15-150 Assignment 8

Karan Sikka
ksikka@andrew.cmu.edu
G
April 11, 2012

1: Task 3.1

Claim 1: The empty queues are related.
Formally, $R($`LQ.emp`$,$`LLQ.emp`$)$.

Proof: Let `f` and `b` be lists. Consider `LLQ.emp` $\cong$ `(f,b)` $\cong$ `([],[])`.

Consider:

`f@(rev b)`	
$\cong$`[]@(rev [])`	def of LLQ.emp
$\cong$`[]@(case [] of [] => [] | ...)`	step
$\cong$`[]@[]`	evaluation of case
$\cong$`[]`	Lemma 3

Since `f@(rev b)` is equivalent to `LQ.emp` $\cong$ `[]`, R(LQ.emp,LLQ.emp) holds by definition of R.

Claim 2: Insertion preserves relatedness.
Formally, for all `x:int`, `l:int list`, `f:int list`, `b:int list`, if R(`l`, `(f,b)`), then R(`LQ.ins(x,l)`, `LLQ.ins(x,(f,b))`)

Proof: Let `x:int`, `l:int list`, `f:int list`, `b:int list` be arbitrary, and assume that:
R(`l`, `(f,b)`)

We want to show that R(`LQ.ins(x,l)`, `LLQ.ins(x,(f,b))`).

Define `(f',b')` $\cong$ `LLQ.ins(x,(f,b))` Consider:

`(f',b')`	
$\cong$`LLQ.ins(x,(f,b))`)	
$\cong$`(f,x::b)`	step
$\cong$`(f,x::b)`	step

Now we know that `(f',b')` $\cong$ `(f,x::b)`. Now consider:

`f' @ (rev b')`	
$\cong$`f @ (rev (x::b))`	proven above
$\cong$`f @ (case (x::b) of [] => [] | x::xs => (rev xs) @ x)`	step
$\cong$`f @ ((rev b) @ [x])`	case evaluation
$\cong$`(f @ (rev b)) @ [x]`	Lemma 1
$\cong$`l @ [x]`	Lemma 1
$\cong$`LQ.ins(l,x)`	step, equiv is transitive

Therefore, by the definition of R, we have proven that R(`LQ.ins(x,l)`, `(f',b')`). Then, by the our definition of `(f',b')`, we have proven the claim.

Claim 3: On related queues, removal gives equal integers and related queues.
More formally, we must prove the following: For all `x:int`, `l:int list`, `f:int list`, `b:int list`, if R(`l`, `(f,b)`) then one of the following is true:

`LQ.rem l` ≅ `NONE` and `LLQ.rem (f,b)` ≅ `NONE`
or
There exist `x:int, y:int`, `l':int list`, `f':int list`, `b':int list`, such that

1. `LQ.rem l` ≅ `SOME(x,l')`
2. `LLQ.rem (f,b)` ≅ `SOME(y,(f',b'))`
3. `x = y`
4. R(`l'`, `(f',b')`)

Proof: Let `l:int list`, `f:int list`, `b:int list`.
Assume R(`l`, `(f,b)`).

First, note that l may be either `[]` or `x::xs` (empty or non-empty). We will examine these two cases separately.

Case `l`≅`[]` Since we know that `l` and `(f,b)` are related, we know `[]` and `(f,b)` are related. This means that `[]` ≅ `f @ (rev b)`. We can see by the definition of the `@` function, that `[]` is only outputted when both inputs are `[]`. Therefore, `f`≅`[]` and `(rev b)` ≅ `[]`. However, `rev b` only outputs `[]` when `b`≅`[]`. Therefore `b`≅`[]`.

Consider:

`LQ.rem l`	
≅`LQ.rem []`	given
≅`case [] => NONE | ...`	step
≅`NONE`	case evaluation

And consider:

`LLQ.rem (f,b)`	
≅`LLQ.rem ([],[])`	shown above
≅`case ([],[]) of ([],[]) => NONE | ...`	step
≅`NONE`	case evaluation

We have shown that `LQ.rem l` ≅ `NONE` and `LLQ.rem (f,b)` ≅ `NONE`. Therefore the claim holds for this case.

Case `l`≅ `x::xs` where `x:int` and `xs:int list` Consider:

`LQ.rem l`	
≅`LQ.rem x::xs`	given
≅`case x::xs of [] = [] | x::xs => SOME(x,xs)`	step
≅`SOME(x,xs)`	step

Therefore we have shown that `LQ.rem l` ≅ `SOME(x,l')` where `l'` is `xs`.

Since we know that `l` and `f`,`b` are related, we know that `l` ≅ `x::xs` ≅ `f @ rev b`. By the spec of rev, we know that the length of rev b is equal to the length of b. By the spec of @, we know that the length of l is equal to the sum length of f and rev b, which, by the above statement, is equal to the sum of f and b. Since l has a positive length because it is not `[]`, then `f+b` has a positive length. Therefore, f or b, or both have positive length.

There are 3 cases for the lengths of f and b.

1. length of f = length of l (b is empty)
2. length of b = length of l (f is empty)
3. f and b are both non-zero

We will see that case 2 becomes case 1 after 1 iteration of LLQ.rem. Consider case 2: `LLQ.rem ([],b)` which steps to `case ([],b) of ... | ([],_) => rem (rev back,[])` which steps to `rem (rev b, [])`. This is clearly case 1 where `f` ≅ `rev b`. We can confirm that this is true, since in case 1, when we flatten the LLQ we get f@rev[] which is equivalent to f@[], which by lemma 3 is equivalent to f. And, since we assumed relatedness, `l` ≅ `f` . In case 2, when we flatten the LLQ we get `[]@rev b`, which according to Lemma to is equivalent to `rev b`, and since we assumed relatedness to l, `l` ≅ `rev b`. After it all, we see case 2 is equivalent to case 1 by transitivity through l.

Therefore, we wil only prove the properties for Case 1 and 3, and say that case 2 becomes case 1.

Now to show `LLQ.rem (f,b)` ≅ `SOME(y,(f',b'))` for case 1 and 3:

Consider (where f = z::zs and b is arbitrary): `LLQ.rem (f,b)` which is the same as
`LLQ.rem (z::zs,b)` which steps to
`case (z::zs,b) of ... | (x::xs,_) => SOME (x, (xs,back))` which evaluates to
`SOME (z, (zs,b))`

Clearly, the claim is true for some `y, f' and b'. where y is z, f' is zs, and b' is b.`

Now we claim that `x = y`. Recall that `x` refers to `l` ≅ `x::xs` and y refers to `f` ≅ `z::zs`. Also recall that `l` ≅ `f@rev b` by relatedness. Proof:
`l` ≅ `f@rev b` is true by relatedness, so by equivalence,
`x::xs` ≅ `z::zs@rev b`

Clearly, `x = z` because the first elements of two non-empty but congruent lists are congruent and above we stated "y is z" so `x` ≅ `y`.

The proof is trivial... I value my sleep...