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

Johannes Waldmann

F-IMN, HTWK Leipzig, Germany
`https://www.imn.htwk-leipzig.de/~waldmann/`

Abstract. We comment on the over-use of lists in functional programming. With this respect, we review history of Haskell and some of its libraries, and hint at current developments.

1 Introduction

It seems that the designers of the programming language Haskell [Mar10] were madly in love with singly linked lists. They even granted notational privileges: e.g., the data constructor for a non-empty list ("cons") uses just one letter (:). Looking at the Haskell standard, it also seems that singly linked lists are more important than the concept of static typing. When declaring the type of an identifier, we must use two letters ::, because the natural (that is, mathematical) notation : is taken.

One goal of language design is to make typical expected usage easy to write. Once the design is cast in stone, this becomes self-fulfilling: unsuspecting users of the language will write programs in what they think is idiomatic style, as suggested by built-in syntactical convenience and sugar. Haskell lists have plenty of both: convenience by short notation, and sugar in the form of list comprehensions. So it seems that it is typical Haskell to not declare types for identifiers, and use lists all over the place.

I will explain on the following pages that the purported connection between functional programming and lists is largely historical, and is detrimental to teaching and programming. Lists do indeed serve a purpose, but a rather limited one:

- *iteration*: you use each element of a collection *in order* (FIFO or LIFO), on demand, and at most once.

  If you use lists for anything else, in particular

- *storage*: you use elements *out of order*, e.g., by known index, or you access ("search") elements by a property,

then you're doing it wrong, *wrong, wrong.* I will explain why, and what you should do instead.

As an appetizer, here are some bumper-sticker slogans extracted from the sections that follow.

- If your program accesses a list by index (with the `!!` operator), then your program is wrong. (Section 4)
- 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.
- The ideal use of a list is such that will be removed by the compiler. (Section 6)
- The enlightened programmer writes list-free code with `Foldable`. (Section 7)

I will not enter arguments about syntax. (The notation of *cons* as ":", and type declarations with "::", came to Haskell from Miranda [Tur87], and has recently been reversed in Agda [Nor07].) This paper is about the semantics (and consequently, performance) of singly linked lists, as we have them in Haskell.

Note that the name `List` has a different meaning in, e.g., the Java standard library [Ora18], where it denotes the abstract data type of sequences (collections whose elements can be accessed via a numerical index). They have linked lists as one possible realization, but there are others, for which `length` and `!!` (indexing) are not atrocious. Of course, there are Haskell libraries for efficient sequences [Les10].

2 Where Do All These Lists Come From?

Certainly Haskell builds on the legacy of LISP [McC60,Gra01], the very first functional programming language - that has lists in its name already. Indeed, singly linked lists were the only way to structure nested data in LISP.

This is a huge advantage (it's soo uniform), but also a huge dis-advantage (you cannot distinguish data - they all look like nested lists). For discriminating between different shapes of nested data, people then invented algebraic data types, pattern matching, and static typing. The flip side of static typing is that it seems to hinder uniform data processing. So, language designers invented polymorphism [Hin69] and generic programming, to get back flexibility and uniformity, while keeping static safety.

This was all well known at the time of Haskell's design, so why are lists that prominent in the language? The one major design goal for Haskell was: to define a functional programming language with static typing and *lazy evaluation*. For functional programming with static typing and strict evaluation, the language ML was already well established [Tur12]. Haskell has lazy evaluation for function calls, including constructor applications, and a prime showcase for this is that we can handle apparently infinite data, like streams:

```
data Stream a = Cons a (Stream a)
```

Such streams do never end, but we can still do reasonable things with them, by evaluating finite prefixes on demand, in other words, lazily.

Now a standard Haskell list is just such a stream that *can* end, because there is an extra constructor for the empty stream

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

and the problem with Haskell lists is that people tend to confuse what the type `[ a ]` stands for:

– is it: a lazy (possibly infinite) stream (yes)
– or is it: a random-access sequence (no); or a set (hell no)

If you have an object-oriented background, then this is the confusion

– between an iterator (that allows to traverse data)
– and its underlying collection (that holds the actual data).

A Haskell list is a very good iterator - we even might say, it is the *essence* of iteration, but it makes for a very bad implementation of a collection.

3 And Where Are These Lists Today?

The (historic) first impression of "functional programming is inherently list processing" seems hard to erase. Indeed it is perpetuated by some Haskell teaching texts, some of which are popular. If a text(book) aimed at beginners presents `Int`, `String` and `[Int]` on pages 1, 2, 3; but keywords `data` and `case` (and assorted concepts of algebraic data type and pattern matching) appear only after the first two hundred pages, well, then something's wrong.

For example, the shiny new (in 2015) `http://haskell.org/` gets this exactly wrong. At the very top of the page, it presents example code that is supposed to highlight typical features of the language. We see a version of Eratosthenes' sieve that generates an infinite list of primes. Declarative it certainly is. But see - Lists, numbers. Must be typical Haskell.

Also, that example code features list comprehensions - that's more syntactic sugar to misguide beginners. And it claims "statically typed code" in the previous line, but - there is no type annotation in the code. Yes, I know, the types are still there, because they are inferred by the compiler, but what does a beginner think? The icing on all this is that the one thing that this example code really exemplifies, namely: infinite streams with lazy evaluation, is not mentioned anywhere in the vicinity.

4   What Lists Are (Not) Good At

The answer is simple: *never use lists for data storage with out-of-order access.*

Accessing an element of a singly linked list at some arbitrary position i takes time proportional to i, and you do not want that.

The only operations that are cheap with a singly linked list are

- accessing its first element element (the head of list),
- and prepending a fresh first element.

So can we access any of the other elements in any reasonable way? Yes, we can access the second element, but only after we drop the first one. The second element becomes the "new head", and the "old head" is then gone for good. We are singly linked, we cannot go back. If we want to access any element in a list, we have to access, and then throw away, all the elements to the left of it.

In other words, lists in Haskell realize the *iterator* design pattern [GHJV95] (Iterator in Java, IEnumerator in C#). So - they are a means of expressing control flow. As Cale Gibbard (2006) puts it [Gib06], "lists largely take the place of what one would use loops for in imperative languages."

Or, we can think of a list as a stack, and going to the next element is its "pop" operation.

Can we really never go back? Well, if `xs` is an iterator for some collection, then `x : xs` is an iterator that first will provide `x`, and after that, all elements of `xs`. So, thinking of a list as a stack, the list constructor (`:`) realizes the "push" operation.

On the other hand, the "merge" function (of "mergesort" fame, it merges two monotonically increasing sequences) is a good use case for Haskell lists, since both inputs are traversed in order (individually - globally, the two traversals are interleaved). Its source code is

```haskell
merge :: Ord a => [a] -> [a] -> [a]
merge as@(a:as') bs@(b:bs')
  | a `compare` b == GT = b:merge as  bs'
  | otherwise           = a:merge as' bs
merge [] bs             = bs
merge as []             = as
```

We never copy lists, and we only ever pattern match on their beginning.

Indeed you can easily (in Java or C#) write a `merge` function that takes two iterators for monotone sequences, and produces one. Try it, it's a fun exercise. Use it to learn about `yield return` in C#.

Back to Haskell: how should we implement merge-sort then, given `merge`? The truthful answer is: we should not, see below. But for the moment, let's try anyway, as an exercise. A straightforward approach is to split long input lists at (or near) the middle:

```
msort :: Ord a => [a] -> [a]
msort [] = [] ; msort [x] = [x]
msort xs = let (lo,hi) = splitAt (div (length xs) 2) xs
           in  merge (msort lo) (msort hi)
```

but this is already wrong. Here, xs is not used in a linear way, but twice: first in length xs (to compute to position of the split), and secondly, in splitAt ... xs, to actually do the split. This is inefficient, since the list is traversed twice, which costs time – and space, since the spine of the list will be held in memory fully.

Indeed, the Haskell standard library contains an implementation of mergesort that does something different. The basic idea is

```
msort :: Ord a => [a] -> [a]
msort xs = mergeAll (map (\x -> [x]) xs)
  where mergeAll [x] = x
        mergeAll xs  = mergeAll (mergePairs xs)
        mergePairs (a:b:xs) = merge a b: mergePairs xs
        mergePairs xs       = xs
```

But the actual implementation [Uni01] has an extra provision for handling monotonic segments of the input more efficiently. Make sure to read the references mentioned in the comments atop the source code of Data.OldList.sort, including [Aug97].

But - if you sort a list in your application *at all*, then you're probably doing something wrong already. There's a good chance that your data should have *never* been stored in a list from the start.

How *do* we store data, then? This is *the* topic of each and every "Data Structures and Algorithms" course, and there *are* corresponding Haskell libraries. For an overview, see [Wal17a].

5   Is There No Use For Lists As Containers?

There are two exemptions from the rules stated above.

If you are positively certain that your collections have a *bounded* size throughout the lifetime of your application, then you can choose to implement it in any way you want (and even use lists), because all operations will run in constant time anyway. But beware - "the constant depends on the bound". So if the bound is three, then it's probably fine.

Then, *education*. With lists, you can exemplify pattern matching and recursion (with streams, you exemplify co-recursion). Certainly each student should see and do recursion early on in their first term.

But the teacher must make very clear that this is in the same ballpark with exercises of computing the Fibonacci sequence, or implementing multiplication and exponentiation for Peano numbers: the actual output of the computation is largely irrelevant. What is relevant are the techniques they teach (algebraic

data types, pattern matching, recursion, induction) because *these* have broad applications.

So, indeed, when I teach [Wal17b] algebraic data types, I do also use singly linked lists, among other examples (Bool, Maybe, Either, binary trees), but I always write (and have students write)

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

I do avoid Haskell's built-in lists as long as possible, that is, until students understand their use (representing iterators) in the application programmer interfaces of recommended libraries like `Data.Set` and `Data.Map`.

6   Lists That Are Not Really There

We learned above that the one legitimate use case for lists is iteration. This is an important one. It's so important that the Glasgow Haskell compiler (GHC) applies clever program transformations [GLP93] in order to generate efficient machine code for programs that combine iterators (that is, lists) in typical ways.

The idea is that the most efficient code is always the one that is not there at all. Indeed, several program transformations in the compiler remove intermediate iterators (lists). Consider the expression

```
sum $ map (\x -> x*x) [1 .. 1000]
```

which contains two iterations: the list `[1 .. 1000]` (the *producer*), and another list (of squares, built by `map`, which is a *transformer*). Finally, we have `sum` as a *consumer* (its output is not a list, but a number).

We note that a transformer is just a consumer (of the input list) coupled to a producer (of the output list). Now the important observation is that we can remove the intermediate list between a producer and the following consumer (where each of them may actually be part of a transformer).

This removal is called *fusion*. With fusion, lists are used prominently in the source code, but the compiler will transform this into machine code that does not mention lists *at all*! In particular, the above program will run in constant space, as it allocates *nothing* on the heap, and it needs *no* garbage collector.

For historic reference and detail, see [Wad90].

7   Lists That Are Really Not There

Now I will present a more recent development (in the style of writing libraries for Haskell) that aims to remove lists from the actual *source* code. Recall that deforestation removes lists from *compiled* code.

We have, e.g., `and [True,False,True]=False`, and you would think that `and` has type `[Bool] -> Bool`, as it maps a list of Booleans to a Boolean. But no, the type is `and :: Foldable t => t Bool -> Bool`, and indeed we can apply `and` to any object of a type `t a` where `t` implements `Foldable`.

The constraint `Foldable t` for a type constructor `t` of kind `* -> *` means that there is an associated function `toList :: t a -> [a]`. So, `c` of type `t a` with `Foldable t` can be thought of as some kind of collection, and we can get the stream of its elements by `toList c`. This is similar to `c.iterator()` for `Iterable<A> c` in `java.util`.

For example, `Data.Set` [Lei02] does this, so we can make a set `s = S.fromList [False, True]` and then write `and s`. It will produce the same result as the expression `and (toList s)`.

Why all this? Actually, this is not about `toList`. Quite the contrary - this `Foldable` class exists to make it easier to *avoid* lists. Typically, we would not implement or call the `toList` function, but use `foldMap` or `foldr` instead — or functions (like `and`) that are defined via `foldMap`.

We give another example. Ancient libraries used to have a function with name and type `sum :: Num a => [a] -> a` but now its actual type is different: `sum :: (Num a, Foldable t) => t a -> a`, and so we can take the sum of all elements in a set `s :: Set Int` simply via `sum s`, instead of writing `sum (toList s)`. See - the lists are gone from the source code.

Let us look at the type of `foldMap`, and the class `Monoid` that it uses.

```
class Foldable t where
    foldMap :: Monoid m => (a -> m) -> t a -> m

class Monoid m where
    mempty :: m ; mappend :: m -> m -> m
```

This is fancy syntax for the following (slightly re-arranged)

```
foldMap :: m -> (a -> m) -> (m -> m -> m) -> t a -> m
```

where the first and third argument are the dictionary for `instance Monoid m`.

So when you call `foldMap` over some collection (of type `t a`), then you have to supply (via the `Monoid` instance) the result value (`mempty :: m`) for the empty collection, the mapping (of type `a -> m`) to be applied to individual elements, and a binary operation `mappend :: m -> m -> m` to be used to combine results for sub-collections.

Let us see how this is used to `sum` over `Data.Set`, the type mentioned above, internally defined as size-balanced search trees in `Data.Set.Internal` [Lei02] and repeated here for convenience:

```
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 `mappend` go r)
```

This means that `foldMap f s`, for `s :: Set a` of size larger than one, does the following: each branch node `n` is replaced by two calls of `mappend` that combine results of recursive calls in subtrees `l` and `r` of `n`, and `f k` for the key `k` of `n`, and we make sure to traverse the left subtree, then the key at the root, then the right subtree.

The definition [Pat05] of `sum` is

```haskell
class Foldable t where
  sum :: Num a => t a -> a
  sum = getSum #. foldMap Sum
```

with the following auxiliary types [Gil01]

```haskell
newtype Sum a = Sum { getSum :: a }
instance Num a => Monoid (Sum a) where
    mempty = Sum 0 ; mappend (Sum x) (Sum y) = Sum (x + y)
```

So, when we take the `sum` of some `Set a`, we do a computation that adds up all elements of the set, but without an explicit iterator - there is no list in sight.

8 Discussion and Related Opinions

Specifically for Section 7 : When the types for `sum, and`, and many more functions were generalized from lists to `Foldable` a while back, some `Foldable` instances were introduced that result in funny-looking semantics like `maximum (2,1) == 1`, which is only "natural" if you think of the curried type constructor application `(,) a b`, remove the last argument, and demand that `(,) a` "obviously" must be `Functor`, `Foldable`, and so on. This might save some keystrokes, but is has been criticized [Thi16] because it "moves Haskell into the Matlab league of languages where any code is accepted, with unpredictable results."

In general : This text is based on my experience in teaching (advanced) programming language concepts, for two decades now, and using Haskell for real-world code, e.g., an E-Learning/Assessment system [RW02,Wal02] in use since 2001, and an administration and presentation layer for Termination Competitions [WMvdK14].

Other researchers and practitioners have expressed similar opinion on using (or, avoiding) lists. Breitner teaches a course [Bre16] that is similar to mine, and Thielemann recommends [Thi07] to "choose types properly", in particular, to not confuse lists with arrays.

Previous discussion of this text : see `https://mail.haskell.org/pipermail/haskell-cafe/2017-March/thread.html#126457` and `https://www.reddit.com/r/haskell/comments/5yiusn/when_you_should_use_lists_in_haskell_mostly_you/`.

References

Aug97.   Lennart Augustsson. Re: heap sort or the wonder of abstraction. messsage
              on mailing list `https://www.mail-archive.com/haskell@haskell.org/`
              `msg01822.html`, 1997.

Bre16.   Joachim Breitner. CIS 194: Introduction to Haskell (Fall 2016). `http:`
              `//cis.upenn.edu/~cis194/fall16/`, 2016.

GHJV95.   Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *De-*
              sign Patterns: Elements of Reusable Object-Oriented Software. Addison-
              Wesley, 1995.

Gib06.   Cale Gibbard.   Export lists in modules.   message on mail-
              ing list `https://mail.haskell.org/pipermail/haskell-prime/2006-`
              `February/000795.html`, 2006.

Gil01.   Andy Gill. `Data.Monoid`: A class for monoids with various general-
              purpose instances. `https://hackage.haskell.org/package/base-4.11.`
              `1.0/docs/Data-Monoid.html`, 2001.

GLP93.   Andrew John Gill, John Launchbury, and Simon L. Peyton Jones. A Short
              Cut to Deforestation. In John Williams, editor, *Proc. Conf. on Func-*
              tional programming languages and computer architecture, FPCA 1993,
              pages 223–232. ACM, 1993.

Gra01.   Paul Graham.   The Roots of LISP.   `http://www.paulgraham.com/`
              `rootsoflisp.html`, 2001.

Hin69.   J. Roger Hindley. The Principal Type-Scheme of an Object in Combina-
              tory Logic. *Transactions of the American Mathematical Society*, 146:29–
              60, 1969.

Lei02.   Daan Leijen.   `Data.Set`: Finite Sets, part of the containers li-
              brary. `https://hackage.haskell.org/package/containers-0.6.0.1/`
              `docs/Data-Set.html`, 2002.

Les10.   Roman Leshchinskiy. `Data.Vector`: Efficient Arrays. `https://hackage.`
              `haskell.org/package/vector-0.12.0.1/docs/Data-Vector.html`,
              2010.

Mar10.   Simon Marlow. Haskell 2010 Language Report. `https://www.haskell.`
              `org/onlinereport/haskell2010/`, 2010.

McC60.   John McCarthy. Recursive Functions of Symbolic Expressions and Their
              Computation by Machine, Part I. *Commun. ACM*, 3(4):184–195, 1960.

Nor07.   Ulf Norell. *Towards a practical programming language based on dependent*
              type theory. PhD thesis, Department of Computer Science and Engineer-
              ing, Chalmers University of Technology, SE-412 96 Göteborg, Sweden,
              September 2007.

Ora18.   Oracle.   Interface `List<E>`, Java Platform Version 10 API Specifica-
              tion. `https://docs.oracle.com/javase/10/docs/api/java/util/List.`
              `html`, 2018.

Pat05.   Ross Paterson. `Data.Foldable`: class of data structures that can be folded
              to a summary value. `https://hackage.haskell.org/package/base-4.`
              `11.1.0/docs/Data-Foldable.html`, 2005.

RW02.   Mirko Rahn and Johannes Waldmann. The Leipzig autotool System for
              Grading Student Homework. In *Workshop Functional and Declarative*
              Programming in Education (FDPE 2002), 2002.

Thi07.   Henning Thielemann. Choose types properly—lists are not good for every-
              thing. `https://wiki.haskell.org/Haskell_programming_tips#Lists_`
              `are_not_good_for_everything`, 2007.

Thi16. Henning Thielemann. Haskell Foldable Wats. message on mailing list `https://mail.haskell.org/pipermail/libraries/2016-February/026678.html`, 2016.

Tur87. David Turner. An overview of miranda. *Bulletin of the EATCS*, 33:103–114, 1987.

Tur12. D. A. Turner. Some History of Functional Programming Languages - (Invited Talk). In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming (TFP)*, volume 7829 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2012.

Uni01. University of Glasgow. Data.List: Operations on Lists. `https://hackage.haskell.org/package/base-4.11.1.0/docs/Data-List.html#g:21`, 2001.

Wad90. Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

Wal02. Johannes Waldmann. Leipzig autotool. `https://gitlab.imn.htwk-leipzig.de/autotool/all0#leipzig-autotool`, 2002.

Wal17a. Johannes Waldmann. How Do We Store Data, Then? `https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/data/`, 2017.

Wal17b. Johannes Waldmann. How I Teach Functional Programming. In *Workshop on Functional Logic Programming (WFLP 2017), Würzburg*, 2017.

WMvdK14. Johannes Waldmann, Rene Muhl, and Stefan von der Krone. star-exec-presenter: A Web Application for Displaying Results of the Termination Competition. `https://github.com/jwaldmann/star-exec-presenter`, 2014.