9 Semantics of Programming Languages (PMS)

Consider a language with abstract syntax

$e ::= \quad n \mid x \mid \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \mid \textbf{alloc} \mid \textbf{free}\, e \mid e_1 := e_2 \mid !e \mid e_1; e_2 \mid e_1 + e_2$

This is intended to allow computation over data allocated in a concrete block of memory: n ranges over a set $W = \{0, \ldots, 2^{32} - 1\}$ of machine words, used both as values and as addresses. A memory state is described by a total function $m : W \to W$, giving the value at each address, and a set $a \subseteq W$, identifying the locations that are currently allocated. The term x ranges over a set of non-mutable variables, not allocated in memory. The expression $e := e'$, $!e$, **alloc**, and **free** e are respectively assignment, dereferencing, allocation, and free of single words.

(*a*) Define a reasonable deterministic operational semantics for this language, as a transition relation

$$\langle e, m, a\rangle \longrightarrow \langle e', m', a'\rangle$$

and a predicate

$$\langle e, m, a\rangle\, \textbf{error}$$

that identifies the configurations that are runtime errors. You can omit the rules for $e_1; e_2$ and $e_1 + e_2$ and the standard definition of substitution.

Your definition should ensure (though you need not prove) that for any configuration $\langle e, m, a\rangle$, either e is a value n, or there is exactly one transition $\langle e, m, a\rangle \longrightarrow \langle e', m', a'\rangle$ from that configuration, or there is exactly one derivation of a runtime error $\langle e, m, a\rangle\, \textbf{error}$.

Note and explain your choices. [17 marks]

(*b*) One could rule out some of those runtime errors with a simple type system that keeps addresses and the numbers used for arithmetic distinct, with types

$T ::= \quad \textbf{address} \mid \textbf{number}$

and type rules that constrain assignment, dereferencing, allocation, free, and arithmetic.

Discuss which of your runtime errors could be prevented by this. [3 marks]