Q1.1

((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3)-

Assign type variables for every sub expression:

|  |  |
| --- | --- |
| ((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3) | T0 |
| (lambda (x1 y1) (if (> x1 y1) #t #f)) | T1 |
| (if (> x1 y1) #t #f) | T2 |
| (> x1 y1) | T4 |
| #t | T5 |
| #f | T6 |
| > | T7 |
| x1 | T8 |
| x2 | T9 |
| 8 | T10 |
| 3 | T11 |

Construct type equations:

|  |  |
| --- | --- |
| ((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3) | T1 = [T10\*T11 →T0] |
| (lambda (x1 y1) (if (> x1 y1) #t #f)) | T1 = [T8\*T9→T2] |
| (if (> x1 y1) #t #f) | T2 = T5 |
| (> x1 y1) | T4 = Boolean |
| #t | T5 = Boolean |
| #f | T6 = Boolean |
| > | T7 = [Number\*Number→Boolean] |
| 8 | T10 = Number |
| 3 | T11 = Number |

Solving the equations:

|  |  |
| --- | --- |
| ~~T~~~~1~~ ~~= [T~~~~10~~~~\*T~~~~11~~ ~~→~~~~T~~~~0~~~~]~~ | T1 = [T10\*T11 →T0] |
| T1 = [T8\*T9→T2] |
| T2 = T5 |
| T4 = Boolean |
| T5 = Boolean |
| T6 = Boolean |
| T7 = [Number\*Number→Boolean] |
| T10 = Number |
| T11 = Number |

|  |  |
| --- | --- |
| ~~T~~~~1~~ ~~= [T~~~~8~~~~\*T~~~~9~~~~→T~~~~2~~~~]~~ | T1 = [T10\*T11 →T0] |
| T2 = T5 |
| T4 = Boolean |
| T5 = Boolean |
| T6 = Boolean |
| T7 = [Number\*Number→Boolean] |
| T10 = Number |
| T11 = Number |
| T8 = T10 |
| T9 = T11 |
| T2 = T0 |

|  |  |
| --- | --- |
| ~~T~~~~2~~ ~~= T~~~~5~~ | T1 = [T10\*T11 →T0]  T2 = T5  T4 = Boolean |
| ~~T~~~~4~~ ~~= Boolean~~ |
| T5 = Boolean |
| T6 = Boolean |
| T7 = [Number\*Number→Boolean] |
| T10 = Number |
| T11 = Number |
| T2 = T0 |

|  |  |
| --- | --- |
| ~~T~~~~5~~ ~~= Boolean~~ | T1 = [T10\*T11 →T0]  T2 = Boolean  T4 = Boolean  T5 = Boolean  T6 = Boolean |
| ~~T~~~~6~~ ~~= Boolean~~ |
| T7 = [Number\*Number→Boolean] |
| T10 = Number |
| T11 = Number |
| T2 = T0 |

|  |  |
| --- | --- |
| ~~T~~~~7~~ ~~= [Number\*Number→Boolean]~~ | T1 = [T10\*T11 →T0]  T2 = Boolean  T4 = Boolean  T5 = Boolean  T6 = Boolean  T7 = [Number\*Number→Boolean] |
| T10 = Number |
| T11 = Number |
| T2 = T0 |

|  |  |
| --- | --- |
| ~~T~~~~10~~ ~~= Number~~ | T1 = [Number\*Number→T0]  T2 = Boolean  T4 = Boolean  T5 = Boolean  T6 = Boolean  T7 = [Number\*Number→Boolean]  T10 = Number  T11 = Number |
| ~~T~~~~11~~ ~~= Number~~ |
| T2 = T0 |

|  |  |
| --- | --- |
| ~~T~~~~2~~ ~~= T~~~~0~~ | T1 = [Number\*Number→Boolean]  T2 = Boolean  T4 = Boolean  T5 = Boolean  T6 = Boolean  T7 = [Number\*Number→Boolean]  T10 = Number  T11 = Number  T0 = Boolean |

T0 = Boolean => ((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3) is of type Boolean.

Q1.2

a. {f:[T1->T2], x: T1} |- (f x): T2

True.

In the given environment; x is from type T1, and f is a function from T1 to T2.

Hence, the output of provoking f on x, is from type T2.

b. {f:[T1->T2] ,g: [T2->T3]}, x: T2} |- (f g x): T3

False.

(f g x) is not a valid expression because f accept 1 operand of type T1, hence False

c. {f:[T2->T1],g: [T1->T2], x: T1}|- (f (g x)): T1

True.

In the given environment x is from type T1, and g operates on T1 and output T2 => g(x) : T2

f is a function from T2 to T1 => g(x) is in the range of f so the operation is valid, and the output will be from type T1

=> f (g(x))

x:T1

=>g(x) :T2

=>f(g(x)) : T1

d. {f:[T2->Number], x: Number}|- (f x x): Number

False.

f doesn’t operate on 2 operands, hence the expression is invalid.

3.

a. cons :: [ T1\*T2 -> Pair <T1,T2>]

b. car :: [ Pair<T1,T2> -> T1]

c. cdr :: [ Pair<T1,T2> -> T2]

4.

(Define f (lambda (x) (values x x x)))

[T1-> (T1 \* T1 \* T1)]

If x:T1 => (values x x x) = (x \* x \* x )

5. Write the MGU of the following expressions, or state that there is no such MGU.

a. T1 , T2 => {T1 = T2}

b. Number , Number => {}

c. [T1\*[T1->T2]->Number] , [[T3->Number]\*[T4->Number]->Number]

=> {T1=T4=[T3->Number],T2=Number}

d. [T1->T1] , [T1->[Number->Number]] =>

{T1 = [Number -> Number]}

There is no unifier that will satisfies those expressions.

Q2.3

(define f (number -> (number \* number))

(lambda ((x: number)): (number \* number)

(values x (+ x 1))))

(define g (T -> (string \* T))

(lambda (x: T):(string \* T)

(values “x” x)))

Q4.b

The use of promises helps us write a cleaner code, using chained promises.  
Also we have the ability to handle errors of few promises with one catch.