15-150 Assigment 5

Jonathan Li
jlli
Section S
June 2, 2016

Task 2.1

```
fun elephant (dumbo, ears) =
case dumbo of
[] => ears
| oooo::TRUMPET => "BEEP" ^ elephant TRUMPET, ears)
```

Here, `elephant : 'a list -> string`.

Task 2.2

`fn x => (fn y => x)`

The most general type of this expression is `'a -> ('b -> 'a)`.

Task 2.3

`(fn x => (fn y => x)) []`

The most general type of this expression is `'a -> 'b list`.

Task 5.1

Theorem 1: ∀ types `t` and values `L : t list list`,

$$\texttt{concat L} \cong \texttt{concatap L}.$$

Proof: By structural induction on L.
Base Case: `L` ≅ `[]`
To show: `concat []` ≅ `concatap []`
Proof:

`concat []`	≅	`case [] of [] => [] | ...`	[step]
	≅	`[]`	[step]
`concatap []`	≅	`case [] of [] => [] | ...`	[step]
	≅	`[]`	[step]

From the above, we can see that:

`concat []`	≅	`[]`	
`concatap []`	≅	`[]`	
∴ `concat []`	≅	`concatap []`	[Extensional Equivalence]

Inductive Step: `L` ≅ `l::ls` for some `l : 'a list`, `ls : 'a list list`
Inductive Hypothesis: For `ls : 'a list list`, `concat ls` ≅ `concatap ls`.
To show: For `L` ≅ `l::ls`, where `l : 'a list`, `concat L` ≅ `concatap L`.
Proof: By nested structural induction on `l`.
Nested Base Case: `l` ≅ `[]`
To show: `concat []::ls` ≅ `concatap []::ls`.
Proof:

`concat []::ls`	≅	`case []::ls of ...`	[step, totality of `concat`, `[]::ls` is a value]
	≅	`concat ls`	[step]
	≅	`concatap ls`	[Outer IH]
`concatap []::ls`	≅	`case []::ls of ...`	[step, totality of `concatap`, `[]::ls` is a value]
	≅	`append ([], concatap ls)`	[step]
	≅	`concatap ls`	[step]

From the above, we can see that:

`concat []::ls`	≅	`concatap ls`	
`concatap []::ls`	≅	`concatap ls`	
∴ `concat []::ls`	≅	`concatap []::ls`	[Extensional Equivalence]

Nested Inductive Step: `l` $\cong$ `x::xs` for some `x : 'a` and `xs : 'a list`
Nested Inductive Hypothesis: For `xs : 'a list`,`concat xs::ls` $\cong$ `concatap xs::ls` To show: `concat (x::xs)::ls` $\cong$ `concatap (x::xs)::ls`
Proof:

`concat (x::xs)::ls`	$\cong$	`case (x::xs)::ls of ...`	[step, totality of `concat`, `(x::xs)::ls` is a value]
	$\cong$	`case x::xs of ...`	[step]
	$\cong$	`x::concat(xs::ls)`	[step]
	$\cong$	`x::concatap(xs::ls)`	[Nested IH, Referential Transparency]
`concatap (x::xs)::ls`	$\cong$	`case (x::xs)::ls of ...`	[step, totality of `concatap`, `(x::xs)::ls` is a value]
	$\cong$	`append((x::xs), concatap ls)`	[step]
	$\cong$	`x::append(xs, ls)`	[step]
	$\cong$	`x::concatap(xs::ls)`	[Lemma 1, Referential Transparency]

From the above, we can see that:

`concat (x::xs)::ls`	$\cong$	`x::concatap(xs::ls)`	
`concatap (x::xs)::ls`	$\cong$	`x::concatap(xs::ls)`	
$\therefore$ `concat (x::xs)::ls`	$\cong$	`concatap (x::xs)::ls`	[Extensional Equivalence]

Since the Nested Base Case and the Nested Inductive Step hold, then the (outer) Inductive Step must be true. Furthermore, since the (outer) Base Case and the (outer) Inductive Step hold, Theorem 1 must be true.

Task 6.2

My implementation of `all_available` involves the following function calls:

```
map (fn intervs : (int * int) list => (startList intervs, endList intervs)) L
map (fn (sList : int list, eList : int list) => getFreeTimes (sList, eList) 0) startsEnds
foldr mergeFreeIntervs initial rest
```

Essentially, my implementation of `all_available` is $O(n\ log\ n)$ because each of the helper functions, including `map`,`foldr`, and `getFreeTimes`, is at most $O(n)$, linearly stepping through each element of the list provided. The only function call that takes $O(n\ log\ n)$ is the call to `msort` in each of the functions `startList` and `endList`, which sort the lists of start and end times. Those functions are called in parallel by the `val` declaration for `startsEnds`, so their runtimes do not compound into a larger big-O.