Part 3: Higher-order functions

Introduction to Haskell

Andres Löh

Well-Typed
The Haskell Consultants

Accumulators

Efficiency of `reverse`

Let's look at the standard solutions for `(++)` and `reverse` earlier:

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

reverse :: [a] -> [a]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

What is the efficiency of `reverse` ?

Well-Typed

Efficiency of `reverse`

Let's look at the standard solutions for `(++)` and `reverse` earlier:

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

reverse :: [a] -> [a]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

What is the efficiency of `reverse` ?

Answer: it has quadratic complexity, as `(++)` is linear in its left argument and `reverse` reduces to a linear chain of `(++)` invocations.

Well-Typed

A better `reverse`

Let's build the reversed list as we go in an additional, accumulating argument:

```
reverseAcc :: [a] -> [a] -> [a]
reverseAcc acc []       = acc
reverseAcc acc (x : xs) = reverseAcc (x : acc) xs

reverse :: [a] -> [a]
reverse = reverseAcc []
```

A better `reverse`

Let's build the reversed list as we go in an additional, accumulating argument:

```haskell
reverseAcc :: [a] -> [a] -> [a]
reverseAcc acc []        = acc
reverseAcc acc (x : xs) = reverseAcc (x : acc) xs

reverse :: [a] -> [a]
reverse = reverseAcc []
```

Note:

- we need the extra creative step to move from `reverse` to `reverseAcc`,
- but the function `reverseAcc` uses the standard design principle for lists again;
- we now traverse the list only once.

Accumulators in general

We have seen that `reverse` benefits significantly from introducing an accumulator.

Accumulators in general

We have seen that `reverse` benefits significantly from introducing an accumulator.

Can the same idea be applied elsewhere?

The `sum` function

Applying the standard design pattern:

```haskell
sum :: [Int] -> Int
sum []       = 0
sum (x : xs) = x + sum xs
```

The `sum` function

Applying the standard design pattern:

```
sum :: [Int] -> Int
sum []       = 0
sum (x : xs) = x + sum xs
```

Trying in GHCi with artificially limited stack size (`ghci +RTS -K1M`):

```
GHCi> sum [1 . . 100000]
*** Exception: stack overflow
```

Why is the function consuming so much stack space?

Equational reasoning

```
  sum [1, 2, 3]
= sum (1 : 2 : 3 : [])       -- removing syntactic sugar
= 1 + sum (2 : 3 : [])       -- by definition of sum
= 1 + (2 + sum (3 : []))     -- by definition of sum
= 1 + (2 + (3 + sum []))     -- by definition of sum
= 1 + (2 + (3 + 0))          -- by definition of sum
= 1 + (2 + 3)                -- by definition of (+)
= 1 + 5                      -- by definition of (+)
= 6                          -- by definition of (+)
```

Well-Typed

Equational reasoning

```
  sum [1, 2, 3]
= sum (1 : 2 : 3 : [])      -- removing syntactic sugar
= 1 + sum (2 : 3 : [])      -- by definition of sum
= 1 + (2 + sum (3 : []))    -- by definition of sum
= 1 + (2 + (3 + sum []))    -- by definition of sum
= 1 + (2 + (3 + 0))         -- by definition of sum
= 1 + (2 + 3)               -- by definition of (+)
= 1 + 5                     -- by definition of (+)
= 6                         -- by definition of (+)
```

We build an entire right-nested chain of additions until we reach the end of the list. Only then can we start reducing them.

Well-Typed

An accumulator for the sum?

Can an accumulator help? Let’s try:

```
sumAcc :: Int -> [Int] -> Int
sumAcc acc []       = acc
sumAcc acc (x : xs) = sumAcc (x + acc) xs

sum :: [Int] -> Int
sum = sumAcc 0
```

Well-Typed

An accumulator for the sum?

Can an accumulator help? Let’s try:

```
sumAcc :: Int -> [Int] -> Int
sumAcc acc []        = acc
sumAcc acc (x : xs) = sumAcc (x + acc) xs

sum :: [Int] -> Int
sum = sumAcc 0
```

Trying in stack-limited GHCi again:

```
GHCi> sum [1 . . 100000]
*** Exception: stack overflow
```

Well-Typed

More equational reasoning

```
  sum [1, 2, 3]
= sum (1 : 2 : 3 : [])             -- removing syntactic sugar
= sumAcc 0 (1 : 2 : 3 : [])        -- by definition of sum
= sumAcc (0 + 1) (2 : 3 : [])      -- by definition of sumAcc
= sumAcc ((0 + 1) + 2) (3 : [])    -- by definition of sumAcc
= sumAcc (((0 + 1) + 2) + 3) []    -- by definition of sumAcc
= ((0 + 1) + 2) + 3                -- by definition of sumAcc
= (1 + 2) + 3                      -- by definition of (+)
= 3 + 3                            -- by definition of (+)
= 6                                -- by definition of (+)
= 1 + (2 + (3 + sum []))           -- by definition of sum
= 1 + (2 + (3 + 0))                -- by definition of sum
= 1 + (2 + 3)                      -- by definition of (+)
= 1 + 5                            -- by definition of (+)
= 6  -- by definition of (+)
```

Well-Typed

The problem

We build the expression

```
sumAcc (0 + 1) (2 : 3 : [])
```

Perhaps surprisingly, `0 + 1` is not reduced, because it is not needed until much later.

The problem

We build the expression

```
sumAcc (0 + 1) (2 : 3 : [])
```

Perhaps surprisingly, `0 + 1` is not reduced, because it is not needed until much later.

We therefore build a (left-nested) chain of additions again until we reach the end of the list . . .

The problem

We build the expression

```
sumAcc (0 + 1) (2 : 3 : [])
```

Perhaps surprisingly, `0 + 1` is not reduced, because it is not needed until much later.

We therefore build a (left-nested) chain of additions again until we reach the end of the list . . .

Accumulators are problematic in a **lazy evaluation** scenario: we update them often, but do not really need them evaluated for a long time.

Perhaps we can provide a hint to the compiler that we want this evaluated sooner?

Making the accumulator strict with bang patterns

```haskell
sumAcc :: Int -> [Int] -> Int
sumAcc !acc []       = acc
sumAcc !acc (x : xs) = sumAcc (x + acc) xs

sum :: [Int] -> Int
sum = sumAcc 0
```

Note that we write `!acc` now. A bang pattern means that GHC will evaluate the passed value to its outermost constructor (but only to that!) even though the pattern is just a variable and would normally match without evaluation.

For an `Int`, evaluating to the outermost constructor means evaluating it completely.

Well-Typed

Bang patterns language extension

Bang patterns require the `BangPatterns` language extension, which can be enabled via a compiler pragma:

```
{-# LANGUAGE BangPatterns #-}
```

at the top of the file.

Problem fixed

In our stack-restricted GHCi:

```
GHCi> sum [1 . . 100000]
5000050000
```

Or even:

```
GHCi> sum [1 . . 10000000]
50000005000000
```

This version of `sum` actually runs in constant space.

The equational reasoning is similar to the previous accumulating version, only that the accumulator is now evaluated immediately on every update.

Make accumulators strict

Accumulators should nearly always be strict.

Make them strict by default.

There are few situations (such as `reverse`) where strictness of the accumulator is not required, but even there it is not harmful.

Always accumulators?

No!

The accumulating parameter pattern always has to traverse the entire list before producing a result! It cannot stop early or produce results incrementally. It can also not work for infinite lists:

```
GHCi> and (False : repeat True)
False

GHCi> take 10 (map (+ 1) [1 . . 10])
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Both of these functions should be defined using the standard design pattern for lists – without an accumulator – to enable the behaviour observed above.

Well-Typed

Capturing design patterns

Abstraction

One of the strengths of Haskell’s flexibility with functions is that they really allow to abstract from reoccuring patterns and thereby save code.

Abstraction

One of the strengths of Haskell’s flexibility with functions is that they really allow to abstract from reoccuring patterns and thereby save code.

The standard design principle for lists we’ve been using all the time works as follows:

```haskell
fun :: [someType] -> someResult
fun []       = ...  -- code
fun (x : xs) = ...  -- code that can use x and fun xs
```

Well-Typed

From an informal pattern to a function

```
fun :: [someType] -> someResult
fun []       = ...  -- code
fun (x : xs) = ...  -- code that can use x and fun xs
```

We have two **interesting** positions where we have to fill in situation-specific code. Let's abstract!

From an informal pattern to a function

```
fun :: [someType] -> someResult
fun []       = nil
fun (x : xs) = cons x (fun xs)
```

- We give names to the cases that correspond to the constructors.
- The case `cons` can use `x` and `fun xs`, so we turn it into a function.
- At the moment, this is not a valid function, because `nil` and `cons` come out of nowhere – but we can turn them into parameters of `fun`!

Well-Typed

From an informal pattern to a function

```
fun :: ... -> ... -> [someType] -> someResult
fun cons nil []       = nil
fun cons nil (x : xs) = cons x (fun cons nil xs)
```

We now have to look at the types of `cons` and `nil`:

- `nil` is used as a result, so `nil :: someResult`;
- `cons` takes a list element and a result to a result, so `cons :: someType -> someResult -> someResult`.

From an informal pattern to a function

```
fun :: (someType -> someResult -> someResult)
    -> someResult
    -> [someType] -> someResult
fun cons nil []       = nil
fun cons nil (x : xs) = cons x (fun cons nil xs)
```

We can give shorter names to `someType` and `someResult` …

From an informal pattern to a function

```
fun :: (a -> r -> r) -> r -> [a] -> r
fun cons nil []       = nil
fun cons nil (x : xs) = cons x (fun cons nil xs)
```

This function is called `foldr` …

From an informal pattern to a function

```
foldr :: (a -> r -> r) -> r -> [a] -> r
foldr cons nil []       = nil
foldr cons nil (x : xs) = cons x (foldr cons nil xs)
```

We could equivalently define it using `where` …

From an informal pattern to a function

```haskell
foldr :: (a -> r -> r) -> r -> [a] -> r
foldr cons nil = go
  where
    go []        = nil
    go (x : xs) = cons x (go xs)
```

The arguments `cons` and `nil` never change while traversing the list, so we can just refer to them in the local definition `go`, without explicitly passing them around.

Using `foldr`

```haskell
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

```haskell
foldr :: (a -> r -> r) -> r -> [a] -> r
foldr cons nil = go
  where
    go []       = nil
    go (x : xs) = cons x (go xs)
```

Using `foldr`

```haskell
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

```haskell
foldr :: (a -> r -> r) -> r -> [a] -> r
foldr cons nil = go
  where
    go []       = nil
    go (x : xs) = cons x (go xs)
```

```haskell
length = foldr (\ x r -> 1 + r) 0
```

or (using `const` and an operator section)

```haskell
length = foldr (const (1 +)) 0
```

Examples of using `foldr`

```haskell
(++) :: [a] -> [a] -> [a]
(++) xs ys = foldr (:) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\ x r -> if p x then x : r else r) []

map :: (a -> b) -> [a] -> [b]
map f = foldr (\ x r -> f x : r) []
```

Well-Typed

Examples of using `foldr`

```haskell
(++) :: [a] -> [a] -> [a]
(++) xs ys = foldr (:) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\ x r -> if p x then x : r else r) []

map :: (a -> b) -> [a] -> [b]
map f = foldr (\ x r -> f x : r) []
```

```haskell
and :: [Bool] -> Bool
and = foldr (&&) True

any :: (a -> Bool) -> [a] -> Bool
any p = foldr (\ x r -> p x || r) False
```

And many more.

The role of `foldr`

- When a list function is easy to express using `foldr`, then you should.
- Makes it immediately recognizable for the reader that it follows the standard design principle.
- Some functions can be expressed using `foldr`, but that does not necessarily make them any clearer. In such cases, aim for clarity.

Accumulating parameter pattern

```haskell
reverse :: [a] -> [a]
reverse = go []
  where
    go !acc []       = acc
    go !acc (x : xs) = go (x : acc) xs
```

```haskell
sum :: Num a => [a] -> a
sum = go 0
  where
    go !acc []       = acc
    go !acc (x : xs) = go (x + acc) xs
```

See something to abstract here?

Accumulating parameter pattern

```
reverse :: [a] -> [a]
reverse = go []
  where
    go !acc []       = acc
    go !acc (x : xs) = go (x : acc) xs
```

```
sum :: Num a => [a] -> a
sum = go 0
  where
    go !acc []       = acc
    go !acc (x : xs) = go (x + acc) xs
```

See something to abstract here?

Abstracting

```haskell
fun :: [a] -> r
fun = go ...
  where
    go !acc []       = acc
    go !acc (x : xs) = go (... acc ... x ...) xs
```

We apply the same tactics as before: let's abstract from the interesting positions and introduce names.

Abstracting

```
fun :: [a] -> r
fun = go e
  where
    go !acc []       = acc
    go !acc (x : xs) = go (op acc x) xs
```

Now we need to introduce `e` and `op` as parameters.

Abstracting

```haskell
fun :: ... -> ... -> [a] -> r
fun op e = go e
  where
    go !acc []       = acc
    go !acc (x : xs) = go (op acc x) xs
```

And we have to figure out the types (or let the compiler infer them).

Abstracting

```haskell
fun :: (r -> a -> r) -> r -> [a] -> r
fun op e = go e
  where
    go !acc []       = acc
    go !acc (x : xs) = go (op acc x) xs
```

This function is called `foldl'` (in the module `Data.List`).

Abstracting

```haskell
foldl' :: (r -> a -> r) -> r -> [a] -> r
foldl' op e = go e
  where
    go !acc []       = acc
    go !acc (x : xs) = go (op acc x) xs
```

This function is called `foldl'` (in the module `Data.List`).

foldr and foldl'

```
foldr  (⊕) e [x, y, z] = x ⊕ (y ⊕ (z ⊕ e))
foldl' (⊕) e [x, y, z] = ((e ⊕ x) ⊕ y) ⊕ z
```

Well-Typed

`foldr` and `foldl'`

```
foldr  (⊕) e [x, y, z] = x ⊕ (y ⊕ (z ⊕ e))
foldl' (⊕) e [x, y, z] = ((e ⊕ x) ⊕ y) ⊕ z
```

Performance advice

There's a function `foldl` with the same type as `foldl'`, which does not have the strict accumulator and should therefore almost never be used!

Well-Typed

Higher-order functions

Functions, functions, functions

A function parameterized by another function or returning a function is called a **higher-order function**.

Functions, functions, functions

A function parameterized by another function or returning a function is called a **higher-order function**.

Currying

Strictly speaking, every curried function in Haskell is a function returning another function:

```
elem   :: Eq a => a -> ([a] -> Bool)
elem 3 :: (Eq a, Num a) => [a] -> Bool
```

Well-Typed

Filtering and mapping

Two of the most useful list functions are higher-order, as they each take a function as an argument:

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

Filtering and mapping

Two of the most useful list functions are higher-order, as they each take a function as an argument:

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

The use of a function `a -> Bool` to express a predicate is generally common. And mapping a function over a data structure is an operation that isn't limited to lists.

Folds

The functions `foldr` and `foldl'` are among the most general higher-order functions on lists. Nearly any other list function can be expressed in terms of them.

Some rules of thumb:

- If a more specific function applies (such as `filter` or `map`), use that.
- If a function becomes more readable by using a fold, then use it. It has the advantage that it also signals to the programmer that a common pattern is being used and nothing special is going on.
- If a function becomes harder to read by using a fold, then do not force it into that pattern (except perhaps for training purposes).

Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: ...
(f . g) x = f (g x)
```

For once – rather than starting from a type – let's infer the type from the code.

Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: ... -> ... -> ... -> ...
(f . g) x = f (g x)
```

It’s apparently a curried function taking three arguments `f` , `g` and `x` .

Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (... -> ...) -> (... -> ...) -> ... -> ...
(f . g) x = f (g x)
```

Both `f` and `g` are applied to something, so they must be functions.

Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (... -> ...) -> (... -> ...) -> a -> ...
(f . g) x = f (g x)
```

No requirements seem to be made about the type of `x`, except that its passed to `g`, so let's assume a type variable here . . .

Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (... -> ...) -> (a -> ...) -> a -> ...
(f . g) x = f (g x)
```

... which then should be the source type of `g` as well.

Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (b -> ...) -> (a -> b) -> a -> ...
(f . g) x = f (g x)
```

The target type of `g` should match the source type of `f`.

Function composition

One of the most ubiquitous higher-order functions is function composition:

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

The target type of `f` is also the type of the overall result.

Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Putting extra parentheses in the type may make it more obvious that we are indeed composing two matching functions.

Composing functions

We can often build functions from existing functions simply by composing them.

Composing functions

We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

```
example :: [Int]
example =
                                                      [1 . .]
```

We start by generating all numbers (lazy evaluation in action).

Composing functions

We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

```
example :: [Int]
example =
                                map (\ x -> x * x)  [1 ..]
```

We use `map` to compute the square numbers. Note that `map` and `filter` are often used with **anonymous** functions.

Composing functions

We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

```
example :: [Int]
example =
  (            filter odd . map (\ x -> x * x)) [1 ..]
```

We use function composition composition (and partial application) to subsequently filter the odd square numbers.

Composing functions

We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

```
example :: [Int]
example =
  (take 100 . filter odd . map (\ x -> x * x)) [1 ..]
```

Finally, we use composition again to take the first 100 elements of this list.

Composition as a design pattern

- Function composition gives you a way to split one programming problem into several, possibly smaller, programming problems.
- In general, higher-order functions are part of your toolbox for attacking programming problems. Recognizing something as a `map` or `filter` is also useful.
- Of course, you should never forget the standard design principle of following the datatype structure as a good way of defining most functions, if applying a higher-order function fails.

Well-Typed

Lessons

- Function composition is a bit like the functional semicolon. It allows us to decompose larger tasks into smaller ones.
- Lazy evaluation allows us to separate the generation of possible results from selecting interesting results. This allows more modular programs in many situations.
- Partial application and anonymous functions help to keep such composition chains concise.

Well-Typed

Operating on functions

Flipping a function

If you want to change the order of arguments of a two-argument curried function, you can use

```haskell
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

Flipping a function

If you want to change the order of arguments of a two-argument curried function, you can use

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Note once again that the function arrow associates to the right, so `flip` can really be seen as a function with three arguments:

```
f :: a -> b -> c
x :: b
y :: a
```

Flipping a function

If you want to change the order of arguments of a two-argument curried function, you can use

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Example use:

```
foreach = flip map

example = foreach [1, 2, 3] (\ x -> x * x)
```

Currying and uncurrying

Sometimes, you end up with a pair and want to apply a function to it that typically (in Haskell) is in curried form. Fortunately, we can convert between curried and uncurried form easily:

```haskell
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y

curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
```

Currying and uncurrying

Sometimes, you end up with a pair and want to apply a function to it that typically (in Haskell) is in curried form. Fortunately, we can convert between curried and uncurried form easily:

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

curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
```

Example:

```
map (uncurry (*)) (zip [1 . . 3] [4 . . 6])
```