What does the correctness proof for TLS-let look like?

TLS-let refers to TLS with the let form added.

Notes

(1) I won't give details on coding the addition of let, except to say that you need to add a new type — ie — a new action function, *let

(2) Realize that let "can occur anywhere" — you may not base your code adaptations or your proof on the mistaken assumption that only top-level let needs to be dealt with..

(3) What is the standard of correctness? Basically, we'll punt; we will say that the goal is to prove that TLS-let evaluates its (syntactically correct) inputs to give the same values as R5RS would give for those inputs.

④ Do we have to give a proof for the entire TLS-let system? NO - we could — but instead we will assume that TLS (without let) works correctly [by the same standard]

On to the proof

The proof is structural induction on TLS-let.

Basis: numbers
booleans
primitives

IH: The conclusion is valid for all proper components of the current TLS-let expression

IS: as you expect - I'll fill in enough of the details, below.

We are to show that every TLS-let expression is correctly evaluated by our (extended) interpreter.

Basis: numbers, booleans and primitives all belong to TLS. So - by our assumption they are correctly evaluated.

IH: as above

IS: (By cases)

① The current exp. is (quote e), where e is in TLS-let. Referring to the specific coding of the system, we argue that (quote e) is reduced to (*quote e)
which returns e — according to the implementation of *quote.
And this is exactly what R5RS would do.

(i) → squeeze the identifier case in here.
e

(iii) The current expression is an application, say $(e_1\ e_2)$.

If let occurs in e, then it occurs entirely within e_1 or e_2 — by syntactic definition, let cannot "span" these two subcomponents. And so — entirely by the IH — we know that e_1 and e_2 (which may contain let) are evaluated correctly. So then — using our assumption about TLS — the application is evaluated correctly as well.

(iv) cond is handled similarly —

(v) what about e being a

lambda expression?

We step through the evaluation to get to *lambda, which creates a closure — and any occurrence of let is entirely in the body of the closure (with no further evaluation carried out at this stage). So — done

(vi) the remaining case? let itself. For this, we'd to argue that our *let function is correct. Details left for you.