When You Should Use Lists in Haskell (Mostly, You Should Not)

Johannes Waldmann, HTWK Leipzig, Germany

WFLP 2018

What's Wrong With This Program?

(e.g., `http://learnyouahaskell.com/recursion#quick-sort`)

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
  let smallerSorted = quicksort [a | a <- xs, a <= x]
      biggerSorted = quicksort [a | a <- xs, a > x]
  in  smallerSorted ++ [x] ++ biggerSorted
```

- singly linked lists!
- append (++) copies the left argument
- never use this in production
- should you use it in teaching? it depends.

What Is Wrong With These Functions?

(from `base:Data.List`)

```
(\\) :: Eq a => [a] -> [a] -> [a]
union :: Eq a => [a] -> [a] -> [a]
intersect :: Eq a => [a] -> [a] -> [a]
```

- These specifications cannot be implemented efficiently. (they all need quadratic time)
- Before you use them, try *very* hard to come up with an instance of one of
  `Ord, Enum, Hashable, Serialize`
  then replace `Data.List` with `Data.{,Int,Hash}Set`

What Lists Are, And What They Are Not

```
data List a = Nil | Cons a (List a)
```

- these operations are efficient:
  add, read, remove the *first* element
  (call the constructors, match on the constructors)
- all others (length, indexed access) are *terribly inefficient*

- Lists are potentially infinite streams (a.k.a. Iterators):
  access each element once, in order, on demand.
- Lists are very bad collections:
  access elements more than once, out of order.

- Exercise: why don't we store the length in each cell?

  ```
  data List a = Nil | Cons Int a (List a)
  ```

Take-Home Messages of This Talk

- If your program accesses a list by index (with `(!!)`), then your program is wrong.
- If your program uses the `length` function, then your program is wrong.
- If your program sorts a list, then your program is wrong.
- If you wrote this `sort` function yourself, then it is doubly wrong.

- Use lists for streams, not for random-access collections
- The ideal use of a list is such that will be removed by the compiler.
- The enlightened programmer writes list-free code with `Foldable`.

Where Do These Haskell Lists Come From?

- lists seem connected to functional programming from the beginning of time (= LISP, 1959)
- but the only reason is that LISP does not have algebraic data types (ADT), and uses nested lists for trees (well, for everything)
- textbook authors never noticed that ADTs had been introduced (ML, 1973) — Haskell (1990) was designed to accomodate such teaching . . . well,
- the defining feature of Haskell is *lazy evaluation*, and Streams are a perfect use case (and showcase)
- thus we have the confusion between
  - lists as container structures (obsolete, inefficient)
  - and lists as streams (important, useful)

Do We Have Good Containers? Plenty!

- sequences:
  - constant-time access, linear concatenation:
    - `Data.Vector` — arrays (with slicing)
    - `Data.ByteString`, `Data.Text`
      (next: Why You Should Never Ever Use `String`)
  - logarithmic access, logarithmic concatenation:
    `Data.Sequence` — size-balanced trees
- sets, maps:
  -
    - logarithmic insert, member/lookup
      `Data.{Set,Map}` — size-balanced trees
    - linear in key size: `Data.Int{Set,Map}` — tries
  - with efficient *bulk operations*: union, intersection, ...
    for *point-free* programming (no explicit iteration)

Stream Processing in Constant Space

```
sum $ map (^ 2) $ [ 1 :: Int .. 10^8 ]
```

- separation of concerns
  (consumer, transformer, producer)
- interleaved computation (on-demand evaluation)
- runs in constant space (intermediate data will be garbage-collected immediately)
- this is a *good* use of lists
  (they represent streams, we access each element once)
- confirm by experiment
  (`./space +RTS -M80k -A10k -S`)
  for detail: `https://mail.haskell.org/pipermail/haskell-cafe/2018-September/129913.html`

Stream Processing in No Space

```
sum $ map (^ 2) $ [ 1 :: Int .. 10^8 ]
```

- compile with `ghc -O2`: get tight non-allocating inner loop

```
$wgo_s5we (w_s5w8 :: GHC.Prim.Int#) (ww1_s5wc :: G
 = case GHC.Prim.==# w_s5w8 ww_s5w5 of {
  __DEFAULT ->
  jump $wgo_s5we
   (GHC.Prim.+# w_s5w8 1#)
   (GHC.Prim.+# ww1_s5wc (GHC.Prim.*# w_s5w8 w_s5w
```

- because of code transformations (rewriting the AST)

```
ghc  .. --dump-rule-firings
  Rule fired: map (GHC.Base)
  Rule fired: fold/build (GHC.Base)
```

No-Stream Processing (How To Avoid Lists)

- example: the sum of the elements of a set
  `m :: Data.Set.Set Int`
- first (“obvious”) solution: `sum ( S.toList m )`
  assuming `sum :: Num a => [a] -> a`
- correct solution: `sum m` , because

```
sum :: (Num a, Foldable t) => t a -> a
instance Foldable Set where ...
```

- avoid production of intermediate list in the source already (don’t defer to compiler or garbage collector)

No-Stream Processing: How Does It Work

- ```
  sum :: (Num a, Foldable t) => t a -> a
  sum = getSum . foldMap Sum
  class Foldable t where
   foldMap :: Monoid m => (a -> m) -> t a -> m
  ```
- ```
  class Monoid m where
   mempty :: m ; mappend :: m -> m -> m
  ```
- ```
  newtype Sum a = Sum { getSum :: a }
  instance Num a => Monoid (Sum a) where
   mempty = Sum 0
   mappend (Sum x) (Sum y) = Sum (x + y)
  ```
- ```
  data Set a = Bin Size a (Set a) (Set a) | Tip
  ```
- ```
  instance Foldable Set where
   foldMap f t = go t where
    go Tip = mempty ; go (Bin 1 k _ _) = f k
    go (Bin _ k l r) = go l `mappend` (f k `mapp
  ```
  ```

Take-Home Messages of This Talk

- If your program accesses a list by index (with `(!!)`), then your program is wrong.
- If your program uses the `length` function, then your program is wrong.
- If your program sorts a list, then your program is wrong.
- If you wrote this `sort` function yourself, then it is doubly wrong.

- Use lists for streams, not for random-access collections
- The ideal use of a list is such that will be removed by the compiler.
- The enlightened programmer writes list-free code with `Foldable`.