CS558 Programming Languages – Fall 2023 – Suggested Study Question Solutions for Lecture 7a

1.

(a)

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \texttt{1} : \text{Int}}\;(\text{Int}) \qquad \dfrac{}{\emptyset \vdash \texttt{2} : \text{Int}}\;(\text{Int})}{\emptyset \vdash \texttt{(+ 1 2)} : \text{Int}}\;(\text{Add}) \qquad \dfrac{}{\emptyset \vdash \texttt{3} : \text{Int}}\;(\text{Int})}{\emptyset \vdash \texttt{(+ (+ 1 2) 3)} : \text{Int}}\;(\text{Add})$$

(b) This expression is not typable in the empty environment, because `x` is free. More formally, any typing tree for this expression would need to have the shape

$$\dfrac{\dfrac{\emptyset(x) = \text{Int}}{\emptyset \vdash \texttt{2} : \text{Int}}\;(\text{Var}) \qquad \dfrac{}{\emptyset \vdash \texttt{1} : \text{Int}}\;(\text{Int})}{\emptyset \vdash \texttt{(+ x 1)} : \text{Int}}\;(\text{Add})$$

but the (Var) inference is invalid, since $\emptyset(x)$ is not defined.

(c) The (While) rule is based on the assumption that the final value of a `while` expression is always the integer 0.

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{0} : \text{Int}}\;(\text{Int}) \qquad \dfrac{\dfrac{\dfrac{\mathit{TE}_1(\mathrm{x}) = \text{Int}}{\mathit{TE}_1 \vdash \mathrm{x} : \text{Int}}\;(\text{Var}) \qquad \dfrac{}{\mathit{TE}_1 \vdash \texttt{10} : \text{Int}}\;(\text{Int})}{\mathit{TE}_1 \vdash \texttt{(<= x 10)} : \text{Bool}}\;(\text{Leq}) \qquad \dfrac{\mathit{TE}_1(\mathrm{x}) = \text{Int} \qquad \dfrac{\dfrac{\mathit{TE}_1(x) = \text{Int}}{\mathit{TE}_1 \vdash \mathrm{x} : \text{Int}}\;(\text{Var}) \qquad \dfrac{}{\mathit{TE}_1 \vdash \texttt{1} : \text{Int}}\;(\text{Int})}{\mathit{TE}_1 \vdash \texttt{(+ x 1)} : \text{Int}}\;(\text{Add})}{\mathit{TE}_1 \vdash \texttt{(:= x (+ x 1))} : \text{Int}}\;(\text{Assgn})}{\mathit{TE}_1 \vdash \texttt{(while (<= x 10) (:= x (+ x 1)))} : \text{Int}}\;(\text{While})}{\emptyset \vdash \texttt{(let x 0 (while (<= x 10) (:= x (+ x 1))))} : \text{Int}}\;(\text{Let})$$

where $\mathit{TE}_1 = \{\mathrm{x} \mapsto \text{Int}\}$.

(d)

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \texttt{0} : \text{Int}}\;(\text{Int}) \qquad \dfrac{}{\emptyset \vdash \texttt{1} : \text{Int}}\;(\text{Int})}{\emptyset \vdash \texttt{(<= 0 1)} : \text{Bool}}\;(\text{Leq}) \qquad \dfrac{\dfrac{\dfrac{\mathit{TE}_1(\mathrm{x}) = \text{Bool}}{\mathit{TE}_1 \vdash \mathrm{x} : \text{Bool}}\;(\text{Var}) \qquad \dfrac{}{\mathit{TE}_1 \vdash \texttt{0} : \text{Int}}\;(\text{Int}) \qquad \dfrac{}{\mathit{TE}_1 \vdash \texttt{1} : \text{Int}}\;(\text{Int})}{\mathit{TE}_1 \vdash \texttt{(if x 1 2)} : \text{Int}} \qquad \dfrac{}{\mathit{TE}_1 \vdash \texttt{3} : \text{Int}}\;(\text{Int})}{\mathit{TE}_1 \vdash \texttt{(+ (if x 1 2) 3)} : \text{Int}}\;(\text{Add})}{\emptyset \vdash \texttt{(let x (<= 0 1) (+ (if x 1 2) 3))} : \text{Int}}\;(\text{Let})$$

where $\mathit{TE}_1 = \{\mathrm{x} \mapsto \text{Bool}\}$.

(e) There is no valid proof tree for this expression because the two arms of the `if` don't have the same type: `(<= 1 2)` is Bool but `3` is Int. Thus, the (If) rule cannot be applied. Note that the expression is untypable even though it would evaluate without any problem under the usual dynamic semantics, since the second arm of the `if` will always be taken—but the type system doesn't know about this.

2. Here is a suitable rule:

$$\dfrac{\mathit{TE} \vdash e_1 : t_1 \qquad \mathit{TE} \vdash e_2 : t_2}{\mathit{TE} \vdash \texttt{(before } e_1 \; e_2\texttt{)} : t_1}\;(\text{Before})$$

Note that is very important to insist (in the second hypothesis) that e_2 has *some* type t_2 even though we do not care what that type is. It might be tempting to write the rule like this:

$$\frac{TE \vdash e_1 : t_1}{TE \vdash \texttt{(before } e_1 \ e_2\texttt{)} : t_1} \text{ (Before')}$$

but this would be a mistake: using this rule, e_2 could be completely ill-typed yet we would still say the overall `before` expression was well-typed.

3. The key idea is to realize that we can define a `struct` with a single field. (Since C structs are not boxed, this doesn't even have any runtime cost.)

```
// defining abbreviations isn't essential, but makes the code more readable
typedef struct ftemp {float t;} FTemp;
typedef struct ctemp {float t;} CTemp;

FTemp x,y;
CTemp z;
x = y;
x.t = 10.0;
x = z;  // should be type error
x.t = 1.8 * z.t * 32.0;
```

4. (a) The second assignment is obviously valid, since left and right sides have the same type (`B`). Using the subsumption rule, we see that the first and third assignments are valid (we are providing a value of subtype `B` where a value of the supertype `A` is expected). Only the fourth assignment is (statically) invalid: all we know statically is that `a` has type `A`, which is not a subtype of `B`. In fact, at run time it turns out that `a` actually contains a `B` object, but the typechecker cannot know this in general (even though conceivably it could figure it out for this particularly simple program) so its rules say to reject the program.

(b) `C1` implements `I1` and `I3`. `C2` implements `I1`, `I2`, `I3`, and `I4`. `C3` implements only `I3` (even though from the body of the function we can see that it actually returns a `B` and therefore really would behave like `I4.f` is supposed to—again, a typechecker cannot know about the dynamic behavior of functions in general). `C4` implements `I3` and `I4`.