Monad P3 : forall keyword (1E)

Young Won Lim
3/31/21

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

Three different usages for **forall**

Basically, there are 3 different common uses for the forall keyword (or at least so it seems), and each has its own Haskell extension:

ScopedTypeVariables

specify types for code inside **where** clauses

RankNTypes/Rank2Types,

The type is labeled "Rank-N" where N is the number of **forall**s which are nested and cannot be merged with a previous one.

ExistentialQuantification

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

forall

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

A set of possible values

One way to think about **forall** is

to consider **types** as a set of possible **values**.

Bool is the set **{True, False, ⊥}**

(remember that **bottom**, ⊥, is a member of every type!),

Integer is the set of integers (and bottom),

String is the set of all possible strings (and bottom), and so on.

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

Intersection of the specified types

forall serves as a way to assert a **commonality** or **intersection** of the specified types (i.e. sets of values).

forall a. a is the **intersection** of all **types**.

this **subset** turns out to be the set **{⊥}**,

since it is an implicit value in every **type**.

that is, [the **type** whose only available **value** is **bottom**]

However, since every Haskell **type** includes bottom, **{⊥}**,

this quantification in fact stipulates all Haskell **types**.

But the only permissible operations on it are

those available to [a **type** whose only available value is **bottom**]

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

A list of bottoms type (1)

1. The list **[forall a. a]**
2. The list **[forall a. Show a => a]**
3. The list **[forall a. Num a => a]**
4. The list **forall a. [a]**

a list of bottoms. **[⊥] , [⊥,⊥], ...**

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

A list of bottoms type (2)

The list, **[forall a. a]**, is the **type of a list**

whose **elements** all have the type **forall a. a**, i.e.

a list of bottoms. **[⊥] , [⊥,⊥], ...**

The list, **[forall a. Show a => a]**, is the **type of a list**

whose **elements** all have the type **forall a. Show a => a**.

the **Show** class constraint requires the possible types

also to be **a member of the class**, **Show**.

However, ⊥ is still the only value common to all these types, **{⊥}**,

so this too is **a list of bottoms**. **[forall a. a]**

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

A list of bottoms type (3)

The list, **[forall a. Num a => a]**, requires each element

to be a member of the class, **Num**.

Consequently, the possible values include **numeric literals**,

which have the specific type, **forall a. Num a => a**,

as well as **bottom**.

forall a. [a] is the type of **the list**

whose elements all have the same type **a**.

since we cannot presume any particular type at all,

this too is **a list of bottoms**.

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

Intersections over types

most **intersections** **over types** just lead to **bottoms ⊥⊥⊥⊥**

types generally don't have **any values in common**

presumptions cannot be made about a **union of their values**.

a **heterogeneous list** using a **type hider**

type hider' functions as a **wrapper type**

which guarantees certain facilities

by implying a **predicate** or **constraint** on the permissible **types**.

the purpose of **forall** is to impose **type constraint**

on the permissible types within a **type declaration**

guaranteeing certain facilities with such types.

```
data ShowBox = forall s. Show s => SB s
heteroList :: [ShowBox]
heteroList = [SB (), SB 5, SB True]
```

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

Summary of heterogeneous list examples (1)

An existential datatype

data T = forall a. MkT a

This defines a polymorphic constructor,

or a family of constructors for **T**

MkT :: forall a. (a -> T)

Pattern matching on our existential constructor

foo (MkT x) = ... -- what is the type of x?

Constructing the hetereogeneous list

heteroList = [MkT 5, MkT (), MkT True, MkT map]

data ShowBox = forall s. Show s => SB s

heteroList :: [ShowBox]

heteroList = [SB (), SB 5, SB True]

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

Summary of heterogeneous list examples (2)

A new existential data type, with a class constraint

```
data T' = forall a. Show a => MkT' a
data T = forall a. MkT a
```

Using our new heterogenous setup

```
heteroList' = [MkT' 5, MkT' (), MkT' True, MkT' "Sartre"]
```

```
main = mapM_ (\(MkT' x) -> print x) heteroList'
 {- prints:
 5
 ()
 True
 "Sartre"
 -}
```

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

runST

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

State and ST monads

the **ST** monad is essentially

a more powerful version of the **State** monad:

It was originally written to provide Haskell with **IO**.

IO is basically just a **State** monad

with an environment of all the information about the real world.

In fact, inside GHC at least, **ST** is used,

and the environment is a **type** called **RealWorld**.

To get out of the **State** / **ST** monad,

use **runState** / **runST**

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

runST – rank-2 polymorphism

runST :: forall a. (forall s. ST s a) -> a

This is actually an example of **rank-2 polymorphism**

a **forall** appearing within the left-hand side of (->)

cannot be moved up, and therefore forms another level or **rank**

therefore, there are **2 levels** of universal quantification.

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

runST – initial state

runST :: forall a. (forall s. ST s a) -> a

there is **no parameter** for the **initial state** … **s**

Indeed, **ST** uses a different notion of *state* to **State**;

State allows you to **get** and **put** the *current state*,

ST provides an **interface** to **references**

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

runST – reference interfaces

To <u>create</u> **references** of the type **STRef**

newSTRef :: a -> ST s (STRef s a)

To provide an **initial value**

readSTRef :: STRef s a -> ST s a

To <u>manipulate</u> them.

writeSTRef :: STRef s a -> a -> ST s ()

runST :: forall a. (forall s. ST s a) -> a

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

runST – a mapping

the **internal environment** of a **ST** computation

is not one specific **value**,

but a **mapping** from **references** to **values**. **… (STRef s a)**

newSTRef :: a -> ST s (STRef s a)

No need to provide an **initial state** to **runST**,

as the **initial state** is just the **empty mapping** **… ()**

containing **no references**.

runST :: forall a. (forall s. ST s a) -> a

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

runST – no specific references

It is not allowed

to create a **reference** in one **ST** computation,

then to use the created **reference** in another **ST** computation.

for reasons of thread-safety

because no **ST** computation should be allowed

to assume that the initial internal environment

contains any specific references.

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

runST

```
runST :: forall a. (forall s. ST s a) -> a
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
```

Example: Bad ST code

```
let v = runST (newSTRef True)   … one ST computation
in runST (readSTRef v)          … another ST computation
```

Example: Briefer bad ST code

```
... runST (newSTRef True) ...
```

```
newSTRef True :: ST s (STRef s a)
runST (newSTRef True) :: STRef s a
v :: STRef s a

readSTRef v :: ST s a
runST (readSTRef v) :: a
```

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

runST

Example: Bad ST code

```
let v = runST (newSTRef True)
in runST (readSTRef v)
```

runST :: forall a. (forall s. ST s a) -> a

the **rank-2 polymorphism** in **runST**'s type

to constrain the scope of the **type variable s**

to be within the first parameter (the left hand side of ->)

if the **type variable s** appears in the first parameter

it cannot also appear in the second.

(the right hand side of ->)

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

runST

Example: Briefer bad ST code

... runST (newSTRef True) ...

Example: The compiler's typechecking stage

newSTRef True :: forall s. ST s (STRef s Bool)

runST :: forall a. (forall s. ST s a) -> a

runST (newSTRef True) ::

(forall s. ST s (STRef s Bool)) -> STRef s Bool

runST :: forall a. (forall s. ST s a) -> a

newSTRef :: a -> ST s (STRef s a)

readSTRef :: STRef s a -> ST s a

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

forall

The importance of the forall in the first bracket is
that we can change the name of the s.

runST (newSTRef True) ::

(forall s. ST s (STRef s Bool)) -> STRef s Bool

Example: A type mismatch!

(forall s'. ST s' (STRef s' Bool)) -> STRef s Bool

This is similar to ∀ x . x > 5 is precisely the same as ∀ y . y > 5
giving the variable a different label.

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

forall

Example: A type mismatch!

```
(forall s'. ST s' (STRef s' Bool)) -> STRef s Bool
```

Notice that as the **forall** does <u>not</u> scope over

the return type of **runST**, **STRef s Bool**

we don't rename the s there as well.

But suddenly, we've got a **type mismatch**!

The result type of the ST computation in the **first parameter**

must match the **result type** of **runST**, but now it doesn't!

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

forall

(forall s'. ST s' (STRef s' Bool)) -> STRef s Bool

The key feature of the **existential** is that
it allows the compiler to generalise
the **type** of the **state** in the **first parameter**,
and so the **result type** cannot depend on it.

This neatly sidesteps our **dependence problem**s,
'**compartmentalises**' each call to **runST**
into its own little heap,
with **references** not being able
to be shared between different **calls**.

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

forall – quantifier (1)

- quantifier in predicate calculus
- type quantifier polymorphic types
- to encode a type in **type isomorphism**

Isomorphism

from . to = id

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

forall – quantifier (2) type isomorphism

the class of **isomorphic types**, i.e. those which
can be **cast** to each other without loss of information.

type isomorphism is an **equivalence relation**
(**reflexive**, **symmetric**, **transitive**),
but due to the limitations of the type system,
only **reflexivity** is implemented for all types

Isomorphism

from . to = id

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

forall – quantifier (3)

```
foo :: (forall a. a -> a) -> (Char, Bool)
bar :: forall a. ((a -> a) -> (Char, Bool))
```

To call **bar**, any **type a** can be chosen,

and it is possible to pass a **function** from **type a** to **type a**.

the **function (+1)** or the **function reverse**.

the **forall** is considered to be as saying

"I get to pick the type now". (**instantiating**.)

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

forall – quantifier (4)

```
foo :: (forall a. a -> a) -> (Char, Bool)
bar :: forall a. ((a -> a) -> (Char, Bool))
```

The restrictions on calling **foo** are much more stringent:
the argument to **foo** must be a **polymorphic function**.

With that **type**, the only functions that can be passed to **foo**
are **id** or a **function** that always **diverges** or **errors**, like **undefined**.

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

forall – quantifier (5)

foo :: (forall a. a -> a) -> (Char, Bool)

bar :: forall a. ((a -> a) -> (Char, Bool))

The reason is that with **foo**, the **forall** is to the **left of the arrow**,
so as the **caller** of **foo** I don't get to pick what **a** is
—rather it's the **implementation** of **foo** that gets to pick what **a** is.

Because **forall** is to the **left of the arrow**,
rather than **above** the **arrow** as in **bar**,
the **instantiation** takes place in the **body** of the **function**
rather than at the **call** site.

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

forall – quantifier (6) above, below, left

Jargon "**above**", "**below**", "**to the left of**".

nothing to do with the *textual ways* types are written

everything to do with **abstract-syntax trees**.

In the **abstract syntax**,

- a **forall** takes the **name** of a **type variable**,
  and then there is a **full type** "**below**" the **forall**.
- an **arrow** takes **two types** (**argument** and **result type**)
  and forms a **new type** (the **function type**).
- the **argument type** is "**to the left of**" the **arrow**;
- it is the **arrow**'s **left child** in the **abstract-syntax tree**.

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

forall – quantifier (7)

forall a . [a] -> [a],

the **forall** is **above the arrow**;

what's to **the left of the arrow** is **[a]**.

```
forall n f e x . (forall e x . n e x -> f -> Fact x f)
                     -> Block n e x -> f -> Fact x f
```

(forall e x . n e x -> f -> Fact x f)

the type in parentheses would be called

"a **forall** to the **left of an arrow**".

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

foo :: a -> a (1)

foo :: a -> a

given this type signature, there is only one function

that can satisfy this type and

the identity function **id**.

foo 5 = 6

foo True = False

they both satisfy the above type signature,

then why do Haskell folks claim

that it is **id** alone which satisfies the type signature?

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

foo :: a -> a (2)

That is because there is an implicit forall hidden in the type signature.

id :: forall a. a -> a

Constraints liberate, liberties constrain

A constraint at the **type level**,
becomes a liberty at the **term level**

A liberty at the **type level**,
becomes a constraint at the **term level**

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

foo :: a -> a (3)

A **constraint** at the **type** level..

So putting a constraint on our type signature

foo :: (Num a) => a -> a

becomes a **liberty** at the term level gives us

the liberty or flexibility to write all of these

foo 5 = 6

foo 4 = 2

foo 7 = 9

...

Same can be observed by constraining a with any other **typeclass** etc

A constraint at the **type level**,

becomes a liberty at the **term level**

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

foo :: a -> a (4)

foo :: (Num a) => a -> a translates to

∃a , st a -> a, ∀a ∈ Num

existential quantification

which translates to there exists some instances of **a**

for which a function of **a -> a**

and those instances all belong to the set of **Numbers**.

adding a **constraint** (**a** should belong to the set of **Nnumbers**),

liberates the **term** level to have multiple possible implementations.

A constraint at the **type level**,

becomes a liberty at the **term level**

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

foo :: a -> a (5)

the explanation of **forall**:

So now let us **liberate** the the **function** at the **type** level:

foo :: forall a. a -> a translates to:

∀a , a -> a

the **implementation** of this type signature

should be such that it is **a -> a** for all circumstances.

A liberty at the **type level**, becomes

a constraint at the **term level**

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

foo :: a -> a (6)

So now this starts **constraining** us at the **term** level.

We can no longer write

foo 5 = 7

because this **implementation** would not satisfy

when a **Bool** type value is passed to **foo**

this is because

under all circumstances **∀a , a -> a**

it should return something of the similar type.

a can be a **Char** or a **[Char]** or a custom datatype.

A liberty at the **type level**, becomes

a constraint at the **term level**

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

foo :: a -> a (7)

∀a , a -> a	the **liberty** at the **type** level
foo 5 = 7	a constraint at the **term** level
	(impossible implementation)

this **liberty** at the **type** level is what is known

as **Universal Quantification**

the only **function** which can satisfy **foo :: forall a. a -> a**

foo a = a	the **identity** function

A liberty at the **type level**, becomes

a constraint at the **term level**

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

foo :: a -> a (8)

Runar Bjarnason titled "Constraints Liberate, Liberties Constrain".

CONSTRAINTS LIBERATE,

LIBERTIES CONSTRAIN

Its very important to digest and believe this statement

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

RunST (1)

runST :: forall a. (forall s. ST s a) -> a

runST has to be able to produce a **value** of **type a**,
no matter what **type** we give as **a**.

runST uses an **argument** of **type (forall s. ST s a)**
which certainly must somehow produce the **a**.

runST must be able to produce a **value** of **type a**
no matter what **type** the **implementation** of **runST**
decides to give as **s**.

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

RunST (2)

runST :: forall a. (forall s. ST s a) -> a

the benefit is that this puts a constraint on the caller of **runST**
in that the **type a** cannot involve the **type s** at all.
you can't pass it a value of type **ST s [s]**, for example.

the implementation of **runST** is free
to perform **mutation** with the value of **type s**.

The **type** guarantees that this **mutation** is
local to the implementation of **runST**.

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

RunST : rank-2 polymorphic type

runST :: forall a. (forall s. ST s a) -> a

The **type** of **runST** is an example of

a **rank-2 polymorphic type**

because the **type** of its **argument**

contains a **forall** quantifier.

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

Existential Quantifiation

```
-- test.hs
{-# LANGUAGE ExistentialQuantification #-}
data EQList = forall a. EQList [a]
eqListLen :: EQList -> Int
eqListLen (EQList x) = length x

ghci> :l test.hs
ghci> eqListLen $ EQList ["Hello", "World"]
2
```

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

Existential Quantifiation

ghci> **:set -XRankNTypes**

ghci> **length (["Hello", "World"] :: forall a. [a])**

Couldnt match expected type 'a' against inferred type '[Char]'

...

With Rank-N-Types, forall a meant that your expression

must fit all possible as. For example:

ghci> **length ([] :: forall a. [a])**

0

https://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do

References

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

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