Monad P3 : Mutability and Strictness (1C)

Young Won Lim
6/13/20

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

Mutability

Mutability demanding cases

1) When a **library** written in another language
   which assumes mutable state. is called in Haskell
   eg) event-callback GUI toolkits.

2) Using Haskell to implement a **language**
   that provides imperative-style mutable variables.

3) Implementing **algorithms** that inherently require
   destructive updates to variables.

4) Dealing with volumes of bulk data massive enough
   to justify squeezing every drop of computational power available
   to make the problem at hand feasible.

https://en.wikibooks.org/wiki/Haskell/Mutable_objects

External and internal demands on mutability

external demands can impose **mutability** on the code

- **library** written in non-functional language (1)
- **language** with imperative style variable (2)

internal demands can impose **mutability** on the code

- **algorithms** may require **mutability** (3)
- **extreme** computational demands (4)

these do not include all the cases

https://en.wikibooks.org/wiki/Haskell/Mutable_objects

Sorting problem

sorting a list does not require

mutability in any essential way,

a **function** that sorts a list and returns a new list,

should be **functionally pure**

even if the sorting algorithm uses

destructive updates

to swap the position of the elements.

In such case, the **mutability** is

just an **implementation detail**.

https://en.wikibooks.org/wiki/Haskell/Mutable_objects

Functional purity and mutability

Keeping **functional purity**

even though the **mutability** is allowed

in an **implementation detail**.

The standard libraries provide **ST monad**

as a nifty tool for handling such situations

while maintaining **pure functions**

the **ST monad** in **Control.Monad.ST**

allows **mutability**

keeps **functional purity**

https://en.wikibooks.org/wiki/Haskell/Mutable_objects

Temporary and local mutable effects

ST monad allows **temporary** and **local mutable effects**.

Because of the way that **ST monad** is implemented,

- none of the effects can be visible from outside of a function

- with the the same input, the function always has the same output. (**purity**)

https://www.snoyman.com/blog/2017/12/what-makes-haskell-unique

Mutable Data Structure

Mutable data structures can be found in the **libraries**

mutable arrays (alongside with **immutable arrays**)
in the **vector package** or the **array package**
bundled with GHC

There are also **mutable hash tables**,
such as those from the **hashtables package**.

In all cases mentioned,
both **ST** and **IO** versions are provided.

https://en.wikibooks.org/wiki/Haskell/Mutable_objects

Mutability Haskell Approaches

to sort more efficiently a vector,

allow **mutable access**

instead of using only **pure operations**.

Haskell has two approaches for **mutable access**

1) mutable data structures

2) mutable copy

https://www.snoyman.com/blog/2017/12/what-makes-haskell-unique

Mutable data structure approach

The first is the ability

to explicitly **create** **mutable data structures**,

and **mutate** them in place. (**mutable arrays**)

if you need the performance, it's available.

unlike **mutable-by-default** approaches,

you now know exactly

which pieces of data you need to handle with care

when coding to avoid tripping yourself up.

https://www.snoyman.com/blog/2017/12/what-makes-haskell-unique

Mutable copy approach

The other approach is

to **create** a **mutable copy** of the original data,

perform your **mutable algorithm** on it,

then **freeze the new copy**

into **an immutable version**.

While this approach requires

an **extra memory buffer**

an **extra copy** of the elements in the vector,

it avoids completely the worries of your data

being changed behind your back.

https://www.snoyman.com/blog/2017/12/what-makes-haskell-unique

Mutable copy approach – sorting examples

```
sortMutable :: MutableVector a -> ST (MutableVector a)
sortMutable = ... -- normal sorting algorithm

sortImmutable :: Vector a -> Vector a
sortImmutable orig = runST $ do
  mutable <- newMutableVector (length orig)
  copyValues orig mutable
  sort mutable
  freeze mutable
```

https://www.snoyman.com/blog/2017/12/what-makes-haskell-unique

Two phase arrays (Mutable copy approach)

An **immutable array** cannot directly **update** it elements **in-place**

semantically simplicity of **immutable array** allow
efficient indexed-based array construction for **mutable** arrays.

Hence, computationally demanding Haskell array code
typically adopts a **two-phase array** life cycle:

(1) arrays are allocated as **mutable** arrays and
initialised using in-place array update;

(2) they are **frozen** by making them **immutable**,
once **initialised**,

https://www.tweag.io/posts/2017-09-27-array-package.html

Immutable array type

to implement custom array algorithms

Haskell has a simple array API in the **Data.Array** module.

These are **immutable**, **boxed**, and **non-strict**.

This allows for the elegant, high-level description

of many array algorithms,

But **boxing** and **non-strictness**

give suboptimal performances

for compute-intensive applications

.

https://www.tweag.io/posts/2017-09-27-array-package.html

Mutable array types

Mutable arrays come in various flavours,
distinguished by the **monad**
in which the **array operations** take place.
Usually, either **IO** or **ST**, and the array package provides
both **boxed** and **unboxed** variants for both monads.

mutable boxed	**IOArray**
mutable unboxed	**IOUArray**
mutable boxed	**STArray**
mutable unboxed	**STUArray**.

https://www.tweag.io/posts/2017-09-27-array-package.html

Two phase array usage example

The above definition of generate uses **STUArray** to **initialise** the array, and then, **freezes** it into a **UArray**, which is returned.

STUArray	**Mutable Unboxed Array**
Uarray	**Immutable Unboxed Array**

The choice of **STUArray** is implicit in the use of **runSTUArray**, which executes the code in the state transformer **monad ST** and **freezes** the **STUArray** into a **UArray**

https://www.tweag.io/posts/2017-09-27-array-package.html

Boxed vs. Unboxed

Boxed Arrays

For **lazy evaluation**,

values are represented at runtime as **pointers** to

either their **value**, or

code for computing their value.

Box :

this extra level of **indirection**

any **extra tags** needed by the runtime

The default **boxed arrays** consist of

many of these boxes,

each of which may compute its value separately.

https://www.tweag.io/posts/2017-09-27-array-package.html

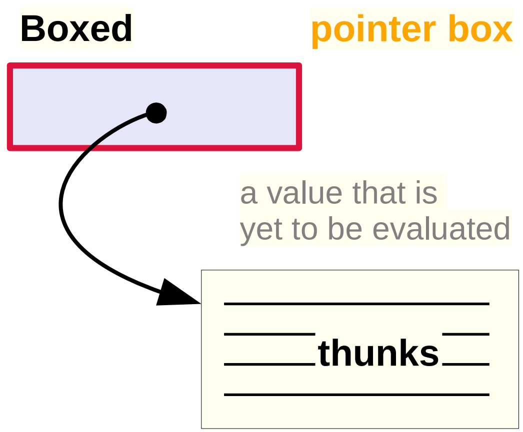
Boxed representation

the **expressiveness** of **non-strict** arrays comes at a price,
especially if the array elements are simple numbers (**values**).

Instead of direct storing those numeric elements,
non-strict arrays require a **boxed** representation

the **elements** are **pointers** to **heap objects**
containing the **numeric values**.

This **additional indirection** requires extra **memory** and
drastically reduces the **efficiency** of array access,
especially in **tight loops**.



https://www.tweag.io/posts/2017-09-27-array-package.html

Boxed Arrays – pros and cons

allow **recursively defining**

an array's **elements** in terms of one another

can compute only the **specific elements** of the array

which are ever needed

for large arrays, it costs a lot in terms of overhead,

and if the entire array is always needed, it can be a waste.

https://www.tweag.io/posts/2017-09-27-array-package.html

Unboxed Arrays

Unboxed arrays are more like **arrays in C** -
they contain just the **plain values**
without this **extra level** of **indirection**,

an array of 1024 values of type Int32
will use only 4 KB (=4*1024) of memory.

https://www.tweag.io/posts/2017-09-27-array-package.html

Unboxed Arrays – pros and cons

indexing of **unboxed arrays** can be significantly faster.

can only have **plain values** having a **fixed size**

can not have **types** defined with **variable size**

without the extra level of indirection,

all of the elements must be evaluated,

when the array is evaluated,

no benefits of **lazy evaluation**.

can not **define recursively** the array elements

in terms of each other

https://www.tweag.io/posts/2017-09-27-array-package.html

Non-strict boxed vs. Strict unboxed arrays

While **boxed** representation can be used in

both **strict** and **non-strict** data structures.

Generally **non-strict** structures typically <u>require</u> **boxing**.

non-strict boxed	**Data.Array.IArray.IArray**	**Immutable**
strict unboxed	**Data.Array.Unboxed.Uarray**	**Immutable**

the **unboxed array element type**

is restricted to **basic types** (fixed size)

such as **integral** and **floating-point** numeric types

https://www.tweag.io/posts/2017-09-27-array-package.html

Applications which require non-strictness

wavefront example

the recursive definition of the array arr.

arr!(i,j-1) + arr!(i-1,j-1) + arr!(i-1,j)

the elements are accessed

to the left, top, and top-left of the current one

Such a **recursive dependency** is

only valid for a **non-strict** data structure.

non-strict boxed

https://www.tweag.io/posts/2017-09-27-array-package.html

Strict v.s Non-strict (Lazy)

Strict (Eager) Evaluation

Strict evaluation, or **eager evaluation**

expressions are evaluated

as soon as they are bound to a variable.

with **strict evaluation**,

when **x = 3 * 7** is read,

3 * 7 is immediately computed

and **21** is bound to **x**.

https://en.wikibooks.org/wiki/Haskell/Strictness

Lazy (Non-strict) Evaluation

Conversely, with **lazy evaluation**

values are computed

only when they are needed.

In the example **x = 3 * 7**,

3 * 7 will not be evaluated until it's needed,

like if you needed to output the value of x.

https://en.wikibooks.org/wiki/Haskell/Strictness

Function call and argument evaluation

In most languages, **calling a function**
with non-prime-expressions as **arguments** (atomic values)
requires **strict evaluation**

foo (x + 1, bar(3, 7));

- first evaluate all the arguments **x+1** and **bar(3,7)**
- and then call foo on the results.

https://www.reddit.com/r/haskellquestions/comments/6xk5hv/the_sequence_function/

Prime expression in Haskell

The prime (') is treated like any number in variable names,
i.e. unless it's at the beginning you can use it just like a letter.

Hence names such as foldl';
generally those will refer some kind of "alternative" of a similar thing,

But surrounding a function with backticks
lets you use it like an **infix operator**, e.g.

```
plus :: Int -> Int -> Int
plus = (+)

   Prelude> 4 `plus` 5
   9
```

https://stackoverflow.com/questions/22873663/what-does-prime-mean-in-haskell

Lazy Evaluation

lazy evaluation : a core feature of GHC

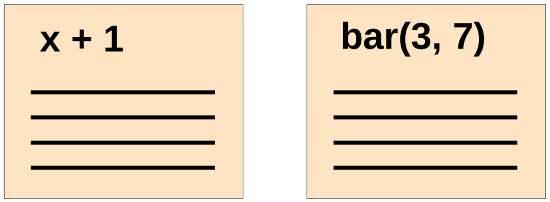
It doesn't matter whether you have **monads** or anything involved.

foo (x + 1, bar(3, 7));

Haskell just packages up each argument in a data structure containing everything needed to compute it later, and passes those data structures into **foo**,

they (data structiure: **thunks**) are only evaluated if **foo** requires access to their values.

code ….. thunks



https://www.reddit.com/r/haskellquestions/comments/6xk5hv/the_sequence_function/

Lazy evaluation of an infinite list

lazy evaluation is applied almost everywhere in Haskel

takeWhile (\x -> x < 4) [1..]

can returns **[1,2,3]**

without getting stuck evaluating the infinite list,

with no monads involved.

takeWhile :: (a -> Bool) -> [a] -> [a]

creates a list from another one, it inspects the original list

and takes from it its elements to the moment

when the condition fails, then it stops processing

https://www.reddit.com/r/haskellquestions/comments/6xk5hv/the_sequence_function/

Lazy evaluation in function calls – thunks

By default, Haskell uses **lazy evaluation**

when you call a function,

the body won't execute immediately,

rather it will return something (**thunks**)

that represents executing the body.

The body will only be actually executed

when the **result** of the function

is used in an **IO computation**,

either directly or via being used in another function

or chain of functions that is used in an **IO computation**.

thunks
unevaluated function executions

https://www.reddit.com/r/programming/comments/3sux1d/strict_haskell_xstrict_has_landed/

Strict evaluation for performance

Having unevaluated function executions

(**thunks**) makes it harder to reason

about **memory usage** and **performance**.

Bookkeeping for these thunks can also

impose a slight **performance penalty**.

Strict Haskell gives Haskell **strict evaluation**,

which is the kind of evaluation most other languages have,

and hence makes it easier to reason about **performance**.

strict evaluation

⇕

lazy evaluation ………. thunks

https://www.reddit.com/r/programming/comments/3sux1d/strict_haskell_xstrict_has_landed/

Performance issues of laziness

Haskell is a **non-strict** language,
and most implementations use a strategy
called **laziness** to run your program.

laziness = non-strictness + sharing

Laziness can be a useful tool
for **improving performance**,
but *sometimes* it reduces performance
by adding a constant overhead to everything.

https://wiki.haskell.org/Performance/Strictness

Sharing

Sharing means that **temporary data** is physically stored,
if it is used multiple times.

```
let x = sin 2
in  x*x
```

x is used twice as factor in the product **x*x**.

Due to **referential transparency** it does not play a role,
whether **sin 2** is computed twice or
whether it is computed once and
the result is stored and reused.

https://wiki.haskell.org/Performance/Strictness

Sharing when computation is cheap

However, when you let the Haskell compiler decide

whether to compute or to store the result.

sharing can be the wrong way,

if a computation is cheap but storing the result is huge.

[0..1000000] ++ [0..1000000]

where it is much cheaper to compute the list of numbers

than to store it with full length.

https://wiki.haskell.org/Performance/Strictness

Cost of thunks

Because of **laziness**, the compiler can't
evaluate a **function argument**
and pass the value to the function,

it has to record the expression in the **heap**
in a **suspension** (or **thunk**)
in case it is evaluated later.

storing and evaluating **suspensions** is *costly*,
and *unnecessary* if the expression was going
to be evaluated anyway.

https://wiki.haskell.org/Performance/Strictness

Non-Strict and Strict Semantics

An expression language is said to have

non-strict semantics

if expressions can have a **value**

even if some of their subexpressions do not.

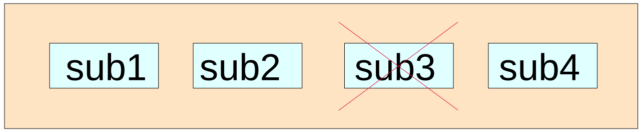
strict semantics

if any subexpression fail to have a value,

the whole expression fails with it.

Haskell has **non-strict semantics by default**:

nearly every **other language** has **strict semantics**



https://wiki.haskell.org/Non-strict_semantics

Strict Semantics

strict semantics

the opposite of **non-strict semantics**.

an undefined argument of a function

leads to an undefined function value.

forall f.	**f undefined**	**=**	**undefined**
	argument		returned value

It may be implemented by **eager evaluation**.

https://wiki.haskell.org/Sstrict_semantics

Order of lazy evaluations

To evaluate an **expression**,

replace all **function applications** by their **definitions**.

The order in which you do this

does not matter much, but it's still important:

start with the **outermost application**

and proceed **from left to right**;

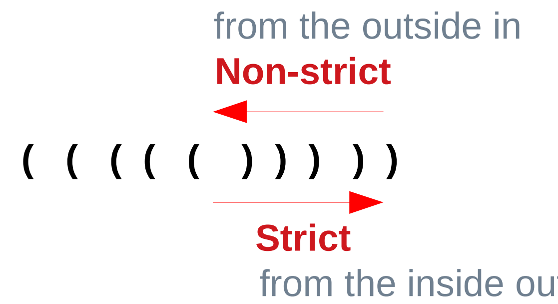
this is called **lazy evaluation**.

https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form

Order of evaluations – strict vs. non-strict

Non-strictness means that **reduction**
(the mathematical term for evaluation)
proceeds from the outside in,
so if you have **(a+(b*c))**
then first you reduce the **+**,
then you reduce the inner **(b*c)**.

Strict languages work the other way around,
starting with the innermost brackets and working outwards.



https://wiki.haskell.org/Lazy_vs._non-strict

Order of evaluations – the bottom value

Direction of evaluation matters to the **semantics**

Consider an expression that evaluates to **bottom** (i.e. an error or endless loop)

any **strict** language **(Strict case)**

that starts at the inside and works outwards

will always find that **bottom** value,

and hence the **bottom** will propagate outwards.

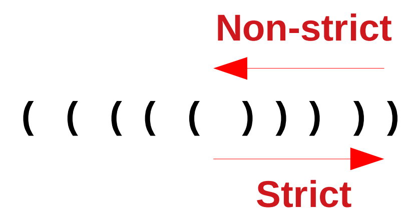
if you start from the outside and work inside **(Non-strict case)**

then some of the sub-expressions

are eliminated by the outer **reductions**,

so they may not be evaluated and

you don't get **bottom**



https://wiki.haskell.org/Lazy_vs._non-strict

Order of evaluations – the bottom value

Direction of evaluation matters to the **semantics**

Consider an expression that evaluates to **bottom**

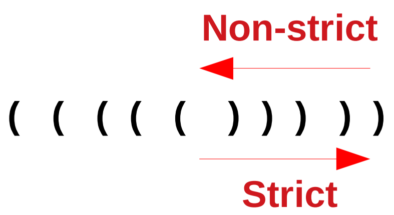
(i.e. an error or endless loop)

any **strict** language **(Strict case)**

the **bottom** will <u>propagate</u> <u>outwards</u>.

Any non-strict language **(Non-strict case)**

The bottom may be omitted



https://wiki.haskell.org/Lazy_vs._non-strict

Lazy evaluation – thunks

Lazy evaluation means
only evaluating an expression **when** its results are needed
(note the shift from "**reduction**" to "**evaluation**").

So when the evaluation engine sees an expression
it builds a **thunk** data structure containing
whatever **values** are needed to evaluate the expression,
plus a **pointer** to the expression itself.

When the result is actually needed
the evaluation engine **calls** the expression
and then **replaces** the thunk with the result for future reference.

https://wiki.haskell.org/Lazy_vs._non-strict

Lazy vs. non-strict

Obviously there is a strong correspondence
between a **thunk** and a **partial evaluation**

Hence in most cases
the terms **lazy** and **non-strict** are synonyms.

But not quite. For instance you could imagine
an evaluation engine on highly parallel hardware
that fires off sub-expression evaluation eagerly,
but then *throws away* results that are not needed.

https://wiki.haskell.org/Lazy_vs._non-strict

Strictness in pattern matching

In practice Haskell is not a purely **lazy language**:

for instance **pattern matching** is usually **strict**

So trying a pattern match forces evaluation

to happen at least far enough to accept or reject the match.

you can prepend a ~ in order to make pattern matches **lazy**

https://wiki.haskell.org/Lazy_vs._non-strict

Strictness in strictness analyzer

In practice Haskell is not a purely **lazy language**:

the **strictness analyzer** also looks for cases

where sub-expressions are always

required by the outer expression,

and converts those into eager evaluation.

It can do this because

the **semantics** (in terms of "bottom") don't change.

https://wiki.haskell.org/Lazy_vs._non-strict

Seq

Programmers can also use the **seq** primitive

to force an expression to evaluate regardless of

whether the result will ever be used.

$! is defined in terms of **seq**.

https://wiki.haskell.org/Lazy_vs._non-strict

$! – strict application

$! is **strict application**,

```
f $! x  = x `seq` f x
```

Consider the following example

```
do state1 <- act state
   dispatch $! state1
```

the difference from **dispatch state1** is
that **state1** is guaranteed to be evaluated and
not just kept as a **lazy thunk**.

forcing evaluation in this way can be
important for efficiency issues, such as preventing memory leaks.

https://stackoverflow.com/questions/25987726/what-does-mean-do-in-haskell

$! – strict application

Non-strict refers to semantics: the mathematical meaning of an expression. The world to which non-strict applies has no concept of the running time of a function, memory consumption, or even a computer. It simply talks about what kinds of values in the domain map to which kinds of values in the codomain. In particular, a strict function must map the value ⊥ ("bottom" -- see the semantics link above for more about this) to ⊥; a non strict function is allowed not to do this.

https://stackoverflow.com/questions/7140978/haskell-how-does-non-strict-and-lazy-differ

$! – strict application

Lazy refers to operational behavior: the way code is executed on a real computer. Most programmers think of programs operationally, so this is probably what you are thinking. Lazy evaluation refers to implementation using thunks -- pointers to code which are replaced with a value the first time they are executed. Notice the non-semantic words here: "pointer", "first time", "executed".

https://stackoverflow.com/questions/7140978/haskell-how-does-non-strict-and-lazy-differ

$! – strict application

Lazy evaluation gives rise to non-strict semantics, which is why the concepts seem so close together. But as FUZxxl points out, laziness is not the only way to implement non-strict semantics.

https://stackoverflow.com/questions/7140978/haskell-how-does-non-strict-and-lazy-differ

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf