3 Compiler Construction (tgg22)

(a) Suppose we are writing a compiler for an ML-like language and we want to employ the equation

\[(\text{map } f \text{ l1}) @ (\text{map } f \text{ l2}) = \text{map } f \ (\text{l1} @ \text{l2})\]

as a left-to-right rewrite rule for optimisation. The symbol @ represents list append.

Discuss the merits of this idea. Is it always correct? If so, state clearly what assumptions you are making about @ and map. [5 marks]

(b) A compiler’s front-end will often expand some syntactic constructs into lower-level representations. Consider the following fragment for the abstract syntax of a SLANG-like language.

\[
\begin{align*}
\text{type var} & = \text{string} \\
\text{type exp} & = \\
& \quad (* \text{abstract syntax *}) (* \text{concrete syntax *}) \\
& \quad | \ \text{Var of var} (* x *) \\
& \quad | \ \text{Project of int * exp} (* \text{proj i e *}) \\
& \quad | \ \text{Tuple of exp list} (* (e1, e2, ... , en) *) \\
& \quad | \ \text{Let of var * exp * exp} (* \text{let x = e1 in e2 *}) \\
& \quad | \ \text{Apply of exp * exp} (* e1 e2 *) \\
& \quad | \ \text{Function of var * arg_pattern * exp} (* \text{fun f p = e *}) \\
\end{align*}
\]

and \text{arg_pattern} =

\[
\begin{align*}
& \quad | \ \text{APvar of var} (* x *) \\
& \quad | \ \text{APtuple of arg_pattern list} (* (p1, p2, ... , pn) *) \\
\end{align*}
\]

This language has general projections for \(n\)-tuples so

\[\text{proj i } (e_1, e_2, \cdots , e_k)\]

will evaluate to \(v_i\), the value of \(e_i\). If \(i\) is not in the range between 1 and \(k\) there will be a run-time error.

In this language we can write functions with simple (possibly nested) patterns for function arguments:

\[
\text{fun f } (a, b, (c, (d, e))) = b \ a
\]

[continued …]
Now suppose we want our front-end to eliminate such patterns. That is, we want to write a function of type

\[ \text{eliminate_tuple_patterns} : \text{exp} \rightarrow \text{exp} \]

so that the resulting expression contains functions with patterns only of the form \text{APvar} \space x for some (new) variable \space x. For example, the code for \text{f} above should be translated to a semantically equivalent expression of the form

\[
\text{fun f x = } \ldots
\]

that contains only simple variable arguments (that is, only \text{APvar} patterns in the abstract syntax).

Your task is to write this function in OCaml. You can assume that you have a function for generating fresh variable strings.

\[ \text{new_var} : \text{unit} \rightarrow \text{string} \]

[15 marks]