

Reading

1. Read chapter 6 (Types) of the text.
2. “A history of Haskell: Being lazy with class”, Section 3 (skip 3.9), Section 6 (skip 6.4 and 6.7)
“How to Make Ad Hoc Polymorphism less ad hoc”, Sections 1 - 7
“Real World Haskell”, Chapter 6: Using typeclasses (don’t worry about JSON example).

Problems

1. ML Types

Explain the ML type for each of the following declarations:

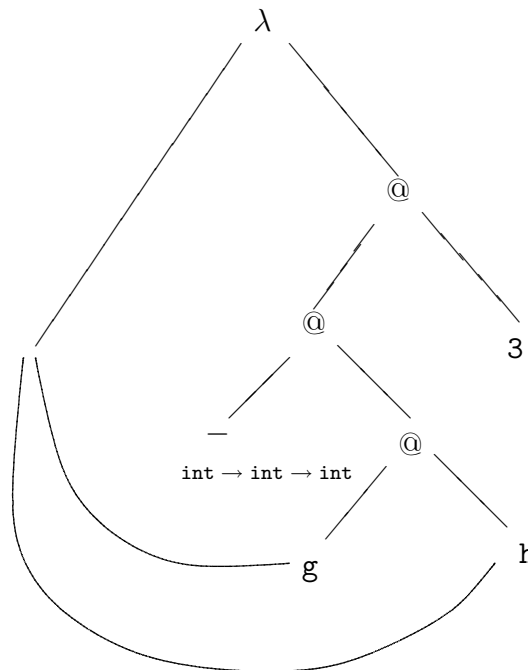
- (a) `fun a(x,y) = x div y + 2;`
- (b) `fun b(x,y) = x + y / 2.0;`
- (c) `fun c(x, y) = fn z => x(z) + y + 2.0;`
- (d) `fun d(f,x) = f(f(x));`
- (e) `fun e(w, x, y, z) = if w(x) then x(y) else z;`

Since you can simply type these expressions into an ML interpreter to determine the type, be sure to write a short *explanation* to show that you understand why the function has the type you give.

2. Parse Graph

Use the parse graph below to calculate the ML type for the function

```
fun f(g,h) = g(h) - 3;
```



3. Polymorphic Fixed Point

A *fixed point* of a function f is some value x such that $x = f(x)$. There is a connection between recursion and fixed points that is illustrated by this Haskell definition of the factorial function `fact :: Int → Int`.

```
y f = f (y f)
factRec g x = case x of
    0 -> 1
    _ -> x * (g (x - 1))

fact = y factRec;
```

The first function, `y`, is a fixed-point operator. The second function, `factRec`, is a function on functions whose fixed point is `fact`, which is the factorial function. Note that `y` is the only recursive function here and both `fact` and `factRec` are not recursively defined. Both of these are Curried functions. Using the Haskell syntax `\x → (...)` for $\lambda x.(...)$, the function `factRec` could also be written as

```
factRec g = \x -> case x of
    0 -> 1
    _ -> x * (g (x - 1))
```

This `factRec` is a function that, when applied to argument `g`, returns a function that, when applied to argument `x`, has the value given by the expression 'if `x=0` then 1 else `x*g(x-1)`'.

- (a) What type will the Haskell compiler deduce for `factRec` and Why?
- (b) What type will the Haskell compiler deduce for `y` and Why?
- (c) Write a function `fibRec` so that the function `fib`, described below, could be written as `fib = y fibRec`.

```
fib n = case n of
    0 -> 0
    1 -> 1
    n -> (fib (n - 1)) + (fib (n - 2))
```

- (d) In pure lambda calculus, the fixed point operator `y` can also be written as

$$y = \lambda f.((\lambda g.f(gg)) (\lambda g.f(gg)))$$

- i. Use β reduction to show that for this lambda expression, $y(f) = f(y(f))$.
- ii. We try to write the function `y` in Haskell as follows

```
y = \f -> (\g -> f (g g)) (\g -> f (g g))
```

However, the Haskell compiler reports a type error when type checking this function definition:

```
Occurs check: cannot construct the infinite type: t = t -> t1
Probable cause: `g' is applied to too many arguments
In the first argument of `f', namely `(g g)'
In the expression: f (g g)
```

Explain Why?

- (e) (BONUS problem) Write a function `reduceRec` so that the function `reduce`, described below, could be written as `reduce = y reduceRec`.

```

reduce f l = case l of
    [] -> undefined
    [x] -> x
    (x:xs) -> f x (reduce f xs)

```

(The definition of 'y' is the same as that mentioned in the beginning of this problem: $y f = f y(f)$).

4. Haskell Type Inference

A friend of yours is working on a programming project in Haskell. Unfortunately, your friend is confused by an compile-time error message and has turned to you for help.

The programming project involves writing a function f that, given a list of positive integers as input, produces a string that is the concatenation of the string representations of each of the integers. For example, $f [1, 2] = "12"$, $f [31, 41, 59, 26] = "31415926"$ and so forth.

Your friend has written the following *buggy* code:

```

concatS :: String -> String -> String
concatS s1 s2 = s1 ++ s2

showI :: Int -> String
showI i = show i

g s n = concatS (showI n) s

foldright h y [] = y
foldright h y (x:xs) = h x (foldright h y xs)

f l = foldright g "" l

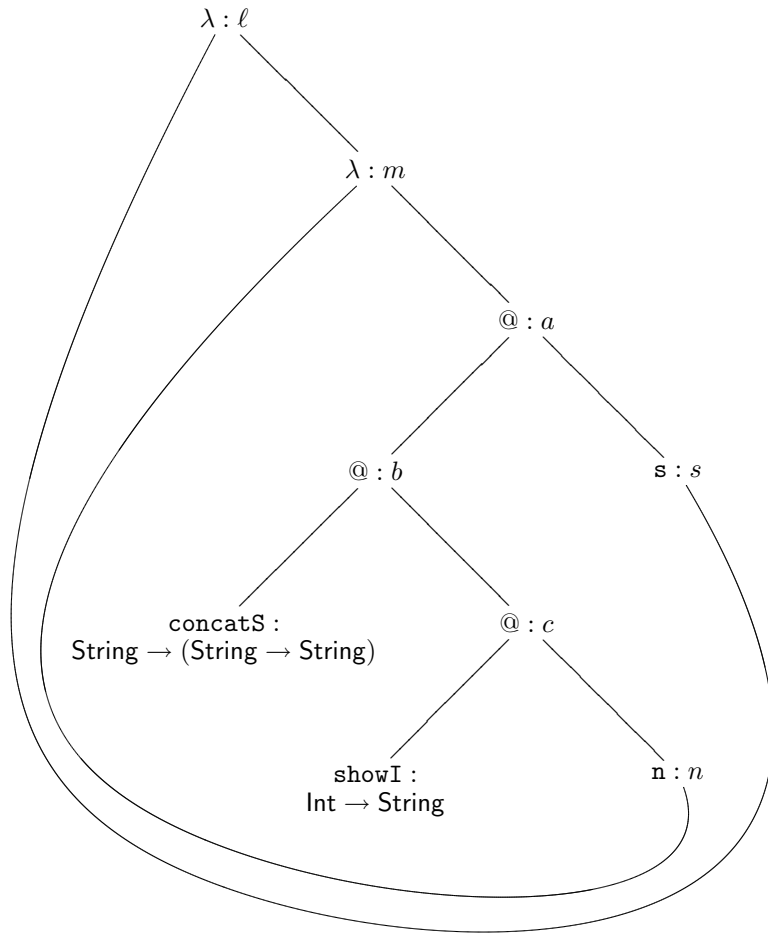
```

`showI` is a function with type $\text{Int} \rightarrow \text{String}$ which converts an integer to a string, and `concatS`, as defined above with type $\text{String} \rightarrow \text{String} \rightarrow \text{String}$, concatenates two strings.

The basic idea, explained in more detail below, is that the Curried function `g` concatenates a string and the string representation of a number. The `foldright` function, explained in part (b) below, is a standard list-manipulation function that recursively uses a function `h` to combine elements of a list. Using these two functions, `f` can be defined by applying the higher-order function `foldright` to the function `g`. In Haskell, `""` is the empty string, which is used in the base case of `foldright` for the empty list.

The following parts of this question ask you to diagnose and fix the problem in the code.

- (c) The Haskell compiler reports that function `g` has type $\text{String} \rightarrow \text{Int} \rightarrow \text{String}$. Using the parse graph and equations below, show how Haskell infers that function `g` has this particular type. Types are already assigned to each node in the graph for you.



Generate constraints from graph

Solve constraints

$\text{Int} \rightarrow \text{String} = \underline{\hspace{2cm}} \rightarrow \underline{\hspace{2cm}}$	$n = \underline{\hspace{2cm}}$
$\text{String} \rightarrow (\text{String} \rightarrow \text{String}) = \underline{\hspace{2cm}} \rightarrow \underline{\hspace{2cm}}$	$c = \underline{\hspace{2cm}}$
$b = \underline{\hspace{2cm}} \rightarrow \underline{\hspace{2cm}}$	$b = \underline{\hspace{2cm}}$
$m = \underline{\hspace{2cm}} \rightarrow \underline{\hspace{2cm}}$	$s = \underline{\hspace{2cm}}$
$\ell = \underline{\hspace{2cm}} \rightarrow \underline{\hspace{2cm}}$	$a = \underline{\hspace{2cm}}$
	$m = \underline{\hspace{2cm}}$
	$\ell = \underline{\hspace{2cm}}$

- (b) The function `foldright` given above correctly implements a *fold right* function. That is, given a function `h`, a value `y` and a list `xs = [x1, ..., xn-1, xn]`, `foldright h y xs` computes the value of the expression:

$$(h\ x_1\ (\dots\ (h\ x_{n-1}\ (h\ x_n\ y))))$$

Using this information and the definition of `foldright` given above, what type will the Haskell compiler assign to `foldright`? For this part of the question it is not necessary to show how you derive the type. (Hint: do both arguments to `h` have to be of the same type?) Recall that in Haskell, the type of lists of elements of type `t` is written `[t]`.

$$\underbrace{(__ \rightarrow __ \rightarrow __)}_{\text{type of } h} \rightarrow \underbrace{__}_{\text{type of } y} \rightarrow \underbrace{[__]}_{\text{type of } xs} \rightarrow \underbrace{__}_{\text{type of } \text{foldright } h\ y\ xs}$$

- (c) Now, given your answers to the previous two parts of the question, why does the function `f` as defined generate a type-check error at compile time?

- (d) Show how to fix the definition of function `g` so that the program as a whole type-checks and correctly implements the specification given above.

`g` _____ = `concatS` _____

5. Haskell Typeclasses and Ambiguous Resolution

Suppose there is a library that defines a datatype `H_XML` in Haskell representing simple XML structures. `H_XML` is an instance of the `Show` and `Read` classes with `show` and `read` functions that print and parse `H_XML` structures as legal XML. Such a library is useful if we want to serialize our data in XML to interoperate with other systems, etc. To use the library for our data structures, we need to give functions to map our data structure into XML. Because there are many different data structures library users might want to serialize, the library defines the type class `XML_Rep` to convert to and from the `H_XML` datatype:

```
data H_XML = Tag { name::String, children::[H_XML]
                 | Value { name::String, value::String }

instance Show H_XML where
  show x = xml_show x

instance Read H_XML where
  read x = xml_read x

class XML_Rep a where
```

```
toXML :: a -> H_XML
fromXML :: H_XML -> a
```

For simplicity's sake, we have used two undefined functions `xml_show` and `xml_read`. You can assume that `xml_show` takes `H_XML` and turns it into a `String`, while `xml_read` reverses this process. Also, all values are represented in `String` form in `H_XML`, so the `Int` value `1` might be represented in `H_XML` as `(Value "Int" "1")`.

- (a) Give instance declarations for `Int` and `String` for `XML_Rep`. Note: You will have to set the flag `-XTypeSynonymInstances` to use the `String` synonym for `[Char]`. You can use the command `:set -XTypeSynonymInstances`.
- (b) Give an instance declaration for the pair `(a, b)` for `XML_Rep`, where the types `a` and `b` both instance the typeclass `XML_Rep`.
- (c) Suppose you wanted to write a `QuickCheck` property to test if converting from an `XML` term to your data structure and back to `XML` is correct:

```
prop_XMLConvert s = s == toXML (fromXML s)
```

This code generates the error message:

```
> typeclass_homework.hs:47:33:
>   Ambiguous type variable `a' in the constraint:
>     `ToXML a'
>     arising from a use of `fromXML' at typeclass_homework.hs:
>       47:33-41
>   Probable fix: add a type signature that fixes these type
>     variable(s)
> Failed, modules loaded: none.
```

In contrast, the property

```
prop_XMLConvert2 i = i == fromXML (toXML i)
```

type checks without error. Explain why the compiler produces the error message in the first case but not the second.