# Monad P3 : Existential Types (1C)



Young Won Lim
10/29/20

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.

# 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

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

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

**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 <u>any type</u>
that results from <u>substituting</u> 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 <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**.

https://wiki.haskell.org/Polymorphism

# Parametric and ad hoc polymorphism

| Parametric polymorphism | ad hoc polymorphism |
|---|---|
| Type variables<br>(a, b, etc) | Type calsses<br>(Eq, Num, etc) |
| Universal | Existential? |
| Compile time | Runtime (also) |
| C++ templates<br>Java generics | Classical<br>(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:

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

# Scoped Type Variables

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

# 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
```

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

# 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.

This feature should be better documented in the Wiki, but this is a start.

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

# Hiding a type variable on the RHS

Normally when creating a new type
using **type**, **newtype**, **data**, etc.,
every **type variable** that appears on the right-hand side
must also appear on the left-hand side.

```
newtype ST s a = ST (State# s -> (# State# s, a #))
```

**Existential types** are a way of escaping

Existential types can be used for several different purposes.
But what they do is to **hide** a **type variable** on the right-hand side.

https://wiki.haskell.org/Existential_type

# Type Variable Example – (1) error

Normally, any type variable appearing on the right must also appear on the left:

**data Worker x y = Worker {buffer :: b, input :: x, output :: y}**

This is an **error**, since the **type** b of the **buffer**
Is not specified on the right (b is a **type variable** rather than a **type**)
but also is not specified on the left (there's no b in the left part).

In Haskell98, you would have to write

**data Worker b x y = Worker {buffer :: b, input :: x, output :: y}**

https://wiki.haskell.org/Existential_type

# Type Variable Example – (2) explicit type signature

However, suppose that a **Worker** can use any type **b**

so long as it belongs to some particular **class**.

Then every **function** that uses a **Worker** will have a type like

```
foo :: (Buffer b) => Worker b Int Int
```

In particular, failing to write an **explicit type signature** **(Buffer b)**

will invoke the dreaded monomorphism restriction.

Using **existential types**, we can avoid this:

https://wiki.haskell.org/Existential_type

# Type Variable Example – (3) existential type

**Using existential type :**

```
data Worker x y =  forall b. Buffer b =>  Worker {buffer :: b, input :: x, output :: y}
foo :: Worker Int Int
```

The **type** of the **buffer** (**Buffer**) now does not appear in the **Worker** type at all.

**Explicit type signature :**

```
data Worker b x y = Worker {buffer :: b, input :: x, output :: y}
foo :: (Buffer b) => Worker b Int Int
```

https://wiki.haskell.org/Existential_type

# Type Variable Example – (4) characteristics

- it is now impossible for a function
  to demand a **Worker** having a specific type of **buffer**.

- the **type** of **foo** can now be derived automatically
  without needing an explicit **type signature**.
  (No **monomorphism** restriction.)

- since code now has no idea
  what **type** the buffer function returns,
  you are more limited in what you can do to it.

```
data Worker x y = forall b. Buffer b => Worker {buffer :: b, input :: x, output :: y}
foo :: Worker Int Int
```

https://wiki.haskell.org/Existential_type

# Hiding a type

In general, when you use a **hidden type** in this way,
you will usually want that **type** to belong to a **specific class**,
or you will want to **pass some functions** along
that can work on that type.

Otherwise you'll have some value belonging
to a **random unknown type**,
and you won't be able to do anything to it!

https://wiki.haskell.org/Existential_type

# Conversion to less a specific type

Note: You can use **existential types**

to **convert** a **more specific type**

into a **less specific one**.

**constrained type variables**

There is no way to perform the reverse conversion!

https://wiki.haskell.org/Existential_type

# A heterogeneous list example

This illustrates **creating a heterogeneous list**,
all of whose members implement "**Show**",
and progressing through that list to show these items:

```
data Obj = forall a. (Show a) => Obj a

    xs :: [Obj]
    xs = [Obj 1, Obj "foo", Obj 'c']

    doShow :: [Obj] -> String
    doShow [] = ""
    doShow ((Obj x):xs) = show x ++ doShow xs
```

With output: **doShow xs ==> "1\"foo\"'c'"**

https://wiki.haskell.org/Existential_type

# Existentials in terms of **forall** (1)

It is also possible to express existentials with RankNTypes
as **type expressions** directly (without a **data** declaration)

**forall r. (forall a. Show a => a -> r) -> r**

(the leading forall r. is optional
unless the expression is part of another expression).

the equivalent type **Obj** :

**data Obj = forall a. (Show a) => Obj a**

https://wiki.haskell.org/Existential_type

# Existentials in terms of **forall** (2)

The conversions are:

```
fromObj ::  Obj -> forall r. (forall a. Show a => a -> r) -> r
fromObj (Obj x) k = k x

toObj :: (forall r. (forall a. Show a => a -> r) -> r)  ->  Obj
toObj f = f Obj
```

https://wiki.haskell.org/Existential_type

# Existentials

**Existential types**, or 'existentials' for short,
are a way of 'squashing' a group of types into one, single type.

Existentials are part of GHC's type system extensions.
They aren't part of Haskell98, and as such you'll have
to either compile any code that contains them
with an extra command-line parameter of **-XExistentialQuantification**,
or put **{-# LANGUAGE ExistentialQuantification #-}**
at the top of your sources that use existentials.

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

# forall

Example: A polymorphic function

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

Example: Explicitly quantifying the type variables

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

instantiating the general type of map to a more specific type

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

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

Example: Two equivalent type statements

**id :: a -> a**

**id :: forall a . a -> a**

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

# forall

Suppose we have a group of values.
We don't know if they are all the same type,
but we do know they are all members of some class
(and, by extension, that all the values have a certain property).

It might be useful to throw all these values into a list.
We can't do this normally because lists elements
must be of the same type (homogeneous with respect to types).

However, **existential types** allow us
to loosen this requirement by defining a **type hider** or **type box**:

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

# Heterogeneous list example

Example: Constructing a heterogeneous list

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

**heteroList :: [ShowBox]**

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

calling the constructor on three values of different types,

**[SB (), SB 5, SB True]**, to place them all into a single list

so we must somehow have the same type for each one.

Use the **forall** in the constructor

**SB :: forall s. Show s => s -> ShowBox.**

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

# forall

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

If we were now writing a function
to which we intend to pass **heteroList**,
we couldn't apply a function such as not to the values inside the **SB**
because their type might not be Bool.

But we do know something about each of the elements:
they can be converted to a string via show.
In fact, that's pretty much the only thing we know about them.

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

# forall

```
instance Show ShowBox where
 show (SB s) = show s        -- (*) see the comment in the text below

f :: [ShowBox] -> IO ()
f xs = mapM_ print xs

main = f heteroList
```

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

# forall

Example: Using our heterogeneous list

```
 instance Show ShowBox where
show (SB s) = show s        -- (*) see the comment in the text below
f :: [ShowBox] -> IO ()
f xs = mapM_ print xs
main = f heteroList
```

Example: Types of the functions involved

```
print :: Show s => s -> IO ()        -- print x = putStrLn (show x)
mapM_ :: (a -> m b) -> [a] -> m ()
mapM_ print :: Show s => [s] -> IO ()
```

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

# forall

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

# forall a. a

**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

# forall a. a

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
to also 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**.

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

# forall a. a

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

# forall

We see that most **intersections over types** just lead to **bottoms** because **types** generally don't have **any values in common** and so presumptions cannot be made about a **union of their values**.

a heterogeneous list using a '**type hider**'.
This **'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** and thereby guaranteeing certain facilities with such types.

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

# forall

Example: An existential datatype

**data T = forall a. MkT a**

Example: This defines a polymorphic constructor,

or a family of constructors for T

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

Example: Pattern matching on our existential constructor

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

Example: Constructing the hetereogeneous list

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

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

# forall

Example: A new existential data type, with a class constraint

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

Example: 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

# forall

Example: The runST function

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

Example: Bad ST code

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

Example: Briefer bad ST code

**... runST (newSTRef True) ...**

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

# forall

Example: The compiler's typechecking stage

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

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

**together, (forall s. ST s (STRef s Bool)) -> STRef s Bool**

Example: A type mismatch!

together, **(forall s'. ST s' (STRef s' Bool)) -> STRef s Bool**

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

# forall

Example: Identity function

**id :: forall a. a -> a**

**id a = a**

Example: Polymorphic value

**x :: forall a. Num a => a**

**x = 0**

Example: Existential type

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

Example: Sum type

**data ShowBox = SBUnit | SBInt Int | SBBool Bool | SBIntList [Int] | ...**

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

# forall

```
{-# LANGUAGE ExistentialQuantification, RankNTypes #-}

newtype Pair a b = Pair (forall c. (a -> b -> c) -> c)

makePair :: a -> b -> Pair a b
makePair a b = Pair $ \f -> f a b
```

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

# forall

```
λ> :set -XExistentialQuantification
λ> :set -XRankNTypes
λ> newtype Pair a b = Pair {runPair :: forall c. (a -> b -> c) -> c}
λ> makePair a b = Pair $ \f -> f a b
λ> pair = makePair "a" 'b'

λ> :t pair
pair :: Pair [Char] Char

λ> runPair pair (\x y -> x)
"a"

λ> runPair pair (\x y -> y)
'b'
```

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

# Rank-N-Types (1)

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

A Haskell '98 type signature such as

**a -> b -> a**

implies that the type variables are **universally quantified** like so:

**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)

However, a **forall** appearing within the **left-hand side** of (->) cannot be moved up, and therefore forms another level or rank.

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

is a **Rank-2** type because the latter forall can be moved to the start but the former one cannot.
Therefore, there are two levels of universal quantification.

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

# Rank-N-Types (3)

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

# Rank-N-Types (4)

In order to unpack an existential type,
you need a polymorphic function that works on any type
that could be stored in the existential.

This leads to a natural relation
between higher-rank types and existentials;
and an encoding of existentials
in terms of higher rank types
in continuation-passing style.

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

# R

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

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

The type of **foo** above is also 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**.

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

# Rank-N-Types

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

```
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)
```

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

# Rank-N-Types

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)
```

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

# Rank-N-Types

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'
    ...
```

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

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

We really want our function to get a function that accepts any possible x!

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

# Rank-N-Types

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'
    ...
```

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

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

We really want our function to get a function that accepts any possible x!

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

# Rank-N-Types

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 that it gets that does.

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