Type inferencing

Know your type.

2019-09-26

This document contains various type inferencing examples. The world of Haskell is more accessible once you are able to inference types as fast as you can drink your tea or coffee.

Please take note that this document is merely here to provide additional exercises. It is by no means representable for the course or the contents of the course.

Checking your types

Remember that you can check your final answer inside *ghci* via typing:

```
:t +d (functions)

-- For example:
:t +d map (+)
map (+) :: [Integer] -> [Integer -> Integer]
```

The parameter *+d* will simplify the type.

Accessible level

The following exercises are an introduction towards type inferencing. Some contain a few tricks, others are just long and perhaps a bit difficult.

Examples

```
-- Determine the type of: map (1 +)

-- Step 0: Write down the situation.
-- We need to determine the type of map ((+) 1)

map :: (a -> b) -> [a] -> [b]
(+) :: Int -> Int -> Int

-- Step 1: Write down the substitutions.
-- Take note: ((+) 1) is the first argument of map.

a = Int
b = Int

-- Step 2: Solve collisions.
-- Take note: There are no collisions between our substitutions :).

-- Step 3: Write down the remaining type.
-- Take note: The first argument is in use, hence we skip that in the type.

-- The remaining type is:
[a] -> [b]

-- Which we substitute accordingly:
[Int] -> [Int]
```

```
-- Determine the type of: foldr map

-- Step 0: Write down the situation.
-- Take note: We use different symbols for both types.

foldr :: (a -> b -> b) -> b -> [a] -> b
map :: (c -> d) -> [c] -> [d]
```

```
-- Step 1: Write down the substitutions.
a = (c -> d)
b = [c]
b = [d]

-- Step 2: Solve collisions.
-- Take note: we have two b's that hold different values.

-- We have:
b = [c]
b = [d]

-- We can solve this by substituting:
c = d

-- Substituting d for c gives us our resulting types:
a = (d -> d)
b = [d]

-- Step 3: Write down the remaining type.
-- Take note: The first argument is in use, hence we skip that in the type.

-- The remaining type is:
b -> [a] -> b

-- Which we substitute accordingly:
[d] -> [(d -> d)] -> [d]
```

Exercises

```
    kjhjb:t (+) 1

:t map even

:t map (+)

:t map maximum

:t map concat

:t map lines

:t filter even
```

```
:t map filter

-- The following functions may appear in Data.List
-- (https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-List.html)

:t length . init

:t reverse . reverse

-- The function permutations is part of Data.List
:t concat . (map permutations)

-- From the course website:

:t foldr map

:t map . foldr

:t concat . concat

:t map (map map)
```

Novelty level

The following exercises contain types that you're not familiar with yet. This is intended. The goal of these exercises is to realise that they all work the same - regardless of your familiarity.

Take note to no longer use the *+d* parameter when checking your types.

Definitions used

```
-- Given the following functions:
flip :: (a -> b -> c) -> b -> a -> c
curry :: ((a -> b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c

-- Given the data types:
data Point = Pt Float Float
data Maybe a = Nothing | Just a

-- From Data.List:
find :: (a -> Bool) -> [a] -> Maybe a

-- From Data.Maybe:
isJust :: Maybe a -> Bool  -- Take note: bad style. Use pattern matching instead.
fromMaybe :: a -> Maybe a -> a
mapMaybe :: (a -> Maybe b) -> [a] -> [b]

-- From Control.Monad:
(>>=) :: m a -> (a -> m b) -> m b
```

Exercises

```
:t map (flip foldr)

:t map (uncurry (+))

-- Remember to first define the Point data type in GHCi when checking.
:t \xs -> [ (Pt 1.0) x | x <- xs]

:t mapMaybe (find even)

:t map (fromMaybe 1)
```

```
:t map (uncurry fromMaybe)

:t foldr (\a acc -> (+)(fromMaybe 0 a) acc)

:t flip (>>=)

:t filter isJust

:t map (curry (\(a, b) -> a + b))

:t foldr (:)

:t \xs -> [sum x | x <- xs]
```

Advanced level

The following are implemented functions taken from various packages. By now you should be able to type check the functions by merely looking at them.

Take note that we both use known and unknown type definitions. As an additional exercise, you can deduce their types and use GHCi to check if you were correct. Or just look them up.

Also take note that eta-reduction is applied to some of the functions.

Some of the following function definitions contain type errors. Find and solve them.

```
-- from Data.Maybe
catMaybes :: [Maybe a] -> [a]
catMaybes ls = [x | Just x <- ls]

fromMaybe :: a -> Maybe a -> a
fromMaybe d x = case x of
  Nothing -> d
  Just v  -> v

listToMaybe :: [a] -> Maybe a
listToMaybe [] =  Nothing
listToMaybe (a:_) =  Just a

-- from Data.List
find :: (a -> Bool) -> [a] -> Maybe a
find p = filter p

deleteBy                :: (a -> a -> Bool) -> a -> [a] -> [a]
deleteBy _  _ []        = []
deleteBy eq x (y:ys)    = if x `eq` y then ys else y : deleteBy eq x ys

intersperse             :: a -> [a] -> [a]
intersperse _   []      = []
intersperse sep (x:xs)  = x : prependToAll sep xs
```

```haskell
zipWith4                :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
zipWith4 z (a:as) (b:bs) (c:cs) (d:ds)
                        =  z a b c d : zipWith4 z as bs cs ds
zipWith4 _ _ _ _ _      =  []

zip4                    :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip4                    =  zipWith4 (,,,)

group                   :: Eq a => [a] -> [[a]]
group                   =  groupBy (==)

unlines                 :: [String] -> String
unlines                 =  concatMap (++ "\n")

words   :: String -> [String]
words s =  case dropWhile isSpace s of
             "" -> []
             s' -> w : words s''
               where (w, s'') = break isSpace s'
```

Definitions were taken from:

- https://hackage.haskell.org/package/base-4.6.0.1/docs/src/Data-Char.html
- https://hackage.haskell.org/package/base-4.6.0.1/docs/src/Data-List.html
- https://hackage.haskell.org/package/base-4.6.0.1/docs/src/Data-Maybe.html

The following functions contained type errors:

```
-- The following function does filter and therefore 'finds', but it never
selects the first element and turns it into a maybe.
find :: (a -> Bool) -> [a] -> Maybe a
find p = filter p

-- It should be (using the function given above in the list, hurr):
find :: (a -> Bool) -> [a] -> Maybe a
find p = listToMaybe . filter p

-- The following function provides a type for first argument of zipWith4
-- of: (a -> b -> c), but it requires (a -> b -> c -> d).
zip4                    :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip4                    =  zipWith4 (,,)

-- It should be:
zip4                    :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip4                    =  zipWith4 (,,,)

-- Take note that (,), (,,) and (,,,) are constructors for the tuple data
type!
```