15-150 Assignment 5

Karan Sikka
ksikka@andrew.cmu.edu
G
2/25/12

1: Task 2.2

Claim: For all `l1: ` α `list` and all `l2: ` α `list list`,

`ap(l1, concatap l2)` $\cong$ `concatap(l1 :: l2)`

Proof:

`concatap(l1::l2)`	
$\cong$`case l1::l2 of [] => [] | x::xs => ap(x,concatap(xs))`	step

At this point, l1::l2 can be either be `nil` or `x::xs`. We will show both cases.

Case `nil`:
(both `l1` and `l2` are `nil`) since `[]::[] = []`:

`case [] of [] => [] | x::xs => ap(x,concatap(xs))`	def of concatap
$\cong$`[]`	step
$\cong$`l2`	value of l2
$\cong$`case [] of [] =>l2 | ...`	rev. step from ap
$\cong$`ap([],[])`	rev. step, def of ap
$\cong$`ap([],case [] of [] => [])`	rev. step, def of concatap
$\cong$`ap([],concatap([])`	rev. step from concatap
$\cong$`ap(l1,concatap(l2)`	values of l1, l2

Case `x::xs`:

$\cong$`case l1::l2 of [] => [] | x::xs => ap(x,concatap(xs))`	def of concatap
$\cong$`ap(l1,concatap(l2))`	reverse step

In both cases, the lemma holds. QED.

2: Task 2.3

`Claim:` For all `l : ` α `list list`,

ksikka@andrew.cmu.edu

`concat l` ≅ `concatap l`

Proof: We proceed by structural induction on `l`

Case []: Want to show: `concat []` ≅ `concatap []` Consider:

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

Case x::xs: Inductive Hypothesis: `concat xs` ≅ `concatap xs` Inductive Step:
Consider:

`concat x::xs`	
≅`case x::xs of ...  | x::xs => ...`	step
≅`case x of [] => concat xs | y::ys => y::concat (ys::xs) )`	step

In the list of lists xs, the first element can be the empty list, or a non-empty list. We will consider both cases. Case `x = []`:

≅`case [] of [] => concat xs | y::ys => y::concat (ys::xs) )`	cont. from above
≅`concat xs`	step
≅`concatap xs`	IH
≅`ap([],concatap xs)`	Lemma 2
≅`case []::xs of ...  | x::xs => ap(x,concatap xs)`	rev. step
≅`concatap []::xs`	rev. step

Case `x = y::ys`:

≅`case y::ys of [] => concat xs | y::ys => y::concat (ys::xs) )`	cont. from above
≅ `y::concat (ys::xs)`	step
≅ `y::(case ys::xs of ...  | x::xs => case x of ...  )`	step, evaluate the case
≅ `y::(case ys of [] => concat xs | z::zs =>...  )`	step

Here we run into another fork in the road. We said x = y::ys, but this function evaluates differently based on whether or not ys is nil. So we will consider both cases, since both are possible. Case `x`

`= y::[]`

≅ `y::(case [] of [] => concat xs | z::zs =>...  )`	cont. from above
≅ `y::(concat xs)`	step
≅ `y::(concatap xs)`	IH
≅ `y::(ap([],concatap xs))`	Lemma 2
≅ `case y::[] of [] => l2 | x::xs => x::(ap(xs,l2))`	rev. step, from ap
≅ `ap(y::[],concatap(xs))`	rev. step
≅ `ap(x,concatap(xs))`	x=y::[]
≅ `case x::xs of [] => [] | x::xs => ap(x,concatap(xs))`	rev. step (concatap)
≅ `concatap (x::xs)`	rev. step

Here we see that the lemma holds for the subcase when x = y::[].

Case `x = y::ys`:

≅ `y::(case ys of ...  | z::zs => z::(concat...  )`	cont. from above
≅ `y::(z::(concat(zs::ys))`	step
≅ `y::(z::(concat(zs::ys))`	step

This decomposition by concat will happen until the tail of the list is nil. Since lists are assumed to be finite, the decomposition will inevitably terminate. When the tail becomes nil, Case x=y::[] holds.

I can't figure out the rest of this proof. However, I tried starting at the bottom of the proof and working up. Here is what I came up with...

≅ `y::(z::(ap (zs,concat xs)))`	cont. from above.
≅ `y::(z::(ap (zs,concatap xs)))`	cont. from above.
≅ `y::(z::(concatap zs::xs)))`	cont. from above.
≅ `y::(concatap ys::xs)`	rev. step
≅ `y::(ap(ys,concat xs))`	rev. step
≅ `y::(ap(ys,concatap xs))`	IH
≅ `case y::ys of [] => l2 | x::xs => x::(ap(xs,l2))`	rev. step, from ap
≅ `ap(y::ys,concatap(xs))`	rev. step
≅ `ap(x,concatap(xs))`	x=y::[]
≅ `case x::xs of [] => [] | x::xs => ap(x,concatap(xs))`	rev. step (concatap)
≅ `concatap (x::xs)`	rev. step

I was almost there :(. Partial credit please?

3: Task 2.4

---

1. lemma 3
2. lemma 3
3. since g and f are valuable, `gof` is valuable
4. since g and f are total, `gof` are total
5. lemma 3
6. step
7. step
8. `map g(map f[])` $\cong$ `map(gof)[]`
9. step
10. step
11. step, def of map
12. `map g(map f xs)` $\cong$ `map gof xs`
13. `map g(map f xLLxs)` $\cong$ `map (gof) (x::xs)`
14. step
15. def of function composition
16. def of function composition
17. ***, transitivity, and totality of map