fun f x = (* infer val f : int -> int *)
  if x > 3
  then 42
  else x * 2
(*
fun g x = (* report type error *)
  if x > 3
  then true
  else x * 2
val x = 42 (* val x : int *)
fun f (y, z, w) = (* y must be bool *)
  if y
  then z + x
  else 0
  (* z must be int *)
  (* both branches have same type *)
  (* I must return an int *)
  (* I must take a bool * int * ANYTHING *)
  so val f : bool * int * 'a -> int *)
(*
  f : T1 -> T2 [must be a function; all functions take one argument]
  x : T2 [must have type of f's argument]
  y : T3
  z : T4
T1 = T3 * T4 [else pattern-match in val-binding doesn’t type-check]
T3 = int [because (abs y) where abs : int -> int]
T4 = int [because add z to an int]
So T1 = int * int
So (abs y) + z : int, so let-expression : int, so body : int, so T2=int
So f : int * int -> int *)
fun f x = (*
  let val (y,z) = x in
  (abs y) + z
end
(*
  sum : T1 -> T2 [must be a function; all functions take one argument]
  xs : T1 [must have type of f’s argument]
  y : T3
  z : T4
T1 = T3 * T4 [else pattern-match in val-binding doesn’t type-check]
T3 = int [because x : T3 and is argument to addition]
T4 = int [because result of recursive call is argument to addition]
So T1 = int * int
So x + y + z : int, so case-expression : int, so body : int, so T2=int
So T2 = int [because of recursive call is argument to addition]
sum xs’ type-checks because xs’ has type T3 list and T1 = T3 list
sum-case-expression type-checks because both branches have type int
  from T1 = T3 list and T3 = int, we know T1 = int list
  from that and T2 = int, we know f : int list -> int *)
fun sum xs = (*
  case xs of
    [] => 0
  | x::xs’ => x + (sum xs’)
  *)
\[ f : T_1 \\
g : T_2 \\
x : T_4 \]

from body of compose being a function, \( T_3 = T_4 \rightarrow T_5 \) for some \( T_4 \) and \( T_5 \)
from \( g \) being passed \( x \), \( T_2 = T_4 \rightarrow T_6 \) for some \( T_6 \)
from \( f \) being passed result of \( g \), \( T_1 = T_6 \rightarrow T_7 \) for some \( T_7 \)
from \( f \) being body of anonymous function, \( T_7 = T_5 \)

putting it all together:
- \( T_1 = T_6 \rightarrow T_5 \), \( T_2 = T_4 \rightarrow T_6 \), and \( T_3 = T_4 \rightarrow T_5 \)

so compose: \((T_6 \rightarrow T_5) \times (T_4 \rightarrow T_6) \rightarrow (T_4 \rightarrow T_5)\)

now replace unconstrained types /consistently/ with type variables:
\((\alpha \rightarrow \beta) \times (\gamma \rightarrow \alpha) \rightarrow (\gamma \rightarrow \beta)\)

\[
\text{fun compose } (f, g) = \lambda x \Rightarrow f (g x)
\]

(**** the value restriction (important, but optional material) ****)

(* this first line is not polymorphic so next two lines do not type-check *)

\[
\text{val } r = \text{ref } \text{NONE}
\]

(*
\[
\text{val } _ = r := \text{SOME } "hi"
\]

\[
\text{val } t = 1 + \text{valOf } (!r)
\]

(*

\[
\text{type } \alpha \text{ foo } = \alpha \text{ ref}
\]

\[
\text{val } f : \alpha \rightarrow \alpha \text{ foo } = \text{ref}
\]

\[
\text{val } r2 = f \text{ NONE } (* \text{ also need value restriction here } *)
\]

(* where the value restriction arises despite no mutation *)

\[
\text{val } \text{pairWithOne } = \text{List.map } (\lambda x \Rightarrow (x,1))
\]

(* a workaround *)

\[
\text{fun } \text{pairWithOne2 } xs = \text{List.map } (\lambda x \Rightarrow (x,1)) \text{ xs}
\]