Monad P3 : Strict and Lazy Package Examples (2D)

Young Won Lim 6/21/20 Copyright (c) 2016 - 2020 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

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

This document was produced by using LibreOffice.

Young Won Lim 6/21/20 Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

Package Examples

Monad operations and strictness

Monad operations (bind and return)

have to be **non-strict** in fact, always!

However <u>other</u> **operations** can be <u>specific</u> to <u>each</u> monad.

For some **monad instances** are **strict** (like **IO**), and others are **non-strict** (like **[**]).

https://wiki.haskell.org/What_a_Monad_is_not#Monads_are_not_about_strictness

Strict and lazy versions of a package

Some monads have <u>multiple flavours</u>, like **State**.

Control.Monad.Trans.State.Strict

Control.Monad.Trans.State.Lazy

the following example produces a <u>usable result</u> when **Lazy** version is used

runState (sequence . repeat \$ state (\x -> (x,x+1))) 0

https://wiki.haskell.org/What_a_Monad_is_not#Monads_are_not_about_strictness

Strict vs Lazy State Monads

mtl (or its underlying transformers) package provides <u>two</u> types of **State** monad;

> Control.Monad.State.Strict Control.Monad.State.Lazy

Control.Monad.State are the re-export of Control.Monad.State.Lazy.

Strict vs Lazy Package Examples

Example A)

```
main = print $ take 5 (evalState foo ())
```

Example B)

evalState (sequence \$ repeat \$ do { n <- get; put (n*2); return n }) 1

Example C)

```
runState (sequence . repeat $ state (\x -> (x,x+1))) 0
```

Example D)

plus n x = execState (sequence \$ replicate n tick) x

Example A print \$ take 5 (1)

import Control.Monad.State.Lazy import Control.Monad.State.Strict	[1,2,3,4,5] hangs up
foo :: State () [Int] foo = traverse pure [1]	
main = print \$ take 5 (evalState foo ()	
pure [1] return [1] traverse [1,2,3,4,5,6,7,8,9,10,]	

Example A print \$ take 5 (2)

Import Control.Monad.State.Strict

In the strict version,

the pattern matches on the pair

forces its evaluation.

(evalState <u>foo</u> ())

So **traverse pure [1..]** <u>never returns</u> until its evaluation is finished.

because an infinite list is involved

Import Control.Monad.State.Strict

foo :: State () [Int] foo = traverse pure [1..]

main = print \$ take 5 (evalState foo ())

Example A print \$ take 5 (3)

Import Control.Monad.State.Lazy

 \underline{avoids} this evaluation of the pair

(evalState foo ()) evaluation is <u>forced later</u> when the pair is <u>actually needed</u>.

In this way, we can <u>manipulate</u> infinite lists in a **lazy state monad**. Import Control.Monad.State.Lazy

foo :: State () [Int] foo = traverse pure [1..]

main = print \$ take 5 (evalState foo ())

Example A print \$ take 5 (4)

This does <u>not imply</u> that we should <u>always</u> <u>prefer</u> the **lazy** version of state monad

because the **lazy** state monad often builds up **large thunks** and causes **space leaks**.

Example B repeat \$ do (1)

evalState (sequence \$ repeat \$ do { n <- get; put (n*2); return n }) 1

Control.Monad.Trans.State.Lazy

sequencing of computations is lazy, so that for example the following produces a <u>usable result</u>:

Control.Monad.Trans.State.Strict

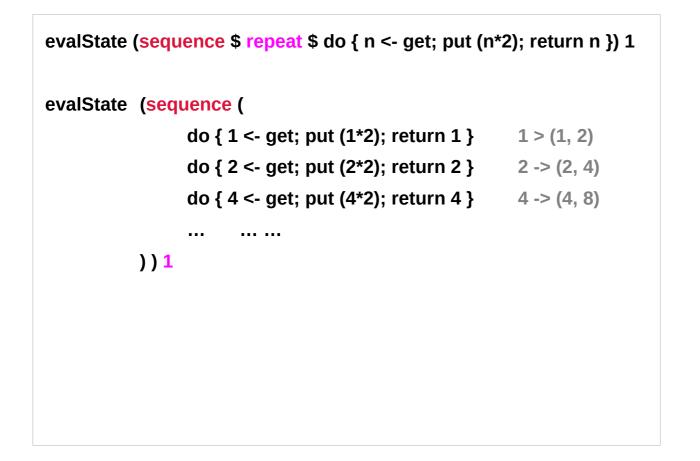
sequencing of computations is strict but computations are <u>not strict</u> in the state unless you force it with 'seq' or the like

repeat \rightarrow an infinite list

lazy sequencing of computations strict sequencing of computations

http://hackage.haskell.org/package/transformers-0.5.6.2/src/Control/Monad/Trans/State/Lazy.hs

Example B repeat \$ do (2)



non-strict computation non-strict computation non-strict computation

lazy sequencing of computations OK strict sequencing of computations Not OK

http://hackage.haskell.org/package/transformers-0.5.6.2/src/Control/Monad/Trans/State/Lazy.hs

Example C repeat \$ state (1)

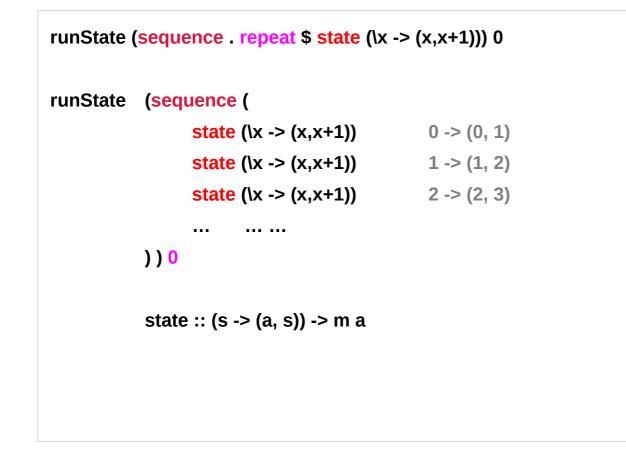
Control.Monad.Trans.State.Strict

Control.Monad.Trans.State.Lazy

runState (sequence . repeat \$ state (\x -> (x,x+1))) 0

https://wiki.haskell.org/What_a_Monad_is_not#Monads_are_not_about_strictness

Example C repeat \$ state (2)



non-strict computation non-strict computation non-strict computation

lazy sequencing of computations OK strict sequencing of computations Not OK

https://wiki.haskell.org/What_a_Monad_is_not#Monads_are_not_about_strictness

Example **D** sequence \$ replicate n (1)

A function to increment a counter.

tick :: State Int Int

tick = do n <- get

put (n+1)

return n

plusOne :: Int -> Int

```
plusOne n = execState tick n
```

plus :: Int -> Int -> Int

plus n x = execState (sequence \$ replicate n tick) x

computations are non-strict

works in both versions

Control.Monad.Trans.State.Lazy Control.Monad.Trans.State.Strict

http://hackage.haskell.org/package/transformers-0.5.6.2/src/Control/Monad/Trans/State/Lazy.hs

Example **D** sequence \$ replicate n (2)

xecState (sequence (
tick	0 -> (0, 1)
tick	1 -> (1, 2)
tick	2 -> (2, 3)
))0	

https://wiki.haskell.org/What_a_Monad_is_not#Monads_are_not_about_strictness

Traverse

traverse turns things <u>inside</u> a Traversable into a Traversable of things <u>inside</u> an Applicative , given a function that makes Applicatives out of things.	t a f (t b) (a -> f b)	
Inside a Tranversable t a Inside an Applicative f (t b)		
traverse :: (Traversable <mark>t</mark> , Applicative f) => (a -> f b) -> t a -> f (t b)		

Traverse example (1)

Let's use **Maybe** as **Applicative** and **list** as **Traversable**.

the transformation function: half :: a -> f b half x = if even x then Just (x `div` 2) else Nothing

traverse half [2,4..10] traverse half [1..10] Just [1,2,3,4,5] Nothing

```
traverse :: (Traversable t, Applicative f) => (a \rightarrow f b) \rightarrow t a \rightarrow f (t b)
```

Traverse example (2)

So if a number is even, we get half of it (inside a **Just**), else we get **Nothing**. If everything goes "well", it looks like this: traverse half [2,4..10] [2,4,6,8,10] -- Just [1,2,3,4,5] But... traverse half [1..10] [1,2,3,4,5,6,7,8,9,10] -- Nothing half x = if even x then Just (x `div` 2) else Nothing

Traverse example (3)

the <*> function is used to build the result, and when <u>one</u> of the arguments is Nothing, we get Nothing back.

(<*>) :: f (a -> b) -> f a -> f b

(<*>) simply represents function application

ap :: (Monad m) => m (a -> b) -> m a -> m b

Traverse of Replicate example (1)

rep x = replicate x x

This function generates a list of <u>length x</u> with the <u>content x</u>, e.g. **rep 3 = [3,3,3]**. What is the result of **traverse rep [1..3]**?

We get the partial results of [1], [2,2] and [3,3,3] using rep.

Now the semantics of lists as Applicatives is take all combinations,

(+) <\$> [10,20] <*> [3,4] is [13,14,23,24].

(10, 3), (10, 4), (20, 3), (20, 4)

Traverse of Replicate example (2)

```
"All combinations" of [1] and [2,2] are two times [1,2].

[1,2], [1,2]

All combinations of two times [1,2] and [3,3,3] are six times [1,2,3].

[1,2], [1,2], [3, 3, 3]

[1], [1, 2, 3], [1, 2, 3], [1, 2, 3]

[1], [1, 2, 3], [1, 2, 3], [1, 2, 3]

[1], 2, 3], [1, 2, 3], [1, 2, 3]

[1], 2, 3], [1, 2, 3], [1, 2, 3]

[1], 2, 3], [1, 2, 3], [1, 2, 3]

So we have:
```

```
traverse rep [1..3]
--[[1,2,3],[1,2,3],[1,2,3],[1,2,3],[1,2,3]]
```

repeat

```
repeat :: a -> [a]
```

it creates an infinite list where all items are the first argument

```
Input: take 4 (repeat 3)
```

```
Output: [3,3,3,3]
```

```
Input: take 6 (repeat 'A')
```

```
Output: "AAAAAA"
```

```
Input: take 6 (repeat "A")
Output: ["A","A","A","A","A","A"]
```

https://wiki.haskell.org/What_a_Monad_is_not#Monads_are_not_about_strictness

sequence Function

[m a] m a m [a]	
[~]	

https://wiki.haskell.org/All_About_Monads

sequence Function Definition 1

sequence :: Monad m ==> [m a] -> m [a] sequence = foldr mcons (return []) where mcons $p q = p \gg |x - y| > return (x:y)$ (>>=) :: m a -> (a -> m b) -> m b p :: m a х :: а q :: m b y :: b x -> q :: a -> m b y -> return (x:y) :: b -> m [b] or m [a]

https://wiki.haskell.org/All_About_Monads

sequence Function Definition 2

```
sequence :: [m a] -> m [a]
sequence [] = return []
sequence (m1:ms) =
     m1 >>= (\x -> sequence ms >>= (\xs -> return $ x:xs))
     m1 : m a
     ms : [m a]
     x :: a
     x -> sequence ms :: a -> m [a]
     xs :: [a]
     xs -> return $ x:xs :: a -> m [a]
```

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

sequence Function Definition 3

<pre>sequence [] = ret sequence (m1:ma x <- m1 xs <- sequence return (x:xs)</pre>	s) = do		
x :: a xs :: [a]	m1 :: m a ms :: [m a]		

https://stackoverflow.com/questions/5299295/why-does-application-of-sequence-on-list-of-lists-lead-to-computation-of-its-c

sequence_ Function

The **sequence**_ function (notice the underscore) has the same behavior as **sequence** but does not return a list of results.

It is useful when <u>only</u> the <u>side effects</u> of the monadic computations are important.

sequence_ :: Monad m ==> [m a] -> m ()
sequence_ = foldr (>>) (return ())

https://wiki.haskell.org/All_About_Monads

sequence Function v.s. Maybe Instance Definitions

The Maybe Monad instance

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f = Nothing
```

(Just x) >>= f = f x

```
sequence function definition
```

```
sequence :: [m a] -> m [a]
```

```
sequence [] = return []
```

```
sequence (m1:ms) =
```

m1 >>= (\x -> sequence ms >>= (\xs -> return \$ x:xs))

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

[] in sequence v.s. Nothing in Maybe

[] in the definition of sequence

the <u>first value</u> in the list is **Nothing**, so Haskell <u>discards</u> the <u>lambda function</u> (which is also where the <u>rest of the list</u> would be evaluated) and <u>returns</u> **Nothing**, because the right-hand side of (>>=)'s **Nothing** pattern doesn't include **f** at all.

```
sequence [] = return []
Nothing >>= f = Nothing
```

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

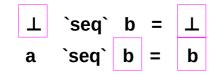
seq – 2 arguments

The **seq** function is the most basic method of introducing **strictness** to a Haskell program.

seq :: a -> b -> b

takes two arguments of any type, and returns the second.

magically strict in the first argument.



https://wiki.haskell.org/Seq

seq – data dependency and evaluation

seq doesn't evaluate anything
just by virtue of existing in the source file,
all it does is introduce an artificial data dependency

when the **result** of **seq** is <u>evaluated</u>, the **first** argument must also be 'sort of' <u>evaluated</u>.

suppose **x** :: Integer, then seq **x** b behaves essentially like

if x == 0 then b else b

<u>unconditionally</u> equal to **b**, but forcing \mathbf{x} along the way.

x `seq` x is completely redundant,

and always has exactly the same effect as just writing x.

https://wiki.haskell.org/Seq

seq – returning b

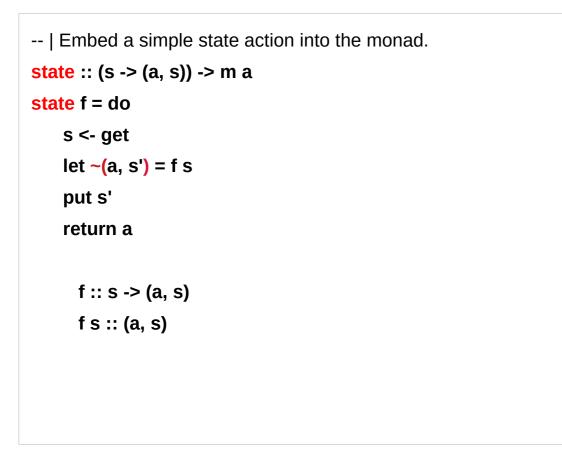
Strictly speaking, the two equations of seq are all it must satisfy,

- \perp `seq` b = \perp
- a `seq` b = b

if the compiler can statically prove	pattern matching		
that the first argument is not \bot , or	⊥`seq`b =⊥		
that its second argument is \bot ,	a`seq`⊥=⊥		
it doesn't have to evaluate anything to meet its obligations.			
In practice, this almost never happens			
However, it is the case that evaluating b and then a ,			
then returning b is a perfectly legitimate thing to do;			

https://wiki.haskell.org/Seq

state method of State monad



https://wiki.haskell.org/What_a_Monad_is_not#Monads_are_not_about_strictness

Irrefutable pattern ~(...)

foo ~(Just x) = "hello" main = putStrLn \$ foo Nothing

This uses an irrefutable pattern (the ~ part). Irrefutable patterns <u>always match</u>, so this always prints hello.

https://stackoverflow.com/questions/6711870/what-causes-irrefutable-pattern-failed-for-pattern-and-what-does-it-mean

Irrefutable pattern match error

foo ~(Just x) = x main = putStrLn \$ foo Nothing

Now, the pattern <u>still matched</u>, but when we <u>tried to use x</u> when it wasn't actually there we got an i**rrefutable pattern match erro**r:

Irr.hs: /tmp/Irr.hs:2:1-17:

Irrefutable pattern failed for pattern (Data.Maybe.Just x)

https://stackoverflow.com/questions/6711870/what-causes-irrefutable-pattern-failed-for-pattern-and-what-does-it-mean

No matching pattern

foo (Just x) = x main = putStrLn \$ foo Nothing

This is subtly distinct from the error you get when there's no matching pattern:

Irr.hs: /tmp/Irr.hs:2:1-16: Non-exhaustive patterns in function foo

https://stackoverflow.com/questions/6711870/what-causes-irrefutable-pattern-failed-for-pattern-and-what-does-it-mean

References

- [1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
- [2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf