Elements of Programming Languages
Tutorial 4: Polymorphism and subtyping
Solution notes

1. **Imperative programming**

   (a) $y := x + x$

$$\frac{\overline{3 + 3 \Downarrow 6}}{\sigma, y := x + x \Downarrow \sigma[y := 6]}$$

   (b) $\texttt{if}\ x == y\ \texttt{then}\ x := x + 1\ \texttt{else}\ y := y + 2$

$$\frac{\dfrac{3 \Downarrow 3 \quad 4 \Downarrow 4}{3 == 4 \Downarrow \texttt{true}} \quad \dfrac{4 + 2 \Downarrow 6}{\sigma, y := y + 2 \Downarrow \sigma[y := 6]}}{\texttt{if}\ x == y\ \texttt{then}\ x := x + 1\ \texttt{else}\ y := y + 2 \Downarrow \sigma[y := 6]}$$

   (c) $\texttt{while}\ x < y\ \texttt{do}\ x := x + 1$

$$\frac{3 < 4 \Downarrow \texttt{true} \quad \dfrac{3 + 1 \Downarrow 4}{\sigma, x := x + 1 \Downarrow \sigma[x := 4]} \quad \dfrac{4 < 4 \Downarrow \texttt{false}}{\sigma[x := 4], \texttt{while}\ x < y\ \texttt{do}\ x := x + 1 \Downarrow \sigma[x := 4]}}{\sigma, \texttt{while}\ x < y\ \texttt{do}\ x := x + 1 \Downarrow \sigma[x := 4]}$$

2. **Subtyping and type bounds**

   (a)

$$Sub1 <: Super \qquad Sub2 <: Super$$

   (b) i. $Sub1 \times Sub2 <: Super \times Super$ This holds:

$$\frac{\overline{Sub1 <: Super} \quad \overline{Sub2 <: Super}}{Sub1 \times Sub2 <: Super \times Super}$$

   ii. $Sub1 \to Sub2 <: Super \to Super$ This does not hold since $Super <: Sub1$ doesn't.

$$\frac{\overset{???}{Super <: Sub1} \quad \overline{Sub2 <: Super}}{Sub1 \to Sub2 <: Super \to Super}$$

   iii. $Super \to Super <: Sub1 \to Sub2$ This does not hold since $Super <: Sub2$ doesn't.

$$\frac{\overline{Sub1 <: Super} \quad \overset{???}{Super <: Sub2}}{Super \to Super <: Sub1 \to Sub2}$$

   iv. $Super \to Sub1 <: Sub2 \to Super$ This holds:

$$\frac{\overline{Sub1 <: Super} \quad \overline{Sub2 <: Super}}{Super \to Sub1 <: Sub2 \to Super}$$

   v. ($\star$) $(Sub1 \to Sub1) \to Sub2 <: (Super \to Sub1) \to Super$ This holds:

$$\frac{\dfrac{\overline{Sub1 <: Super} \quad \overline{Sub1 <: Sub1}}{Super \to Sub1 <: Sub1 \to Sub1} \quad \overline{Sub2 <: Super}}{(Sub1 \to Sub1) \to Sub2 <: (Super \to Sub1) \to Super}$$

(c) If we call `f1` on `Sub2(true)` then the result has type `Super`. We can't access the `b` field because of a type mismatch.

(d) This typechecks, because in either case we return `x` which has type `A`. If we apply it to a value of type `Sub1` or `Sub2` we get the same value back. If we apply it to `42 : Int` then we get a match error.

(e) This typechecks, because as for `f2` we return `x : A` in either case. However, now if we apply to `Sub1` or `Sub2` we get the same value back, while if we apply to something of an unrelated type we get a type error. This seems to solve the problem.

3. **Lists**

(a) Typing rules:

$$\frac{}{\Gamma \vdash \texttt{nil} : \texttt{list}[\tau]} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{list}[\tau]}{\Gamma \vdash e_1 :: e_2 : \texttt{list}[\tau]}$$

(b)

$$\Lambda\alpha.\Lambda\beta.\lambda f{:}\alpha \to \beta.\texttt{rec}\ map(x{:}\texttt{list}[\alpha]).\\ \texttt{case}_{\texttt{list}}\ x\ \texttt{of}\ \{\texttt{nil} \Rightarrow \texttt{nil}\ ;\ x :: xs \Rightarrow (fx) :: map(xs)\}$$

Notice that the `rec` only handles the inner function call.

(c)

$$\dfrac{\dfrac{\dfrac{}{\vdash map : \forall A.\forall B.(A \to B) \to (\texttt{list}[A] \to \texttt{list}[B])}}{\vdash map[\texttt{int}] : \forall B.(\texttt{int} \to B) \to (\texttt{list}[\texttt{int}] \to \texttt{list}[B])} \quad \dfrac{\dfrac{x{:}\texttt{int} \vdash x{:}\texttt{int} \quad x{:}\texttt{int} \vdash 1 : \texttt{int}}{x{:}\texttt{int} \vdash x + 1 : \texttt{int}}}{\vdash \lambda x{:}\texttt{int}.x + 1 : \texttt{int} \to \texttt{int}}}{\dfrac{\vdash map[\texttt{int}][\texttt{int}] : (\texttt{int} \to \texttt{int}) \to (\texttt{list}[\texttt{int}] \to \texttt{list}[\texttt{int}])}{\vdash map[\texttt{int}][\texttt{int}](\lambda x.x + 1) : \texttt{list}[\texttt{int}] \to \texttt{list}[\texttt{int}]} \quad \dfrac{\vdash 2 : \texttt{int} \quad \vdash \texttt{nil} : \texttt{list}[\texttt{int}]}{\vdash (2 :: \texttt{nil}) : \texttt{list}[\texttt{int}]}}{\vdash map[\texttt{int}][\texttt{int}](\lambda x.x + 1)(2 :: \texttt{nil}) : \texttt{list}[\texttt{int}]}$$

(d) Evaluation rules:

$$\frac{}{\texttt{nil} \Downarrow \texttt{nil}} \qquad \frac{e_1 \Downarrow v_2 \quad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2}$$

$$\frac{e_0 \Downarrow \texttt{nil} \quad e \Downarrow v}{\texttt{case}_{\texttt{list}}\ e_0\ \texttt{of}\ \{\texttt{nil} \Rightarrow e\ ;\ \cdots\} \Downarrow v} \qquad \frac{e_0 \Downarrow v_1 :: v_2 \quad e_1[v_1/x, v_2/y] \Downarrow v}{\texttt{case}_{\texttt{list}}\ e_0\ \texttt{of}\ \{\cdots\ ;\ x :: y \Rightarrow e\} \Downarrow v}$$

(e) This question is intended to provoke discussion; the answer to this question depends on what "definable" means, which is not a concept we have carefully defined.

In one sense, lists and the list operations are not definable, because there is no way to create a data structure of infinite "size" using just pairs and sums (e.g. for any finite program, we can bound the maximum size of a data structure the program constructs.)

In another reasonable sense, lists could be defined (in principle) by encoding pairs, sums, and lists into natural numbers (assuming infinite precision arithmetic). However, this too might be unsatisfactory, since we would not easily be able ot do this uniformly in the type of list elements τ, and it would be very difficult to translate a polymorphic program operating over lists.