Homework 3

Due 17 October

Handout 3
CS242: Autumn 2012
10 October

Reading

1. Read the revised chapter 6 (Types) of the text. Available on CourseWare under the Types lecture.

2. Read the new chapter 7 (Type Classes) of the text. Available on CourseWare under the Type Classes lecture.

Problems

1. Polymorphic Sorting

Recall the Haskell implementation of quicksort discussed in lecture. The code compiles and runs correctly.

```
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

What is the type of this function? What are the types of the two helper functions, `lesser` and `greater`?

Feel free to use `ghci` to check your answer. You do not have the provide a formal argument; just explain why the average Haskell programmer would expect the code to have these types.

2. 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 $\texttt{fact} :: \texttt{Int} \to \texttt{Int}$.

```
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 λx.(...), 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)`).

3. .... Haskell Type Inference and Program Analysis

You should download the file `Inference.hs` from Courseware for parts (a) and (b) of this question. You should make your edits directly in your copy of this file.

To submit parts (a) and (b), submit your edited version of this file by uploading it to Courseware.

Please make sure your code compiles properly and contains only the changes that we told you to make. We will be grading your code using automated scripts, so if it doesn't compile, then you will get zero points. Also, our scripts cannot grade what you write in comments, so please make sure to un-comment all of the code you want us to grade. If you are working on this with a partner you should both submit a copy of the code.

Submit parts (c) and (d) *on paper,* in class or in the homework drop box.

(a) Give a Haskell expression named `myDecl` of type `Decl` that represents the uHaskell function:

```
f (x,y) = let z = 2 * x in z + y
```

There is a place for you to write your expression in `Inference.hs`.

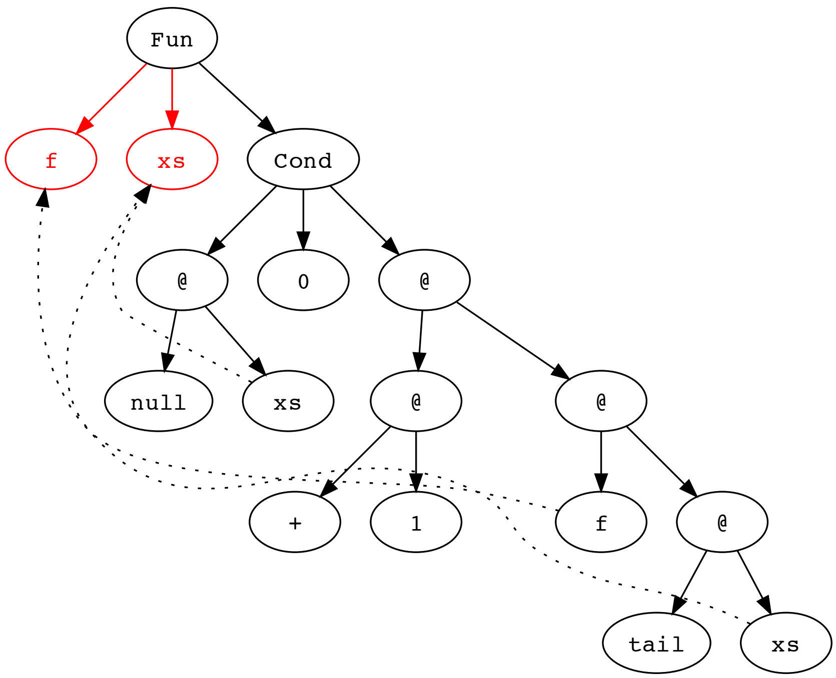
(b) Fill in the missing pieces of the functions `varsInPat` and `freeVarsE`. The function skeletons are in the same file `Inference.hs`.

(c) Draw a parse tree like the ones we saw in lecture for the function `f` defined in part (a). Note that this parse tree is a graphical representation of the data structures defined in part (a). You will need to add a new kind of node for the `let` construct.

(d) Consider the uHaskell function:

```
f xs = if null xs then 0 else 1 + f(tail xs)
```

with the parse tree shown below.



i. Explain the constraints generated for the `Cond` node.

ii. Annotate each node with the constraints it contributes to the type inference problem, assuming:

```
null :: [a] -> Bool
tail :: [a] -> [a]
```

iii. Solve the constraints from part (ii) to produce the type of `f`. Show your work.

4. .... uHaskell Type Inference

A friend of yours is working on a programming project in uHaskell. Unfortunately, your friend is confused by a 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* uHaskell code:

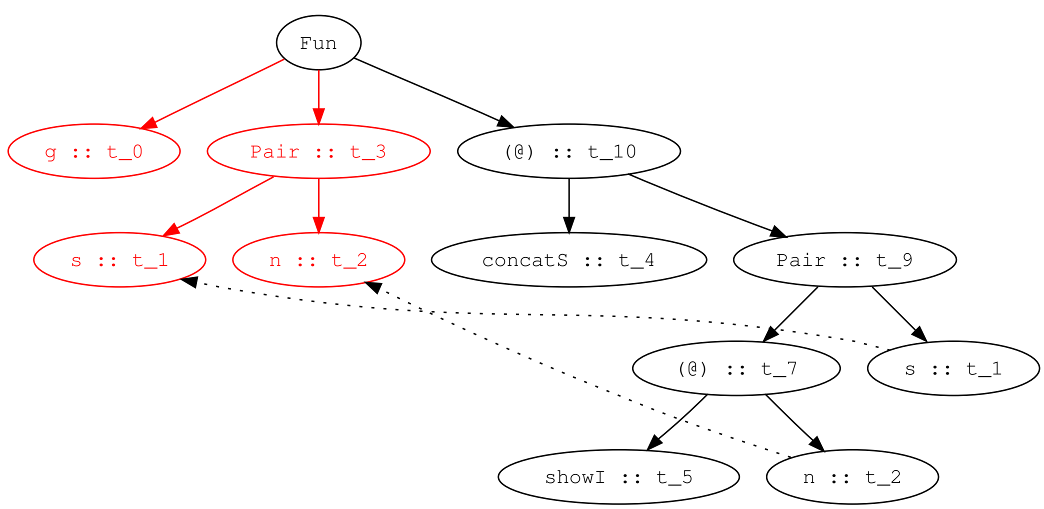
```
concatS :: (String, String) -> String
```

```
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
```

This code includes the function `concatS`, with type `(String, String) -> String`, which concatenates two strings, and `showI`, with type `Int -> String`, which converts an integer to a string. The basic idea behind this code, explained in more detail below, is that the 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. Part (a) ask you to explain how uHaskell determines the type of `g`.

(a) The uHaskell compiler reports that function `g` has type `(String, Int) -> String` We would like to understand how the uHaskell compiler infers this type.
Using the parse graph and type variables below, determine the type of the uHaskell function `g`. Use the type variables that have already been assigned to each node in the graph. You only need to generate and solve enough constraints to clearly show that you have found the correct type for the function.



(b) The function `foldright` given above correctly implements a *fold right* function. That is, given a function `h`, a value `y` and a list $\mathtt{xs} = [\mathtt{x}_1, \ldots, \mathtt{x}_{n-1}, \mathtt{x}_n]$, `foldright h y xs` computes the value of the expression:

$$(\mathtt{h}\ (\mathtt{x}_1, (\ \cdots\ (\mathtt{h}\ (\mathtt{x}_{n-1}, (\mathtt{h}\ (\mathtt{x}_n, \mathtt{y})))))))$$

Using this information and the definition of `foldright` given above, what type will the uHaskell compiler assign to `foldright`? For this part of the question it is not necessary to show how you derive the type. Recall that in uHaskell, the type of lists of elements of type `t` is written `[t]`.

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

(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 Type Classes

This problem involves the following Haskell program on type class `Comp`. First, we define a ternary `Ordering` which stores the result of a comparison: `LT` (less-than), `EQ` (equal), `GT` (greater-than). Functions `compareInt` and `compareChar` compare the values of a pair of `Int` and `Char` respectively.

```
data Ordering = LT | EQ | GT

compareInt :: Int -> Int -> Ordering
compareInt x y = if (x < y) then LT else (if (x > y) then GT else EQ)

compareChar :: Char -> Char -> Ordering
compareChar x y = if (x < y) then LT else (if (x > y) then GT else EQ)
```

We define a new type class `Comp` comprised of a single operator `?=`.

```
class Comp a where
  (?=) :: a -> a -> Ordering

-- Integer comparison
instance Comp Int where
  (?=) x y = compareInt x y

-- Character comparison
instance Comp Char where
  (?=) x y = compareChar x y

-- Lists are compared element by element
instance Comp a => Comp [a] where
  (?=) [] [] = EQ
  (?=) (x:xs) [] = GT
  (?=) [] (y:ys) = LT
  (?=) (x:xs) (y:ys) = if ((x ?= y) /= EQ) then (x ?= y) else (xs ?= ys)

-- Tuples are compared by first element then by second element
instance (Comp a, Comp b) => Comp (a,b) where
  (?=) (x1,x2) (y1,y2) = if ((x1 ?= y1) /= EQ) then (x1 ?= y1) else (x2 ?= y2)
```

We further define the following function `f`.

```
f x y = let
  xx = (length x, x)
  yy = (length y, y)
  in ( xx ?= yy )
```

(a) When processing the type class declaration for `Comp`, the Haskell compiler will generate the following internal type and function declaration.

```
data CompD a = MakeCompD (a -> a -> Ordering)
(?=) (MakeCompD comp) = comp
```

For each instance declaration, the compiler will also generate code to construct a corresponding dictionary. Fill in the following dictionary construction code for comparing integers and lists.

```
-- Integer comparison
dCompInt :: CompD Int
dCompInt = ________________________________________

-- List Comparison
dCompList :: CompD a -> CompD [a]
dCompList d = MakeCompD compList where
    compList []     []     = EQ
    compList (x:xs) []     = GT
    compList []     (y:ys) = LT
    compList (x:xs) (y:ys) =
          if ( ((?=)______________________________) /= EQ )
        then ((?=)________________________________________)
        else ((?=)________________________________________)
```

(b) Consider the application

```
r = f "Hello" "World"
```

What implementations of `(?=)` are involved in the computing r? More specifically, the operator `(?=)` is called four times during the execution. How are `(?=)` calls re-written by the compiler? In the space provided below, fill in the parameters passed to `(?=)` during these four calls.

```
(?=) ________________________ (length "Hello","Hello") (length "World","World")

(?=) ________________________  ___________________  ___________________
```

```
(?=) ___________________________    "Hello"   "World"

(?=)  dCompChar  'H'  'W'
```

(c) What is the type of `f`? Explain the inference process that produces this type in English. You do not need to draw an inference diagram.

6. .... Implementing Haskell Typeclasses

Suppose we are interested in considering two Haskell `Int`s `i` and `j` equal if the absolute value of `i` is equal to the absolute value of `j`:

```
(abs i) == (abs j)
```

Suppose we are further interested in considering data structures containing `Int`s as equal if the corresponding `Int`s in those structures are equal up to absolute value. We can use Haskell's type class mechanism to define a new type class `MyEq` comprised of a single operator === that denotes this notion of equality:

```
class MyEq a where
  (===) :: a -> a -> Bool
```

Using an instance declaration, we can make `Int` an instance of this type class:

```
instance MyEq Int where
  i === j  =  abs i == abs j
```

(a) When processing the type class declaration for `MyEq`, the Haskell compiler will generate the following internal type and function declarations:

```
data MyEqD a = MkMyEqD (a -> a -> Bool)
(===) (MkMyEqD eq) = eq
```

Explain what the generated datatype `MyEqD` represents and what the generated function === does. (All of this can be answered in a few sentences.)

(b) Suppose that we are using the following `Tree` data structure and want to compare such trees using the === operator.

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
  deriving (Show)
```

Fill in the following instance declaration to make `Tree`s an instance of the type class `MyEq`:

```
instance ___________ => MyEq (____________) where

  (===) (Leaf v1) (Leaf v2) = v1 === v2

  (===) (Node v1 tl1 tr1)

        (Node v2 tl2 tr2) = ________________________________________

  (===) _ _ = False
```

(c) From such an instance declaration, the compiler will generate code to construct `Tree` dictionaries. Fill in the following dictionary construction code:

```
dMyEqTree :: __________________________

dMyEqTree d = MkMyEqD myEqTree where

  myEqTree (Leaf v1) (Leaf v2) = _________________________

  myEqTree (Node v1 tl1 tr1)

           (Node v2 tl2 tr2)   = _________________________

                                 _________________________

                                 _________________________
```

(d) The `cmp` function compares two values from any type that belongs to the `MyEq` type class and returns a `String` indicating whether the values were equal.

```
cmp :: (MyEq a) => a -> a -> String
cmp t1 t2 = if t1 === t2 then "Equal" else "Not Equal"
```

The value `result`

```
  result = cmp test1 test2
```

uses the function `cmp` to compare two test trees where:

```
   test1 :: Tree Int
   test2 :: Tree Int
```

The compiler will rewrite the `cmp` function and its uses. Explain why it does so.

(e) Assume that the compiler generated a dictionary named `dMyEqInt` for the `Int` instance of `MyEq`:

```
dMyEqInt :: MyEqD Int
```

Fill in the following rewritten versions of the `cmp` function and `result` definition:

```
cmp' :: ______________________________

cmp' ____________ = if _____________________
```

```
                    then "Equal" else "Not Equal"

result' = cmp' ____________________________ test1 test2
```