hack.hands( )
A PLURALSIGHT COMPANY

BECOME AN EXPERT

SIGN IN

hack.hands() is an online service for live programing help available 24/7.

GET STARTED

Posted on January 22nd 2015

Modular Code and Lazy Evaluation in Haskell

This tutorial is part of a larger series, The Incomplete Guide to Lazy Evaluation (in Haskell).

Modular Code and Lazy Evaluation in Haskell

minimum = head . sort

Lazy evaluation is the most widely used method for executing Haskell program code on a computer. It can help with making our code simpler and more modular, but it can also make it harder to understand how a program is evaluated in detail. In the last tutorial, we have looked at how exactly a Haskell program is executed with lazy evaluation, and what this means for time and memory usage. Today, we want to reap the benefits: How exactly can lazy evaluation help improve our code?

We will look at several examples and idioms that are made possible by lazy evaluation, for example an equality test for strings that returns early, or the celebrated *infinite lists*. The main tenet is that lazy evaluation allows us to reuse code in a larger context while keeping a good level of efficiency. At the end of the tutorial, we will take this to an extreme and find the minimum element of a list by first *sorting* the list -- and rely on lazy evaluation to do that in

linear time.

Of course, lazy evaluation is a trade-off: It is always possible to rewrite any code that uses lazy evaluation to code that does not use it. Doing this can make our code faster and make memory usage more predictable. However, the result will also be more specialized and less modular. In this tutorial, we want to look at the lazy side of the trade-off and explore what can be gained in terms of modularity.

(This tutorial assumes that you know how program execution is done with lazy evaluation, for instance because you've read the previous tutorial).

List equality that returns early

To gain a basic understanding of how lazy evaluation can make our code more modular, let us consider the function `(&&)` which implements a logical AND. Its definition is

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```

We have already seen that thanks to lazy evaluation, it returns a result early when the first argument is `False`. For instance, the expression

```
False && ((4*2 + 34) == 42)
```

immediately reduces to `False` and the second argument is not evaluated at all. This behavior is known as short-circuit evaluation and also available in many other programming languages.

However, let us now consider a function `and` that implements a logical AND which operates not just on two values, but on a whole list of values. (This function can be found in the Haskell Prelude: and.) A recursive definition would be

```
and :: [Bool] -> Bool
and []     = True
and (b:bs) = b && and bs
```

(Exercise: Write this in terms of foldr!) If we apply it to a list whose first element is `False`, then lazy evaluation will proceed as follows:

```
and [False, True, True, True]
```

```
=>  False && and [True, True, True]
=>  False
```

In other words, after finding that the first element is `False`, the function already returns `False` and ignores the rest of the list. In general, the function will inspect the first few list elements and return `False` as soon as it encounters one element that is `False`. This can save a substantial amount of computation time.

We want to go even further. Consider the following function that compares two lists:

```
prefix :: Eq a => [a] -> [a] -> Bool
prefix xs ys = and (zipWith (==) xs ys)
```

It returns `True` whenever one of the argument lists is a prefix of the other one. (For the sake of clarity, I do not wish to use actual list equality `(==)` as an example because it cannot be implemented with the `zipWith` function.) Remember that the library function zipWith pairs two lists with a binary function

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

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys zipWith f _ _ = []
```

Here is what lazy evaluation does when we compare the string `"Haskell"` to the string `"eager"`:

```
prefix "Haskell" "eager"
=>  and (zipWith (==) "Haskell "eager")
=>  and ('H' == 'e' : zipWith (==) "askell" "ager")
=>  and ( False     : zipWith (==) "askell" "ager")
=>  False && and (zipWith (==) "askell" "ager")
=>  False
```

As you can see, the comparison returns `False` as soon as it finds that the first letters differ. Very efficient!

Now, the point here is that the library function `zipWith` is completely generic, its implementation knows nothing about the functions `and` and `(&&)`. The three functions are independent and modular. Yet, thanks to lazy evaluation, they work together and yield an efficient prefix test when combined.

What is it that makes them work together so well? The key is that lazy evaluation tries to evaluate only as much as necessary, nothing more. It is often also called "demand-driven evaluation". The function `zipWith` will only calculate as much of the result list as the enclosing application of the function `and` demands. Likewise, the function `and` supplies the remaining list to the function `(&&)` in a way so that the latter can decide how much of the remaining list it is going to demand. By returning and requesting results only piece by piece, functions can be reused in many more different contexts than if they were to return the whole result at once.

Of course, it is also possible to write a prefix test that returns early in languages that use eager evaluation, like LISP or F#. But we would have to write special code for it. One possible implementation would be

```
prefix (x:xs) (y:ys) = if x == y then eq xs ys else False
prefix _      _      = True
```

We can no longer reuse the library functions like `(&&)` and `zipWith`. Rather, we have to reimplement their combined functionality anew.

Infinite lists

Continuing our exploration of lazy evaluation and code modularity, we will now look at one of the most striking data structures made possible by lazy evaluation, one that every Haskell programmer should know about: *infinite lists*. How is it even possible to hold an infinite list in memory? Well, in a way, the trick is to never evaluate all of it.

We begin with a very simple example: The infinite list consisting of all natural number starting at `0` can be written as

```
[0..]
```

If you type this into your favorite Haskell interpreter, it will start printing `[0,1,2,…` and never stop unless you interrupt it; after all, the list is infinite. It is more reasonable to print only a finite part of it. For instance, the expression

```
head [0..]
```

will return the head of this infinite list, which is `0`. How does this work? The expression `[0..]` is syntactic sugar for the function application `enumFrom 0`, which can be defined as

```
enumFrom :: Integer -> [Integer]
enumFrom n = n : enumFrom (n+1)
```

(I have specialized the type signature for clarity.) As you can see, this function puts the starting number in the head of the list and puts an application of itself in the tail of the list. Since this never stops, the result list must be infinite. Lazy evaluation will reduce the expression `head [0..]` as follows:

```
head (enumFrom 0)
=>  head (0 : enumFrom (0+1))
=>  0
```

The point is that `head` returns the head of the list right away and the tail is discarded without evaluating it.

Infinite lists can be very useful. Here is a very common idiom for decorating the elements of a list `xs` with their position in the list:

```
zip [0..] xs
```

The function zip is from the Haskell Prelude. Let us consider an example to see how lazy evaluation proceeds:

```
zip [0..] "Oh"
=   zip (enumFrom 0) "Oh"
=>  zip (0 : enumFrom (0+1)) "Oh"
=>  (0,'O') : zip (enumFrom (0+1)) "h"
=>  (0,'O') : zip ((0+1) : enumFrom ((0+1)+1)) "h"
=>  (0,'O') : ((0+1), 'h') : zip (enumFrom ((0+1)+1)) []
=>  (0,'O') : ((0+1), 'h') : []
```

The letter `'O'` is paired with position `0` and the letter `'h'` is paired with position `(0+1) = 1`. In the last step, the application of `zip` was reduced to an empty list because the second argument was empty.

Why is this idiom useful? One might be tempted to put more effort into making the list of positions fit the list `xs` and write something like this

```
zip [0 .. (length xs)] xs
```

But this effort is not necessary. Thanks to lazy evaluation, the `zip` function will only take as much of the first list as needed.

(Exercise: You may have noticed that the second element of the infinite list `enumFrom 0` was the unevaluated expression `(0+1)` and not the expression `1`. It turns out that a better implementation of `enumFrom` uses the `seq` combinator.

```
enumFrom n = n `seq` (n : enumFrom (n+1))
```

Try this!)

The general usage pattern for infinite lists is this: Create an infinite list of all values of potential interest, then keep only those that are actually of interest. This way, we disentangle generation from selection, making our code clearer and more modular.

A prototypical example is Heron's method for calculating the square root of a number. (This is a more advanced example from numerics, but certainly worth studying.) Given an approximation `x` to the square root of the number `a`, a function `better` calculates a better approximation to the square root by taking an average

```
better a x = 1/2*(x + a/x)
```

The Prelude function iterate can be used to generate an infinite list of values `[x, f x, f (f x), …]` by repeatedly applying a function `f`:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

The idea is to use repeatedly apply the function `better` to generate an infinite list of better and approximations. But we also have to select a single value out of these infinitely many. The following function `within` returns the first value that is closer than a threshold `eps` from its predecessor

```
within eps (x:y:ys) =
    if abs (x-y) <= eps then y else within eps (y:ys)
```

Putting this together, we can build a function that calculates the square root:

```
squareRoot a = within (1e-14) $ iterate (better a) (a/2)
```

The beautiful thing about this approach is that the process of generating and selecting approximations are independent of each other, our code has become more modular. For instance, we can implement a different approximation method by changing just the generation part. Or we can implement a different selection scheme, for instance by replacing the function `within` the following function:

```
relative eps (x:y:ys) =
    if abs (x-y) <= eps * abs x then y else relative eps (y:ys)
```

This and many more examples of modularity by lazy evaluation can be found in the classic article "Why Functional Programming Matters" by John Hughes. I particularly like the example about programs for finding winning strategies for games like chess. Every Haskell programmer is encouraged to read it!

In a way, lazy evaluation allows us to calculate with *potentially* infinite lists. However, it is often easier to pretend that the lists are *actually* infinite. After all, it can be quite tedious to trace all steps of lazy evaluation in detail; the issue with `(0+1)` vs `1` in the `enumFrom` example already hints at that. Treating lists as actually infinite can be justified by denotational semantics (and this is the subject of the next tutorial). The point is that it is possible to understand how the `zip [0..]` and `squareRoot` functions work *without* knowing how lazy evaluation works! In fact, the Haskell standard does *not* specify that it uses lazy evaluation, rather, it only specifies that Haskell is a *non-strict* language. This guarantees that infinite lists still work, but implementations are free to use execution models other than lazy evaluation.

By the way, in languages like Python, there is a concept called "iterator" which closely related to Haskell's infinite lists. You can think of iterators as "lists on demand". In Haskell, there is no distinction: all lists are "on demand" by default.

User-defined control structures

So far, we have looked at how lazy evaluation allows us to reuse library functions in a larger context. However, it also has another benefit which has less to do with reuse and more with expressiveness, namely the ability to implement *control structures* as functions.

Most imperative languages like C or Java have a limited set of control structures like `if … then … else`, `while`, `for` and a few others. Some languages like C# add new ones like `foreach`, which add convenience in some use cases. Yet, all these constructs are baked into the language, the programmer cannot add any new control structures.

In Haskell, we are not limited to a fixed set of control structures -- in fact, there are no predefined control structures, everything can be expressed as functions. One ingredient to user-defined control structures in Haskell are higher order functions, giving us combinators like `map`, `filter` and `foldr`. The other ingredient is -- lazy evaluation. As an example, let us try to reimplement the standard `if … then … else` structure as a function:

```haskell
myIf :: Bool -> a -> a -> a
myIf True  x y = x
myIf False x y = y
```

In a language with eager evaluation, like LISP or F#, this is *not* a drop-in replacement for `if … then … else`. Consider the expression

```haskell
myIf (x /= 0) (100 / x) 0
```

In those languages, the second argument of the `myIf` function is evaluated even when the value `x` is equal to `0`, resulting in a division by zero exception. In LISP, one would have to resort to a macro. Not so in Haskell, where the division is only carried out when the value is nonzero. Thanks to lazy evaluation, we can express many constructs as plain functions where other languages would have to resort to macros.

Another example of a user-defined controls structure would be the forM function, which is occasionally useful when programming in the imperative style. It can be used as a `for` loop:

```haskell
forM [1..100] (\i -> …)
```

The first argument is a list of values that are passed to the action in the second argument. Thanks to lazy evaluation, the list of numbers is not evaluated to full normal form, but its elements are demanded step by step whenever the action is executed, just like in a traditional `for` loop. Of course, we are not limited to lists of numbers, this function essentially becomes a `foreach` structure when applied to, say, a list of strings:

```haskell
filenames <- getDirectoryContents "."
forM filenames (\filename -> …)
```

To summarize, lazy evaluation is one ingredient that allows us to implement custom control flow. One can take these ideas further and embed entire mini-languages in Haskell, in a way using it as a host, or "macro" language, where lazy evaluation ensures that "macros" are only expanded when needed. This is the subject of domain specific languages (DSLs), but that's a story for another day.

The extreme minimum

We now go back to the topic of code reuse and take it to an extreme. Imagine that we want to find the smallest element in a list. Certainly, we can do that by first sorting the list and then

taking the head of the result:

```
minimum = head . sort
```

Of course, this seems rather inefficient. If the list has *n* elements, then a good sorting algorithm will take at least *O(n log n)* time to sort it. But to find the minimum of a list, scanning each element would be enough, and this would take only linear time, *O(n)*. However, this calculation was done without lazy evaluation! We know that `head` will not evaluate the tail of the result, and as we will see below, most sorting algorithms actually only need linear time to return the smallest element, so thanks to lazy evaluation, this seeminlgy crazy implementation is in the same efficiency class as the direct version.

Of course, the question is: Should you really write code like this in practice? I would say: Yes! You see, we often write code that we later throw away: Maybe we didn't need to calculate the minimum at all, but rather the length of the list. Instead of thinking very long about how to best implement the function `minimum`, we can save a lot of time by jotting down the first thing that works, for instance the combination of the library functions `head` and `sort`. Thanks to lazy evaluation, the result will perform reasonably well. Of course, once our code has settled, we can go back and optimize any functions that are too inefficient. Still, in general, it is better to start with code that works correctly, if a little slow, than code that produces incorrect results very fast.

We now consider a concrete sorting algorithm and show that the combination of `head` and `sort` takes linear time. We will look a variant of quicksort, but mergesort would work as well. Here is the algorithm:

```
sort :: Ord a => [a] -> [a]
sort []     = []
sort (x:xs) = sort ls ++ [x] ++ sort rs
    where
    ls = filter ( < x) xs
    rs = filter (>= x) xs
```

We assume that we always choose a good pivot element `x`, so that the lists `ls` and `rs` of elements that are smaller, resp. larger than the pivot have length about *n/2* where *n* is the length of the total list `xs`; likewise for the recursive applications.

To estimate the running time, we make use of the fact that lazy evaluation takes no more reduction steps than eager evaluation. Hence, we can freely intermingle lazy and eager evaluation and count the number of reduction steps, knowing that this can only be more than what lazy evaluation really needs.

The idea is that we evaluate the application `sort` recursively, but only reduce the left applications of `sort` and leave the right ones unevaluated:

```
head (sort xs)
~> head (sort ls ++ [x] ++ sort rs)
~> head ((sort lls ++ [lx] ++ sort lrs) ++ [x] ++ sort rs)
~> head (((sort llls ++ [llx] ++ sort llrs) ++ [lx] ++ sort lrs) ++ …)
```

To perform these reduction steps, we have to evaluate the left lists `ls`, `lls` and so on to normal form, which takes about

O(n) + O(n/2) + O(n/4) + ... = O(n)

reduction steps, since the evaluation of `filter` takes linear time and each list has half the length of the previous one. At some point, the left list will be empty

```
=> head ((([] ++ [y] ++ sort …) ++ […] ++ sort …) ++ …)
```

and lazy evaluation proceeds to evaluate the `head` of the concatenations `(++)` in another *O(log n)* reduction steps, because that's the number of concatenations that `head` has to "pass through" before reaching the smallest element `y`. This is significantly faster than linear, so overall, the combination `head (sort xs)` only takes *O(n)* reduction steps to produce the smallest element! Amazing!

Conclusion

We have looked at various examples of how lazy evaluation can help with making our code more modular: First, we have looked at short-circuiting logical operations, and seen that general purpose library functions will work well together if we adhere to the principle that they only generate as much of the result as demanded. This principle also allows us to program with infinite lists, enabling useful idioms like `zip [0..]` or the "generate-then-select" style. After a short aside on the topic of custom control structures, we have used the idea of combining general purpose functions in the seemingly extreme, but hopefully illuminating example of calculating the minimum of a list by sorting it.

In short, lazy evaluation is a key ingredient for a programming style that is also known as *wholemeal programming*: Instead of looking at small pieces, like the individual elements of a list, look at the "big picture", the whole list, and build programs by combining general purpose functions on these. By manipulating data from this higher pointer of view, it becomes easier to write more correct programs more quickly.

Written by

Heinrich Apfelmus

Europe/Berlin

Haskell programmer for more than 10 years. I maintain several open source libraries, currently focusing on graphical user interfaces (GUI) and functional reactive programming (FRP). Background in mathematics, physics and theoretical computer science. Besides teaching in person, I also try to pass on what I learned in the form of tutorials and blog posts. You can find them either here on HackHands, or on my personal website.

REQUEST EXPERT

$2 / min

5 Comments HackHands 1 Login

Recommend Share Sort by Best

Join the discussion…

hr • a year ago

It's a clear article, it'll help me explain lazy evaluation to others.

One note: eq [1..3] [1..4] evaluates to True because zipWith shortens the second list, maybe call it prefixEq?

• Reply • Share ›

Heinrich Apfelmus hr • a year ago

Ouch. Thanks a lot! I would like to keep the use of `zipWith`, so I'm going to call it `prefix`, indeed.

• Reply • Share ›

maxigit • a year ago

is not O(n) + O(n/2) + O(n/4) ... = O(n*log(n)) ???

• Reply • Share ›

pk maxigit • a year ago

No, because the infinite geometric series converges as:
1 + 1/2 + 1/4 + ... = 2 = O(1)

• Reply • Share ›

maxigit pk • a year ago

Of course. Thanks

• Reply • Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy

Get on the mailing list!

The latest major updates, and nothing else.

your name

you@domain.com

Favorite technologies

SUBSCRIBE

Get live help for these popular subjects plus many more!

Java Node.js Android C# & .NET PHP
Ember.js iOS Javascript Angular.js Golang
Meteor.js SQL Ruby on Rails HTML & CSS CoffeeScript
Haskell Python & Django Scala

Latest Tweets - @hackhands

@1Kyandi That's no bueno. We can certainly help with that. Message us at https://t.co/mph0ePw1nm so we can look up your account.

---

hack.hands( )

© 2015 HackHands Inc. All rights reserved.

Terms of use | Privacy Policy