5 Compiler Construction (TGG)

Functional programmers will often rewrite a recursive function such as

\[
\text{fun fact1 } n = \\
\text{ if } n <= 1 \\
\text{ then } 1 \\
\text{ else } n \times (\text{fact1 } (n - 1))
\]

to one such as

\[
\text{fun fact2 } n = \\
\text{ let fun aux } (m, a) = \\
\text{ if } m <= 1 \\
\text{ then } a \\
\text{ else aux}(m-1, m \times a) \\
\text{ in aux } (n, 1) \text{ end}
\]

using an accumulator (the parameter \(a\) of \(aux\)) and tail recursion.

\(a\) Clearly explain the optimisation such programmers are expecting from the compiler and how that optimisation might improve performance. [4 marks]

\(b\) The desired optimisation can be performed by a compiler either directly on the source program or on lower-level intermediate representations. Treating it as a source-to-source transformation, rewrite \(\text{fact2}\) to ML code that has been transformed by this optimisation. You will probably use references and assignments as well as the construct \(\text{while EXP do EXP}\). [8 marks]

\(c\) Suppose that the programmer used instead a function as an accumulator.

\[
\text{fun fact3 } n = \\
\text{ let fun aux } (m, h) = \\
\text{ if } m <= 1 \\
\text{ then } h(1) \\
\text{ else aux}(m-1, \text{fn r => m \times (h r)}) \\
\text{ in aux } (n, \text{fn x => x}) \text{ end}
\]

Will your optimisation still work in this case? Explain your answer in detail. [8 marks]