2 Compiler Construction (tgg22)

This question involves the derivation of “stack machines” using the CPS transformation.

(a) Consider the following OCaml code of type

\[
\text{add_right} : \text{int list} \rightarrow \text{int}
\]

that returns the sum of the integers in its argument list.

\[
\text{let rec add_right l =}
\text{match l with}
\text{| [] -> 0}
\text{| h::tl -> h + (add_right tl)};;
\]

Explain why this code, as presented, is not tail recursive. [2 marks]

(b) Use the CPS transformation to rewrite \text{add_right} to a function that could be given the type

\[
\text{add_right_cps} : \text{int list} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{int}
\]

[6 marks]

(c) Apply defunctionalisation to your code for \text{add_right_cps}. That is, define a (non-functional) data type \text{cnt} and a transformed function \text{add_right_dfc} of type

\[
\text{add_right_dfc} : \text{int list} \rightarrow \text{cnt} \rightarrow \text{int}
\]

[6 marks]

(d) The function \text{add_right} from Part (a) could be generalised to the following function.

\[
\text{let rec fold_right f l accu =}
\text{match l with}
\text{| [] -> accu}
\text{| a::l -> f a (fold_right f l accu)};;
\]

For simplicity, we will treat this code as if it had the type

\[
\text{fold_right} : (\text{int} \rightarrow \text{int} \rightarrow \text{int}) \rightarrow \text{int list} \rightarrow \text{int} \rightarrow \text{int}
\]
and not worry about polymorphism. Rewrite this program using the CPS transformation. Justify your treatment of the variable \( f \). What problems might you encounter in attempting to defunctionalise your CPS version? [6 marks]