Finding the Type of Haskell Expressions

Lucas W.

2024-12-06 13:31:13Z

If you just want a one-page explained example of the algorithm see Ex1 on page 10.

0 Overview / What this is for

A common task given in FMFP is to determine the *type* of an expression.

Sometimes you'll be asked to find the type using a formal proof tree, other times they will give you an odd-looking Haskell expression and just ask for the answer, where you can usually (somehow) figure out the type on your own, by fast means.[1]

This document is for when you need a method that lets you systematically find the type of an (arbitrarily complicated) expression without guessing (*and* without using a full-blown proof tree).

It could be useful at an exam where for easier cases like[2] `(\x y -> x y y)` where you can guess the type yourself by looking at it long enough, while for trickier ones like[3] `map . (,)` you'll be able to use this method to get a reliable result. 🫠

Preview `uncurry curry`

This is what the algorithm looks like when applied to `uncurry curry` *(See task Ty5):*

Assume

- `uncurry :: (a -> b -> c) -> (a,b) -> c`
- `curry :: ((a,b) -> c) -> a -> b -> c`

```
{- Find type of ‘uncurry curry‘ -}
uncurry :: (      x     -> y ->    z    ) -> (x,y) -> z
curry   :: ((a,b) -> c) -> a -> (b -> c)
-- x = ((a,b) -> c),  y = a,  z = (b -> c),  therefore:
uncurry curry :: (x, y) -> z
               = ((a,b) -> c, a) -> (b -> c)
               = ((a,b) -> c, a) -> b -> c
```

[1] You could technically *always* rely on the formal approach, but that's obviously overkill; Some practice makes it much easier to eyeball a type.

[2] See 1.A ii) for the type derivation of `(\x y -> x y y)`.

[3] See 1.A iii) for the type derivation of `map . (,)`.

Contents

0 Overview / What this is for **1**
Preview `uncurry curry` . . . 1

1 Algorithm Information **5**
1.1 Useful Haskell Details . . . 5
1.1.1 Associativity . . . 5
1.1.2 Operators and operator sections . . . 5
1.1.3 Several Lambda Inputs . . . 5
1.1.4 Polymorphic Type Variables . . . 6
1.1.5 Typeclasses . . . 6
1.2 Typing Function Applications . . . 6
1.3 Typing Lambdas . . . 8
1.4 Explained Examples . . . 10
Ex1 `uncurry (flip (,))` . . . 10
Ex2 `(\x -> (head x) True)` . . . 11

2 Tasks and Solutions **12**
2.1 Exercise Sheet 4 . . . 12
2.(a) `(\x y z -> (x y) z)` . . . 12
2.(b) `(\f -> fst (f,f))` . . . 13
2.(c) `(\p f -> filter p . map f)` . . . 14
2.2 Exercise Sheet 5 . . . 15
2.(a) 1. `(\x y z -> x (y z))` . . . 15
2.(a) 2. `(\f -> (\h -> (f,h)))` . . . 15
2.(a) 3. `(\x -> x (<))` . . . 16
2.(a) 4. `(.) . (.)` . . . 16
2.(a) 5. `flip id` . . . 17
2.3 Exercise Sheet 7 . . . 17
1.(a) 1. `(\x y z -> (y x, x z))` . . . 17
1.(a) 2. `(\x y -> (\z -> y))` . . . 18
1.(a) 3. `map (1:)` . . . 18
1.(a) 4. `(\x ys -> inits (map (x <) ys))` . . . 19
2.4 Exam FS23 . . . 20
1.A i) `(\x y -> x (y,y))` . . . 20
1.A ii) `(.) zipWith (\x y -> x y)` . . . 21
2.5 Exam FS22 . . . 22
1.A 1. `foldl (==)` . . . 22
1.A 2. `(\x y -> (y x, x y))` . . . 22
2.6 Exam HS21 . . . 23
1.A i) `zipWith elem` . . . 23
1.A ii) `(\x -> map x [x])` . . . 23
1.A iii) `(\x y z -> (x z) (y z))` . . . 24
2.7 Exam FS21 . . . 24
1.A i) `(\x y -> y y x)` . . . 24
1.A ii) `(\x y -> x y y)` . . . 25
1.A iii) `map . (,)` . . . 25
2.8 Exam FS20 . . . 26
1.A i) `(\x y -> \z -> y z x))` . . . 26
1.A ii) `map ($ [])` . . . 27
1.A iii) `(\x y -> y x == y)` . . . 28
2.9 Exam FS19 . . . 29
1.A i) `(\x y -> x y + y)` . . . 29
1.A ii) `(\x y -> x y + x)` . . . 30
1.A iii) `(\x -> map (\y -> x fst y) [])` . . . 32

2.10 Exam FS18 . . . 33
1.1. (a) TODO `map (filter (<3))` . . . 33
1.1. (b) TODO `(\x -> x map)` . . . 33
2.11 Exam FS17 . . . 33
1.(a) 1. TODO `(\x y -> y x x)` . . . 33
1.(a) 2. TODO `(\x y -> y x y)` . . . 33
1.(a) 3. TODO `(\x -> fst x (snd x))` . . . 33
1.(a) 4. TODO `map (\x -> x + x) []` . . . 33
2.12 Exam FS16 . . . 34
1.(a) 1. TODO `(\x y z -> (y x, x z))` . . . 34
1.(a) 2. `map . map` . . . 34
1.(a) 3. TODO `(\x y -> x (map y x))` . . . 34
1.(a) 4. TODO `(\x y -> snd x y) (True, \y -> 3 + y)` . . . 34
2.13 Exam FS15 . . . 35
1. (a) TODO `(\x y z -> z y x)` . . . 35
1. (b) TODO `(\x y -> (==) (map x y))` . . . 35
1. (c) TODO `(\x -> filter x ((<) : []))` . . . 35
1. (d) `(\x y -> map x (y (. x)))` . . . 36
2.14 Exam FS14 . . . 37
1.(a) i. TODO `(\x z -> z x)` . . . 37
1.(a) ii. TODO `(\x y -> (x + 1) < y [] x)` . . . 37
2.15 Exam FS13 . . . 37
1.(a) 1. TODO `(\x y -> x y True)` . . . 37
1.(a) 2. TODO `(\x y z -> if z x then y x else x)` . . . 37
1.(a) 3. TODO `(\x y -> takeWhile (x 0) y)` . . . 37
2.16 Exam FS12 . . . 38
1. 1.) `(\x -> [x 0])` . . . 38
1. 2.) `(\x z -> z (\y -> x))` . . . 38
1. 3.) `map map` . . . 39
1. 4.) `(\x -> x >>= (\y -> y))` . . . 39
2.17 Rep. Exam FS09 . . . 40
1.(a) 1. TODO `(\a b c -> c b a)` . . . 40
1.(a) 2. TODO `(\b -> ((\x -> x), b 1))` . . . 40
1.(a) 3. TODO `(\xs x -> head xs True > x)` . . . 40
2.18 Midterm FS11 . . . 40
1.(a) 1. `(\x y z -> (y x, x z)` . . . 40
1.(a) 2. `(\x y -> x (\z -> y))` . . . 40
1.(a) 3. `map (1:)` . . . 41
1.(a) 4. `(\x ys -> inits (map (x <) ys))` . . . 41
2.19 Midterm FS10 . . . 41
1. 1. `(\x y z -> y z x)` . . . 41
1. 2. `zipWith (==)` . . . 41
1. 3. `(\x y -> y ((==) x)` . . . 42
1. 4. `takeWhile (\x -> x 1)` . . . 43
2.20 Midterm FS09 . . . 44
1.(a) 1. `(\x y z -> x (y z))` . . . 44
1.(a) 2. `(\f -> (\h -> (f,h)))` . . . 44
1.(a) 3. `map (elem 0)` . . . 44
1.(a) 4. `(\x -> x (<))` . . . 44
2.21 Midterm FS08 . . . 45
1.(a) I. `(\x -> snd x) (fst, \y z -> map y ('e':z))` . . . 45
1.(a) II. `(\x y z -> tail x == map (/= z) y)` . . . 46
2.22 Session Exercises . . . 47
TA1 `(\x -> x (<) + 1)` . . . 47
TA2 `(\x a b -> x (a (b x) x))` . . . 48
2.23 MaxS Exercises . . . 49

W4 `(\x -> \y -> x + y)` . . . 49
W4' `filter (<3)` . . . 50
W5 `(\x -> (head x) True)` . . . 51
W5' `(\x y -> map (y<) x)` . . . 52
W5" `(\x y -> head (x map) + 100)` . . . 53
2.24 Further Exercises . . . 54
Ty1 `(head .) . zip` . . . 54
Ty2 `reverse >>= (==)` . . . 55
Ty3 `(\op f -> \x y -> op (f x) (f y))` . . . 56
Ty4 `uncurry fst` . . . 57
Ty5 `uncurry curry` . . . 57
Ty6 `uncurry (flip (,))` . . . 57
Ty7 `foldr (.) id` . . . 58
Ty8 `\f -> foldr ((:) . f) []` . . . 59
Ty9 `(\p f -> head . filter p . iterate f)` . . . 60

1 Algorithm Information

To be able to type any complicated Haskell expression three things will be explained:

1.1 Some important Haskell details;

1.2 How to find the type of function applications (like '`map ((+) 1)`');

1.3 How to find the type of lambda expressions (like `(\x y -> y x)`).

1.1 Useful Haskell Details

Knowing the following things might be very useful, and specifically understanding associativity (= the parenthesis rules/mantras; *'arrow associates to the right'*, *'function application associates to the left'*) will help a lot.

1.1.1 Associativity

There are two parenthesis rules, for (function application-) *expressions* and *function types*:

Function Expressions are applied from the left, so the following are all equivalent (left-associativity):

```
  f x y z
= (f x) y z
= ((f x) y) z
= (f x y) z
```

Function types can return other functions, so the following are all equivalent (right-associativity of `-> (...)`):

```
  a -> b -> c -> d
= a -> (b -> c -> d)
= a -> (b -> (c -> d))
= a -> b -> (c -> d)
```

Associativity will be especially useful to properly *match* two types. In fact, it is so important and frequently used that adding or removing parentheses like this will **not** be explicitly mentioned every time it is used in this document.

$\Longrightarrow$ It will just be assumed that `a -> b -> c` and `a -> (b -> c)` are exactly the same thing!

1.1.2 Operators and operator sections

Operators in Haskell are just functions too. (What a surprise)

An infix operator is just syntax sugar for calling the operator as a function, and more importantly, operator sections are just partially applied functions:

```
a + b = (+) a b     (a +) = (+) a     (+ a) = (\x -> x + a) = (\x -> (+) x a)
```

$\Longrightarrow$ Rewriting operators as functions allows us to systematically type things like `(head .)` because we know that's just equivalent to '`(.) head`'.

1.1.3 Several Lambda Inputs

It is handy to have seen the Haskell syntax equivalence

```
(\x -> (\y -> x+y)) = (\x y -> x+y)
```

$\Longrightarrow$ If you see a long, nested lambda like `(\x -> (\y -> (\z -> x+y+z)))`, you can just collapse its arguments to `(\x y z -> x+y+z)`.

1.1.4 Polymorphic Type Variables

Often you see type signatures containing lowercase letters instead of explicit types:

```
flip :: (a -> b -> c) -> (b -> a -> c)    g :: Int -> a -> [a]
```

These functions are polymorphic, i.e. they work for any type[4] you use in place of `a`, `b`, `c`...

$\Longrightarrow$ When solving a problem you can take polymorphic type signatures like for `f` and `g` and *rename* the type variable(s) before you use them (so they don't overlap and cause confusion later):

```
g :: Int -> a -> [a]  ⟹  g :: Int -> d -> [d]
```

1.1.5 Typeclasses

Typeclasses might come up when solving typing problems. Important things to keep in mind when seeing something like `(+) :: Num a => a -> a -> a` are:

- (What it means: `Num a` asserts that `(+)` works for any type `a`, as long as it behaves like a `Num`ber.)
- Assertions (`Num a => ...`) *always come in front of* the actual type.
- (When there are several assertions the syntax looks like this: `maximum :: (Ord a, Foldable l) => l a -> a`)
- Not only functions can be polymorphic, Haskell constants/literals can be too! By default, Haskell will say `42 :: Num a => a`, meaning the symbols `42` could either become e.g. an `Int`, or a `Double`, in different contexts.

1.2 Typing Function Applications

Typing pure function applications is straightforward because you just have to match the *type of the argument* with the *input type the function expects* and see what the type is that the function now returns.

Assume `id :: a -> a` and `7 :: Int`. When asked the type of `id 7` you probably figure out it'll be `Int`. You know this because the first function input type of `id`, `a`, must match the argument `7`'s type, `Int`:

```
{- Find type of 'id 7' -}
id ::  a -> a
7  :: Int
-- a = Int,  therefore:
id 7 :: a
      = Int
```

The way we simply figured out how we need to replace `a = Int` in the output type `a` is generalizable to arbitrary applications!

Assume you're given more interesting types

[4] This is analogous to bound variables in predicate logic, $\forall a.P(a)$, and renaming between $\forall a.P(a) \iff \forall d.P(d)$. *In fact*, this literally what happens in Haskell with more explicit type annotations:

```
g :: (forall a. Int -> a -> [a]) ⟺ g :: (forall d. Int -> d -> [d])
```

- `map :: (a -> b) -> [a] -> [b]`
- `even :: Int -> Bool`

Then we would type `map even` like this:

```
{- Find type of 'map even' -}
map  :: ( a  ->  b  ) -> ([a] -> [b])
even ::  Int -> Bool

-- a = Int,  b = Bool,  therefore:

map even :: ([a] -> [b])
         =  [a] -> [b]
         = [Int] -> [b]
         = [Int] -> [Bool]
```

Notice how we match the argument type, `even :: (Int -> Bool)`, with the *first function input* type of `map :: (a -> b) -> ...`. This leaves us the *function return* type `map even :: ...`.

Matching and substituting like this is essentially all that you need to understand!

Here are a couple disclaimers concerning non-obvious typing details:

Typing nested expressions. This means situations that look like `f ((g a) b) (h c)` etc. Note that with nested expressions you'll be forced to start from the *inside* – because you can't start matching types outside if you don't know the inner types. Specific example: '`map ((:) 1)`'.

- (Assume the types of `map`, `(:)`, and `1` are already given by the task.)
- You cannot type `map (...)` because we don't know the type of `...` yet.
- So you find the type of `(:) 1` first.
- Now you can type `map ((:) 1)`.

Matching nested types. Sometimes you need to match types nested in other types, like lists. Make sure you don't get confused:

```
{- Find type of 'zip [('h','i')]' -}
zip         :: [     a     ] -> [b] -> [(a,b)]
[('h','i')] :: [(Char,Char)]
-- a = (Char,Char),  therefore:
zip [('h','i')] :: [b] -> [(a,b)]
                 = [b] -> [((Char,Char),b)]
```

The tricky part being that matching `[a]` and `[(Char,Char)]` only requires you to match the `a` and `(Char,Char)` *inside* the list type. A similar thing will happen to types inside tuples, e.g. `(String,Double) ~ (c,d)`.

Tricky associativity. Take the case of matching '`a -> b -> c -> d`' and '`(x -> y) -> z -> w`'. The trick is to consider all explicit associativity parentheses for `(a -> (b -> (c -> d)))` and `((x -> y) -> (z -> w))` and then align the parentheses and arrows:

```
(   a     -> (b -> (c -> d)))   <==>     a     -> b -> c -> d
((x -> y) -> (z ->    w    ))   <==>  (x -> y) -> z ->   w
```

Understanding this is mostly a question of practice and being able to imagine the parentheses that always implicitly exist.

Typeclasses Typeclass annotations are assertions on a type. You need to carry assertions together with the types you replace (don't forget that they're always in front of the whole type):

```
{- Find the type of `map negate` -}
map    ::         (a -> b) -> [a] -> [b]
negate :: Num c => c -> c
-- a = Num c => c,  b = Num c => c,  therefore:
map negate :: [a] -> [b]
            = Num c => [c] -> [c]
```

1.3 Typing Lambdas

Often you will be able to spot the type of a lambda through other, possibly more convenient methods (like guessing). For example, if you determine the type of the lambda `(\x y -> x y) :: (s -> t) -> s -> t` then you can solve `(.) zipWith (\x y -> x y)` with the simple function application approach from before alone.

This section explains how to (technically) solve any lambda task systematically, similar to the function application case.

I'd say it's unlikely anyone will use the following approach exactly at a time-critical exam, but it's included for completeness *(because I don't think I ever saw an explanation of a systematic approach to finding the type of a general lambda expression elsewhere – Feel free to send me ideas if you see something).*

The trick to finding the type of some lambda (e.g. `(\x y -> ...)`) is to

- start with general types for the arguments (e.g. `x :: a`, `y :: b`),
- solve the type of everything that's *inside* the lambda (e.g. `... :: [Bool]`), while keeping track of updating argument types (e.g. you discover `y :: Int`),
- and at the end just assemble the argument- and result types into a full type of the lambda (i.e. `(\x y -> ...) :: a -> Int -> [Bool]`).

The most frequent type update you'll see is when a variable `x :: x'` is used as a function like `x 7`, so you notice that actually `x :: Int -> x1` 'all along': you realize `x' = Int -> x1` and replace `x'` everywhere it still appears. (Notice that `x1` is a new type you need to introduce for what the function(!) `x` returns.)

Doing this in practice will look like this:

```
{- Find type of `(\x -> x 7)` -}
{- Initial input types of `(\x -> ...)` -}
x :: x'

{- Find type of `x 7` -}
x :: ___ -> x1
7 :: Int
-- x = Int -> x1,  therefore:
x :: x'
   = Int -> x1
x 7 :: x1

{- Assemble types of `(\x -> x 7)` -}
(\x -> x 7) :: x' -> x1
               = (Int -> x1) -> x1
```

That's all there's to it.

You just do, *Initialize* → *Solve+Update* → *Assemble*, every time you enter a lambda.

We illustrate how to do this for several variables using the example `(\x y -> x y)` mentioned at the start:

```
{- Find type of `(\x y -> x y)` -}
{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `x y` -}
x :: __ -> x1
y :: y'
-- x = y' -> x1,  therefore:
x :: x'          y :: y'
   = y' -> x1
x y :: x1

{- Assemble types of `(\x y -> x y)` -}
(\x y -> x y) :: x' -> y' -> x1
               = (y' -> x1) -> y' -> x1
```

1.4 Explained Examples

Ex1 `uncurry (flip (,))`

Assume

- `uncurry :: (a -> b -> c) -> (a,b) -> c`
- `flip :: (a -> b -> c) -> b -> a -> c`
- `(,) :: a -> b -> (a,b)`

We want to type `uncurry (flip (,))`.
To find what type `uncurry` applied to `(flip (,))` evaluates to we first need to find the type of the second expression `flip (,)` which we don't know yet:

```
{- Find type of `flip (,)` -}
```

We already have the types of `flip` and `(,)` given to us in the task description, so figuring out the type of `flip` just requires us to match the *type of the argument* `(,)` to the *input type that the function `flip` expects*:

```
flip :: (a -> b ->   c  ) -> b -> a -> c
(,)  ::  u -> v -> (u,v)
```

Note: For clarity we may rename type variables `a,b` to `u,v` at the beginning, see here if confused.
We see that `a` matches `u`, `b` matches `v`, and `c` matches the entire type `(u,v)`:

```
-- a = u,  b = v,  c = (u,v),  therefore: ...
```

Now last thing we need to do with this information is see what `flip` returns under these restrictions:

```
flip (,) :: b -> a -> c
          = v -> u -> (u,v)
```

We used the types we matched earlier and substituted them in the returned type*!*
Having `flip (,) :: v -> u -> (u,v)` we can already type the actual goal `uncurry (flip (,))`, as usual through matching argument and input type:

```
{- Find type of `uncurry (flip (,))` -}
uncurry  :: (x -> y ->   z  ) -> (x,y) -> z
flip (,) ::  v -> u -> (u,v)
-- x = v,  y = u,  z = (u,v),  therefore: ...
```

And as always finish by doing the necessary substitutions on the returned type:

```
uncurry (flip (,)) :: (x,y) -> z
                    = (v,u) -> (u,v)
```

(For a clean presentation of the task without commentary see Ty6.)

Ex2 `(\x -> (head x) True)`

Assume

- `head :: [a] -> a`

We start to type.

```
{- Type of `(\x -> (head x) True)` -}
```

Because we have a lambda, we must now first 'abstract' over its inputs:

```
{- Initial input types of `(\x -> ...)` -}
x :: x'
```

And we now type *everything inside the lambda* and to see what our input types need to be; At the end we 'un-abstract' these types to make a function type from the variable types we figured out. In practice we can just continue with our normal typing procedure:

```
{- Find type of `head x` -}
head :: [a] -> a
x    ::  x'
-- x' = [a],  therefore
x :: x'
   = [a]
head x :: a
```

Although special to notice is that while we're inside the lambda, besides typing function applications, we must keep track of and update the type of the variables, here `x`.
We continue like this:

```
{- Find type of `(head x) True` -}
head x :: ____ -> a1
True   :: Bool
-- a = Bool -> a1,  therefore:
x :: [a]
   = [(Bool -> a1)]
(head x) True :: a1
```

Finally we are done with the body of the lambda. Now we can take the type of the input parameters + the type of the output expression, and put together the full type of the lambda quite easily:

```
{- Assemble types of `(\x -> (head x) True)` -}
(\x -> (head x) True) :: x' -> a1
                       = [(Bool -> a1)] -> a1
```

(For a clean solution of the problem without comments see W5.)

2 Tasks and Solutions

The following tasks are taken from various sources.

* Some are simply taken from Source: previous exercise sheets (FS24).
* Some are from previous exams, available on Community Solutions: Formal Methods And Functional Programming.
* Additional tasks, 2.22, from general TA slides (FS24).
* Some more tasks under 2.23 are due to Maximilian Schlegel, whose TA documents for FMFP FS23 have helped me immensely and contained additional typing exercises. Thank you!
* Finally, I have come up with further exercises myself (2.24).

Note that you can always check the type of an expression through GHCi, (‘`:t (\x -> x)`’), or by using a tool specifically developed by FMFP TAs for formal type inference of Minihaskell expressions (especially useful for pure lambdas): `https://n.ethz.ch/~rawick/typers/index.html`.

2.1 Exercise Sheet 4

2.(a) `(\x y z -> (x y) z)`

```
{- Initial input types of ‘(\x y z -> ...)‘ -}
x :: x’    y :: y’    z :: z’

{- Find type of ‘x y‘ -}
x :: __ -> x1
y :: y’
-- x’ = y’ -> x1,  therefore:
x :: x’          y :: y’    z :: z’
   = y’ -> x1
x y :: x1

{- Find type of ‘x y z‘ -}
x y :: __ -> x2
z   :: z’
-- x1 = z’ -> x2,  therefore:
x :: y’ -> x1            y :: y’    z :: z’
   = y’ -> (z’ -> x2)
x y z :: x2

{- Assemble type of ‘(\x y z -> (x y) z)‘ -}
(\x y z -> (x y) z)
    :: x’ -> y’ -> z’ -> x2
    :: (y’ -> z’ -> x2) -> y’ -> z’ -> x2
```

2.(b) `(\f -> fst (f,f))`

Assume

- `fst :: (a,b) -> a`

```
{- Initial input types of ‘(\f -> ...)‘ -}
f :: f’

{- Find type of ‘fst (f,f)‘ -}
fst   :: (x, y) -> x
(f,f) :: (f’,f’)
-- x = f’,  y = f’,  therefore:
f :: f’
fst (f,f) :: x
           = f’

{- Assemble type of ‘(\f -> fst (f,f))‘ -}
(\f -> fst (f,f)) :: f’ -> f’
```

2.(c) `(\p f -> filter p . map f)`

Assume

- `filter :: (a -> Bool) -> [a] -> [a]`
- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `map :: (a -> b) -> [a] -> [b]`

```
{- Transform expression into clearer form -}
  (\p f -> filter p . map f)
= (\p f -> (.) (filter p) (map f))

{- Initial input types of `(\p f -> ...)` -}
p :: p'    f :: f'

{- Find type of `filter p` -}
filter :: (x -> Bool) -> [x] -> [x]
p      ::    p'
-- (x -> Bool) = p',  therefore:
p :: p'             f :: f'
   = (x -> Bool)
filter p :: [x] -> [x]

{- Find type of `(.) (filter p)` -}
(.)      :: ( b  ->  c ) -> (a -> b) -> (a -> c)
filter p ::  [x] -> [x]
-- b = [x],  c = [x],  therefore:
p :: x -> Bool    f :: f'
(.) (filter p) :: (a -> b) -> (a -> c)
                = (a -> [x]) -> (a -> [x])

{- Find type of `map f` -}
map :: (u -> v) -> [u] -> [v]
f   ::    f'
-- f' = (u -> v),  therefore:
p :: x -> Bool    f :: f'
                     = (u -> v)
map f :: [u] -> [v]

{- Find type of `(.) (filter p) (map f)` -}
(.) (filter p) :: ( a  -> [x]) -> (a -> [x])
map f          ::  [u] -> [v]
-- a = [u],  x = v,  therefore:
p :: x -> Bool    f :: u -> v
   = v -> Bool
(.) (filter p) (map f)
    :: a -> [x]
     = [u] -> [v]

{- Assemble type of `(\p f -> (.) (filter p) (map f))` -}
(\p f -> (.) (filter p) (map f))
    :: p' -> f' -> [u] -> [v]
     = (v -> Bool) -> (u -> v) -> [u] -> [v]
```

2.2 Exercise Sheet 5

2.(a) 1. `(\x y z -> x (y z))`

Remark: This is the function '(.)' from Prelude.

```
{- Find type of `(\x y z -> x (y z))` -}
{- Initial input types of `(\x y z -> ...)` -}
x :: x'    y :: y'    z :: z'

{- Find type of `(y z)` -}
y :: __ -> y1
z :: z'
-- y' = z' -> y1,  therefore:
x :: x'    y :: y'          z :: z'
              = z' -> y1
y z :: y1

{- Find type of `x (y z)` -}
x   :: __ -> x1
y z :: y1
-- x' = y1 -> x1,  therefore:
x :: x'          y :: z' -> y1    z :: z'
   = y1 -> x1
x (y z) :: x1

{- Assemble type of `(\x y z -> x (y z))` -}
(\x y z -> x (y z))
    :: x' -> y' -> z' -> x1
     = (y1 -> x1) -> (z' -> y1) -> z' -> x1
```

2.(a) 2. `(\f -> (\h -> (f,h)))`

```
{- Transform expression into easier-to-type form -}
  (\f -> (\h -> (f,h)))
= (\f h -> (f,h))

{- Initial input types of `(\f h -> ...)` -}
f :: f'    h :: h'

{- Find type of `(f,h)` -}
(f,h) :: (f',h')

{- Assemble type of `(\f h -> (f,h))` -}
(\f h -> (f,h)) :: f' -> h' -> (f',h')
```

2.(a) 3. `(\x -> x (<))`

Assume

- `(<) :: Ord a => a -> a -> Bool`

```
{- Find type of ‘(\x -> x (<))‘ -}
{- Initial input types of ‘(\x -> ...)‘ -}
x :: x’

{- Find type of ‘x (<)‘ -}
x   ::          _________________ -> x1
(<) :: Ord a => (a -> a -> Bool)
-- x’ = Ord a => (a -> a -> Bool) -> x1,  therefore:
x :: x’
   = Ord a => (a -> a -> Bool) -> x1
x (<) :: x1

{- Assemble type of ‘(\x -> x (<))‘ -}
(\x -> x (<)) :: x’ -> x1
              = Ord a => ((a -> a -> Bool) -> x1) -> x1
```

2.(a) 4. `(.) . (.)`

Assume

- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`

```
{- Transform expression into clearer form -}
  (.) . (.)
= (.) (.) (.)
= ((.) (.)) (.)

{- Find type of ‘(.) (.)‘ -}
(.)  :: (   b     ->          c           ) -> (a -> b) -> (a -> c)
(.)  ::  (y -> z) -> ((x -> y) -> (x -> z))
-- b = (y -> z),  c = ((x -> y) -> (x -> z)),  therefore:
(.) (.) :: (a -> b) -> (a -> c)
         = (a -> (y -> z)) -> (a -> ((x -> y) -> (x -> z)))
         = (a -> y -> z) -> a -> (x -> y) -> x -> z

{- Find type of ‘(.) (.) (.)‘ -}
(.) (.) :: (   a     ->    y     ->    z    ) -> a -> (x -> y) -> x -> z
(.)     ::  (v -> w) -> (u -> v) -> (u -> w)
-- a = (v -> w),  y = (u -> v),  z = (u -> w),  therefore:
(.) (.) (.) :: a -> (x -> y) -> x -> z
             = (v -> w) -> (x -> (u -> v)) -> x -> (u -> w)
             = (v -> w) -> (x -> u -> v) -> x -> u -> w
```

2.(a) 5. `flip id`

Assume

- `flip :: (a -> b -> c) -> b -> a -> c`
- `id :: a -> a`

```
{- Find type of ‘flip id‘ -}
flip :: (a -> (b -> c)) -> b -> a -> c
id   ::  x ->    x
-- a = x,  (b -> c) = x,  therefore:
flip id :: b -> a -> c
         = b -> x -> c
         = b -> (b -> c) -> c
```

2.3 Exercise Sheet 7

1.(a) 1. `(\x y z -> (y x, x z))`

```
{- Find type of ‘(\x y z -> (y x, x z))‘ -}
{- Initial input types of ‘(\x y z -> ...)‘ -}
x :: x’    y :: y’    z :: z’

{- Find type of ‘y x‘ -}
y :: __ -> y1
x :: x’
-- y’ = x’ -> y1,  therefore:
x :: x’    y :: y’          z :: z’
              = x’ -> y1
y x :: y1

{- Find type of ‘x z‘ -}
x :: __ -> x1
z :: z’
-- x’ = z’ -> x1,  therefore:
x :: x’          y :: x’ -> y1            z :: z’
   = z’ -> x1       = (z’ -> x1) -> y1
x z :: x1

{- Find type ‘(y x, x z)‘ -}
(y x, x z) :: (y1, x1)

{- Assemble type of ‘(\x y z -> (y x, x z))‘ -}
(\x y z -> (y x, x z))
    :: x’ -> y’ -> z’ -> (y1, x1)
     = (z’ -> x1) -> ((z’ -> x1) -> y1) -> z’ -> (y1, x1)
```

1.(a) 2. `(\x y -> (\z -> y))`

```
{- Transform into clearer form -}
  (\x y -> (\z -> y))
= (\x y z -> y)

{- Initial input types of `(\x y z -> ...)` -}
x :: x'    y :: y'    z :: z'

{- Assemble type of `(\x y z -> y)` -}
(\x y z -> y) :: x' -> y' -> z' -> y'
```

1.(a) 3. `map (1:)`

Assume

- `map :: (a -> b) -> [a] -> [b]`
- `(:) :: a -> [a] -> [a]`
- `1 :: Num a => a`

```
{- Transform expression into clearer form -}
  map (1:)
= map ((:) 1)

{- Find type of `(:) 1` -}
(:) ::          a -> [a] -> [a]
1   :: Num b => b
-- a = Num b => b,  therefore:
(:) 1 :: [a] -> [a]
       = Num b => [b] -> [b]

{- Find type of `map ((:) 1)` -}
map   ::          ( x  ->  y ) -> [x] -> [y]
(:) 1 :: Num b =>  [b] -> [b]
-- x = Num b => [b],  y = Num b => [b],  therefore:
map ((:) 1) :: [x] -> [y]
             = Num b => [[b]] -> [[b]]
```

1.(a) 4. `(\x ys -> inits (map (x <) ys))`

Assume

- `inits :: [a] -> [[a]]`
- `map :: (a -> b) -> [a] -> [b]`
- `(<) :: Ord a => a -> a -> Bool`

```
{- Transform into clearer form -}
  (\x ys -> inits (map (x <) ys))
= (\x ys -> inits (map ((<) x) ys))

{- Initial input types of `(\x ys -> ...)` -}
x :: x'    ys :: y'

{- Find type of `(<) x` -}
(<) :: Ord o => o -> o -> Bool
x   ::          x'
-- x' = Ord o => o,  therefore:
x :: x'            ys :: y'
   = Ord o => o
(<) x :: Ord o => o -> Bool

{- Find type of `map ((<) x)` -}
map   ::         (a ->  b  ) -> [a] -> [b]
(<) x :: Ord o => o -> Bool
-- a = Ord o => o,  b = Bool,  therefore:
x :: Ord o => o    ys :: y'
map ((<) x) :: [a] -> [b]
             = Ord o => [o] -> [Bool]

{- Find type of `(map (x <) ys)` -}
map ((<) x) :: Ord o => [o] -> [Bool]
ys          ::           y'
-- y' = Ord o => [o],  therefore:
x :: Ord o => o    ys :: y'
                       = Ord o => [o]
map ((<) x) ys :: [Bool]

{- Find type of `inits (map (x <) ys)` -}
inits          :: [ c  ] -> [[c]]
map ((<) x) ys :: [Bool]
-- c = Bool,  therefore:
x :: Ord o => o    ys :: Ord o => [o]
inits (map (x <) ys) :: [[c]]
                      = [[Bool]]

{- Assemble type of `(\x ys -> inits (map (x <) ys))` -}
(\x ys -> inits (map (x <) ys))
    :: x' -> y' -> [[Bool]]
     = Ord o => o -> [o] -> [[Bool]]
```

2.4 Exam FS23

Source: https://exams.vis.ethz.ch/exams/g7u8zopy.pdf

1.A i) `(\x y -> x (y,y))`

```
{- Find type of `(\x y -> x (y,y))` -}
{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `x (y,y)` -}
x     :: _______ -> x1
(y,y) :: (y',y')
-- x' = (y',y') -> x1,  therefore:
x :: x'               y :: y'
   = (y',y') -> x1
x (y,y) :: x1

{- Assemble type of `(\x y -> x (y,y))` -}
(\x y -> x (y,y)) :: x' -> y' -> x1
                   = ((y',y') -> x1) -> y' -> x1
```

1.A ii) `(.) zipWith (\x y -> x y)`

Remark: This the same as the function 'zipWith' from Prelude.

Assume

- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

```
{- Find type of '(.) zipWith' -}
(.)     :: (      b       ->          c         ) -> (a -> b) -> (a -> c)
zipWith ::  (x -> y -> z) -> ([x] -> [y] -> [z])
-- b = (x -> y -> z),  c = [x] -> [y] -> [z],  therefore:
(.) zipWith
    :: (a -> b) -> (a -> c)
     = (a -> (x -> y -> z)) -> (a -> ([x] -> [y] -> [z]))
     = (a -> x -> y -> z) -> a -> [x] -> [y] -> [z]

{- Find type of '(\x y -> x y)' -}
{- Initial input types of '(\x y -> ...)' -}
x :: x'    y :: y'

{- Find type of 'x y' -}
x :: __ -> x1
y :: y'
-- x = y' -> x1,  therefore:
x :: x'          y :: y'
   = y' -> x1
x y :: x1

{- Assemble type of '(\x y -> x y)' -}
(\x y -> x y) :: x' -> y' -> x1
               = (y' -> x1) -> y' -> x1

{- Find type of '(.) zipWith (\x y -> x y)' -}
(.) zipWith   :: (    a    ->  x -> (y -> z)) -> a -> [x] -> [y] -> [z]
(\x y -> x y) ::  (t -> s) ->  t ->    s
-- a = (t -> s),  x = t,  (y -> z) = s,  therefore:
(.) zipWith (\x y -> x y)
    :: a -> [x] -> [y] -> [z]
     = (t -> s) -> [x] -> [y] -> [z]
     = (x -> (y -> z)) -> [x] -> [y] -> [z]
     = (x -> y -> z) -> [x] -> [y] -> [z]
```

2.5 Exam FS22

Source: https://exams.vis.ethz.ch/exams/7uousppj.pdf

1.A 1. `foldl (==)`

Assume

- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `(==) :: Eq a => a -> a -> Bool`

```
{- Find type of `foldl (==)` -}
foldl ::        (b -> a ->  b  ) -> b -> [a] -> b
(==)  :: Eq e => e -> e -> Bool
-- e = b,  a = e,  b = Bool;  a = b = Bool  therefore:
foldl (==) :: b -> [a] -> b
            = Bool -> [Bool] -> Bool
```

1.A 2. `(\x y -> (y x, x y))`

```
{- Find type of `(\x y -> (y x, x y))` -}
{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `y x` -}
y :: __ -> y1
x :: x'
-- y' = x' -> y1,  therefore:
x :: x'    y :: y'
              = x' -> y1
y x :: y1

{- Find type of `x y` -}
x :: ________ -> x1
y :: x' -> y1
-- x' = (x' -> y1) -> x1,  therefore:
error "Recursive type found. Not well-typed."
```

2.6 Exam HS21

Source: https://exams.vis.ethz.ch/exams/kq31s35t.pdf

1.A i) `zipWith elem`

Assume

- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `elem :: Eq a => a -> [a] -> Bool`

```
{- Find type of ‘zipWith elem‘ -}
zipWith ::        (a ->  b  ->  c  ) -> [a] -> [b] -> [c]
elem    :: Eq e => e -> [e] -> Bool
-- a = Eq e => e,  b = Eq e => [e],  c = Bool,  therefore:
zipWith elem :: [a] -> [b] -> [c]
             = Eq e => [e] -> [[e]] -> [Bool]
```

1.A ii) `(\x -> map x [x])`

Assume

- `map :: (a -> b) -> [a] -> [b]`

```
{- Find type of ‘(\x -> map x [x])‘ -}

{- Abstract types of ‘(\x -> ...)‘ -}
x :: x'

{- Find type of ‘map x‘ -}
map :: (a -> b) -> [a] -> [b]
x   ::    x'
-- x' = (a -> b),  therefore:
x :: x'
   = (a -> b)
map x :: [a] -> [b]

{- Find type of ‘map x [x]‘ -}
map x :: [   a    ] -> [b]
[x]   :: [(a -> b)]
-- a = (a -> b),  therefore:
error "Recursive type found. This is not well-typed."
```

1.A iii) `(\x y z -> (x z) (y z))`

```
{- Find type of `(\x y z -> (x z) (y z))` -}
{- Initial input types of `(\x y z -> ...)` -}
x :: x'    y :: y'    z :: z'

{- Find type of `(x z)` -}
x :: __ -> x1
z :: z'
-- x' = z' -> x1,  therefore:
x :: x'          y :: y'    z :: z'
   = z' -> x1
x z :: x1

{- Find type of `(y z)` -}
y :: __ -> y1
z :: z'
-- y' = z' -> y1,  therefore:
x :: z' -> x1    y :: y'    z :: z'
                    = z' -> y1
y z :: y1

{- Find type of `(x z) (y z)` -}
x z :: __ -> x2
y z :: y1
-- x1 = y1 -> x2,  therefore:
x :: z' -> x1    y :: z' -> y1    z :: z'
   = z' -> y1 -> x2
(x z) (y z) :: x2

{- Assemble type of `(\x y z -> (x z) (y z))` -}
(\x y z -> (x z) (y z)) :: x' -> y' -> z' -> x2
                         = (z' -> y1 -> x2) -> (z' -> y1) -> z' -> x2
```

2.7 Exam FS21

Source: https://exams.vis.ethz.ch/exams/fgll1udm.pdf

1.A i) `(\x y -> y y x)`

```
{- Find type of `(\x y -> y y x)` -}
{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `y y` -}
y :: __ -> y1
y :: y'
-- y' = y' -> y1,  therefore
error "Recursive type found. Not well-typed."
```

1.A ii) `(\x y -> x y y)`

```
{- Find type of `(\x y -> x y y)` -}
{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `x y` -}
x :: __ -> x1
y :: y'
-- x' = y' -> x1,  therefore:
x :: x'          y :: y'
   = y' -> x1
x y :: x1

{- Find type of `x y y` -}
x y :: __ -> x2
y   :: y'
-- x1 = y' -> x2,  therefore:
x :: y' -> x1          y :: y'
   = y' -> y' -> x2
x y y :: x2

{- Assemble type of `(\x y -> x y y)` -}
(\x y -> x y y) :: x' -> y' -> x2
                 = (y' -> y' -> x2) -> y' -> x2
```

1.A iii) `map . (,)`

Assume

- `map :: (a -> b) -> [a] -> [b]`
- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `(,) :: a -> b -> (a,b)`

```
{- Transform expression into clearer form -}
  map . (,)
= (.) map (,)

{- Find type of `(.) map` -}
(.) :: (    b    ->      c      ) -> (a -> b) -> (a -> c)
map ::  (x -> y) -> ([x] -> [y])
-- b = (x -> y),  c = ([x] -> [y]),  therefore:
(.) map :: (a -> b) -> (a -> c)
         = (a -> (x -> y)) -> (a -> ([x] -> [y]))
         = (a -> x -> y) -> a -> [x] -> [y]

{- Find type of `(.) map (,)` -}
(.) map :: (a -> x ->   y  ) -> a -> [x] -> [y]
(,)     ::  u -> v -> (u,v)
-- a = u,  x = v,  y = (u,v),  therefore:
(.) map (,) :: a -> [x] -> [y]
             = u -> [v] -> [(u,v)]
```

2.8 Exam FS20

Source: https://exams.vis.ethz.ch/exams/ghigz2p5.pdf

1.A i) `(\x y -> \z -> y z x))`

```
{- Transform expression into clearer form -}
  (\x y -> \z -> y z x))
= (\x y z -> y z x)

{- Initial input types of `(\x y z -> ...)` -}
x :: x'    y :: y'    z :: z'

{- Find type of `y z` -}
y :: __ -> y1
z :: z'
-- y' = z' -> y1,  therefore:
x :: x'    y :: y'           z :: z'
              = z' -> y1
y z :: y1

{- Find type of `y z x` -}
y z :: __ -> y2
x   :: x'
-- y1 = x' -> y2,  therefore:
x :: x'    y :: z' -> y1          z :: z'
              = z' -> x' -> y2
y z x :: y2

{- Assemble type of `(\x y z -> y z x)` -}
(\x y z -> y z x) :: x' -> y' -> z' -> y2
                   = x' -> (z' -> x' -> y2) -> z' -> y2
```

1.A ii) `map ($ [])`

Assume

- `map :: (a -> b) -> [a] -> [b]`
- `($) :: (a -> b) -> a -> b`

```
{- Transform expression into clearer form -}
  map ($ [])
= map (\x -> x $ [])
= map (\x -> ($) x [])

{- Initial input types of `(\x -> ...)` -}
x :: x'

{- Find type of `($) x` -}
($) :: (a -> b) -> a -> b
x   ::    x'
-- x' = (a -> b),  therefore:
x :: x'
   = (a -> b)
($) x :: a -> b

{- Find type of `($) x []` -}
($) x ::  a  -> b
[]    :: [e]
-- a = [i],  therefore:
x :: a -> b
   = [i] -> b
($) x [] :: b

{- Assemble type of `(\x -> x $ [])` -}
(\x -> x $ []) :: x' -> b
                = ([e] -> b) -> b
```

1.A iii) `(\x y -> y x == y)`

Assume

- `(==) :: Eq a => a -> a -> Bool`

```
{- Transform expression into clearer form -}
  (\x y -> y x == y)
= (\x y -> (==) (y x) y)

{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `(y x)` -}
y :: __ -> y1
x :: x'
-- y' = x' -> y1,  therefore
x :: x'    y :: y'
              = x' -> y1
y x :: y1

{- Find type of `(==) (y x)` -}
(==) :: Eq a => a -> a -> Bool
y x  ::         y1
-- y1 = Eq a => a,  therefore:
x :: x'    y :: x' -> y1
              = Eq a => x' -> a
(==) (y x) :: Eq a => a -> Bool

{- Find type of `(==) (y x) y` -}
(==) (y x) :: Eq a =>     a    -> Bool
y          :: Eq a => (x' -> a)
-- a = (x' -> a),  therefore:
error "Recursive type found. Not well-typed."
```

2.9 Exam FS19

Source: https://exams.vis.ethz.ch/exams/rh0fbq8b.pdf

1.A i) `(\x y -> x y + y)`

Assume

- `(+) :: Num a => a -> a -> a`

```
{- Transform into clearer form -}
  (\x y -> x y + y)
= (\x y -> (+) (x y) y)

{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `(x y)` -}
x :: __ -> x1
y :: y'
-- x' = y' -> x1,  therefore:
x :: x'          y :: y'
   = y' -> x1
x y :: x1

{- Find type of `(+) (x y)` -}
(+) :: Num a => a  -> a -> a
x y ::          x1
-- x1 = Num a => a,  therefore:
x :: y' -> x1               y :: y'
   = Num a => y' -> a
(+) (x y) :: Num a => a -> a

{- Find type of `(+) (x y) y` -}
(+) (x y) :: Num a => a  -> a
y         ::          y'
-- y' = Num a => a,  therefore:
x :: Num a => y' -> a    y :: y'
   = Num a => a -> a        = Num a => a
(+) (x y) y :: Num a => a

{- Assemble type of `(\x y -> (+) (x y) y)` -}
(\x y -> (+) (x y) y) :: Num a => x' -> y' -> a
                       = Num a => (a -> a) -> a -> a
```

1.A ii) `(\x y -> x y + x)`

Assume

- `(+) :: Num a => a -> a -> a`

```
{- Transform into clearer form -}
  (\x y -> x y + x)
= (\x y -> (+) (x y) x)

{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `(x y)` -}
x :: __ -> x1
y :: y'
-- x' = y' -> x1,  therefore:
x :: x'          y :: y'
   = y' -> x1
x y :: x1

{- Find type of `(+) (x y)` -}
(+) :: Num a => a  -> a -> a
x y ::          x1
-- x1 = Num a => a,  therefore:
x :: y' -> x1            y :: y'
   = Num a => y' -> a
(+) (x y) :: Num a => a -> a

{- Find type of `(+) (x y) x` -}
(+) (x y) :: Num a =>     a    -> a
x         :: Num a => (y' -> a)
-- a = Num a => y' -> a,  therefore:
error "Recursive type found. Not well-typed.
```

1.A iii) `(\x -> map (\y -> x fst y) [])`

Assume

- `map :: (a -> b) -> [a] -> [b]`
- `fst :: (a, b) -> a`

```
{- Transform into clearer form -}
  (\x -> map (\y -> x fst y) [])
= (\x -> map (\y -> (x fst) y) [])

{- Find type of `(\x -> map (\z -> (x fst) z) [])` -}
{- Initial input types of `(\x -> ...)` -}
x :: x'

{- Find type of `(\y -> (x fst) y)` -}
{- Initial input types of `(\y -> ...)` -}
y :: y'

{- Find type of `x fst` -}
x   :: ___________ -> x1
fst :: (t, s) -> t
-- x' = ((t, s) -> t) -> x1,  therefore:
x :: x'
   = ((t, s) -> t) -> x1
y :: y'
x fst :: x1

{- Find type of `(x fst) y` -}
x fst :: __ -> x2
y     :: y'
-- x1 = y' -> x2,  therefore:
x :: ((t, s) -> t) -> x1
   = ((t, s) -> t) -> z' -> x2
y :: y'
(x fst) y :: x2

{- Assemble type of `(\y -> (x fst) y)` -}
(\y -> (x fst) y) :: y' -> x2

{- Find type of `map (\y -> (x fst) y)` -}
map               :: (a  -> b ) -> [a] -> [b]
(\y -> (x fst) y) ::  y' -> x2
-- a = y',  b = x2,  therefore:
x :: ((t, s) -> t) -> z' -> x2
map (\y -> (x fst) y) :: [a] -> [b]
                       = [y'] -> [x2]

{- Find type of `map (\y -> (x fst) y) []` -}
map (\y -> (x fst) y) :: [y'] -> [x2]
[]                    :: [_ ]
-- _ = y',  therefore:
x :: ((t, s) -> t) -> y' -> x2
map (\y -> (x fst) y) [] :: [x2]

{- Assemble type of `(\x -> map (\z -> (x fst) z) [])` -}
(\x -> map (\z -> (x fst) z) [])
    :: x' -> [x2]
     = (((t, s) -> t) -> y' -> x2) -> [x2]
```

2.10 Exam FS18

Source: https://exams.vis.ethz.ch/exams/3fiscy9n.pdf

1.1. (a) todo `map (filter (<3))`

Assume

- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `(<) :: Ord a => a -> a -> Bool`
- `3 :: Num a => a`

1.1. (b) todo `(\x -> x map)`

Assume

- `map :: (a -> b) -> [a] -> [b]`

2.11 Exam FS17

Source: https://exams.vis.ethz.ch/exams/3o2sjb4b.pdf

1.(a) 1. todo `(\x y -> y x x)`

1.(a) 2. todo `(\x y -> y x y)`

1.(a) 3. todo `(\x -> fst x (snd x))`

Assume

- `fst :: (a, b) -> a`
- `snd :: (a, b) -> b`

1.(a) 4. todo `map (\x -> x + x) []`

Assume

- `map :: (a -> b) -> [a] -> [b]`
- `(+) :: Num a => a -> a -> a`

2.12 Exam FS16

Source: https://exams.vis.ethz.ch/exams/y05fwfbb.pdf

1.(a) 1. todo `(\x y z -> (y x, x z))`

1.(a) 2. `map . map`

Assume

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

```
{- Transform into clearer form -}
  map . map
= (.) map map
= ((.) map) map

{- Find type of `(.) map` -}
(.) :: (   b     ->      c      ) -> (a -> b) -> a -> c
map ::  (u -> v) -> ([u] -> [v])
-- b = (u -> v),  c = ([u] -> [v]),  therefore:
(.) map :: (a -> b) -> a -> c
         = (a -> (u -> v)) -> a -> ([u] -> [v])
         = (a -> u -> v) -> a -> [u] -> [v]

{- Find type of `((.) map) map` -}
(.) map :: (   a     ->  u  ->  v ) -> a -> [u] -> [v]
map     ::  (s -> t) -> [s] -> [t]
-- a = (s -> t),  u = [s],  v = [t],  therefore:
((.) map) map :: a -> [u] -> [v]
               = (s -> t) -> [[s]] -> [[t]]
```

1.(a) 3. todo `(\x y -> x (map y x))`

Assume

- `map :: (a -> b) -> [a] -> [b]`

1.(a) 4. todo `(\x y -> snd x y) (True, \y -> 3 + y)`

Assume

- `snd :: (a, b) -> b`
- `True :: Bool`
- `(+) :: Num a => a -> a -> a`

2.13 Exam FS15

Source: https://exams.vis.ethz.ch/exams/qklu6gph.pdf

1. (a) todo `(\x y z -> z y x)`

```
{- Why are so many typing exercises lambdas??? Screw this: -}
(\x y z -> z y x) :: x' -> y' -> z' -> ??
-- z' = y' -> x' -> z2,  therefore:
(\x y z -> z y x) :: x' -> y' -> (y' -> x' -> z2) -> z2
```

1. (b) todo `(\x y -> (==) (map x y))`

Assume

- `(==) :: Eq a => a -> a -> Bool`
- `map :: (a -> b) -> [a] -> [b]`

1. (c) todo `(\x -> filter x ((<) : []))`

Assume

- `filter :: (a -> Bool) -> [a] -> [a]`
- `(<) :: Ord a => a -> a -> Bool`
- `(:) :: a -> [a] -> [a]`

1. (d) `(\x y -> map x (y (. x)))`

Assume

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

```
{- Transform into clearer form -}
  (\x y -> map x (y (. x)))
= (\x y -> (map x) (y (\z -> (.) z x)))

{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `(\z -> (.) z x)` -}
{- Initial input types of `(\z -> ...)` -}
z :: z'

{- Find type of `(.) z` -}
(.) :: (b -> c) -> (a -> b) -> a -> c
z   ::    z'
-- z' = b -> c,  therefore:
z :: z'
   = b -> c
x :: x'    y :: y'
(.) z :: (a -> b) -> a -> c

{- Find type of `(.) z x` -}
(.) z :: (a -> b) -> a -> c
x     ::    x'
-- x' = a -> b,  therefore:
z :: b -> c
x :: x'        y :: y'
   = a -> b
(.) z x :: a -> c

{- Assemble type of `(\z -> (.) z x)` -}
(\z -> (.) z x) :: z' -> a -> c
                 = (b -> c) -> a -> c

{- Find type of `y (\z -> (.) z x)` -}
y               :: __________________ -> y1
(\z -> (.) z x) :: (b -> c) -> a -> c
-- y' = ((b -> c) -> a -> c) -> y1,  therefore:
x :: a -> b    y :: y'
                  = ((b -> c) -> a -> c) -> y1
y (\z -> (.) z x) :: y1

{- Find type of `map x` -}
map :: (u -> v) -> [u] -> [v]
x   ::  a -> b
-- u = a,  v = b,  therefore:
x :: a -> b    y :: ((b -> c) -> a -> c) -> y1
map x :: [u] -> [v]
       = [a] -> [b]

{- Find type of `(map x) (y (\z -> (.) z x))` -}
map x             :: [a] -> [b]
y (\z -> (.) z x) ::  y1
-- y1 = [a],  therefore:
x :: a -> b    y :: ((b -> c) -> a -> c) -> y1
                  = ((b -> c) -> a -> c) -> [a]
(map x) (y (\z -> (.) z x)) :: [b]

{- Assemble type of `(\x y -> (map x) (y (\z -> (.) z x)))` -}
(\x y -> (map x) (y (\z -> (.) z x)))
    :: x' -> y' -> [b]
     = (a -> b) -> (((b -> c) -> a -> c) -> [a]) -> [b]
```

2.14 Exam FS14

Source: https://exams.vis.ethz.ch/exams/dhm62tei.pdf

1.(a) i. todo `(\x z -> z x)`

1.(a) ii. todo `(\x y -> (x + 1) < y [] x)`

Assume

- `(+) :: Num a => a -> a -> a`
- `1 :: Num a => a`
- `(<) :: Ord a => a -> a -> Bool`
- `[] :: [a]`

2.15 Exam FS13

Source: https://exams.vis.ethz.ch/exams/jvkpag10.pdf

1.(a) 1. todo `(\x y -> x y True)`

Assume

- `True :: Bool`

1.(a) 2. todo `(\x y z -> if z x then y x else x)`

1.(a) 3. todo `(\x y -> takeWhile (x 0) y)`

Assume

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
- `0 :: Num a => a`

2.16 Exam FS12

Source: https://exams.vis.ethz.ch/exams/9yw6bhmq.pdf

1. 1.) `(\x -> [x 0])`

Assume

- `0 :: Num a => a`

```
{- Transform into clearer form -}
  (\x -> [x 0])
= (\x -> [(x 0)])

{- Find type of `(\x -> [(x 0)])` -}
{- Initial input types of `(\x -> ...)` -}
x :: x'

{- Find type of `(x 0)` -}
x ::          _ -> x1
0 :: Num a => a
-- x' = Num a => a -> x1,  therefore:
x :: x'
   = Num a => a -> x1
x 0 :: x1

{- Find type of `[(x 0)]` -}
[(x 0)] :: [x1]

{- Assemble type of `(\x -> [x 0])` -}
(\x -> [x 0]) :: x' -> [x1]
               = Num a => 0(a -> x1)@ -> [x1]
```

1. 2.) `(\x z -> z (\y -> x))`

```
{- Find type of `(\x z -> z (\y -> x))` -}
{- Initial input types of `(\x z -> ...)` -}
x :: x'    z :: z'

{- Find type of `z (\y -> x)` -}
z         :: ________ -> z1
(\y -> x) :: y' -> x'
-- z' = (y' -> x') -> z1,  therefore:
x :: x'    z :: z'
             = (y' -> x') -> z1
z (\y -> x) :: z1

{- Assemble type of `(\x z -> z (\y -> x))` -}
(\x z -> z (\y -> x)) :: x' -> z' -> z1
                       = x' -> ((y' -> x') -> z1) -> z1
```

1. 3.) `map map`

Assume

- `map :: (a -> b) -> [a] -> [b]`

```
{- Find type of ‘map map‘ -}
map :: (   a     ->      b      ) -> [a] -> [b]
map ::  (x -> y) -> ([x] -> [y])
-- a = (x -> y),  b = ([x] -> [y]),  therefore:
map map :: [a] -> [b]
         = [(x -> y)] -> [([x] -> [y])]
```

1. 4.) `(\x -> x >>= (\y -> y))`

Remark: This is the function ‘join’ from Control.Monad.

Assume

- `(>>=) :: Monad m => m a -> (a -> m b) -> m b`

```
{- Transform into clearer form -}
  (\x -> x >>= (\y -> y))
= (\x -> (>>=) x (\y -> y))

{- Initial input types of ‘(\x -> ...)‘ -}
x :: x'

{- Find type of ‘(>>=) x‘ -}
(>>=) :: Monad m => m a -> (a -> m b) -> m b
x     ::             x'
-- x' = Monad m => m a,  therefore:
x :: x'
   = Monad m => m a
(>>=) x :: Monad m => (a -> m b) -> m b

{- Find type of ‘(>>=) x (\y -> y)‘ -}
(>>=) x   :: Monad m => (a -> m b) -> m b
(\y -> y) ::             e ->  e
-- a = e,  e = Monad m => m b,  therefore:
x :: Monad m => m a
   = Monad m => m e
   = Monad m => m (m b)
(>>=) x (\y -> y) :: m b

{- Assemble type of ‘(\x -> (>>=) x (\y -> y))‘ -}
(\x -> (>>=) x (\y -> y)) :: Monad m => x' -> m b
                           = Monad m => m (m b) -> m b
```

2.17 Rep. Exam FS09

Source: https://exams.vis.ethz.ch/exams/3wdsgia4.pdf

1.(a) 1. todo `(\a b c -> c b a)`

1.(a) 2. todo `(\b -> ((\x -> x), b 1))`

Assume

- `1 :: Num a => a`

1.(a) 3. todo `(\xs x -> head xs True > x)`

Assume

- `head :: [a] -> a`
- `True :: Bool`
- `(>) :: Ord a => a -> a -> Bool`

2.18 Midterm FS11

Source: https://exams.vis.ethz.ch/exams/908spyt5.pdf

1.(a) 1. `(\x y z -> (y x, x z)`

Solved in 1.(a) 1..

1.(a) 2. `(\x y -> x (\z -> y))`

```
{- Find type of `(\x y -> x (\z -> y))` -}
{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `(\z -> y)` -}
{- Initial input types of `(\z -> ...)` -}
z :: z'

{- Assemble type of `(\z -> y)` -}
(\z -> y) :: z' -> y'

{- Find type of `x (\z -> y)` -}
x         :: ________ -> x1
(\z -> y) :: z' -> y'
-- x' = (z' -> y') -> x1,  therefore:
x :: x'                  y :: y'
   = (z' -> y') -> x1
x (\z -> y) :: x1

{- Assemble type of `(\x y -> x (\z -> y))` -}
(\x y -> x (\z -> y)) :: x' -> y' -> x1
                       = ((z' -> y') -> x1) -> y' -> x1
```

1.(a) 3. `map (1:)`

Solved in 1.(a) 3..

1.(a) 4. `(\x ys -> inits (map (x <) ys))`

Solved in 1.(a) 4..

2.19 Midterm FS10

Source: https://exams.vis.ethz.ch/exams/vnfwzzo2.pdf

1. 1. `(\x y z -> y z x)`

```
{- Find type of `(\x y z -> y z x)` -}
{- Initial input types of `(\x y z -> ...)` -}
x :: x'    y :: y'    z :: z'

{- Find type of `y z` -}
y :: __ -> y1
z :: z'
-- y' = z' -> y1,  therefore:
x :: x'    y :: y'          z :: z'
              = z' -> y1
y z :: y1

{- Find type of `y z x` -}
y z :: __ -> y2
x   :: x'
-- y1 = x' -> y2,  therefore:
x :: x'    y :: z' -> y1           z :: z'
              = z' -> (x' -> y2)
y z x :: y2

{- Assemble type of `(\x y z -> y z x)` -}
(\x y z -> y z x) :: x' -> y' -> z' -> y2
                   = x' -> (z' -> x' -> y2) -> z' -> y2
```

1. 2. `zipWith (==)`

Assume

- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `(==) :: Eq a => a -> a -> Bool`

```
{- Find type of `zipWith (==)` -}
zipWith ::        (a -> b ->  c  ) -> [a] -> [b] -> [c]
(==)    :: Eq e => e -> e -> Bool
-- a = Eq e => e,  b = Eq e => e,  c = Bool,  therefore:
zipWith (==) :: [a] -> [b] -> [c]
             = Eq e => [e] -> [e] -> [Bool]
```

1. 3. `(\x y -> y ((==) x)`

Assume

- `(==) :: Eq a => a -> a -> Bool`

```
{- Find type of `(\x y -> y ((==) x)` -}
{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `(==) x` -}
(==) :: Eq a => a -> a -> Bool
x    ::         x'
-- x' = Eq a => a,  therefore:
x :: x'           y :: y'
   = Eq a => a
(==) x :: Eq a => a -> Bool

{- Find type of `y ((==) x)` -}
y      ::         _________ -> y1
(==) x :: Eq a => a -> Bool
-- y' = Eq a => (a -> Bool) -> y1,  therefore:
x :: Eq a => a    y :: y'
                     = Eq a => (a -> Bool) -> y1
y ((==) x) :: y1

{- Assemble type of `(\x y -> y ((==) x)` -}
(\x y -> y ((==) x) :: x' -> y' -> y1
                     = Eq a => a -> ((a -> Bool) -> y1) -> y1
```

1. 4. `takeWhile (\x -> x 1)`

Assume

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
- `1 :: Num a => a`

```
{- Find type of `(\x -> x 1)` -}
{- Initial input types of `(\x -> ...)` -}
x :: x'

{- Find type of `x 1` -}
x ::          _ -> x1
1 :: Num a => a
-- x' = Num a => a -> x1,  therefore:
x :: x'
   = Num a => a -> x1
x 1 :: x1

{- Assemble type of `(\x -> x 1)` -}
(\x -> x 1) :: x' -> x1
             = Num a => (a -> x1) -> x1

{- Find type of `takeWhile (\x -> x 1)` -}
takeWhile   ::          (   b      -> Bool) -> [b] -> [b]
(\x -> x 1) :: Num a =>  (a -> x1) ->  x1
-- b = Num a => a -> x1,  x1 = Bool,  therefore:
takeWhile (\x -> x 1) :: [b] -> [b]
                       = Num a => [a -> x1] -> [a -> x1]
                       = Num a => [a -> Bool] -> [a -> Bool]
```

2.20 Midterm FS09

Source: https://exams.vis.ethz.ch/exams/5w58is4g.pdf

1.(a) 1. `(\x y z -> x (y z))`

Solved in 2.(a) 1..

1.(a) 2. `(\f -> (\h -> (f,h)))`

Solved in 2.(a) 2..

1.(a) 3. `map (elem 0)`

Assume

- `map :: (a -> b) -> [a] -> [b]`
- `elem :: Eq a => a -> [a] -> Bool`
- `0 :: Num a => a`

```
{- Find type of ‘elem 0‘ -}
elem :: Eq a  => a -> [a] -> Bool
0    :: Num b => b
-- Eq a => a = Num b => b,  therefore:
elem 0 :: [a] -> Bool
        = Num b => [b] -> Bool

{- Find type of ‘map (elem 0)‘ -}
map    ::          ( u  ->  v  ) -> [u] -> [v]
elem 0 :: Num b =>  [b] -> Bool
-- u = Num b => [b],  v = Bool,  therefore:
map (elem 0) :: [u] -> [v]
              = Num b => [[b]] -> [Bool]
```

1.(a) 4. `(\x -> x (<))`

Solved in 2.(a) 3..

2.21 Midterm FS08

Source: https://exams.vis.ethz.ch/exams/tahr0y7j.pdf

1.(a) I. `(\x -> snd x) (fst, \y z -> map y ('e':z))`

Assume

- `snd :: (a, b) -> b`
- `fst :: (a, b) -> a`
- `map :: (a -> b) -> [a] -> [b]`
- `'e' :: Char`
- `(:) :: a -> [a] -> [a]`

```
{- Transform into clearer form -}
  (\x -> snd x) (fst, \y z -> map y ('e':z))
= snd (fst, \y z -> map y ('e':z))
= \y z -> map y ('e':z)
= (\y z -> map y ((:) 'e' z))

{- Initial input types of `(\y z -> ...)` -}
y :: y'    z :: z'

{- Find type of `map y` -}
map :: (a -> b) -> [a] -> [b]
y   ::    y'
-- y' = a -> b,  therefore:
y :: y'        z :: z'
   = a -> b
map y :: [a] -> [b]

{- Find type of `(:) 'e'` -}
(:) ::  e  -> [e] -> [e]
'e' :: Char
-- e = Char,  therefore:
y :: a -> b    z :: z'
(:) 'e' :: [e] -> [e]
         = [Char] -> [Char]

{- Find type of `(:) 'e' z` -}
(:) 'e' :: [Char] -> [Char]
z       ::   z'
-- z' =  [Char],  therefore:
y :: a -> b    z :: z'
                  = [Char]
(:) 'e' z :: [Char]

{- Find type of `map y ((:) 'e' z))` -}
map y     :: [ a  ] -> [b]
(:) 'e' z :: [Char]
-- a = Char,  therefore:
y :: a -> b       z :: [Char]
   = 0Char@ -> b
map y ((:) 'e' z)) :: [b]

{- Assemble type of `(\y z -> map y ((:) 'e' z))` -}
(\y z -> map y ((:) 'e' z)) :: y' -> z' -> [b]
                             = (Char -> b) -> [Char] -> [b]
```

1.(a) II. `(\x y z -> tail x == map (/= z) y)`

Assume

- `tail :: [a] -> [a]`
- `(==) :: Eq a => a -> a -> Bool`
- `map :: (a -> b) -> [a] -> [b]`
- `(/=) :: Eq a => a -> a -> Bool`

```
{- Transform into clearer form -}
  (\x y z -> tail x == map (/= z) y)
= (\x y z -> tail x == map ((/=) z) y)
= (\x y z -> (==) (tail x) (map ((/=) z) y))

{- Initial input types of `(\x y z -> ...)` -}
x :: x'    y :: y'    z :: z'

{- Find type of `tail x` -}
tail :: [u] -> [u]
x    ::  x'
-- x' = [u],  therefore:
x :: x'     y :: y'    z :: z'
   = [u]
tail x :: [u]

{- Find type `(==) (tail x)` -}
(==)   :: Eq v => v -> v -> Bool
tail x ::        [u]
-- Eq v => v = [u],  therefore:
x :: [u]            y :: y'    z :: z'
   = Eq u => [u]
(==) (tail x) :: Eq v => v -> Bool
               = Eq u => [u] -> Bool

{- Find the type of `(/=) z` -}
(/=) :: Eq w => w -> w -> Bool
z    ::         z'
-- z' = Eq w => w,  therefore:
x :: Eq u => [u]    y :: y'    z :: z'
                                  = Eq w => w
(/=) z :: Eq w => w -> Bool

{- Find the type of `map ((/=) z)` -}
map    : :       (a ->  b  ) -> [a] -> [b]
(/=) z :: Eq w => w -> Bool
-- a = Eq w => w,  b = Bool,  therefore:
x :: Eq u => [u]    y :: y'    z :: Eq w => w
map ((/=) z) :: [a] -> [b]
              = Eq w => [w] -> [Bool]

{- Find the type of `map ((/=) z) y` -}
map ((/=) z) :: Eq w => [w] -> [Bool]
y            ::          y'
-- y' = Eq w => [w],  therefore:
x :: Eq u => [u]    y :: y'             z :: Eq w => w
                       = Eq w => [w]
map ((/=) z) y :: [Bool]

{- Find the type of `(==) (tail x) (map ((/=) z) y)` -}
(==) (tail x)  :: Eq u => [ u  ] -> Bool
map ((/=) z) y ::         [Bool]
-- u = Bool,  therefore:
x :: Eq u => [u]    y :: Eq w => [w]    z :: Eq w => w
   = [Bool]
(==) (tail x) (map ((/=) z) y) :: Bool

{- Assemble type of `(\x y z -> (==) (tail x) (map ((/=) z) y))` -}
(\x y z -> (==) (tail x) (map ((/=) z) y)) :: x' -> y' -> z' -> Bool
  = Eq w => [Bool] -> [w] -> w -> Bool
```

2.22 Session Exercises

TA1 `(\x -> x (<) + 1)`

Assume

- `(<) :: Ord a => a -> a -> Bool`
- `(+) :: Num a => a -> a -> a`
- `(1 :: Num a => a)`

```
{- Transform expression into clearer form -}
  (\x -> x (<) + 1)
= (\x -> (+) (x (<)) 1)

{- Initial input types of `(\x -> ...)` -}
x :: x'

{- Find type of `x (<)` -}
x   ::          _________________ -> x1
(<) :: Ord a => (a -> a -> Bool)
-- x' = Ord a => (a -> a -> Bool) -> x1,  therefore:
x :: x'
   = Ord a => (a -> a -> Bool) -> x1
x (<) :: x1

{- Find type of `(+) (x (<))` -}
(+)   :: Num n => n -> n -> n
x (<) ::          x1
-- x1 = Num n => n,  therefore:
x :: Ord a => (a -> a -> Bool) -> x1
   = (Ord a,Num n) => (a -> a -> Bool) -> n
(+) (x (<)) :: Num n => n -> n

{- Find type of `(+) (x (<)) 1` -}
(+) (x (<)) :: Num n => n -> n
1           :: Num n => n
-- therefore:
x :: (Ord a,Num n) => (a -> a -> Bool) -> n
(+) (x (<)) 1 :: Num n => n

{- Assemble type of `(\x -> (x (<)) 1)` -}
(\x -> (x (<)) 1) :: Num n => x' -> n
                   = (Ord a,Num n) => ((a -> a -> Bool) -> n) -> n
```

TA2 `(\x a b -> x (a (b x) x))`

```
{- Find type of `(\x a b -> x (a (b x) x))` -}
{- Initial input types of `(\x a b -> ...)` -}
x :: x'    a :: a'    b :: b'

{- Find type of `b x` -}
b :: __ -> b1
x :: x'
-- b' = x' -> b1,  therefore:
x :: x'    a :: a'    b :: b'
                         = x' -> b1
b x :: b1

{- Find type of `a (b x)` -}
a   :: __ -> a1
b x :: b1
-- a' = b1 -> a1,  therefore:
x :: x'    a :: a'          b :: x' -> b1
              = b1 -> a1
a (b x) :: a1

{- Find type of `a (b x) x` -}
a (b x) :: __ -> a2
x       :: x'
-- a1 = x' -> a2,  therefore:
x :: x'    a :: b1 -> a1          b :: x' -> b1
              = b1 -> x' -> a2
a (b x) x :: a2

{- Find type of `x (a (b x) x)` -}
x         :: __ -> x1
a (b x) x :: a2
-- x' = a2 -> x1,  therefore:
x :: x'          a :: b1 -> x' -> a2            b :: x' -> b1
   = a2 -> x1       = b1 -> (a2 -> x1) -> a2       = (a2 -> x1) -> b1
x (a (b x) x) :: x1

{- Assemble type of `(\x a b -> x (a (b x) x))` -}
(\x a b -> x (a (b x) x))
    :: x' -> a' -> b' -> x1
     = (a2 -> x1) -> (b1 -> (a2 -> x1) -> a2) -> ((a2 -> x1) -> b1) -> x1
```

2.23 MaxS Exercises

Source: https://n.ethz.ch/~mschlegel/fmfp23/fmfp.html

W4 `(\x -> \y -> x + y)`

Assume

- `(+) :: Num a => a -> a -> a`

```
{- Transform expression into clearer form -}
  (\x -> \y -> x + y)
= (\x y -> x + y)
= (\x y -> (+) x y)

{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `(+) x` -}
(+) :: Num a => a -> a -> a
x   ::          x'
-- x' = Num a => a,  therefore
x :: x'            y :: y'
   = Num a => a
(+) x :: Num a => a -> a

{- Find type of `(+) x y` -}
(+) x :: Num a => a -> a
y     ::          y'
-- y' = Num a => a,  therefore
x :: Num a => a    y :: y'
                      = Num a => a
(+) x y :: Num a => a

{- Assemble type of `(\x y -> (+) x y)` -}
(\x y -> (+) x y) :: Num a => x' -> y' -> a
                   = Num a => a -> a -> a
```

W4' `filter (<3)`

Assume

- `filter :: (a -> Bool) -> [a] -> [a]`
- `(<) :: Ord a => a -> a -> Bool`
- `3 :: Num a => a`

```
{- Transform expression into clearer form -}
  filter (<3)
= filter ((<) 3)

{- Find type of '(<) 3' -}
(<) :: Ord a => a -> a -> Bool
3   :: Num n => n
-- Num n => n = Ord a => a,  therefore
(<) 3 :: Ord a => a -> Bool
       = (Ord n,Num n) => n -> Bool

{- Find type of 'filter ((<) 3)' -}
filter ::                 (x -> Bool) -> [x] -> [x]
(<) 3  :: (Ord n,Num n) => n -> Bool
-- x = (Ord n,Num n) => n,  therefore
filter ((<) 3) :: [x] -> [x]
               = (Ord n,Num n) => [n] -> [n]
```

W5 `(\x -> (head x) True)`

Assume

- `head :: [a] -> a`

```
{- Find type of ‘(\x -> (head x) True)‘ -}
{- Initial input types of ‘(\x -> ...)‘ -}
x :: x’

{- Find type of ‘head x‘ -}
head :: [a] -> a
x    ::  x’
-- x’ = [a],  therefore
x :: x’
   = [a]
head x :: a

{- Find type of ‘(head x) True‘ -}
head x :: ____ -> a1
True   :: Bool
-- a = Bool -> a1,  therefore:
x :: [a]
   = [(Bool -> a1)]
(head x) True :: a1

{- Assemble type of ‘(\x -> (head x) True)‘ -}
(\x -> (head x) True) :: x’ -> a1
                      = [(Bool -> a1)] -> a1
```

W5' `(\x y -> map (y<) x)`

Assume

- `map :: (a -> b) -> [a] -> [b]`
- `(<) :: Ord a => a -> a -> Bool`

```
{- Transform expression into clearer form -}
  (\x y -> map (y<) x)
= (\x y -> map ((<) y) x)

{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `(<) y` -}
(<) :: Ord o => o -> o -> Bool
y   ::          y'
-- y' = Ord o => o,  therefore:
x :: x'    y :: y'
              = Ord o => o
(<) y :: Ord o => o -> Bool

{- Find type of `map ((<) y)` -}
map   ::         (a ->  b  ) -> [a] -> [b]
(<) y :: Ord o => o -> Bool
-- a = Ord o => o,  b = Bool,  therefore:
x :: x'    y :: Ord o => o
map ((<) y) :: [a] -> [b]
             = Ord o => [o] -> [Bool]

{- Find type of `map ((<) y) x` -}
map ((<) y) :: Ord o => [o] -> [Bool]
x           ::          x'
-- x' = Ord o => [o],  therefore:
x :: x'              y :: Ord o => o
   = Ord o => [o]
map ((<) y) x :: [Bool]

{- Assemble type of `(\x y -> map ((<) y) x)` -}
(\x y -> map ((<) y) x) :: x' -> y' -> [Bool]
                         = Ord o => [o] -> o -> [Bool]
```

W5” `(\x y -> head (x map) + 100)`

Assume

- `head :: [a] -> a`
- `map :: (a -> b) -> [a] -> [b]`
- `(+) :: Num a => a -> a -> a`
- (`100 :: Num a => a`)

```
{- Transform expression into clearer form -}
  (\x y -> head (x map) + 100)
= (\x y -> (+) (head (x map)) 100)

{- Initial input types of `(\x y -> ...)` -}
x :: x'    y :: y'

{- Find type of `x map` -}
x   :: ______________________ -> x1
map :: (a -> b) -> [a] -> [b]
-- x' = ((a -> b) -> [a] -> [b]) -> x1,  therefore:
x :: x'                                y :: y'
   = ((a -> b) -> [a] -> [b]) -> x1
x map :: x1

{- Find type of `head (x map)` -}
head  :: [e] -> e
x map :: x1
-- x1 = [e],  therefore
x :: ((a -> b) -> [a] -> [b]) -> x1     y :: y'
   = ((a -> b) -> [a] -> [b]) -> [e]
head (x map) :: e

{- Find type of `(+) (head (x map))` -}
(+)          :: Num n => n -> n -> n
head (x map) ::          e
-- e = n,  therefore:
x :: ((a -> b) -> [a] -> [b]) -> [e]             y :: y'
   = Num n => ((a -> b) -> [a] -> [b]) -> [n]
(+) (head (x map)) :: Num n => n -> n

{- Find type of `(+) (head (x map)) 100` -}
(+) (head (x map)) :: Num n => n -> n
100                :: Num n => n
-- therefore:
x :: Num n => ((a -> b) -> [a] -> [b]) -> [n]    y :: y'
(+) (head (x map)) 100 :: Num n => n

{- Assemble type of `(\x y -> (+) (head (x map)) 100)` -}
(\x y -> (+) (head (x map)) 100)
    :: Num n => x' -> y' -> n
     = Num n => (((a -> b) -> [a] -> [b]) -> [n]) -> y' -> n
```

2.24 Further Exercises

Ty1 `(head .) . zip`

Assume

- `head :: [a] -> a`
- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `zip :: [a] -> [b] -> [(a,b)]`

```
{- Transform expression into clearer form -}
  (head .) . zip
= (.) (head .) zip
= (.) ((.) head) zip

{- Find type of `(.) head` -}
(.)  :: ( b  -> c) -> (a -> b) -> (a -> c)
head ::  [t] -> t
-- b = [t],  c = t,  therefore:
(.) head :: (a -> b) -> (a -> c)
          = (a -> [t]) -> a -> t

{- Find type of `(.) ((.) head)` -}
(.)      :: (   y       ->    z    ) -> (x -> y) -> (x -> z)
(.) head ::  (a -> [t]) -> (a -> t)
-- y = a -> [t],  z = a -> t,  therefore:
(.) ((.) head) :: (x -> y) -> (x -> z)
                = (x -> (a -> [t])) -> (x -> (a -> t))
                = (x -> a -> [t]) -> x -> a -> t

{- Find type of `(.) ((.) head) zip` -}
(.) ((.) head) :: ( x  ->  a  -> [  t  ]) -> x -> a -> t
zip            ::  [i] -> [j] -> [(i,j)]
-- x = [i],  a = [j],  t = (i,j),  therefore:
(.) ((.) head) zip :: x -> a -> t
                    = [i] -> [j] -> (i,j)
```

Ty2 `reverse >>= (==)`

Remark: This function checks palindromes.

Assume

- `reverse :: [a] -> [a]`
- `(>>=) :: (c -> a) -> (a -> c -> b) -> c -> b`
- `(==) :: Eq a => a -> a -> Bool`

```
{- Transform expression into clearer form -}
  reverse >>= (==)
= (>>=) reverse (==)

{- Find type of `(>>=) reverse` -}
(>>=)   :: ( c  ->  a ) -> (a -> c -> b) -> c -> b
reverse ::  [x] -> [x]
-- c = [x],  a = [x],  therefore:
(>>=) reverse :: (a -> c -> b) -> c -> b
               = ([x] -> [x] -> b) -> [x] -> b

{- Find type of `(>>=) reverse (==)` -}
(>>=) reverse ::        ([x] -> [x] ->  b  ) -> [x] -> b
(==)          :: Eq a =>  a  ->  a  -> Bool
-- [x] = Eq a => a,  b = Bool,  therefore:
(>>=) reverse (==) :: [x] -> b
                    = Eq x => [x] -> Bool
```

Ty3 `(\op f -> \x y -> op (f x) (f y))`

Remark: This is the function 'on' from Data.Function.

```
{- Transform expression into clearer form -}
  (\op f -> \x y -> op (f x) (f y))
= (\op f x y -> op (f x) (f y))

{- Initial input types of `(\op f x y -> ...)` -}
op :: op'    f :: f'    x :: x'    y :: y'

{- Find type of `f x` -}
f :: __ -> f1
x :: x'
-- f' = x' -> f1,  therefore:
op :: op'    f :: f'          x :: x'    y :: y'
                = x' -> f1
f x :: f1

{- Find type of `f y` -}
f :: x' -> f1
y :: y'
-- y' = x',  therefore:
op :: op'    f :: x' -> f1    x :: x'    y :: y'
                                           = x'
f y :: f1

{- Find type of `op (f x)` -}
op  :: __  -> op1
f x :: f1
-- op' = f1 -> op1,  therefore:
op :: op'          f :: x' -> f1    x :: x'    y :: x'
    = f1 -> op1
op (f x) :: op1

{- Find type of `op (f x) (f y)` -}
op (f x) :: __ -> op2
f y      :: f1
-- op1 = f1 -> op2,  therefore:
op :: f1 -> op1            f :: x' -> f1    x :: x'    y :: x'
    = f1 -> (f1 -> op2)
op (f x) (f y) :: op2

{- Assemble type of `(\op f x y -> op (f x) (f y))` -}
(\op f x y -> op (f x) (f y))
    :: op' -> f' -> x' -> y' -> op2
     = (f1 -> f1 -> op2) -> (x' -> f1) -> x' -> x' -> op2
```

Ty4 `uncurry fst`

Assume

- `uncurry :: (a -> b -> c) -> (a,b) -> c`
- `fst :: (a,b) -> a`

```
{- Find type of ‘uncurry fst‘ -}
uncurry :: (  a   -> (b -> c)) -> (a,b) -> c
fst     ::  (x,y) ->    x
-- a = (x,y),  (b -> c) = x,  therefore
uncurry fst :: (a, b) -> c
            = ((x,y), b) -> c
            = ((b -> c,y), b) -> c
```

Ty5 `uncurry curry`

Assume

- `uncurry :: (a -> b -> c) -> (a,b) -> c`
- `curry :: ((a,b) -> c) -> a -> b -> c`

```
{- Find type of ‘uncurry curry‘ -}
uncurry :: (      x     -> y ->    z    ) -> (x,y) -> z
curry   :: ((a,b) -> c) -> a -> (b -> c)
-- x = ((a,b) -> c),  y = a,  z = (b -> c),  therefore
uncurry curry :: (x, y) -> z
              = ((a,b) -> c, a) -> (b -> c)
              = ((a,b) -> c, a) -> b -> c
```

Ty6 `uncurry (flip (,))`

Assume

- `uncurry :: (a -> b -> c) -> (a,b) -> c`
- `flip :: (a -> b -> c) -> b -> a -> c`
- `(,) :: a -> b -> (a,b)`

```
{- Find type of ‘flip (,)‘ -}
flip :: (a -> b ->   c  ) -> b -> a -> c
(,)  ::  u -> v -> (u,v)
-- a = u,  b = v,  c = (u,v),  therefore:
flip (,) :: b -> a -> c
         = v -> u -> (u,v)

{- Find type of ‘uncurry (flip (,))‘ -}
uncurry  :: (x -> y ->   z  ) -> (x,y) -> z
flip (,) ::  v -> u -> (u,v)
-- x = v,  y = u,  z = (u,v),  therefore:
uncurry (flip (,)) :: (x,y) -> z
                   = (v,u) -> (u,v)
```

Ty7 `foldr (.) id`

Assume

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `id :: a -> a`

```
{- Find type of ‘foldr (.)‘ -}
foldr :: (   a     ->    b     ->    b    ) -> b -> [a] -> b
(.)   ::  (y -> z) -> (x -> y) -> (x -> z)
-- a = y -> z,  b = x -> y,  b = x -> z;  y = z,  therefore:
foldr (.) :: b -> [a] -> b
           = (x -> y) -> [(y -> z)] -> (x -> y)
           = (x -> z) -> [(z -> z)] -> (x -> z)

{- Find type of ‘foldr (.) id‘ -}
foldr (.) :: (x -> z) -> [(z -> z)] -> (x -> z)
id        ::  u -> u
-- x = u,  z = u,  therefore:
foldr (.) id :: [(z -> z)] -> (x -> z)
              = [(u -> u)] -> (u -> u)
```

Ty8 `\f -> foldr ((:) . f) []`

Remark: This is the function 'map' from Prelude.

Assume

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

```
{- Transform expression into clearer form -}
  \f -> foldr ((:) . f) []
= \f -> foldr ((.) (:) f) []

{- Initial input types of `(\f -> ...)` -}
f :: f'

{- Find type of `(.) (:)` -}
(.) :: (b ->     c     ) -> (a -> b) -> (a -> c)
(:) ::  e -> [e] -> [e]
-- b = e,  c = [e] -> [e],  therefore:
f :: f'
(.) (:) :: (a -> b) -> (a -> c)
         = (a -> e) -> a -> [e] -> [e]

{- Find type of `(.) (:) f` -}
(.) (:) :: (a -> e) -> a -> [e] -> [e]
f       ::    f'
-- f' = (a -> e),  therefore:
f :: f'
   = (a -> e)
(.) (:) f :: a -> [e] -> [e]

{- Find type of `foldr ((.) (:) f)` -}
foldr     :: (x ->  y  ->  y ) -> y -> [x] -> y
(.) (:) f ::  a -> [e] -> [e]
-- x = a,  y = [e],  therefore:
f :: a -> e
foldr ((.) (:) f) :: y -> [x] -> y
                   = [e] -> [a] -> [e]

{- Find type of `foldr ((.) (:) f) []` -}
foldr ((.) (:) f) :: [e] -> [a] -> [e]
[]                :: [_]
-- _ = e,  therefore:
f :: a -> e
foldr ((.) (:) f) [] :: [a] -> [e]

{- Assemble type of `(\f -> foldr ((.) (:) f) [])` -}
(\f -> foldr ((.) (:) f) []) :: f' -> [a] -> [e]
                              = (a -> e) -> [a] -> [e]
```

Ty9 `(\p f -> head . filter p . iterate f)`

Remark: This is the function 'until' from Prelude.

Assume

- `head :: [a] -> a`
- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `iterate :: (a -> a) -> a -> [a]`

```
{- Transform expression into clearer form -}
  (\p f -> head . filter p . iterate f)
= (\p f -> head . (filter p . iterate f))
= (\p f -> (.) head ((.) (filter p) (iterate f)))

{- Initial input types of `(\p f -> ...)` -}
p :: p'    f :: f'

{- Find type of `filter p` -}
filter :: (x -> Bool) -> [x] -> [x]
p      ::    p'
-- p' = (x -> Bool),  therefore:
p :: p'             f :: f'
   = (x -> Bool)
filter p :: [x] -> [x]

{- Find type of `iterate f` -}
iterate :: (y -> y) -> y -> [y]
f       ::    f'
-- f' = (y -> y),  therefore:
p :: x -> Bool    f :: f'
                    = (y -> y)
iterate f :: y -> [y]

{- Find type of `(.) (filter p)` -}
(.)      :: ( b  ->  c ) -> (a -> b) -> (a -> c)
filter p ::  [x] -> [x]
-- b = [x],  c = [x],  therefore:
p :: x -> Bool    f :: y -> y
(.) (filter p) :: (a -> b) -> (a -> c)
                = (a -> [x]) -> a -> [x]

{- Find type of `(.) (filter p) (iterate f)` -}
(.) (filter p) :: (a -> [x]) -> a -> [x]
iterate f      ::  y -> [y]
-- a = y,  x = y,  therefore:
p :: x -> Bool    f :: y -> y
   = y -> Bool
(.) (filter p) (iterate f) :: a -> [x]
                            = y -> [y]

{- Find type of `(.) head` -}
(.)  :: ( v  -> w) -> (u -> v) -> (u -> w)
head ::  [e] -> e
-- v = [e],  w = e,  therefore:
p :: y -> Bool    f :: y -> y
(.) head :: (u -> v) -> (u -> w)
         = (u -> [e]) -> u -> e

{- Find type of `(.) head ((.) (filter p) (iterate f))` -}
(.) head                    :: (u -> [e]) -> u -> e
(.) (filter p) (iterate f) ::  y -> [y]
-- u = y,  e = y,  therefore:
p :: y -> Bool    f :: y -> y
(.) head ((.) (filter p) (iterate f)) :: u -> e
                                       = y -> y

{- Assemble type of `(\p f -> (.) head ((.) (filter p) (iterate f)))` -}
(\p f -> (.) head ((.) (filter p) (iterate f)))
    :: p' -> f' -> y -> y
     = (y -> Bool) -> (y -> y) -> y -> y
```