15-150 Assignment 09

Jack Kasbeer
jkasbeer@andrew.cmu.edu
Section K
November 10, 2015

2: Bounding Boxes

---

1. Functor `Box` is implemented in `box.sml`.

3: Representation independence

---

1. Prove that for all types `t` and all values `L : t list` that

   `Standard.fromList L` $\equiv$ `Fun.fromList L`

   Proof. Let P(L) be the proposition that `Standard.fromList L` $\equiv$ `Fun.fromList L`.
   We will show that P(L) holds true for all lists L by structural induction (for lists).

   Base Case: L = `[]`.. Let K be an arbitrary `t list` such that `length(K)` is constant.
   By definition of `Standard.fromList`, we have `Standard.fromList ([] @ K) = [] @ K`, which is trivially just `K`, by Lemma 2.
   Notice that `K = foldr op :: K []`, by definition of `foldr`.
   One step further, and `(foldr op :: K []) = (fn xs => foldr op :: xs []) K`, by `[xs:K]`.
   Lastly, this is now the definition of `Fun.fromList`,
   so `(fn xs => foldr op :: xs []) K = (Fun.fromList []) K`.
   Hence, `Standard.fromList []` $\equiv$ `Fun.fromList []`, and the base case holds.

   Now, let L = `y::ys` and assume that P(`ys`) is true (**IH**) (`K : t list` will also be used below). WWTS that P(`y::ys`) also holds.
   Inductive Step: By definition of `Standard.fromList`, we may say that
   `Standard.fromList (y::ys)@K = (y::ys)@K`.
   By Lemma 3, `(y::ys)@K = y::(ys@K) => y::(Standard.fromList (ys@K))`, by definition of `Standard.fromList`.
   Now we can use our IH to say that this is equivalent to
   `y::((Fun.fromList ys) K) = y::((fn xs => foldr op :: xs ys) K)`,
   by definition of `Fun.fromList`. Then, similar to our base case, we now have `y::(foldr op :: K ys)`, by `[xs:K]`. And by definition of `op ::`, we can say that this is equivalent to
   `op :: (1, (foldr op :: K ys)`. By `foldr`, this is `(foldr op :: K y::ys)`.
   Again using `[xs:K]`, we have `(fn xs => foldr op :: xs (y::ys)) K`. This is now, by def. of `Fun.fromList`, equivalent to `Fun.fromList (y::ys) K`.
   Thus, `Standard.fromList (y::ys) @ K` $\equiv$ `Fun.fromList (y::ys) @ K`, by the definition of *equivalence*.

   Hence, by structural induction, P(L) holds for all lists of all sizes and we're done. $\square$

Jack Kasbeer
jkasbeer@andrew.cmu.edu

2. Prove that for all types `t` and all values `L1, L2 : t Standard.List`, `L1', L2' : t Fun.List`,

   if `L1` $\equiv$ `L1'` $\wedge$ `L2` $\equiv$ `L2'`, then `Standard.append L1 L2` $\equiv$ `Fun.append L1' L2'`

   Proof. By equivalence, we can prove the above proposition by showing that for all `K : t list`,

   if `L1` $\equiv$ `L1'` and `L2` $\equiv$ `L2'`, then
   `(Standard.append L1 L2) @ K = (Fun.append L1' L2') K.`

   So assume that `L1` $\equiv$ `L1'` and `L2` $\equiv$ `L2'`, and let `K` be an arbitrary `t list`.
   By def. of `Standard.append`, `(Standard.append L1 L2) @ K = (L1 @ L2) @ K`. Then by Lemma 1, we now have `L1 @ (L2 @ K)`, which is equal to `L1 @ (L2' K)` by assumption and the def. of equivalence. Again by assumption and equivalence, we have `L1' (L2' K)`.
   By def. of `o`, `L1' (L2' K) = (L1' o L2') K`, and finally by the def. of `Fun.append`, this equates to `Fun.append L1' L2'`.
   Thus, `(Standard.append L1 L2) @ K = (Fun.append L1' L2') K`.

   Hence, `Standard.append L1 L2` $\equiv$ `Fun.append L1' L2'`, by the definition of *equivalence*. $\square$

3. `Standard.append` and `Fun.append` differ mainly in the sense that they use a different number of cons operations. `Standard.append` uses more cons operations than `Fun.append`, which is most easily seen in how `append` is implemented in `Standard.append`. It appends lists using the `@` operation, which performs as many cons operations as there are elements in `xs`. In contrast to this, `Fun.append` does not perform cons operations until it's done appending lists (i.e. only on the final list). Hence, the implementation for `Fun.append` is less costly and more efficient.

4: Dictionaries as functions

1. The functor `FunDict` is implemented in `dict.sml`.

5: Polynomials over a ring

1. `RatRing:RING` is implemented in `ring.sml`.
2. The functor `PowerRing` is implemented in `ring.sml`.

6: Red-Black Trees

1. Prove that if `t = Node (l, (c, v), r)` is a valid red-black tree then so are both `l` and `r`.

   Proof. It's sufficient to prove this by showing each property of red-black trees holds.

   Property 1: If `t` is valid, then all of its leaves are black by definition. Since `t` is composed of `l` and `r`, all of the leaves in `l` and `r` are trivially black whether or not they're trees or leaves themselves.
   Property 2: If `t` is valid, then the child of any red node is black. Since all of `t`'s children are either `l`, `r`, or one of these branches' children, the child of every red node in `l` and `r` is black.
   Property 3: If `t` is valid, then each path from the root to a leaf has the same number of black nodes. Suppose there are n black nodes in these paths starting at `t`. **(1)** If the root of `t` is red, then paths in `l` and `r` still have n black nodes. **(2)** If the root of `t` is black, then every path in `l` and `r` still has the same number of black nodes, except it's now $n-1$ (root was black).

   Hence, if `t` is valid, then `l` and `r` satisfy all the properties of a red-black tree, which means that `l` and `r` are also both valid trees. □

2. Prove that for any red-black tree with a black-height of `b` (that is, each path from the root to a leaf has `b` black nodes not counting the leaf) then $2b \leq n$ where n is the number of nodes in the tree (counting all leaves).

   Proof. By structural induction on `T`.
   Base Case: `T` is a leaf. Notice this implies that `T` is black. Since it's a leaf, `b = 0` and `n = 0`.

   $$2^0 \leq 1 \Rightarrow 1 \leq 1$$

   This is trivially true, so the base case holds.

   Let `T = Node(L, (color, val), R)` be a valid red-black tree, and let `b'` denote the black-height of `T`. By these assumptions, we know that

   $$n = n_{left} + n_{right} + 1$$

   Now assume that P(L) holds and P(R) holds. In other words, $2^b \leq n_{left}$ and $2^b \leq n_{right}$ (IH). We may assume that the left and right branches have the same b because `T` is a valid red-black tree, which means both `L` and `R` are also valid (proven in 6.1).
   WWTS

   $$2^{b'} \leq n$$

   To do this, we'll case on the color of the root of `T`...
   Case 1: `color = Red`.. This means $b = b'$ since no additional black nodes are added to any of the paths. Then by our IH,

   $$2^b \leq n_{left} \Rightarrow 2^b \leq n_{left} + n_{right} + 1$$

   by def. of $\leq$.
   Since $n = n_{left} + n_{right} + 1$, it's clear that $2^{b'} \leq n$.

Case 2: `color = Black`.. In contrast to Case 1, this means that $b' = b + 1$ because the root of `T` being `Black` adds a black node to every path. By adding the inequalities from our IH, we have

$$2^b + 2^b \leq n_{left} + n_{right} \Rightarrow 2^{b+1} \leq n_{left} + n_{right} \Rightarrow 2^{b+1} \leq n_{left} + n_{right} + 1$$

by def. of $\leq$.
Hence, $2^{b'} \leq n$ because $b' = b + 1$ and $n = n_{left} + n_{right} + 1$. □

3. Prove that a red-black tree with `n` nodes has a height in $O(\log(n))$.

Proof. By sub-claims..
Sub-claim 1: Black-height of tree with height h, $b \geq h/2$.
We know that h is the longest path to the leaf by definition, and the worst case occurs when the nodes alternate between `Red` and `Black`. This is the worst case because when a node is `Red`, both its children must be `Black`, which means that number of `Red` nodes $\leq h/2$. This then means that the number of `Black` nodes $\geq h/2$.

INCOMPLETE

□