Elements of Programming Languages
Tutorial 4: Subtyping and polymorphism
Solution notes

1. **Subtyping and type bounds**

   (a)

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

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

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

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

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

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

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

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

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

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

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

   (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.

2. **Typing derivations** Construct typing derivations for the following expressions, or argue why they are not well-formed:

   (a) $\Lambda A.\lambda x{:}A.x + 1$ **does not typecheck because** A **is not** `int`**.**

   $$\frac{\frac{\frac{\overset{???}{x{:}A \vdash x : \mathtt{int}} \quad \frac{}{x{:}A \vdash 1 : \mathtt{int}}}{x{:}A \vdash x + 1 : \mathtt{int}}}{\vdash \lambda x{:}A.x + 1 : A \to \mathtt{int}}}{\vdash \Lambda A.\lambda x{:}A.x + 1 : \forall A.A \to \mathtt{int}}$$

(b) ($\star$) $\Lambda A.\lambda x{:}A \times A.\texttt{if fst}\ x\ ==\ \texttt{snd}\ x\ \texttt{then fst}\ x\ \texttt{else snd}\ x$ (and how does its well-formedness depend on the typing rule for equality?)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \texttt{fst}\ x : A} \quad \dfrac{\dfrac{}{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \texttt{snd}\ x : A}}{\Gamma \vdash \texttt{fst}\ x == \texttt{snd}\ x : \texttt{bool}} \quad \dfrac{\dfrac{}{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \texttt{fst}\ x : A} \quad \dfrac{\dfrac{}{\Gamma \vdash x{:}A \times A}}{\Gamma \vdash \texttt{snd}\ x : A}}{\Gamma \vdash \texttt{if fst}\ x == \texttt{snd}\ x\ \texttt{then fst}\ x\ \texttt{else snd}\ x : A}}{\vdash \lambda x{:}A \times A.\texttt{if fst}\ x == \texttt{snd}\ x\ \texttt{then fst}\ x\ \texttt{else snd}\ x : A \times A \to A}}{\vdash \Lambda A.\lambda x{:}A \times A.\texttt{if fst}\ x == \texttt{snd}\ x\ \texttt{then fst}\ x\ \texttt{else snd}\ x : \forall A.A \times A \to A}$$

where $\Gamma = x{:}A \times A$. **this only works because we have defined $==$'s typing rule so that any two values of the same type can be compared for equality, including two values of an unknown type A. However, if $==$ is restricted to base types (as in Coursework 1) then we cannot do this.**

3. **Evaluation derivations**

   Construct evaluation derivations for the following expressions, or explain why they do not evaluate:

   (a) $(\Lambda A.\lambda x{:}A.x + 1)[\texttt{int}]\ 42$ **Notice that this does not typecheck, but still evaluates OK.**

$$\dfrac{\dfrac{\dfrac{}{\Lambda A.\lambda x{:}A.x + 1 \Downarrow \Lambda A.\lambda x{:}A.x + 1} \quad \dfrac{}{\lambda x{:}\texttt{int}.x + 1 \Downarrow \lambda x{:}\texttt{int}.x + 1}}{(\Lambda A.\lambda x{:}A.x + 1)[\texttt{int}] \Downarrow \lambda x{:}\texttt{int}.\ x + 1} \quad \dfrac{}{42 \Downarrow 42} \quad \dfrac{\vdots}{42 + 1 \Downarrow 43}}{(\Lambda A.\lambda x{:}A.x + 1)[\texttt{int}]\ 42 \Downarrow 43}$$

   (b) $(\Lambda A.\lambda x{:}A.x + 1)[\texttt{bool}]\ \texttt{true}$

   This does not typecheck, and does not evaluate either, because when we try to add `true` to 1 we get stuck.

$$\dfrac{\dfrac{\dfrac{}{(\Lambda A.\lambda x{:}A.x + 1) \Downarrow (\Lambda A.\lambda x{:}A.x + 1)} \quad \dfrac{}{\lambda x{:}\texttt{bool}.x + 1 \Downarrow \lambda x{:}\texttt{bool}.x + 1}}{(\Lambda A.\lambda x{:}A.x + 1)[\texttt{bool}] \Downarrow \lambda x{:}\texttt{bool}.x + 1} \quad \dfrac{}{\texttt{true} \Downarrow \texttt{true}} \quad \dfrac{???}{\texttt{true} + 1 \Downarrow ???}}{(\Lambda A.\lambda x{:}A.x + 1)[\texttt{bool}]\ \texttt{true} \Downarrow ??}$$

4. ($\star$) **Lists and polymorphism**

   (a)

$$\begin{array}{l}\Lambda A.\Lambda B.\lambda f{:}A \to B.\texttt{rec}\ \mathit{map}(x{:}\texttt{list}[A]).\\ \quad \texttt{case}_{\texttt{list}}\ x\ \texttt{of}\ \{\texttt{nil} \Rightarrow \texttt{nil}\ ;\ x :: xs \Rightarrow (f x) :: \mathit{map}(xs)\}\end{array}$$

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

   (b)

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

   (c) 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 to do this uniformly in the type of list elements τ, and it would be very difficult to translate a polymorphic program operating over lists.