Monad P3 : Polymorphic Types (1C)

Young Won Lim
12/13/20

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

Overloading

The **literals** **1**, **2**, etc. are often used to represent

both fixed and arbitrary precision integers.

Numeric operators such as + are often defined to work

on many different kinds of numbers.

the **equality operator** (**==** in Haskell) usually works on

numbers and many other (but not all) types.

the **overloaded behaviors** are

different for each type

in fact sometimes **undefined**, or **error**

type classes provide a structured way to control

ad hoc polymorphism, or **overloading**.

In the **parametric polymorphism**

the type truly does **not matter**

(Eq a) =>

Type class

Ad hoc polymorphism

https://www.haskell.org/tutorial/classes.html

Quantification

parametric polymorphism is useful in
defining families of types
by **universally quantifying** over all types.

Sometimes, however, it is necessary
to quantify over some smaller set of types,
eg. those types whose elements can be compared for equality.
ad hoc polymorphism

```
elem :: a -> [a] -> Bool
```

```
elem :: (Eq a) => a -> [a] -> Bool
```

https://www.haskell.org/tutorial/classes.html

Type class and parametric polymorphism

type classes can be seen as providing a structured way
to quantify over a constrained set of types

the **parametric polymorphism** can be viewed
as a kind of **overloading** too!

parametric polymorphism
an **overloading** occurs implicitly over all types

elem :: a -> [a] -> Bool

ad hoc polymorphism
a **type class** for a constrained set of types

elem :: (Eq a) => a -> [a] -> Bool

https://www.haskell.org/tutorial/classes.html

Parametric polymorphism (1) definition

Parametric polymorphism refers to
when the **type** of a **value** contains
one or more (**unconstrained**) **type variables**,
so that the **value** may adopt any type
that results from substituting those variables with **concrete types**.

elem :: a -> [a] -> Bool

https://wiki.haskell.org/Polymorphism

Parametric polymorphism (2) unconstrained type variable

In Haskell, this means any type in which a **type variable**,

denoted by a name in a type

beginning with a **lowercase letter**,

appears **without constraints**

(i.e. does not appear to the left of a **=>**).

In **Java** and some similar languages,

generics (roughly speaking) fill this role.

elem :: a -> [a] -> Bool

https://wiki.haskell.org/Polymorphism

Parametric polymorphism (3) examples

For example, the function **id :: a -> a** contains
an **unconstrained type variable a** in its type,
and so can be used in a context requiring

Char -> Char or

Integer -> Integer or

(Bool -> Maybe Bool) -> (Bool -> Maybe Bool) or

any of a literally infinite list of other possibilities.

Likewise, the empty **list [] :: [a]** belongs to every list type,

and the polymorphic function **map :: (a -> b) -> [a] -> [b]**
may operate on any function type.

https://wiki.haskell.org/Polymorphism

Parametric polymorphism (4) multiple appearance

Note, however, that if a single **type variable** appears multiple times,

it must take the same type everywhere it appears,

so e.g. the result type of **id** must be the same as the argument type,

and the input and output types of the function

given to **map** must match up with the list types.

id :: a -> a

map :: (a -> b) -> [a] -> [b]

https://wiki.haskell.org/Polymorphism

Parametric polymorphism (5) parametricity

Since a parametrically polymorphic value does not "know" anything about the **unconstrained type variables**, it must behave the same regardless of its type.

This is a somewhat limiting but extremely useful property known as **parametricity**

```
id :: a -> a
map :: (a -> b) -> [a] -> [b]
```

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (1)

Ad-hoc polymorphism refers to
when a **value** is able to adopt any one of several **types**
because it, or a value it uses, has been given
a separate definition for each of those **types**.

the **+ operator** essentially does something entirely different
when applied to floating-point values
as compared to when applied to integers

elem :: (Eq a) => a -> [a] -> Bool

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (2)

in languages like C, **polymorphism** is restricted to only *built-in* **functions** and **types**.

Other languages like C++ allow programmers to provide their own **overloading**, supplying **multiple definitions** of a **single function**, to be disambiguated by the **types** of the **arguments**

In Haskell, this is achieved via the system of **type classes** and **class instances**.

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (3)

Despite the similarity of the name,
Haskell's **type classes** are quite different from
the **classes** of most object-oriented languages.

They have more in common with **interfaces**,
in that they specify a series of **methods** or **values**
by their **type signature**,
to be implemented by an **instance declaration**.

```
class Eq a where
  (==)        :: a -> a -> Bool

instance Eq Integer where
  x == y     =  x `integerEq` y

instance Eq Float where
  x == y     =  x `floatEq` y
```

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (4)

So, for example, if **my type** can be compared for **equality** (most types can, but some, particularly function types, cannot) then I can give **an instance declaration** of the **Eq class**

All I have to do is specify the behaviour of the **== operator** on **my type**, and I gain the ability to use all sorts of functions defined using **== operator**, e.g. checking if a value of **my type** is present in a list, or looking up a corresponding value in a list of pairs.

```
class Eq a where
  (==)        :: a -> a -> Bool

instance Eq Integer where
  x == y    =  x `integerEq` y

instance Eq Float where
  x == y    =  x `floatEq` y
```

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (5)

Unlike the **overloading** in some languages,

overloading in Haskell is not limited to **functions**

– **minBound** is an example of an **overloaded value**,

as a **Char**, it will have value **'\NUL**'

as an **Int** it might be **-2147483648**

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (6)

Haskell even allows **class instances** to be defined for **types** which are themselves **polymorphic** (either **ad-hoc** or **parametrically**).

So for example, an **instance** can be defined of **Eq** that says "if **a** has an **equality operation**, then **[a]** has one".

Then, of course, **[[a]]** will automatically also have an instance, and so **complex compound types** can have **instances** built for them out of the instances of their components.

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (7)

```
data List a = Nil | Cons a (List a)

instance Eq a => Eq (List a) where
  (Cons a b) == (Cons c d)        =      (a == c) && (b == d)
  Nil == Nil                      =      True
  _ == _                          =      False
```

https://stackoverflow.com/questions/30520219/how-to-define-eq-instance-of-list-without-gadts-or-datatype-contexts

Ad hoc polymorphism (8)

You can recognise the presence of **ad-hoc polymorphism**
by looking for **constrained type variables**:
that is, variables that appear to the left of **=>**,
like in **elem :: (Eq a) => a -> [a] -> Bool**.

Note that **lookup :: (Eq a) => a -> [(a,b)] -> Maybe b**
exhibits both **parametric** (in **b**) and **ad-hoc** (in **a**) **polymorphism**.

https://wiki.haskell.org/Polymorphism

Parametric and ad hoc polymorphism

Parametric polymorphism	ad hoc polymorphism
Type variables (a, b, etc)	Type calsses (Eq, Num, etc)
Universal	Existential?
Compile time	Runtime (also)
C++ templates Java generics	Classical (ordinary OO)

http://sm-haskell-users-group.github.io/pdfs/Ben%20Deane%20-%20Parametric%20Polymorphism.pdf

Polymorphic data types and functions

```
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)
data Either a b = Left a | Right b


reverse :: [a] -> [a]
fst :: (a,b) -> a
id :: a -> a
```

http://sm-haskell-users-group.github.io/pdfs/Ben%20Deane%20-%20Parametric%20Polymorphism.pdf

Polymorphic Types

types that are universally quantified in some way over all types.

polymorphic type expressions essentially describe families of types.

For example, **(forall a) [a]** is the family of types
consisting of, for every **type a**, the **type of lists of a**.

- lists of integers (e.g. **[1,2,3]**),
- lists of characters (**['a','b','c']**),
- even lists of lists of integers, etc.,

(Note, however, that [2,'b'] is not a valid example,
since there is *no single type* that contains both 2 and 'b'.)

https://www.haskell.org/tutorial/goodies.html

Type variables – universally quantified

Identifiers such as **a** above are called **type variables**,
and are uncapitalized to distinguish them
from specific types such as **Int**.

since Haskell has only universally quantified **types**,
there is no need to explicitly write out the symbol
for **universal quantification**,
and thus we simply write **[a]** in the example above.

In other words, all **type variables** are implicitly universally quantified

https://www.haskell.org/tutorial/goodies.html

List

Lists are a commonly used data structure in functional languages, and are a good tool for explaining the principles of polymorphism.

The list **[1,2,3]** in Haskell is actually shorthand for

the list **1:(2:(3:[]))**,

where **[]** is the **empty list** and

: is the **infix operator**

that adds its first argument to the front

of its second argument (a list).

Since **:** is right associative, we can also write this list as

1:2:3:[].

https://www.haskell.org/tutorial/goodies.html

Polymorphic function example

```
length          :: [a] -> Integer
length []        =  0
length (x:xs)    =  1 + length xs

    length [1,2,3]          =>   3
    length ['a','b','c']    =>   3
    length [[1],[2],[3]]    =>   3
```

an example of a polymorphic function.

It can be applied to a list containing elements of any type,

for example **[Integer]**, **[Char]**, or **[[Integer]]**.

https://www.haskell.org/tutorial/goodies.html

Patterns in functions

```
length          :: [a] -> Integer
length []        =  0
length (x:xs)    =  1 + length xs
```

The left-hand sides of the equations contain **patterns** such as **[]** and **x:xs**.

In a **function application** these **patterns** are matched against **actual parameters** in a fairly intuitive way

https://www.haskell.org/tutorial/goodies.html

Matching patterns

```
length          :: [a] -> Integer
length []        =  0
length (x:xs)    =  1 + length xs
```

[] only matches the empty list,

x:xs will successfully match any list with at least one element,

binding **x** to the first element and **xs** to the rest of the list

If the match succeeds,

the right-hand side is evaluated

and returned as the result of the application.

If it fails, the next equation is tried,

and if all equations fail, an error results.

https://www.haskell.org/tutorial/goodies.html

Not all possible cases – runtime errors

Function **head** returns the first element of a list,

function **tail** returns all but the first.

```
head          :: [a] -> a
head (x:xs)  =  x

tail            :: [a] -> [a]
tail (x:xs)     =  xs
```

Unlike length, these functions are not defined

for all possible values of their argument.

A **runtime error** occurs when these functions

are applied to an empty list.

https://www.haskell.org/tutorial/goodies.html

General types

With polymorphic types, we find that
some types are in a sense strictly more general than others
in the sense that the set of values they define is larger.

the type **[a]** is more general than **[Char]**.

type **[Char]** can be derived from **[a]**
by a suitable substitution for **a**.

https://www.haskell.org/tutorial/goodies.html

Principal type

With regard to this **generalization ordering**,
Haskell's type system possesses two important properties:

1. every **well-typed expression** is guaranteed to have a **unique principal type** (explained below),
2. the **principal type** can be inferred automatically.

In comparison to a monomorphically typed language such as C,
the reader will find that polymorphism improves expressiveness,
and **type inference** lessens the burden of types on the programmer.

https://www.haskell.org/tutorial/goodies.html

Unique principal types

An **expression's** or **function's** **principal type** is

the **least general type** that, intuitively,

"contains all instances of the expression".

For example, the principal type of **head** is **[a]->a**;

[b]->a, **a->a**, or even **a** are correct types, but too general,

whereas something like **[Integer]->Integer** is too specific.

The existence of **unique** **principal types** is

the hallmark feature of the **Hindley-Milner type system**,

which forms the basis of the type systems of Haskell

https://www.haskell.org/tutorial/goodies.html

Explicitly Quantifying Type Variables

to explicitly bring fresh **type variables** into **scope**.

Explicitly quantifying the **type variables**

map :: forall a b. (a -> b) -> [a] -> [b]

for any combination of types **a** and **b**

choose **a** = **Int** and **b** = **String**

then it's valid to say that map has the type

(Int -> String) -> [Int] -> [String]

Here we are **instantiating** the general type of **map**

to a more specific type.

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

Implicit **forall**

any introduction of a **lowercase type parameter**

implicitly begins with a **forall** keyword,

Example: Two equivalent type statements

id :: a -> a

id :: forall a . a -> a

We can apply additional **constraints**

on the quantified **type variables**

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

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:

Scoped Type Variables

specify types for code inside **where** clauses

RankN Types / Rank2 Types,

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.

Existential Quantification

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

Extensions for forall

```
foob :: forall a b. (b -> b) -> b -> (a -> b) -> Maybe a -> b
foob postProcess onNothin onJust mval =
    postProcess val
    where
      val :: b
      val = maybe onNothin onJust mval
```

This code doesn't compile (syntax error) in plain Haskell 98.

It requires an **extension** to support the forall keyword.

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

Scoped Type Variables

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

Scoped Type Variables

ScopedTypeVariables helps one

to specify types for code inside **where** clauses.

It makes the **b** in **val :: b** the same one as

the **b** in **foob :: forall a b. (b -> b) -> b -> (a -> b) -> Maybe a -> b**

A confusing point:

when you omit the **forall** from a type it is actually still implicitly there.

(normally these languages omit the **forall** from polymorphic types).

This claim is correct, but it refers to the other uses of **forall**,

and not to the ScopedTypeVariables use.

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

Scoped Type Variables

```
{-# LANGUAGE ScopedTypeVariables #-}
...
mkpair1 :: forall a b. a -> b -> (a,b)
mkpair1 aa bb = (ida aa, bb)
    where
      ida :: a -> a -- This refers to a in the function's type signature
      ida = id
```

https://wiki.haskell.org/Scoped_type_variables

Scoped Type Variables

```
mkpair2 :: forall a b. a -> b -> (a,b)
mkpair2 aa bb = (ida aa, bb)
    where
      ida :: b -> b -- Illegal, because refers to b in type signature
      ida = id

mkpair3 :: a -> b -> (a,b)
mkpair3 aa bb = (ida aa, bb)
    where
      ida :: b -> b -- Legal, because b is now a free variable
      ida = id
```

forall a. a -> (forall b. b -> (a,b))

https://wiki.haskell.org/Scoped_type_variables

Scoped Type Variables

Scoped type variables make it possible
to specify the particular type of a **function** in situations
where it is not otherwise possible,
which can in turn help avoid problems
with the **Monomorphism restriction**.

https://wiki.haskell.org/Scoped_type_variables

Scoped Type Variables

ScopedTypeVariables breaks GHC's usual rule

that explicit **forall** is optional and

doesn't affect **semantics**.

the explicit **forall** is required

If omitted, usually the program will not compile;

in a few cases it will compile

but the functions get a different **signature**.

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#ghc-flag--XScopedTypeVariables

Scoped Type Variables

to trigger those forms of ScopedTypeVariables,

the **forall** must appear against

the **top-level signature** (or **outer expression**)

but not against **nested signatures**

referring to the same **type variables**.

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#ghc-flag--XScopedTypeVariables

Scoped Type Variables

```
f :: forall a. [a] -> [a]
f xs = ys ++ ys
    where
      ys :: [a]
      ys = reverse xs
```

the **explicit** forall in the **type signature f**

brings the **type variable** a into scope,

The **type variables** a bound by a forall **scope**

over the entire definition of **f**

the type variable **a scopes** over the whole **definition** of **f**,

including over the type signature for **ys**.

In Haskell 98 it is not possible to declare a type for ys;

a major benefit of scoped type variables is that it becomes possible to do so.

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#ghc-flag--XScopedTypeVariables

Scoped Type Variables

```
f :: [a] -> [a]
f (xs :: [aa]) = xs ++ ys
  where
    ys :: [aa]
    ys = reverse xs
```

```
f :: [a] -> [a]
f (xs :: [a]) = xs ++ ys
  where
    ys :: [a]
    ys = reverse xs
```

without the explicit **forall** form, type variable **a** from **f**'s signature

is not scoped over **f**'s equation(s).

type variable **aa** bound by the **pattern signature**

is scoped over the **right-hand side** of **f**'s equation.

therefore there is no need to use a distinct **type variable a**

using **a** would be equivalent

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#ghc-flag--XScopedTypeVariables

Scoped Type

Let's start with that

mayb :: b -> (a -> b) -> Maybe a -> b

is equivalent to

mayb :: forall a b. b -> (a -> b) -> Maybe a -> b

except for when ScopedTypeVariables is enabled.

This means that it works for every **a** and **b**.

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

Rank N Types / Rank 2 Types

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

Rank-N types (1)

Normal Haskell '98 types are considered **Rank-1 types**.

a -> b -> a

implies that the **type variables** are **universally quantified**

forall a b. a -> b -> a

forall can be **floated** out of the **right-hand side** of **->**

forall a. a -> (forall b. b -> a)

is also a **Rank-1 type** because it is equivalent to the previous signature.

https://wiki.haskell.org/Rank-N_types

Rank-N types (2)

forall can be **floated**

(xxx) -> (forall o. ooo)

out of the **right-hand side** of **->**

forall o. ((xxx) -> ooo)

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

(forall x. xxx) -> ooo

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

forall x. ((xxx) -> ooo)) not equivalent

forall o. ((forall x. xxx)) -> ooo)

https://wiki.haskell.org/Rank-N_types

Rank-N types (3)

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.

```
forall o. ( ( forall x. xxx ) ) -> ooo )
```

Rank-2

https://wiki.haskell.org/Rank-N_types

Rank-N types (4)

(forall a. a -> a) -> (forall b. b -> b)

is a **Rank-2** type

the **forall b** can be moved to the start

forall b. (forall a. a -> a) -> (b -> b) **(O)**

but the **forall a** cannot.

forall a b. (a -> a) -> (b -> b)) – not equivalent **(X)**

there are two **levels** of universal quantification.

https://wiki.haskell.org/Rank-N_types

Rank-N types (5)

Rank-N type reconstruction is undecidable in general,

and some **explicit type annotations** are required in their presence.

Rank-2 or **Rank-N types** may be specifically enabled

by the language extensions

{-# LANGUAGE Rank2Types #-} or

{-# LANGUAGE RankNTypes #-}.

https://wiki.haskell.org/Rank-N_types

Polymorphic arguments of Rank-N

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

The type of **foo** above is of **rank 2**.
An ordinary **polymorphic** type, like that of **bar**, is **rank-1**,
but it becomes **rank-2** if the **types** of **arguments** are
required to be **polymorphic**, with their own **forall** quantifier.

And if a function takes **rank-2 arguments**
then its type is **rank-3**, and so on.
In general, a **type** that takes **polymorphic arguments** of **rank n**
has **rank n + 1**.

(forall a. a -> a)

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

Rank-N example **A** (1)

In the normal case

(forall n. Num n => (n -> n) -> (Int, Double))

we choose an **n** first and

then provide a **function**. **(n -> n)**

So we could pass in a **function** of type **(n -> n)**

Int -> Int,

Double -> Double,

Rational -> Rational

and so on.

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

Rank-N example **A** (2)

the previous case is normal (**Rank-1**)

because that's how **type variables** work by default.

If you don't have a **forall** at all, your **type signature** is equivalent to having **forall** at the very beginning.

prenex form

Num n => (n -> n) -> (Int, Double)

is implicitly the same as

forall n. Num n => (n -> n) -> (Int, Double).

https://stackoverflow.com/questions/33446759/understanding-haskells-rankntypes

Rank-N example **A** (3)

In the **Rank-2** case

((forall n. Num n => n -> n) -> (Int, Double))

we have to provide the **function** before we know **n**.

the **type** of a **function** that works **for any n**

It's exactly **forall n. Num n => n -> n**.

https://stackoverflow.com/questions/33446759/understanding-haskells-rankntypes

Rank-N example **A** (4)

In the **rank-N** case **f** has to be a **polymorphic** function

which is valid for all numeric types **n**.

In the **rank-1** case **f** only has to be defined

for a single numeric type **n**

(+1) for Num n

```
{-# LANGUAGE RankNTypes #-}

rankN :: (forall n. Num n => n -> n) -> (Int, Double)
rankN = undefined

rank1 :: forall n. Num n => (n -> n) -> (Int, Double)
rank1 = undefined
```

https://stackoverflow.com/questions/33446759/understanding-haskells-rankntypes

Rank-N example **A** (5)

rankN :: (forall n. Num n => n -> n) -> (Int, Double)

rankN takes a parameter **f :: Num n => n -> n** and returns **(Int, Double)**, where for any numeric type **n**, **f** can take an **n** and return an **n**

rank1 :: forall n. Num n => (n -> n) -> (Int, Double)

for any numeric type **n**, **rank1** takes an argument **f :: n -> n** and returns an **(Int, Double)**

by default all foralls are implicitly placed at the outer-most position (resulting in a rank-1 type).

https://stackoverflow.com/questions/33446759/understanding-haskells-rankntypes

Rank-N example **A** (6)

```
main = print $ rankN (+1)

rankN :: (forall n. Num n => n -> n) -> (Int, Double)
```

requires a function from **n** to **n** for **some** Num n;

```
rankN f = (f 1, f 1.0)                    (2, 2.0)
```

Int -> Int (+1) — Double -> Double (+1)

```
rank1 :: forall n. Num n => (n -> n) -> (Int, Double)
```

requires a function from n to n for **every** Num n.

the example code given
because the **function f**
that's passed in
is applied to two different types:
an **Int** and a **Double**.
So it has to work for both of them.

https://stackoverflow.com/questions/33446759/understanding-haskells-rankntypes

Rank-N example **A** (7)

```
foo :: Int -> Int                -- monomorphic
foo n = n + 1

test1 = rank1 foo                -- OK
test2 = rankN foo                -- does not type check
test3 = rankN (+1)               -- OK since (+1) is polymorphic
                                 --     (+1) :: Int -> Int
                                 --     (+1) :: Double -> Double

rank1 :: forall n. Num n => (n -> n) -> (Int, Double)
rankN :: (forall n. Num n => n -> n) -> (Int, Double)
```

https://stackoverflow.com/questions/33446759/understanding-haskells-rankntypes

Rank-N example **B** (1)

```
bar :: (forall n. Num n => n -> n) -> (Int, Double) -> (Int, Double)
bar f (i,d) = (f i, f d)
```

That is, we apply **f** to both an **Int** and a **Double**.

without using RankNTypes it won't type check:

```
{-# LANGUAGE RankNTypes #-}
```

https://stackoverflow.com/questions/33446759/understanding-haskells-rankntypes

Rank-N example **B** (2)

```
bar :: (forall n. Num n => n -> n) -> (Int, Double) -> (Int, Double)
bar f (i,d) = (f i, f d)
```

none of the following signature work

even with using RankNTypes it won't type check:

```
bar' :: Num n =>   (n -> n)  -> (Int, Double) -> (Int, Double)
bar' f (i,d) = (f i, f d)

bar' ::              (Int -> Int)  -> (Int, Double) -> (Int, Double)
bar' :: (Double -> Double)  -> (Int, Double) -> (Int, Double)
```

https://stackoverflow.com/questions/33446759/understanding-haskells-rankntypes

Rank-N example **C** (1)

```
ghci> let putInList x = [x]
ghci> liftTup putInList (5, "Blah")
([5], ["Blah"])
```

the type of this liftTup?

```
liftTup :: (forall x. x -> f x) -> (a, b) -> (f a, f b)
```

(5, "Blah")

([5], ["Blah"])

Num -> [Num]

[Char] -> [[Char]]

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

Rank-N example **C** (2) – code 1

```
ghci> let liftTup liftFunc (a, b) = (liftFunc a, liftFunc b)
ghci> liftTup (\x -> [x]) (5, "Hello")
    No instance for (Num [Char])
    ...

ghci> :t liftTup
liftTup :: (t -> t1) -> (t, t) -> (t1, t1)
              (Num -> Num) -> (Num, Num) -> ([Num], [Num])
```

GHC infer that the tuple must contain two of the same type

5 -> [5] – Num -> [Num]

”Hello” -> [”Hello”] – [Char] -> [[Char]]

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

Rank-N example **C** (3) – code 2

```
– test.hs
liftTup :: (x -> f x) -> (a, b) -> (f a, f b)
liftTup liftFunc (t, v) = (liftFunc t, liftFunc v)

ghci> :l test.hs
    Couldnt match expected type 'x' against inferred type 'b'
    ...
```

so here GHC doesn't let us apply **liftFunc** on **v**

because **v :: b** and **liftFunc** wants an **x**.

Need a function that accepts any possible **x**

t :: a ↔ x

v :: b ↔ x

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

Rank-N example **C** (4) – code 3

```
{-# LANGUAGE RankNTypes #-}
liftTup :: (forall x. x -> f x) -> (a, b) -> (f a, f b)
liftTup liftFunc (t, v) = (liftFunc t, liftFunc v)
```

So it's not **liftTup** that works for all x,

it's the function **liftFunc** that it gets that works for all x.

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

Existential Quantification

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

Existential quantification

```
-- 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
                          ["Hello", "World"]
                          [ [Char] ]                [a]
```

the value contained
can be of **any** suitable type,

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

Existential quantification

With **Rank-N-Types**,

forall a means that

your expression must fit all possible **a**'s.

```
ghci> :set -XRankNTypes
ghci> length (["Hello", "World"] :: forall a. [a])
   Couldnt match expected type 'a' against inferred type '[Char]'
   ...
ghci> length ([] :: forall a. [a])
0
```

An **empty lis**t does work as a **list of any type**.

["Hello", "World"]
[[Char], [Char]]

~~[Int, Int]~~

~~[a , a]~~

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

Existential quantification

```
{-# LANGUAGE ExistentialQuantification #-}
data EQList = forall a. EQList [a]
```

with Existential-Quantification,

foralls in **data** definitions mean that,

the value contained can be of **any** suitable type,

not that it must be of **all** suitable types.

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

Existential quantification

Existential quantification actually works

a lot like **universal quantification**.

```
data Univ a = Univ a
data Exis    = forall a. Exis a

toUniv :: a -> Univ a
toUniv = Univ

toExis :: a -> Exis
toExis = Exis
```

https://stackoverflow.com/questions/9259921/haskell-existential-quantification-in-detail

Existential quantification

```
useUniv :: (a -> b) -> Univ a -> b
useUniv f (Univ x) = f x

useExis :: (forall a. a -> b) -> Exis -> b
useExis f (Exis x) = f x
```

The function **useExis** is useless, but it's still valid code.

https://stackoverflow.com/questions/9259921/haskell-existential-quantification-in-detail

Existential quantification

First, note that **toUniv** and **toExis** are nearly the same.

both take a free **type parameter a**

because both **data constructors** are **polymorphic**.

But while **a** appears in the **return type** of **toUniv**

a doesn't appear in the **return type** of **toExis**

when it comes to the kind of **type errors**

you might get from using a **data constructor**,

there's not a big difference

between **existential** and **universal** types.

```
data Univ a = Univ a
data Exis   = forall a. Exis a

toUniv :: a -> Univ a
toUniv = Univ

toExis :: a -> Exis
toExis = Exis
```

https://stackoverflow.com/questions/9259921/haskell-existential-quantification-in-detail

Existential quantification

note the **rank-2 type (forall a. a -> b)** in **useExis**.
This is the big difference in type inference.

The **existential type a** taken
from the **pattern match (Exis x)** x :: a
acts like an extra, hidden **type variable**
passed to the body of the function, f x :: b
and it must not be unified with other types.

In **useUniv**, the **type variable a** is part of the **function type**.
In **useExis**, it's the **existential type** from the data structure **(Exis x)**

```
data Univ a = Univ a
data Exis   = forall a. Exis a

useUniv :: (a -> b) -> Univ a -> b
useUniv f (Univ x) = f x

useExis :: (forall a. a -> b) -> Exis -> b
useExis f (Exis x) = f x

         x :: a
         f x :: b
```

https://stackoverflow.com/questions/9259921/haskell-existential-quantification-in-detail

Existential quantification

To make this clearer, here are some wrong declarations
of the last two functions **useUniv** and **useExis**
where we try to unify **types** that shouldn't be unified.

In both cases, we force the **type** of **x** to be
unified with an unrelated **type variable**.

https://stackoverflow.com/questions/9259921/haskell-existential-quantification-in-detail

Existential quantification

```
useUniv' :: forall a b c. (c -> b) -> Univ a -> b
useUniv' f (Univ x) = f x       -- Error, can't unify 'a' with 'c'
                                -- Variable 'a' is there in the function type

          Univ x  :: Univ a
               x :: a

               c -> b
```

https://stackoverflow.com/questions/9259921/haskell-existential-quantification-in-detail

Existential quantification

```
useExis' :: forall b c. (c -> b) -> Exis -> b
useExis' f (Exis x) = f x       -- Error, can't unify 'a' with 'c'.
                                -- Variable a comes from the pattern Exis x
                                -- via the existential in
                                -- data Exis = forall a. Exis a

          Exis x :: Exis
          Exis x :: forall a. Exis a
               x :: a

               c -> b
```

https://stackoverflow.com/questions/9259921/haskell-existential-quantification-in-detail

Existential quantification

In Haskell, the things being quantified over are **types**

our **logical statements** are also **types**,

and instead of being "**true**" we think about "**can be implemented**".

a **universally quantified type** like **forall a. a -> a** means that,

for any possible **type** "**a**", we can **implement**

a **function** whose type is **a -> a**.

Since **a** is **universally quantified**, we know nothing about it,

and therefore cannot inspect the **argument** in any way.

So **id** is the only possible function of this type

```
id :: forall a. a -> a
id x = x
```

https://stackoverflow.com/questions/14299638/existential-vs-universally-quantified-types-in-haskell

Existential quantification

In Haskell, **universal quantification** is the "**default**"--

any **type variables** in a signature
are implicitly **universally quantified**

thus the type of **id** is normally written as just **a -> a**

this is also known as **parametric polymorphism**,
often just called "**polymorphism**" in Haskell,
and in some other languages (e.g., C#) known as "**generics**".

https://stackoverflow.com/questions/14299638/existential-vs-universally-quantified-types-in-haskell

Existential quantification

An **existentially quantified** type like **exists a. a -> a** means that,

for some particular **type** "**a**", we can **implement** a **function**

whose type is **a -> a**. Any function will do, so I'll pick one:

```
func :: exists a. a -> a
func True = False
func False = True
```

https://stackoverflow.com/questions/14299638/existential-vs-universally-quantified-types-in-haskell

Existential quantification

```
func :: exists a. a -> a
func True = False
func False = True
```

This is of course the "not" function on booleans.

we can't use it as such,
because all we know about the **type** "**a**" is that it exists.

Any information about *which type it might be*
has been discarded,
which means we can't apply **func** to any **values**.

https://stackoverflow.com/questions/14299638/existential-vs-universally-quantified-types-in-haskell

Existential quantification

it's a **function** with the <u>same</u> **type** for its **input** and **output**,
so we could compose it with itself, for example.
Essentially, <u>the only things</u> you can do
with something that has an **existential type**
are the things that is related to
the <u>**non-existential** parts</u> of the type.

Similarly, given something of type **exists a. [a]**
we can find its **length**, or **concatenate** it <u>to itself</u>,
or **drop** some elements, or anything else we can do to any **list**.

https://stackoverflow.com/questions/14299638/existential-vs-universally-quantified-types-in-haskell

Existential quantification

the reason why Haskell doesn't have **existential types** directly

since things with **existentially quantified types**

can only be used with operations

that have **universally quantified types**,

we can write the type **exists a. a** as

forall r. (forall a. a -> r) -> r

--in other words, for all result types **r**

given a function that for all types **a**

takes an argument of **type a**

and returns a value of **type r**,

we can get a result of **type r**.

https://stackoverflow.com/questions/14299638/existential-vs-universally-quantified-types-in-haskell

Existential quantification

forall r. (forall a. a -> r) -> r

note that the overall type is
not **universally quantified** for **a**
--rather, it takes an argument
that itself is **universally quantified** for **a**,
which it can then use
with whatever specific type **a** it chooses.

the equivalence between **an existential type**
and a **universally quantified argument**

https://stackoverflow.com/questions/14299638/existential-vs-universally-quantified-types-in-haskell

References

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

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