Monad P3 : Polymorphic Types (1C)

1

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.

2

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 <u>type</u> truly does <u>**not**</u> **matter**

(Eq a) => Type class Ad hoc polymorphism

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

Quantification

parametric polymorphism is useful in defining <u>families of types</u> by universally quantifying over <u>all types</u> .	elem :: <mark>a</mark> -> [<mark>a</mark>] -> Bool
Sometimes, however, it is necessary to <u>quantify</u> over some <u>smaller set of types</u> , eg. those types whose elements can be compared for equality. ad hoc polymorphism	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 <u>constrained set of types</u>	
the parametric polymorphism can be viewed	
as a kind of overloading too!	
parametric polymorphism	elem :: <mark>a</mark> -> [<mark>a</mark>] -> Be
an overloading occurs implicitly over all types	
ad hoc polymorphism	elem :: (<mark>Eq a</mark>) => a -
a type class for a <u>constrained set of types</u>	

Bool

-> [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 <u>any type</u>
that results from substituting those variables with concrete types.

7

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

Parametric polymorphism (2) unconstrained type variable

In Haskell, this means any type in which a **type variable**, denoted by a <u>name</u> in a type beginning with a **lowercase letter**, appears **without constraints** (i.e. does <u>not</u> appear to the left of a =>). In **Java** and some similar languages, **generics** (roughly speaking) fill this role.

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

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.

Parametric polymorphism (4) multiple appearance

Note, however, that if a single **type variable** appears <u>multiple times</u>, it must take <u>the same type</u> 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]

Parametric polymorphism (5) parametricity

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

This is a somewhat limiting but extremely useful property known as **parametricity** id :: a -> a map :: (a -> b) -> [a] -> [b]

Ad hoc polymorphism (1)

Ad-hoc polymorphism refers to

when a **value** is able to adopt any one of <u>several **types**</u> because it, or a value it uses, has been given a <u>separate definition</u> for each of <u>those **types**</u>.

the **+ operator** essentially does something entirely different when applied to <u>floating-point values</u> as compared to when applied to <u>integers</u> elem :: (Eq a) => a -> [a] -> Bool

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 <u>disambiguated</u> by the **types** of the **arguments**

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

Ad hoc polymorphism (3)

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

They have more in common with **interfaces**, in that they <u>specify</u> a series of **methods** or **values** by their **type signature**, to be <u>implemented</u> 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

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

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

Ad hoc polymorphism (6)

Haskell even allows **class instances** to be <u>defined</u> 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.

Ad hoc polymorphism (7)

data List a = Nil Cons a (List	: a)	
instance Eq a => Eq (List a) w	here	
(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 <u>to the left of =></u>, 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**.

Parametric and ad hoc polymorphism

Type variables	
	Type calsses
(a, b, etc)	(Eq, Num, etc)
Universal	Existential?
Compile time	Runtime (also)
C++ templates	Classical
Java generics	(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 <u>universally quantified</u> in some way <u>over all types</u>. **polymorphic type expressions** essentially describe <u>families of types</u>.

For example, **(forall a) [a]** is the <u>family of types</u> 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 <u>not</u> a valid example, since there is *no single type* that contains both 2 and 'b'.)

Type variables - universally quantified

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

since Haskell has <u>only universally quantified</u> **types**, there is no need to <u>explicitly</u> 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

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 <u>right associative</u>, we can also write this list as **1:2:3:[]**.

Polymorphic function example

length:: [a] -> Integerlength []= 0length (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]].

Patterns in functions

length:: [a] -> Integerlength []= 0length (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 <u>matched</u> against **actual parameters** in a fairly intuitive way

Matching patterns

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

[] only matches the empty list,

x:xs will successfully match any <u>list with at least one element</u>, 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 <u>returned</u> as the result of the application.

If it fails, the next equation is tried,

and if all equations fail, an error results.

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 <u>not</u> defined for all possible values of their argument. A **runtime error** occurs when these functions are applied to an empty list.

General types

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

the type **[a]** is <u>more general</u> than **[Char]**.

type **[Char]** can be <u>derived</u> from **[a]** by a suitable <u>substitution</u> for **a**.

Principal type

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

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

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

Unique principal types

An **expression's** or **function's principal type** is the <u>least general type</u> that, intuitively, "contains <u>all instances</u> 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 <u>unique</u> principal types is the hallmark feature of the Hindley-Milner type system, which forms the basis of the type systems of Haskell

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 <u>general</u> type of **map** to a more <u>specific</u> type.

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

Implicit forall

any introduction of a **lowercase type parameter** <u>implicitly</u> begins with a **forall** keyword,

Example: Two equivalent type statements

id :: a -> a

id :: forall a . a -> a

We can apply <u>additional</u> **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 **foralls** which are <u>nested</u> and <u>cannot</u> be <u>merged</u> with a previous one.

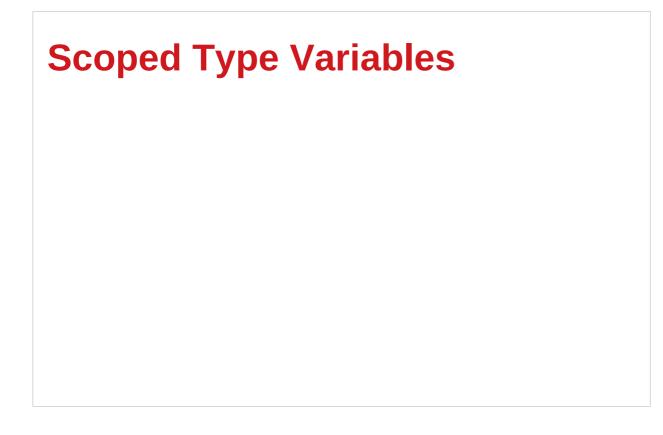
Existential Quantification

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 <u>doesn't</u> 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



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

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 <u>omit</u> the **forall** from a type it is actually still <u>implicitly</u> there. (normally these languages omit the **forall** from polymorphic types).

This claim is correct, but it refers to the other uses of **forall**, and <u>not</u> to the **ScopedTypeVariables** use.

{-# 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

```
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 make it possible to <u>specify</u> 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

ScopedTypeVariables breaks GHC's usual rule that explicit forall is <u>optional</u> and doesn't affect semantics.

the explicit forall is required

If <u>omitted</u>, usually the program will <u>not compile</u>; in a few cases it will <u>compile</u> but the functions get a <u>different</u> **signature**.

to trigger those forms of ScopedTypeVariables, the forall <u>must</u> appear against the top-level signature (or outer expression) but <u>not</u> against nested signatures referring to <u>the same</u> 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 <u>bound</u> by a forall scope
over the entire definition of f
the type variable a <u>scopes</u> 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.

f :: [a] -> [a]	f :: [a] -> [a]			
f (xs :: [aa]) = xs ++ ys	f (xs :: <mark>[a</mark>]) = xs ++ ys			
where	where			
ys :: [aa]	ys :: [<mark>a</mark>]			
ys = reverse xs	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** <u>bound</u> by the **pattern signature**

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

therefore there is <u>no need</u> to use a <u>distinct</u> type variable a

using **a** would be equivalent

Scoped Type

Let's start with that

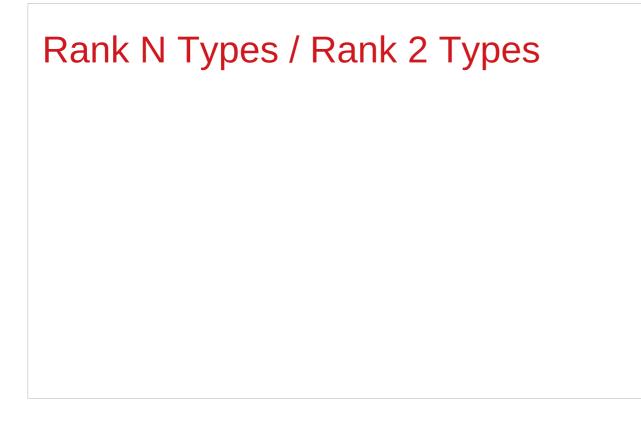
mayb :: b -> (a -> b) -> Maybe a -> b

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 <u>every</u> **a** and **b**.



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 <u>equivalent</u> to the previous signature.

Rank-N types (2)

```
forall can be floated
  (xxx) -> (forall 0. 000)
out of the right-hand side of ->
  forall 0. ((xxx) -> 000)
However, a forall appearing within the left-hand side of (->)
  (forall x. xxx) -> 000
cannot be moved up, and therefore forms another level or rank.
  forall x. ((xxx) -> 000)) not equivalent
forall 0. ((forall x. xxx)) -> 000)
```

Rank-N types (3)

The type is labeled "Rank-N"

where N is the number of **forall**s

which are <u>nested</u> and

cannot be merged with a previous one.

forall o. ((forall x. xxx)) -> 000)

Rank-2

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.

Rank-N types (5)

Rank-N type reconstruction is <u>undecidable</u> 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 #-}.

Polymorphic arguments of Rank-N

foo :: (<mark>forall a. a -> a</mark>) -> (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)

Rank-N example A (1)

In the normal case (forall n. Num n => (n -> n) -> (Int, Double))				
we <u>choose</u> an n <u>first</u> and				
then <u>provide</u> a function .	(n -> n)			
So we could <u>pass</u> in a function of type	(n -> n)			
Int -> Int,				
Double -> Double,				
Rational -> Rational				
and so on.				

Rank-N example A (2)

the previous case is normal (**Rank-1**) because that's how **type variables** work <u>by default</u>.

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

prenex form

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

is *implicitly* the same as

forall n. Num n => $(n \rightarrow n) \rightarrow (Int, Double)$.

Rank-N example A (3)

In the Rank-2 case

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

we have to <u>provide</u> the **function** <u>before</u> we know \mathbf{n} .

the type of a function that works for any n

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

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
{-# LANGUAGE RankNTypes #-}
rankN :: (forall n. Num n => n -> n) -> (Int, Double)
rankN = undefined
```

rank1 :: forall n. Num n => (n -> n) -> (Int, Double) rank1 = undefined (+1) for Num n

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

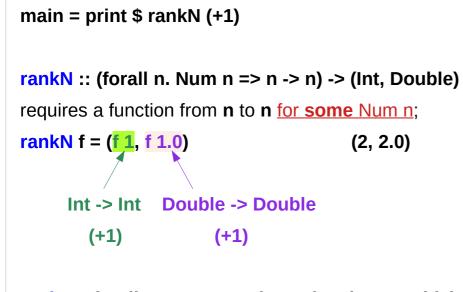
for any numeric type **n**, **f** can take an **n** and return an **n**

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).

Rank-N example A (6)



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.

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)		

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 #-}

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)
```

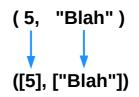
bar' ::	(Int -> Int) -> (Int, Double) -> (Int, Double)
bar' :: <mark>(Double</mark>	-> Double) -> (Int, Double) -> (Int, Double)

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)
```



Num -> [Num] [Char] -> [[Char]]

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]]

Rank-N example C (3) – code 2

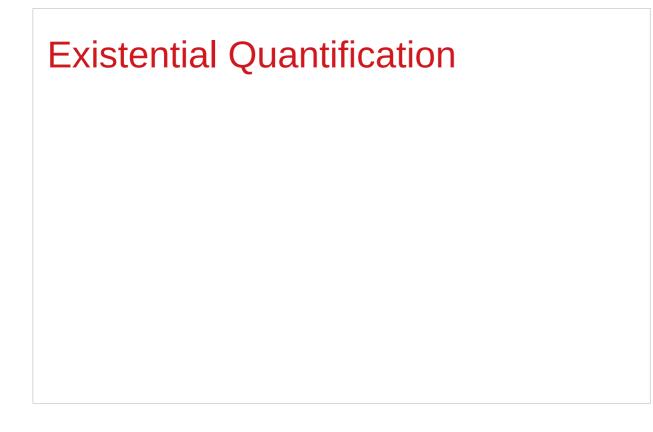
- test.hs liftTup :: (x -> f x) -> (a, b) -> (f a, f b) liftTup liftFunc (t, v) = (liftFunc t, liftFunc v)
ghci> :l test.hs Couldnt match expected type 'x' against inferred type 'b'
so here GHC doesn't let us apply liftFunc on v because v :: b and liftFunc wants an x .
Need a function that accepts <u>any possible x</u>

 $t :: a \leftrightarrow x$ $v :: b \leftrightarrow x$

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 <u>not</u> **liftTup** that works for all x, it's the function **liftFunc** that it gets that works for all x.



```
-- 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 <u>any</u> suitable type,

With Rank-N-Types,

forall a means that

your expression must fit <u>all possible</u> a's.

```
ghci> :set -XRankNTypes
```

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

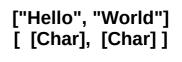
Couldnt match expected type 'a' against inferred type '[Char]'

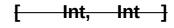
```
ghci> length ([] :: forall a. [a])
```

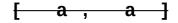
0

. . .

An empty list does work as a list of any type.







{-# LANGUAGE <u>ExistentialQuantification</u> #-} data EQList = forall a. EQList [a]

with Existential-Quantification,

foralls in data definitions mean that, the value contained can be of <u>any</u> suitable type,

not that it must be of <u>all</u> suitable types.

Existential quantification actually works a lot like **universal quantification**.

data <mark>Univ</mark> 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

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 <u>useless</u>, but it's still <u>valid</u> code.

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

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** <u>doesn't</u> 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 <u>not</u> 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

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 <u>extra, hidden</u> 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 <u>part</u> 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

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

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

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

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 \mathbf{x}
	via the existential in
	data Exis = forall a. Exis a
Exis x :: Exis	
Exis x :: forall	a. Exis a
x :: a	
c -> b	

In Haskell, the things being <u>quantified</u> 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, <u>for any possible</u> **type** "**a**", we can <u>implement</u> a **function** whose type is **a -> a**.

Since **a** is **universally quantified**, we <u>know nothing</u> about it, and therefore <u>cannot inspect</u> the **argument** in any way.

So id is the only possible function of this type

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

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

any type variables in a signature are <u>implicitly</u> 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**".

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

func :: exists a. a -> a

func True = False

func False = True

This is of course the "<u>not</u>" function on booleans.

we can't use it as such,

because all we know about the **type** "**a**" is that it <u>exists</u>.

Any information about *which type it might be* has been discarded.

which means we can't apply **func** to any **values**.

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**.

the reason why Haskell <u>doesn't</u> have **existential types** <u>directly</u> 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 \mathbf{r}

given a function that for all types a

takes an argument of type a

and <u>returns</u> a value of **type r**,

we can get a result of type r.

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

note that <u>the overall type</u> is <u>not</u> **universally quantified** for **a** --rather, it takes an <u>argument</u> 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** <u>argument</u>

References

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