v n a. REMOVE (UPDATE v a n) n = REMOVE a n`
7. ```
   !v1 v2 a n1 n2. n1 <> n2 ==>
         ((UPDATE v1 (UPDATE v2 a n2) n1) = (UPDATE v2 (UPDATE v1 a n1) n2))
   ```
8. ```
   !v a n1 n2. n1 <> n2 ==>
         ((UPDATE v (REMOVE a n2) n1) = (REMOVE (UPDATE v a n1) n2))
   ```
9. ```
   !a n1 n2. n1 <> n2 ==>
         ((REMOVE (REMOVE a n2) n1) = (REMOVE (REMOVE a n1) n2))
   ```

2 Hints

If you perform your proofs naively, you need a lot of case-splits and everything gets very lengthy. It is beneficial to use many auxiliary definitions and use them and many tiny lemmas about them to avoid case-splits. It might for example be beneficial to introduce auxiliary functions `VAL_OF_ROOT` and `GEN_GET_SUBARRAY` and derive the following properties:

- `!a.  ILOOKUP a [] = VAL_OF_ROOT a`
- `!a i idx. ILOOKUP a (i::idx) = ILOOKUP (GEN_GET_SUBARRAY i a) idx`

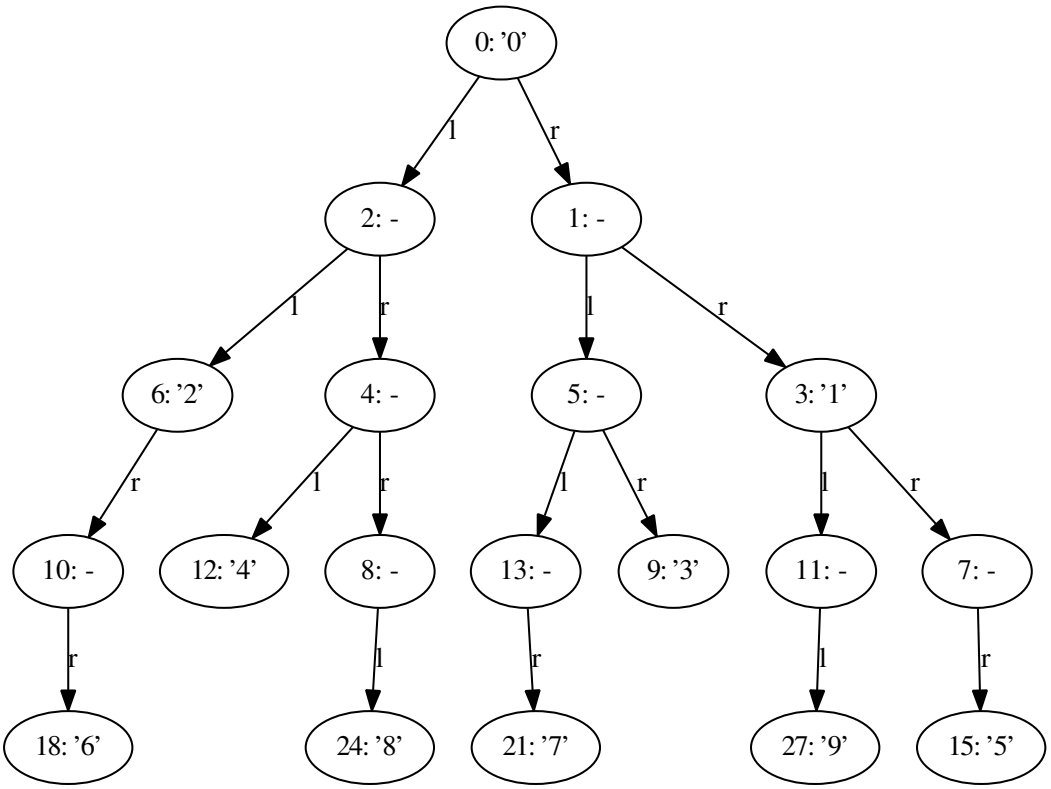
3 Optional: Visualise Trees

Note: this task is recommended but not required.

Let's use the graphviz tool (`http://www.graphviz.org`) to visualise your trees. We provide auxiliary library files, in the form of `dot_graphsLib.sig` and `dot_graphsLib.sml`, to communicate with graphviz, along with some glue code in `hw4ArraysLib.sml` to generate graphs from HOL4 arrays[2]. You may need to adjust the functions `is_array_leaf` and `dest_array_node` in `hw4ArraysLib.sml` for your specific datatype definition.

The example array `a2` should have for $n < 10$ the value n stored at key $3 * n$, and all other keys should have no value stored. When you visualise `a2`, the result should look as follows:

[2] `https://github.com/kth-step/itppv-course/blob/master/homeworks/hw4-supplementary`



This is a good test of whether your `UPDATE` works as expected.