

Monad P3 : Existential Types (1D)

Copyright (c) 2016 - 2021 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

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>

Existential Quantification

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

Hiding a type variable (1)

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

Hiding a type variable (2)

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

Hiding a type variable (3)

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

Hiding a type variable (4)

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. **Worker x y**

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

Hiding a type variable (5)

- 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 variable (6)

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

Less specific types (1)

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

Less specific types (2)

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, provide 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 at the top of your sources that use existentials.

`{-# LANGUAGE ExistentialQuantification #-}`

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

forall type variables

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

Heterogeneous Lists

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

Type hider

Suppose we have a group of values.

they may not be all the same **type**,

but they are all **members** of some **class**

thus, they have a certain **property**

It might be useful to throw all these **values** into a **list**.

normally this is impossible because **lists elements**

must be of **the same type**

(**homogeneous** with respect to **types**).

existential types allow us to loosen this requirement

by defining a **type hider** or **type box**:

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

```
heteroList :: [ShowBox]
```

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

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

Heterogeneous list example (1)

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

```
-- type hider
```

```
heteroList :: [ShowBox]
```

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

[SB (), SB 5, SB True] calls the **constructor** on three values of different types, to place them all into a single list virtually **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

Heterogeneous list example (2)

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

When passing **heteroList** type parameters to a function
we cannot take out the **values** inside the **SB**
because their type might **Bool**, **Int**, **Char**, ...

But each of the elements can be
converted to a **string** via **show**.

In fact, that's the only thing we know about them.

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

Heterogeneous list example (3)

```
instance Show ShowBox where
```

```
  show (SB s) = show s
```

```
f :: [ShowBox] -> IO ()
```

```
f xs = mapM_ print xs
```

```
main = f heteroList
```

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

Heterogeneous list example (4)

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

mapM, mapM_, and map (1)

The core idea is that **mapM** maps an "action" (ie function of type **a -> m b**) over a list and gives you **all** the results as **m [b]**

mapM_ does the same thing, but never collects the results, returning a **m ()**.

If you care about the **results** of your **a -> m b** function, use **mapM**.
If you only care about the **effect**, but not the resulting value, use **mapM_**, because it can be more **efficient**

<https://stackoverflow.com/questions/27609062/what-is-the-difference-between-mapm-and-mapm-in-haskell/27609146>

mapM, mapM_, and map (2)

Always use **mapM_** with functions of the type **a -> m ()**,
like **print** or **putStrLn**.
these functions return **()** to signify that only the **effect** matters.

If you used **mapM**, you'd get **a list of ()** (ie **[(), (), ()]**),
which would be completely useless
but waste some memory.

If you use **mapM_**, you would just get a **()**,
but it would still print everything.

<https://stackoverflow.com/questions/27609062/what-is-the-difference-between-mapm-and-mapm-in-haskell/27609146>

mapM, mapM_, and map (3)

Normal **map** is something different:

it takes a normal function (**a -> b**)

instead of one using a monad (**a -> m b**).

This means that it cannot have any sort of **effect**

besides returning the **changed list**.

You would use it if you want to **transform a list**

using a normal function.

map_ doesn't exist because, since you don't have any effects,

you always care about the **results** of using **map**.

<https://stackoverflow.com/questions/27609062/what-is-the-difference-between-mapm-and-mapm-in-haskell/27609146>

forall

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

A set of possible values

One way to think about **forall** is to consider **types** as a set of possible values.

Bool is the set **{True, False, ⊥}**
(remember that **bottom**, **⊥**, is a member of every type!),

Integer is the set of integers (and bottom),

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

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

Intersection of the specified types

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

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

this **subset** turns out to be the set $\{\perp\}$,

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, $\{\perp\}$** ,
this quantification in fact stipulates all Haskell types.

But the only permissible operations on it are

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

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

A list of bottoms type (1)

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

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

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

A list of bottoms type (2)

The list, `[forall a. a]`, is the **type of a list**
whose **elements** all have the type `forall a. a`, i.e.
a list of bottoms. `[⊥]` , `[⊥,⊥]` , ...

The list, `[forall a. Show a => a]`, is the **type of a list**
whose **elements** all have the type `forall a. Show a => a`.

the **Show** class constraint requires the possible types
also to be **a member of the class, Show.**

However, `⊥` is still the only value common to all these types, `{⊥}`,
so this too is **a list of bottoms.** `[forall a. a]`

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

A list of bottoms type (3)

The list, `[forall a. Num a => a]`, requires each element to be a member of the class, Num.

Consequently, the possible values include **numeric literals**, which have the specific type, `forall a. Num a => a`, as well as **bottom**.

`forall a. [a]` is the type of **the list** whose elements all have the same type **a**.

since we cannot presume any particular type at all, this too is **a list of bottoms**.

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

Intersections over types

most **intersections over types** just lead to **bottoms** $\perp\perp\perp\perp$
types generally don't have **any values in common**
presumptions cannot be made about a **union of their values**.

a **heterogeneous list** using a **type hider**
type hider' functions as a **wrapper type**
which guarantees certain facilities
by implying a **predicate** or **constraint** on the permissible **types**.

the purpose of **forall** is to **impose type constraint**
on the permissible types within a **type declaration**
guaranteeing certain facilities with such types.

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

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

Summary of heterogeneous list examples (1)

An **existential datatype**

```
data T = forall a. MkT a
```

This defines a **polymorphic constructor**,
or a family of constructors for **T**

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

Pattern matching on our existential constructor

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

Constructing the **heterogeneous list**

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

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

```
heteroList :: [ShowBox]
```

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

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

Summary of heterogeneous list examples (2)

A new existential data type, with a **class constraint**

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

```
data T = forall a. MkT a
```

Using our new heterogeneous setup

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

```
main = mapM_ (\(MkT' x) -> print x) heteroList'
```

```
{- prints:
```

```
5
```

```
()
```

```
True
```

```
"Sartre"
```

```
-}
```

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

runST

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

State and ST monads

the **ST** monad is essentially

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

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

IO is basically just a **State** monad

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

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

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

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

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

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

runST – rank-2 polymorphism

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

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

a **forall** appearing within the **left-hand side** of (->)
cannot be moved up, and therefore forms **another level or rank**
therefore, there are **2 levels** of universal quantification.

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

runST – initial state

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

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

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

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

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

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

runST – reference interfaces

To create references of the type **STRef**

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

To provide an **initial value**

readSTRef :: STRef s a -> ST s a

To manipulate them.

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

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

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

runST – a mapping

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

is not one specific value,

but a **mapping** from **references** to **values**.

... (STRef s a)

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

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

as the **initial state** is just the **empty mapping**

... ()

containing **no references**.

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

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

runST – no specific references

It is not allowed

to create a **reference** in one **ST computation**,
then to use the created **reference** in another **ST computation**.
for reasons of **thread-safety**

because no ST computation should be allowed
to assume that the **initial internal environment**
contains any specific references.

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

runST

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

Example: Bad ST code

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

Example: Briefer bad ST code

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

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

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

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

runST

Example: Bad ST code

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

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

the **rank-2 polymorphism** in **runST**'s type
to constrain the scope of the **type variable s**
to be within the first parameter (the left hand side of ->)

if the **type variable s** appears in the first parameter
it cannot also appear in the second.
(the right hand side of ->)

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

runST

Example: Briefer bad ST code

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

Example: The compiler's typechecking stage

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

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

```
runST (newSTRef True) ::
```

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

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

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

```
readSTRef :: STRef s a -> ST s a
```

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

forall

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

`runST (newSTRef True) ::`

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

Example: A type mismatch!

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

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

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

forall

Example: A type mismatch!

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

Notice that as the `forall` does not scope over the return type of `runST`, `STRef s Bool` we don't rename the `s` there as well.

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

The result type of the ST computation in the **first parameter** must match the **result type** of `runST`, but now it doesn't!

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

forall

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

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

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

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

Quantified types as products and sums

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

Quantified Types as Products and Sums

A **universally** quantified type may be interpreted as an **infinite product** of types.

a **polymorphic function** can be understood as a **product**, or a **tuple**, of **individual functions**, one per every possible **type a**.

To construct a **value** of such **type**, we have to provide all the **components** of the **tuple** at once.

-- one formula generating an **infinity** of functions

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

Quantified Types as Products and Sums

Example: Identity function

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

```
id a = a
```

a **polymorphic function** can be understood
as a **product**, or a **tuple**, of **individual functions**,
one per every possible **type a**.

```
Int -> Int, Double -> Double, ...
```

```
Char -> Char, [Char] -> [Char], ...
```

```
...
```

```
...
```

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

Quantified Types as Products and Sums

To construct a **value** of such **type**, we have
to provide all the **components** of the **tuple** at once.

in case of **numeric types**, one **numeric constant**
may be used to initialize **many types** at once.

Example: Polymorphic value

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

```
x = 0
```

x may be conceptualized as a **tuple** consisting
of an **Int value**, a **Double value**, etc.

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

Quantified Types as Products and Sums

Similarly, an **existentially quantified type** may be interpreted as an **infinite sum**.

Example: Existential type

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

may be conceptualized as a **sum**:

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

Quantified Types as Products and Sums

Example: Existential type

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

Example: Sum type

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

to construct a **value** of this **type**,
we only have to pick one of the constructors.

A **polymorphic constructor SB**

combines all those constructors into one.

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

Quantification as a primitive

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

Pair type example (1)

Universal quantification is useful

for defining data types that aren't already defined.

Suppose there was no such thing as **pairs** built into haskell.

Quantification could be used to define them.

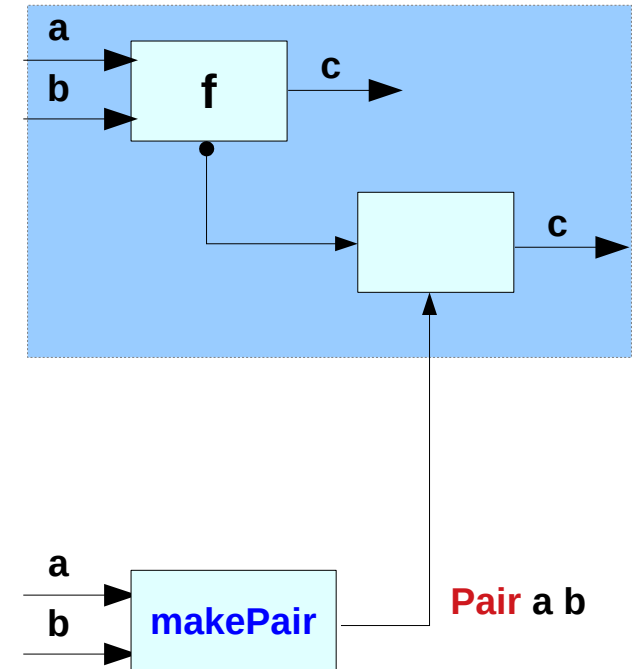
```
{-# LANGUAGE ExistentialQuantification, RankNTypes #-}
```

```
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
```

Pair \$ \f -> f a b :: **Pair** a b



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

Pair type example (2)

In GHCi

```
λ> :set -XExistentialQuantification
```

```
λ> :set -XrankNTypes
```

```
λ> newtype Pair a b = Pair {runPair :: forall c. (a -> b -> c) -> c}
```

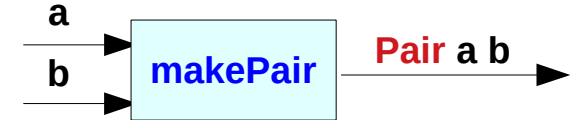
```
λ> makePair a b = Pair $ \f -> f a b
```

```
λ> pair = makePair "a" 'b'
```

```
Pair $ \f -> f "a" 'b'
```

$\backslash f$: function itself $f :: a \rightarrow b \rightarrow c$

$f \text{ "a" 'b'}$: function application result



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

Pair type example (3)

```
λ> newtype Pair a b = Pair {runPair :: forall c. (a -> b -> c) -> c}
```

```
Pair {runPair}
```

```
Pair {forall c. (a -> b -> c) -> c}
```

```
pair :: Pair [Char] Char
```

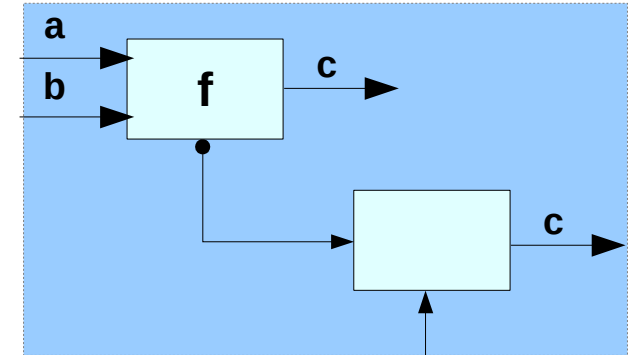
```
pair = makePair "a" 'b'
```

```
Pair $ \f -> f "a" 'b'
```

```
pair (λx y -> x) = Pair $ \f -> f "a" 'b' (λx y -> x)
```

```
runPair pair (λx y -> x)
```

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



```
pair (λx y -> x)
```

```
makePair "a" 'b' (λx y -> x)
```

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

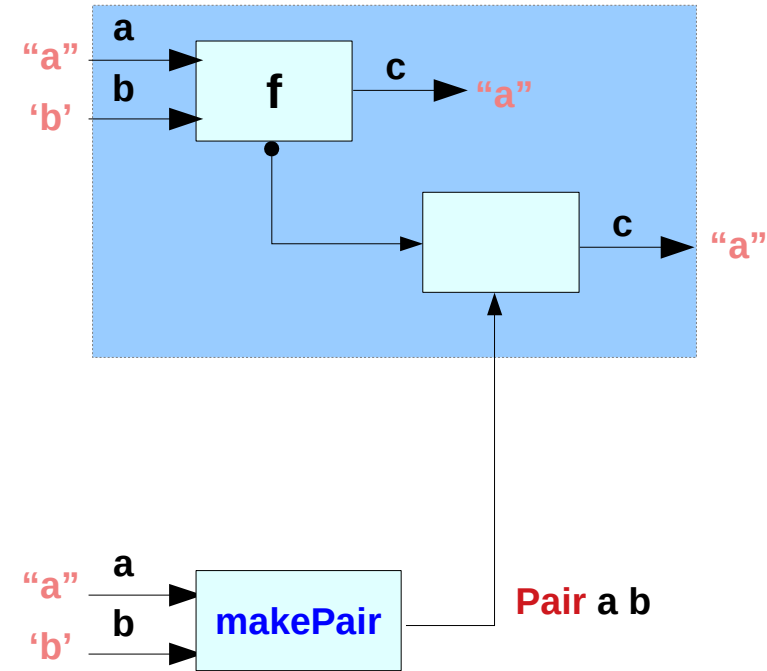
Pair type example (3)

```
runPair pair (λx y -> x)
```

```
runPair makePair "a" 'b' (λx y -> x)
```

```
runPair Pair $ λf -> f "a" 'b' (λx y -> x)
```

Pair \$ λf -> f a b :: **Pair** a b



```
pair (λx y -> x)
```

```
makePair "a" 'b' (λx y -> x)
```

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

Pair type example (4)

```
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 (λx y -> x)           pair
```

```
"a"
```

```
λ> runPair pair (λx y -> y)
```

```
'b'
```

```
pair
```

```
makePair "a" 'b'
```

```
Pair $ \f -> f "a" 'b'
```

```
pair (λx y -> x)
```

```
Pair $ \f -> f "a" 'b'
```

```
f = (λx y -> x)
```

```
f "a" 'b' = "a"
```

```
Pair $ (λx y -> x) -> "a"
```

```
runPair :: forall c. (a -> b -> c) -> c
```

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
 $\exists a, \text{st } a \rightarrow a, \forall a \in \text{Num}$

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:

$\forall a, a -> a$

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 $\forall a, a \rightarrow a$

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)

$\forall a, a \rightarrow a$ the **liberty** at the **type** level
foo 5 = 7 a constraint at the **term** level
 (impossible implementation)

this **liberty** at the **type** level is what is known
as **Universal Quantification**

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

foo a = a the **identity** function

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

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

foo :: a -> a (8)

Runar Bjarnason titled "Constraints Liberate, Liberties Constrain".

CONSTRAINTS LIBERATE,
LIBERTIES CONSTRAIN

Its very important to digest and believe this statement

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

RunST (1)

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

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

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

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

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

RunST (2)

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

the benefit is that this puts a **constraint** on the **caller** of **runST** in that the **type a** cannot involve the **type s** at all.

you can't pass it a value of type **ST s [s]**, for example.

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

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

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

RunST : rank-2 polymorphic type

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

The **type** of **runST** is an example of a **rank-2 polymorphic type** because the **type** of its **argument** contains a **forall** quantifier.

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

Existential 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
```

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

Existential Quantification

```
ghci> :set -XRankNTypes
ghci> length (["Hello", "World"] :: forall a. [a])
  Couldn't 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>