Functional programming - tutorial 3

Arnold Meijster

Dept. computer science (university of Groningen)

September 21, 2020

11.8 `total` function

Define a function

```
total ::  (Integer -> Integer) -> (Integer -> Integer)
```

such that `(total f) n` returns `f 0 + f 1 + ... + f n`

11.8 total function

Define a function

```
total ::  (Integer -> Integer) -> (Integer -> Integer)
```

such that `(total f) n` returns `f 0 + f 1 + ... + f n`

Several (equivalent) solutions are possible:

```
total f n = sum (map f [0..n])
```

```
total f n = foldr (+) 0 (map f [0..n])
```

```
total f = (\n -> foldr (+) 0 (map f [0..n]))
```

Given a function `f` of the type `a -> b -> c`, write a lambda expression that describes the function of type `b -> a -> c` that behaves like `f` but takes its arguments in the other order. Using this expression, give a definition of the function `flip :: (a -> b -> c) -> (b -> a -> c)` which reverses the order in which its function argument takes its arguments.

Given a function `f` of the type `a -> b -> c`, write a lambda expression that describes the function of type `b -> a -> c` that behaves like `f` but takes its arguments in the other order. Using this expression, give a definition of the function `flip :: (a -> b -> c) -> (b -> a -> c)` which reverses the order in which its function argument takes its arguments.

```
-- note: flip is defined in the prelude, therefore
-- we use the name 'flipArgs'.
flipArgs :: (a -> b -> c) -> (b -> a -> c)
flipArgs f = (\x y -> f y x)
```

Using the following definitions:

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

What is the effect of `uncurry ($)`? What is its type?
Answer similar questions for `uncurry (:)`, and `uncurry (.)`.

Using the following definitions:

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

What is the effect of `uncurry ($)`? What is its type?
Answer similar questions for `uncurry (:)`, and `uncurry (.)`.

The type of `uncurry ($)` is `(a -> b, a) -> b`. To understand this, it is a good idea to rename type variables:

```
uncurry :: (x -> y -> z) -> (x, y) -> z
($)     :: (a -> b) -> a -> b
```

Now it is obvious that `x <==> a -> b`, `y <==> a`, and `z <==> b`.
Hence `uncurry ($) :: (a -> b, a) -> b`

The next question is, what does it do?

11.14 (continued)

The next question is, what does it do?
The surprising thing is that you can answer this question without knowing what `uncurry` and `($)` actually do. Only the type `uncurry ($) :: (a -> b, a) -> b` is enough to answer this question.

11.14 (continued)

The next question is, what does it do?
The surprising thing is that you can answer this question without knowing what `uncurry` and `($)` actually do. Only the type `uncurry ($) :: (a -> b, a) -> b` is enough to answer this question.

Note that `uncurry ($)` returns something of type `b`, which is a type variable, not a concrete type. The function in the tuple (1st argument) returns values of type `b` and expects a value of type `a`, which can only be found in the same tuple. There are no concrete types, so the only thing `uncurry ($)` can do is to take the `snd` of the tuple, supply it as an argument to the `fst` of the tuple, and return whatever it returns.
This is easily shown using a few examples in ghci:
`uncurry ($) ((+1), 0)` yields 1
`uncurry ($) (even, 0)` yields `True`
`uncurry ($) ((2^), 3)` yields 8

Now, we do the same for `uncurry (:)`.

```
uncurry :: (x ->  y  ->  z) -> (x, y) -> z
(:)     ::  a -> [a] -> [a]
```

Now, we do the same for `uncurry (:)`.

```
uncurry :: (x ->  y  ->  z) -> (x, y) -> z
(:)     ::  a -> [a] -> [a]
```

Now it is obvious that `x <==> a`, `y <==> [a]`, and `z <==> [a]`.
Hence `uncurry (:) :: (a, [a]) -> [a]`

11.14 (continued)

Now, we do the same for `uncurry (:)`.

```
uncurry :: (x ->  y  ->  z) -> (x, y) -> z
(:)     ::  a -> [a] -> [a]
```

Now it is obvious that `x <==> a`, `y <==> [a]`, and `z <==> [a]`.
Hence `uncurry (:) :: (a, [a]) -> [a]`

Since `uncurry f (x, y) = f x y`, we simply have
`uncurry (:) (x,xs) = (:) x xs = x:xs`

Now, we do the same for `uncurry (:)`.

```
uncurry :: (x -> y -> z) -> (x, y) -> z
(:)     ::  a -> [a] -> [a]
```

Now it is obvious that `x <==> a`, `y <==> [a]`, and `z <==> [a]`.
Hence `uncurry (:) :: (a, [a]) -> [a]`

Since `uncurry f (x, y) = f x y`, we simply have
`uncurry (:) (x,xs) = (:) x xs = x:xs`

For example, `uncurry (:) (1,[2,3])` yields `[1,2,3]`

Finally, we do the same for `uncurry (.)`

```
uncurry :: (x -> y -> z) -> (x, y) -> z
(.)     :: (b -> c) -> (a -> b) -> a -> c
```

Finally, we do the same for `uncurry (.)`

```
uncurry :: (x -> y -> z) -> (x, y) -> z
(.)     :: (b -> c) -> (a -> b) -> a -> c
```

Now, we have `x <==> b->c`, `y <==> a->b`, and `z <==> a->c`

11.14 (continued)

Finally, we do the same for `uncurry (.)`

```
uncurry :: (x -> y -> z) -> (x, y) -> z
(.)     :: (b -> c) -> (a -> b) -> a -> c
```

Now, we have `x <==> b->c`, `y <==> a->b`, and `z <==> a->c`

Hence, `uncurry (.) :: (b -> c, a -> b) -> a -> c`

11.14 (continued)

Finally, we do the same for `uncurry (.)`

```
uncurry :: (x -> y -> z) -> (x, y) -> z
(.)     :: (b -> c) -> (a -> b) -> a -> c
```

Now, we have `x <==> b->c`, `y <==> a->b`, and `z <==> a->c`

Hence, `uncurry (.) :: (b -> c, a -> b) -> a -> c`

Since `uncurry f (x, y) = f x y`, we simply have
`uncurry (.) (f,g) = (.) f g = f.g`

For example,
`(uncurry (.)) ((*2), (+1)) 1` yields `(*2) ((+1) 1) = (*2) 2 = 4`

11.15

What are the types and effects of `uncurry uncurry` and `curry curry`?

11.15

What are the types and effects of `uncurry uncurry` and `curry curry`?

We use the same technique as in 11.14.

```
uncurry :: (x -> y -> z) -> (x, y) -> z
uncurry :: (a -> b -> c) -> (a, b) -> c
```

So, `x <==> a -> b -> c`, `y <==> (a,b)` and `z <==> c`

What are the types and effects of `uncurry uncurry` and `curry curry`?

We use the same technique as in 11.14.

```
uncurry :: (x -> y -> z) -> (x, y) -> z
uncurry :: (a -> b -> c) -> (a, b) -> c
```

So, `x <==> a -> b -> c`, `y <==> (a,b)` and `z <==> c`
Hence `uncurry uncurry :: (a -> b -> c, (a,b)) -> c`

11.15

What are the types and effects of `uncurry uncurry` and `curry curry`?

We use the same technique as in 11.14.

```
uncurry :: (x -> y -> z) -> (x, y) -> z
uncurry :: (a -> b -> c) -> (a, b) -> c
```

So, `x <==> a -> b -> c`, `y <==> (a,b)` and `z <==> c`
Hence `uncurry uncurry :: (a -> b -> c, (a,b)) -> c`
Let `f :: a - > b -> c` and `x::a` and `y::b`, then
`uncurry uncurry (f, (x,y)) = uncurry f (x,y) = f x y`
For example, `uncurry uncurry ((*),(2,3))` yields 6.

What are the types and effects of `uncurry uncurry` and `curry curry`?

We use the same technique as in 11.14.

```
uncurry :: (x -> y -> z) -> (x, y) -> z
uncurry :: (a -> b -> c) -> (a, b) -> c
```

So, `x <==> a -> b -> c`, `y <==> (a,b)` and `z <==> c`
Hence `uncurry uncurry :: (a -> b -> c, (a,b)) -> c`
Let `f :: a - > b -> c` and `x::a` and `y::b`, then
`uncurry uncurry (f, (x,y)) = uncurry f (x,y) = f x y`
For example, `uncurry uncurry ((*),(2,3))` yields 6.

The second part of the exercise is actually not 'fair'.

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

It is not possible to match `((x,y) -> z)` with the type of uncurry.

Is it possible to define the functions

```
curry3 ::  ((a, b, c) -> d) -> (a -> b -> c -> d)
uncurry3 ::  (a -> b -> c -> d) -> ((a, b, c) -> d)
```

which perform the analogue of `curry` and `uncurry` but for three arguments rather than two? Is it possible to use `curry` and `uncurry` in these definitions?

Is it possible to define the functions

```
curry3 ::  ((a, b, c) -> d) -> (a -> b -> c -> d)
uncurry3 ::  (a -> b -> c -> d) -> ((a, b, c) -> d)
```

which perform the analogue of `curry` and `uncurry` but for three arguments rather than two? Is it possible to use `curry` and `uncurry` in these definitions?

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry3 f a b c = f (a,b,c)

uncurry3 :: (a -> b -> c -> d) -> ((a, b, c) -> d)
uncurry3 f (a,b,c) = f a b c
-- I do not see how curry, uncurry would be useful
-- in curry3 and uncurry3.
```

```
iter :: Integer -> (a -> a) -> (a -> a)
iter n f
  | n > 0 = f .  iter (n - 1) f
  | otherwise = id

double :: Num a => a -> a
double x = 2*x

add :: Num a => a -> a -> a
add x y = x + y

sq :: Num a => a -> a
sq x = x * x

succ :: Integer -> Integer
succ n = n + 1

comp2 :: (a -> b) -> (b -> b -> c) -> (a -> a -> c)
comp2 f g = (\x y -> g
```

What is the output of:

```
iter 3 double 1
(comp2 succ (*)) 3 4
comp2 sq add 3 4
```

What is the type and effect of the function `\n -> iter n succ`?

`iter 3 double 1` yields `8`, because

```
     iter 3 double 1
   = (double . iter 2 double) 1
   = (double . double . iter 1 double) 1
   = (double . double . double . iter 0 double) 1
   = (double . double . double . id) 1
   = (double . double . double) 1
   = (double . double) 2
   = double 4 = 8
```

1.19/20 (continued)

```
(comp2 succ (*)) 3 4 yields 20, because
    (comp2 succ (*)) 3 4
  = (\x y -> (*) (succ x) (succ y)) 3 4
  = (*) (succ 3) (succ 4)
  = (*) 4 (succ 4) (*) 4 5
  = 20
```

```
(comp2 succ (*)) 3 4 yields 20, because
    (comp2 succ (*)) 3 4
  = (\x y -> (*) (succ x) (succ y)) 3 4
  = (*) (succ 3) (succ 4)
  = (*) 4 (succ 4) (*) 4 5
  = 20
```

```
comp2 sq add 3 4 yields 25, because
     comp2 sq add 3 4
   = (\x y -> add (sq x) (sq y)) 3 4
   = add (sq 3) (sq 4)
   = add 9 (sq 4)
   = add 9 16
   = 9 + 16
   = 25
```

```
(comp2 succ (*)) 3 4 yields 20, because
    (comp2 succ (*)) 3 4
  = (\x y -> (*) (succ x) (succ y)) 3 4
  = (*) (succ 3) (succ 4)
  = (*) 4 (succ 4) (*) 4 5
  = 20

comp2 sq add 3 4 yields 25, because
     comp2 sq add 3 4
   = (\x y -> add (sq x) (sq y)) 3 4
   = add (sq 3) (sq 4)
   = add 9 (sq 4)
   = add 9 16
   = 9 + 16
   = 25

(\n -> iter n succ) applies n times succ on its argument.

(\n -> iter n succ) 10 32 = succ(succ(....(succ(32)....))=42
```

11.21

Give an alternative 'constructive' definition of `iter` which creates the list of `n` copies of `f`, i.e. `[f, f,..., f]`, and then composes these function by folding the operator `.` to give
`f . f . ... .f`.

Give an alternative 'constructive' definition of `iter` which creates the list of `n` copies of `f`, i.e. `[f, f,..., f]`, and then composes these function by folding the operator `.` to give
`f . f . ... .f`.

```
iter2 :: Int -> (a -> a) -> (a -> a)
iter2 n f = foldr (.) id (replicate n f)
```

Define the function `splits :: [a] -> [([a], [a])]` which defines the list of all the ways that a list can be split in two. For example
`splits "Spy" = [("", "Spy"), ("S", "py"), ("Sp", "y"), ("Spy", "")]`

Define the function `splits :: [a] -> [([a], [a])]` which defines the list of all the ways that a list can be split in two. For example
`splits "Spy" = [("", "Spy"), ("S", "py"), ("Sp", "y"), ("Spy", "")]`

```
splits :: [a] -> [([a],[a])]
splits []     = [([],[])]
splits (x:xs) = ([],(x:xs)):
                (zip (map (x:) (map fst (splits xs))) (map snd (splits xs)))
```

Define the function `splits :: [a] -> [([a], [a])]` which defines the list of all the ways that a list can be split in two. For example
`splits "Spy" = [("", "Spy"), ("S", "py"), ("Sp", "y"), ("Spy", "")]`

```
splits :: [a] -> [([a],[a])]
splits []     = [([],[])]
splits (x:xs) = ([],(x:xs)):
               (zip (map (x:) (map fst (splits xs))) (map snd (splits xs)))


-- A very nice solution is (in case you know the functions inits and tails):
splits2 :: [a] -> [([a],[a])]
splits2 xs = zip (inits xs) (tails xs)
```

17.2

Using the list comprehension notation, define the functions

```
  sublists, subsequences ::  [a]-> [[a]]
```

which return all the sublists and subsequences of a list.

To refresh: a sublist is obtainedby omitting some elements of a list, a subsequence is a continuous block that is part of the list.

17.2

Using the list comprehension notation, define the functions
`sublists, subsequences ::  [a]-> [[a]]`
which return all the sublists and subsequences of a list.

To refresh: a sublist is obtainedby omitting some elements of a list, a subsequence is a continuous block that is part of the list.

```
subLists :: [a] -> [[a]]
subLists [] = [[]]
subLists (x:xs) = [x:sub | sub <- subLists xs] ++ subLists xs
```

Using the list comprehension notation, define the functions

```
sublists, subsequences ::  [a]-> [[a]]
```

which return all the sublists and subsequences of a list.

To refresh: a sublist is obtainedby omitting some elements of a list, a subsequence is a continuous block that is part of the list.

```
subLists :: [a] -> [[a]]
subLists [] = [[]]
subLists (x:xs) = [x:sub | sub <- subLists xs] ++ subLists xs

subSequences :: [a] -> [[a]]
subSequences xs =
  []:[take j (drop i xs) | i <- [0..len-1], j <- [1..len-i]]
    where len = length xs
```

17.23

Define the infinite lists of factorial and Fibonacci numbers.

```
factorial = [1, 1, 2, 6, 24, 120, 720, ...]
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, ...]
```

Define the infinite lists of factorial and Fibonacci numbers.

```
factorial = [1, 1, 2, 6, 24, 120, 720, ...]
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, ...]
```

```
factorials :: [Integer]
factorials = 1 : zipWith (*) factorials [1..]
```

Define the infinite lists of factorial and Fibonacci numbers.

```
factorial = [1, 1, 2, 6, 24, 120, 720, ...]
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, ...]
```

```
factorials :: [Integer]
factorials = 1 : zipWith (*) factorials [1..]

fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Give a definition of the function

```
factors ::  Integer -> [Integer]
```

which returns a list containing the factors of a positive integer.
For example, `factors 12 = [1,2,3,4,6,12]`.

Using this function, define the list of numbers whose only prime factors are 2, 3 and 5, to give the so-called Hamming Numbers.

17.24

Give a definition of the function

```
  factors ::  Integer -> [Integer]
```

which returns a list containing the factors of a positive integer.
For example, `factors 12 = [1,2,3,4,6,12]`.

Using this function, define the list of numbers whose only prime factors are 2, 3 and 5, to give the so-called Hamming Numbers.

```
factors :: Integer -> [Integer]
factors n = [d | d <- [1..n], n `mod` d == 0]
```

Give a definition of the function

```
factors ::  Integer -> [Integer]
```

which returns a list containing the factors of a positive integer.
For example, `factors 12 = [1,2,3,4,6,12]`.

Using this function, define the list of numbers whose only prime factors are 2, 3 and 5, to give the so-called Hamming Numbers.

```
factors :: Integer -> [Integer]
factors n = [d | d <- [1..n], n `mod` d == 0]
```

```
-- BEWARE: error in book. 1 is not a hamming number! Moreove,
-- in my opinion, it is easier compute them without using factors.
hamming :: [Integer]
hamming = tail hamlist
  where
    hamlist = 1:merge3 (map (2*) hamlist) (map (3*) hamlist)
                       (map (5*) hamlist)
    merge3 xs ys zs = merge xs (merge ys zs)
    merge (x:xs) (y:ys)
      | x < y     = x : merge xs (y:ys)
      | x > y     = y : merge (x:xs) ys
      | otherwise = x : merge xs ys
```

Define the function

```
runningSums ::  [Integer] -> [Integer]
```

which calculates the running sums
`[0, a0, a0 + a1, a0 + a1 + a2, ...]` of a list
`[a0, a1,a2, ...]`.

Define the function

`runningSums :: [Integer] -> [Integer]`

which calculates the running sums

`[0, a0, a0 + a1, a0 + a1 + a2, ...]` of a list

`[a0, a1,a2, ...]`.

```
runningSums :: [Integer] -> [Integer]
runningSums xs = sumlist xs 0
    where
      sumlist [] a = []
      sumlist (x:xs) a = (a+x) : sumlist xs (a+x)
```

17.29

How would you merge two infinite lists, assuming that they are ascending? How would you remove duplicates from the list which results? As an example, how would you merge the lists of powers of 2 and 3?

How would you merge two infinite lists, assuming that they are ascending? How would you remove duplicates from the list which results? As an example, how would you merge the lists of powers of 2 and 3?

```
merge xs [] = xs
merge [] xs = xs
merge (x:xs) (y:ys)
  | x < y     = x : merge xs (y:ys)
  | x > y     = y : merge (x:xs) ys
  | otherwise = x : merge xs ys
```

How would you merge two infinite lists, assuming that they are ascending? How would you remove duplicates from the list which results? As an example, how would you merge the lists of powers of 2 and 3?

```
merge xs [] = xs
merge [] xs = xs
merge (x:xs) (y:ys)
  | x < y     = x : merge xs (y:ys)
  | x > y     = y : merge (x:xs) ys
  | otherwise = x : merge xs ys

pow23 = merge pow2 pow3
  where pow2 = 1:map (* 2) pow2
        pow3 = 1:map (* 3) pow3
```