2 1F-2

I both agree and disagree with Hoare's points regarding static type discipline. In brief, he argues for static typing via explicit programmer annotation. First, I agree problems can arise due to languages being too forgiving. Horror stories due to permissive javascript coercions are well-known; I'll provide a slightly more esoteric example from my experience using Clojure in industry (a dynamically-typed lisp dialect). In Clojure both (linked) lists and arrays implement a common 'sequence' interface. Despite different performance characteristics, their value semantics are virtually identical; excepting the function 'conj', a generic procedure used to add elements to any type implementing the very general 'collection' interface. For lists, conj prepends elements; for arrays, it appends them. Since all the other sequence functions behave more-or-less identically, introducing a function producing lists into a codebase expecting arrays will likely produce no visible consequences, unless or until a 'conj' is introduced, whereafter elements will begin be added to the wrong side of collections. This happened at my job, and went unnoticed for more than a week, requiring costly manual database surgery to repair. While the genericity of the interfaces Clojure provides result in terse, readable code, free from distractions, a simple type declaration could have prevented this error.

However, static typing discipline is not free. Maybe this was a simple enough argument to make in 1973, when maybe most type systems were simple agglomerations of primitive machine types. Even then though there arise situations where code duplication is necessary for code to be well-typed, which itself can become a liability. For example, simple type systems without polymorphism often mandate re-implementation of syntactically identical functions, differing only in that one takes (say) floats as opposed to integers. This permits subtle inconsistencies to enter the system if one definition is updated but not the other. The solution here has been to mandate more-and-more complex type system features; parametric polymorphism might address the previous problem, but the resulting type signatures get longer. This in-turn is addressed through (say, Hindley–Milner-style) type inference; explicit annotations are now optional, but type errors become more inscrutable as they rely on non-local constraint satisfaction problems.

More insidiously, users of increasingly complex type systems can sometimes begin relying on them as replacements for tests, even as the code duplication required to satisfy the type system permits more opportunities for inconsistency which the type system is not able to detect, sometimes negating the assurances offered by the system. There are real tradeoffs here, which I strongly believe have different answers in different contexts. Static typing and dynamic typing both require different kinds of discipline on the part of programmers, and language designers shouldn't make decisions about type systems without considering what use-cases they wish to support.

3 1F-3

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 \div e_2, \sigma\rangle \Downarrow n_1 \div n_2} \; \mathsf{n}_2 \neq 0$$

Let the division symbol on the right-hand side of the conclusion denote integer division, i.e. the quotient of Euclidean division, i.e. integer division in the OCaml/C semantics, which is defined on all integers n_1 and all non-zero integers n_2 and gives an integer result. I'm choosing here to just let evaluation get stuck if the denominator evaluates to 0, leaving the semantics non-total. Alternatives include introducing a numeric supertype including a NaN value, or adding a generic error value to all types. These would involve modifying other rules though, to propagate errors through the system.

4 1F-4

We evaluate e in the initial environment, then evaluate c in the original environment augmented by binding x to the value of e. We then return the resulting environment, except with the original value for x (if any) restored. When x is not in sigma, $\sigma(x)$ returns something denoting 'undefined'.

$$\frac{\langle e, \sigma\rangle \Downarrow n \quad \langle c, \sigma[x := n]\rangle \Downarrow \sigma'}{\langle \texttt{let } x = e \texttt{ in } c, \sigma\rangle \Downarrow \sigma'[x := \sigma(x)]}\ \textsf{let}$$

5 1F-5

Additions (where $\texttt{let}_{\texttt{old}}$ denotes a new (temporary) syntactic form) :

New redexes:

- $\texttt{let } x = n \texttt{ in } c$
- $\texttt{let}_{\texttt{old}}\ x = n \texttt{ in } \mathit{skip}$

New reduction rules:

- $\langle \texttt{let } x = n_{new} \texttt{ in } c, \sigma\rangle \rightarrow \langle \texttt{let}_{\texttt{old}}\ x = \sigma(x) \texttt{ in } c, \sigma[x := n_{new}]\rangle$
- $\langle \texttt{let}_{\texttt{old}}\ x = n_{old} \texttt{ in } \mathit{skip}, \sigma\rangle \rightarrow \langle \mathit{skip}, \sigma[x := n_{old}]\rangle$

New contexts:

- $\texttt{let } x = H \texttt{ in } c$
- $\texttt{let}_{\texttt{old}}\ x = n \texttt{ in } H$

The idea here is that when we get to the *let*, the first context causes us to small-step through e until we reduce it to n. Then, $x = n$ is added to σ, but the old value of x, if any, syntactically replaces n in the *let* form. The form is renamed let_{old} to keep the reduction deterministic. Then the second context causes us to small-step c until we end up with *skip*, whereupon the binding of x in σ is replaced with the old value. This feels hacky but I couldn't figure out how else to do it in a way that didn't involve modifying other rules, e.g. making σ stack-valued etc

1 HW1 (select all pages: your first page has your name and bookkeeping, and all others are anonymous))

- **0 pts** Correct