Exercise 1F-2

Throughout my years of programming, I have come to prefer statically typed languages over dynamically typed languages more and more. Not having to specify the type of a variable and allowing a variable to change type saves some time when first writing out a program. However, if there are any bugs to correct, or if I ever want to go back to read or modify the program, the lack of types makes everything take much longer. Instead of being able to tell from a glance what kind of things I expect a variable to contain, I instead have to slog through old code to remember exactly what is going on. Furthermore, I've had the experience of teaching new programmers to code in *Python*, a dynamically typed language. While all the students were exceptionally smart, they were tripped up time and time again by the perils of dynamic typing. Overall, it seems that Hoare agrees with my opinions about types, and brings up many other concerns about dynamic typing such as efficiency, security, and documentation.

Moving away from types, Hoare's opinions on comments have been shown to be precisely correct. Despite being relatively rare at the time of writing, (almost?) all modern languages contain a special symbol (e.g., // or #) that turns the rest of the line into a comment. This convention is exactly what Hoare suggested as the proper way to specify comments. Indeed, even though most languages have /* block comments */, it is often seen as bad practice to use them outside of certain specific scenarios. In fact, block comments were forbidden except as file headers by the style guide at all companies I've been employed by. It's remarkable how correct Hoare was about comments over 50 years ago.

Hoare takes a negative stance on so-called optimizing compilers. He claims that they tend to be inefficient, lack the guarantees of non-optimizing compilers, and remove control from the programmer. I believe that, while he is at least partially correct on all of these points, his criticism of optimizing compilers is overstated. Optimizing compilers are often slow; however, there is no need to run an optimizing compiler while writing or debugging code. Indeed, such heavy optimizations should be saved for some final executable that will be run many times or on very large data. Until then, a non-optimizing (or less-optimizing) compiler should be used. This yields the "best of both worlds," allowing for fast compile times and fast run times whenever they respectively matter. Much work has taken place in the decades since 1973 to produce reliable optimizing compilers, and today there is usually little risk of optimizations producing incorrect output. For example, g++ features an O2 flag that is meant to guarantee the same functionality as when optimizations are turned off (with the more experimental Ofast flag lacking these guarantees). I believe that history has, to an extent, proved Hoare wrong on optimizing compilers. These optimizing compilers are used far and wide in the real world, and are responsible for huge performance improvements.

Question assigned to the following page: 3

Exercise 1F-3

There are two reasonable interpretations of a division operator:

- a division operator that acts on integers, rounding down
- a division operator that acts on rational numbers

The first version is simpler, as we do not then need to consider new types of data. We first amend our definition of an arithmetic expression to include division:

$$\begin{aligned} e ::= & n && n \in \mathbb{Z} \\ \mid & x && x \in L \\ \mid & e_1 + e_2 && e_1, e_2 \in \mathsf{Aexp} \\ \mid & e_1 - e_2 && e_1, e_2 \in \mathsf{Aexp} \\ \mid & e_1 * e_2 && e_1, e_2 \in \mathsf{Aexp} \\ \mid & e_1 / e_2 && e_1, e_2 \in \mathsf{Aexp} \end{aligned}$$

We can implement it via the following rule:

$$\frac{\langle a_1, \sigma \rangle \Downarrow n_1 \quad \langle a_2, \sigma \rangle \Downarrow n_2}{\langle a_1 / a_2, \sigma \rangle \Downarrow \lfloor n_1 / n_2 \rfloor}$$

Here, $\lfloor n_1/n_2 \rfloor$ refers to floor division defined in the usual way. If $n_2 = 0$, this is undefined. The desired behavior for division by zero should be to throw an exception, but IMP does not have exceptions yet (I have looked ahead in the homeworks and seen this is a temporary problem). To avoid these issues, we will simply say that this rule is only defined for $n_2 \neq 0$.

Suppose instead that we wish to have true rational division. Then we should use the same rule we first defined:

$$\frac{\langle a_1, \sigma \rangle \Downarrow n_1 \quad \langle a_2, \sigma \rangle \Downarrow n_2}{\langle a_1 / a_2, \sigma \rangle \Downarrow n_1 / n_2}$$

This time, we take n_1/n_2 to mean the *rational* result of dividing n_1 by n_2, again requiring $n_2 \neq 0$. We should then also amend our definition of an arithmetic expression to allow the supply of rational values:

$$\begin{aligned} e ::= & n && n \in \mathbb{Q} \\ \mid & x && x \in L \\ \mid & e_1 + e_2 && e_1, e_2 \in \mathsf{Aexp} \\ \mid & e_1 - e_2 && e_1, e_2 \in \mathsf{Aexp} \\ \mid & e_1 * e_2 && e_1, e_2 \in \mathsf{Aexp} \\ \mid & e_1 / e_2 && e_1, e_2 \in \mathsf{Aexp} \end{aligned}$$

Question assigned to the following page: 4

Exercise 1F-4

We use the following new rule:

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

Informally, this

1. evaluates e in state σ to find n,
2. evaluates c in state σ with the value of x set to n to find state σ', and
3. returns the state σ' with the value of x set back to its original value.

Thus, this rule evaluates c with x set to the result of e, and carries over the changes from that evaluation modulo the value of x being restored.

Question assigned to the following page: 5

Exercise 1F-5

We first extend the redexes by adding the following option for r:

$$r ::= \ldots \mid \texttt{let}\ x = n\ \texttt{in}\ c$$

We extend the contexts by adding the following option for H:

$$H ::= \ldots \mid \texttt{let}\ x = H\ \texttt{in}\ c$$

Finally, we add the following new reduction rule:

$$\langle \texttt{let}\ x = n\ \texttt{in}\ c, \sigma \rangle \to \langle c; x := \sigma(x), \sigma[x := n] \rangle$$