Semilazy data structures in Haskell

Andrew Lelechenko
1@dxdy.ru

Barclays, London

f(by) 2019, Minsk, 26.01.2019

Be lazy!

Haskell is a lazy language. By default the expression (or any of its subexpressions) is not evaluated until its value is utterly and unavoidably needed.

```
integers :: [Int]
integers = [0..]

f :: Int -> Int
f 42 = 3
f x = <infinite_loop>

> (map f integers) !! 42
3
```

Laziness is an abstraction to handle potentially infinite processes as actually infinite objects in a pure functional way.

Introduction to schedules

A schedule is recursively defined as one of

- the full calendar,
- a literal list of dates,
- all specific weekdays (Mondays, Tuesdays, etc.),
- union $\cup$ of two schedules,
- intersection $\cap$ of two schedules,
- all sorts of random stuff, like each day of a given schedule, which is the fifth Tuesday of the month and is directly preceded by a fourth Monday.

It all boils down to the algebra of sets.

A simple schedule

A trader buys Microsoft stock on New York Stock Exchange and Dubai Financial Market and sell it on Moscow Exchange. What is the trading schedule?

- *NYSE*:
  - Take the full calendar.
  - Remove all Saturdays and Sundays.
  - Remove a list of public holidays in USA.
- *DFM*:
  - Take the full calendar.
  - Remove all Fridays and Saturdays.
  - Remove a list of public holidays in OAE.
- *MOEX*:
  - Take the full calendar.
  - Remove all Saturdays and Sundays.
  - Remove a list of public holidays in Russia.
  - Add a list of working holidays in Russia.
- Return $(NYSE \cup DFM) \cap MOEX$.

List vs. Set vs. Vector

How can we represent a set of unique values?

- A lazy single-linked list `[a]`:
  $O(1)$ insert, $O(n)$ lookup, huge memory overhead.
- A fixed-sized array `Vector a`:
  $O(n)$ insert, $O(\log n)$ lookup, low memory overhead.
- A binary search tree `Set a`:
  $O(\log n)$ insert, $O(\log n)$ lookup, medium memory overhead.

Usually binary search trees are the way to go.

Unless the set is infinite.

How to generate first 100 dates of a schedule?

How many dates of x and y need to be precomputed before we are able to return first 100 dates of $x \cap y$?

We can use try-and-guess: take first 100 dates of both schedules: x_{100} and y_{100}. If $x_{100} \cap y_{100}$ contains at least 100 dates we are done. Otherwise take first 200 dates of x and y and try again, etc. This is ugly and inefficient.

Another idea is to proclaim a Doomsday on 29th of February 2900 and compute everything up to this date, in a vain hope that our system will get decomissioned earlier. This is vastly inefficient.

We'd rather work with infinite sets in a lazy fashion. Unfortunately, we cannot work so with trees or with arrays.

> *...when you have eliminated the impossible, whatever remains, however improbable, must be the truth.*
>
> — *One lazy detective*

Schedule as a lazy distinct sorted list — 1

How to merge (possibly infinite) ordered lists lazily? We cannot use operations from `Data.List`.

```
merge [1,3,6] [2,4,5,7] = [1,2,3,4,5,6,7]
merge [1,3..]  [2,4..]  = [1..]
```

Find inspiration in the merge sort!

```
merge ::  Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = case x `compare` y of
   LT -> x :  merge xs (y:ys)
   _  -> y :  merge (x:xs) ys
```

Similar definitions may be given for set intersection, difference, etc.

Schedule as a lazy distinct sorted list — 2

There are several packages, defining operations on sorted lists:

Package	Fatal flaws
`sorted`	NIH, abandoned
`sorted-list`	NIH, allows repetitions
`data-ordlist`	NIH, provides no type safety

Here I intended to insert the XKCD comix about 15 competing standards, but forgot how to embed pictures in LaTeX.

Solution: write a new package `containers-lazy`. It mimics full `Data.Set` interface, provides a newtype with safe constructors and operates over sets without repetitions only.

Available from
`https://github.com/Bodigrim/containers-lazy`

Semilazy sets — 1

The chosen representation of schedules fits well to listing of first n dates. Complexity of $\cap$ and $\cup$ is $O(n)$, same to binary trees.

But lookups suffer from poor performance: $O(n)$ instead of $O(\log n)$. Can we make sets great again?

```
data Semilazy a = SL
  { strictInit ::  Set a
  ,   lazyTail ::  [a]
  }
```

E. g., `SL (Set.fromList [1,3,5,9]) [10,20..]`.

`Semilazy` maintains the invariant: the last element of `strictInit` is less than the first element of `lazyTail`.

Semilazy sets — 2

Is it a valid definition of merge?

```
merge ::  Semilazy a -> Semilazy a -> Semilazy a
merge (SL s1 ls1) (SL s2 ls2) =
   SL (s1 `Set.merge` s2) (ls1 `merge` ls2)
```

No, because it does not maintain the invariant:

```
merge (SL empty [0..])  (SL (Set.fromList [10]) []) =
   SL (Set.fromList [10] [0..])
```

Valid implementation:

```
merge (SL s1 (l1:ls1)) (SL s2 (l2:ls2))
  | l1 < l2, (xs, ys) <- span (< l2) ls1
  = SL (s1 `Set.merge` Set.fromList xs `Set.merge` s2)
(ys `merge` ls2)
  | otherwise = ...
```

Semilazy sets — 3

To be as lazy as possible the actual implementation maintains not only a strict init and a lazy tail, but also a position of delimiter between them.

```
data Delimiter a = Bottom | Middle a | Top

data Ascension a = Ascension
   { strictInit :: Set a
   , delimiter  :: Delimiter a
   , lazyTail   :: [a]
   }
```

Available from `https://github.com/Bodigrim/ascension`

Full speed astern

It is still not entirely satisfying, because lookups take between $O(\log n)$ and $O(n)$ time. Can we achieve amortized $O(1)$ time?

In finite setting when lookups become a bottleneck and inserts are rare, one can use a bit array. The set is represented by a raw region of memory, where i-th bit equals to 1 when i is an element of the set and equals to 0 otherwise. By the vary nature bit arrays are strict: there is simply no space to store any pointer to deferred computation.

Bit arrays provide superfast set intersection / union by means of bitwise and / or.

There is a Haskell implementation of bit arrays in `bitvec` package.

Can we implement an infinite bit array? Since it is infinite it must somehow involve laziness.

Tricks from a can of worms

How do dynamic arrays work in imperative languages? They occupy memory enough to store 2^k elements. While the actual size remains below 2^k, appending new elements does not require reallocation. Only when the size rises beyond 2^k, new chunk of 2^{k+1} size is allocated and the existing array is copied there.

Let us have an infinite lazy list of strict bit arrays of growing size: [ptr to 64 bit block, ptr to 128 bit block, ptr to 256 bit block, ...]

The lookup function takes an index n, traverses the outer list to extract $m = \log_2(n/64)$-th element and returns the relevant bit. For example, to check whether 200 is an element we traverse until the 3-rd block and return its $200 - (64 + 128) = 8$-th bit.

It is better to store bit blocks in a lazy array with instant indexing. Since chunks grow rapidly, for all practical applications an outer array of size 64 will suffice. This gives us amortized $O(1)$ indexing.

Chimera

This approach (lazy outer array of pointers to growing inner arrays) can be generalized from storing bits to storing any data and is implemented in `chimera` package.

```
data Chimera a = Vector (Vector a)
```

`tabulate` takes predicate and returns an infinite bit array:

```
tabulate ::  (Word -> a) -> Chimera a
```

`index` implements random access in $O(1)$ amortized time:

```
index    ::  Chimera a -> (Word -> a)
```

Caching

Let us use tabulate and index to get a fully functional caching in a purely functional and performant manner:

```
expensive ::  Word -> a
expensive x = <heat_cpu_for_ten_minutes>

cache ::  Chimera a
cache = tabulate expensive

cheap ::  Word -> a
cheap = index cache
```

Fibonacci 101

Let us define Fibonacci numbers in a naïve, exponential way:

```
fibo ::  Word -> Natural
fibo n = if n < 2 then n else fibo (n-1) + fibo (n-2)
```

We can cache it as is:

```
fiboCache ::  Chimera Natural
fiboCache = tabulate fibo

fibo' ::  Word -> Natural
fibo' = index fiboCache
```

But recursive calls still know nothing about cache. Can we make them aware of?

Fixed-point combinator

Any recursive function can be expressed as a non-recursive one and the `fix` combinator a. k. a. Y combinator.

```
fix :: (a -> a) -> a
fix f = let x = f x in x

fiboFix :: (Word -> Natural) -> (Word -> Natural)
fiboFix f n = if n < 2 then n else f (n-1) + f (n-2)

fibo :: (Word -> Natural)
fibo = fix fiboFix
```

Now use `tabulateFix` to cache all recursive calls as well:

```
fiboCache = tabulateFix fiboFix :: Chimera Natural

fibo = index fiboCache :: Word -> Natural
```

Summary

- Hybrid combination of a strict binary search tree and a lazy list allows to work with infinite sets.
- If lookups become a bottleneck, one can trade space for speed and switch to an infinite bit mask, hybrid of a lazy array and a bit array.
- The latter approach can be generalized to store any data, applicable for transparent caching of functions, including recursive ones.

Thank you!