

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

Existential Quantification

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

Existentials

Existential types, or
Existentials for short,
provide a way of
squashing a group of types
into one, single type.

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

Existentials

Existentials are part of GHC's type system **extensions**.

But not part of **Haskell98**

have to either compile with a 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

The forall keyword is to explicitly bring fresh **type variables** into scope

type variables

those variables that begin with a **owercase** letter

the compiler allows **any type** to fill these variables

those variables that are **universally quantified**

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

forall type variables

Example: A polymorphic function

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

a lowercase type parameter

implicitly begins with a **forall** keyword,

Example: Explicitly quantifying the type variables

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

two type declarations for map are equivalent

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

b = String

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

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

Hiding a type variable

<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 this rule

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 (3')

The **monomorphism restriction** is a counter-intuitive rule in Haskell type inference.

If you *forget to provide* a **type signature**, sometimes this rule will **fill** the free type variables with **specific types** using **type defaulting** rules.

always less polymorphic than you'd expect, so often this results in **type errors** when you expected it to infer a perfectly sane type for a polymorphic expression.

https://wiki.haskell.org/Existential_type

Hiding a type variable (3'')

A simple example is **plus = (+)**.

Without an explicit signature for **plus**,
the compiler will not infer the type for **plus**

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

but will apply **defaulting rules** to specify

plus :: Integer -> Integer -> Integer

When applied to **plus 3.5 2.7**, GHCi will then produce
the somewhat-misleading-looking error,
No instance for (Fractional Integer) arising from the literal '3.5'.

https://wiki.haskell.org/Existential_type

Hiding a type variable (4)

Explicit type signature :

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

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**

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

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           -- type hider
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>

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>

Newtype creates a function (1)

```
newtype Parser a = Parser { parse :: String -> Maybe (a,String) }
```

- 1) A **type** named **Parser**.
- 2) A **term level constructor** of **Parser**'s named **Parser**.
The **type** of this (constructor) function is

```
Parser :: (String -> Maybe (a, String)) -> Parser a
```

You give it a function of the type

```
(String -> Maybe (a, String))
```

and it wraps it inside a **Parser**

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (2)

```
newtype Parser a = Parser { parse :: String -> Maybe (a,String) }
```

- 3) A **function** named `parse` to remove the `Parser` wrapper and get your function back. The type of this function is:

```
parse :: Parser a -> String -> Maybe (a, String)
```

A **term level constructor** named `Parser`

```
Parser :: (String -> Maybe (a, String)) -> Parser a
```

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (3)

```
Prelude> newtype
```

```
  Parser a = Parser { parse :: String -> Maybe (a,String) }
```

```
Prelude> :t Parser
```

```
Parser :: (String -> Maybe (a, String)) -> Parser a
```

```
Prelude> :t parse
```

```
parse :: Parser a -> String -> Maybe (a, String)
```

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (4)

```
newtype Parser a = Parser { parse :: String -> Maybe (a,String) }
```

the **term level constructor** (`Parser`)

the **function** to remove the wrapper (`parse`)

Both can have arbitrary names

No need to match the type name.

It's common to write:

```
newtype Parser a = Parser { unParser :: String -> Maybe (a,String) }
```

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (5)

```
newtype Parser a = Parser { unParser :: String -> Maybe (a,String) }
```

this name makes it clear `unParser` removes
the **wrapper** around the parsing function.

```
unParser :: Parser a -> String -> Maybe (a, String)
```

however, it is recommended that the **type** and **constructor**
have **the same name** when using **newtypes**.

```
(Parser, Parser)
```

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (6)

```
newtype Parser a = Parser { parser :: String -> Maybe (a,String) }
```

1) **Parser** is declared as a **type** with a **type parameter a**

2) can instantiate **Parser** by providing a **parser** function

```
p = Parser (\s -> Nothing)
```

3) a function name **parser** defined and

it is capable of *running Parser's*.

unwrap the function

then apply the function

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (7)

```
newtype Parser a = Parser { parser :: String -> Maybe (a,String) }
```

```
parser :: Parser a -> String -> Maybe (a, String)
```

```
parser (Parser (λs -> Nothing)) "my input"
```

```
(λs -> Nothing) "my input"
```

```
Nothing
```

You are **unwrapping** the **function** using **parse** and then calling the unwrapped function with "myInput".

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (8)

First, let's have a look at a parser **newtype** without **record** syntax:

```
newtype Parser' a = Parser' (String -> Maybe (a,String))
```

it stores a function **String -> Maybe (a,String)**.

To run this parser, we will need to make a **new function**:

```
runParser' :: Parser' a -> String -> Maybe (a,String)
```

```
runParser' (Parser' f) i = f i
```

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (9)

```
runParser' :: Parser' a -> String -> Maybe (a,String)
```

```
runParser' (Parser' f) i = f i
```

```
runParser' (Parser' $ \s -> Nothing) "my input".
```

But now note that, since Haskell functions are curried,
we can simply remove the reference to the input `i` to get:

```
runParser'' :: Parser' -> (String -> Maybe (a,String))
```

```
runParser'' (Parser' f') = f'
```

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (10)

```
runParser'' :: Parser' -> (String -> Maybe (a,String))
```

```
runParser'' (Parser' f') = f'
```

This function is exactly equivalent to `runParser'`,
but you could think about it differently:

instead of applying the parser function to the value explicitly,
it simply takes a parser and fetches the parser function from it;

```
(Parser' f') → f'
```

however, thanks to **currying**, `runParser''`
can still be used with two arguments.

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (11)

```
newtype Parser a = Parser { parse :: String -> Maybe (a,String) }  
newtype Parser' a = Parser' (String -> Maybe (a,String))
```

difference : record syntax with only one field

this record syntax automatically defines a function

```
parse :: Parser a -> (String -> Maybe (a,String)),
```

which extracts the `String -> Maybe (a,String)` function
from the `Parser a`.

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Newtype creates a function (12)

```
newtype Parser a = Parser { parse :: String -> Maybe (a,String) }
```

`parse` can be used with two arguments thanks to **currying**, and this simply has the effect of **running** the function stored within the `Parser a`.

equivalent definition to the following code:

```
newtype Parser a = Parser (String -> Maybe (a,String))
```

```
parse :: Parser a -> (String -> Maybe (a,String))
```

```
parse (Parser p) = p
```

<https://stackoverflow.com/questions/60291263/why-the-newtype-syntax-creates-a-function>

Access functions in a record type (1)

```
data Person = Person { firstName :: String ,  
                        lastName :: String ,  
                        age      :: Int   ,  
                        height   :: Float ,  
                        phoneNo  :: String ,  
                        flavor    :: String  
                      } deriving (Show)
```

```
ghci> :t flavor  
flavor :: Person -> String  
ghci> :t firstName  
firstName :: Person -> String
```

return types of
access functions

Person ::
the input type of
access functions

<http://learnyouahaskell.com/making-our-own-types-and-typeclasses>

Access functions in a record type (2)

```
data Car = Car String String Int deriving (Show)
```

```
ghci> Car "Ford" "Mustang" 1967
```

```
Car "Ford" "Mustang" 1967
```

```
data Car = Car {company :: String,  
                model  :: String,  
                year   :: Int} deriving (Show)
```

```
ghci> Car {company="Ford", model="Mustang", year=1967}
```

```
Car {company = "Ford", model = "Mustang", year = 1967}
```

<http://learnyouahaskell.com/making-our-own-types-and-typeclasses>

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
```

```
f :: a -> b -> c
```

```
f a b :: c
```

f is not yet defined

c can be any type (**forall** c)

defining data type c

that aren't already defined

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

Pair type example (2)

```
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
```

using a record type with a single field

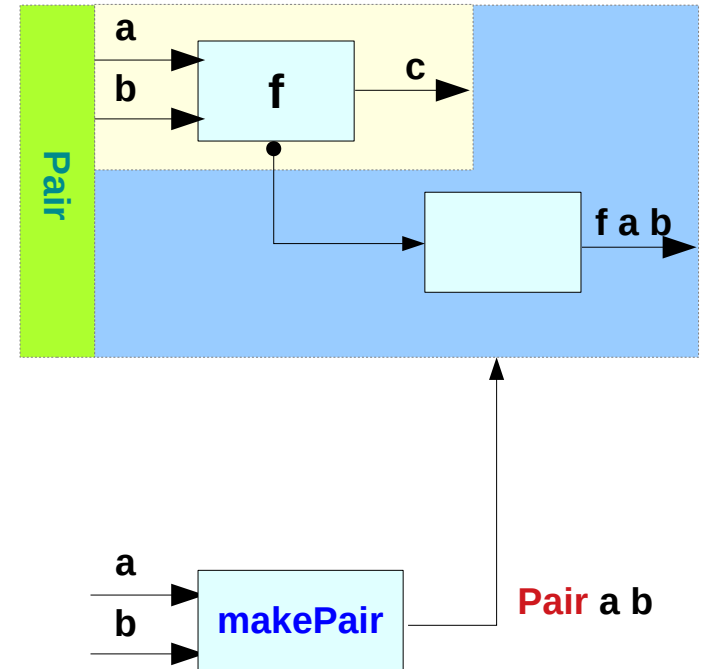
```
λ> newtype Pair a b = Pair {runPair :: forall c. (a -> b -> c) -> c}
```

runPair is an access function

takes an input of the type **Pair a b**

returns an output of the type **forall c. (a -> b -> c) -> c**

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



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

Pair type example (3)

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'
```

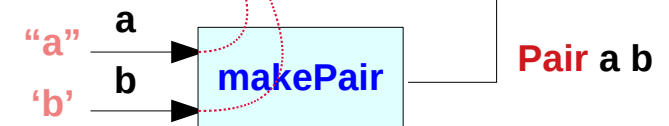
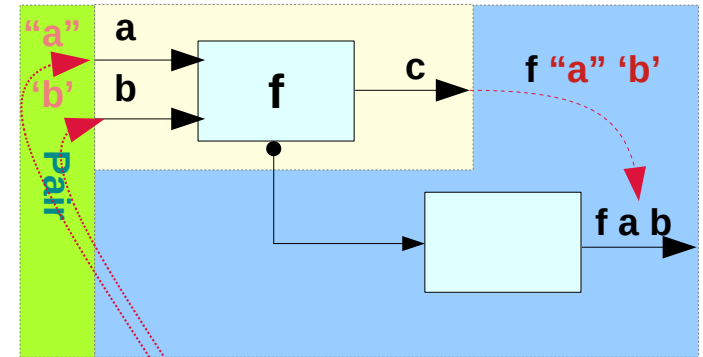
```
λ> :t pair
```

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

```
λ> runPair pair (\x y -> x) -- unwrap (a -> b -> c) -> c then apply  
"a"
```

```
λ> runPair pair (\x y -> y) -- unwrap (a -> b -> c) -> c then apply  
'b'
```

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



makePair "a" 'b'

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

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
```

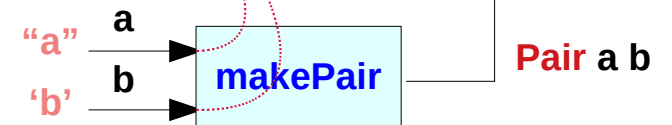
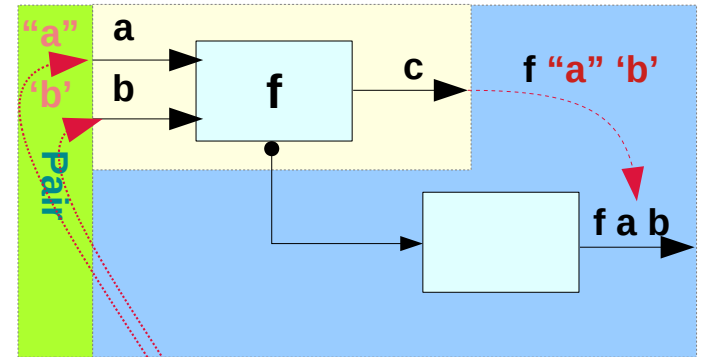
```
λ> pair = makePair "a" 'b'
```

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

```
\f: function itself    f :: a -> b -> c
```

```
f "a" 'b': the result of applying the function
```

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



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

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

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

Pair type example (5)

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

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

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

```
runPair makePair a b = \f -> f a b -- unwrapping
```

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

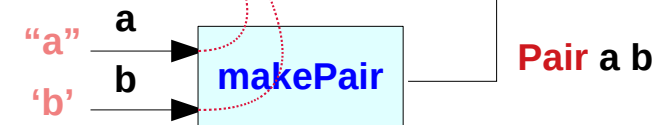
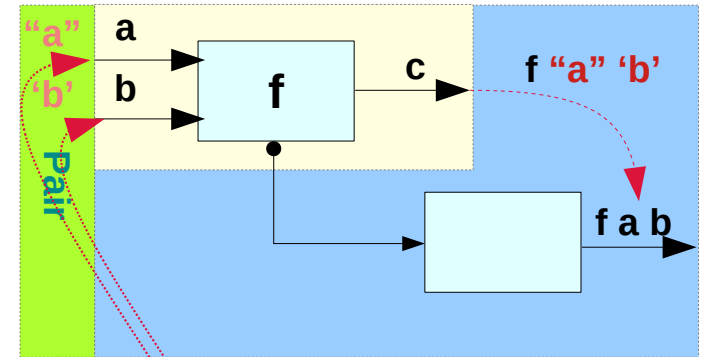
```
runPair makePair "a" 'b' = \f -> f "a" 'b'
```

```
pair = makePair :: Pair [Char] Char
```

```
runPair pair (lx y -> x) = (lx y -> x) "a" 'b'
```

```
runPair pair (lx y -> y) = (lx y -> y) "a" 'b'
```

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



makePair "a" 'b'

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

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

Pair type example (6)

```
runPair pair (lx y -> x) = (lx y -> x) "a" 'b'
```

```
runPair pair (lx y -> y) = (lx y -> y) "a" 'b'
```

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

```
(lx y -> x) "a" 'b'
```

```
"a"
```

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

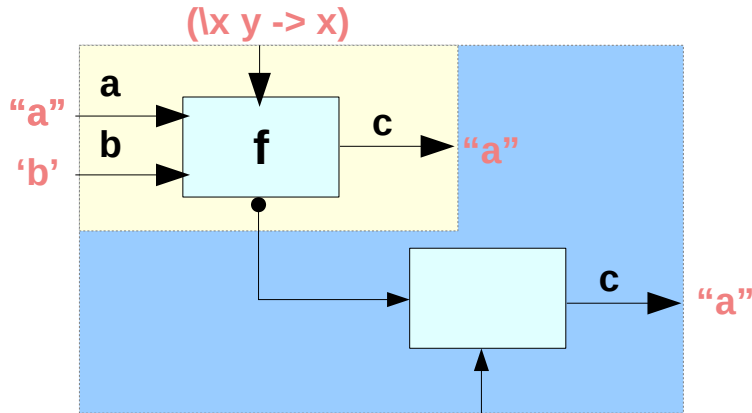
```
(lx y -> y) "a" 'b'
```

```
'b'
```

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

Pair type example (6)

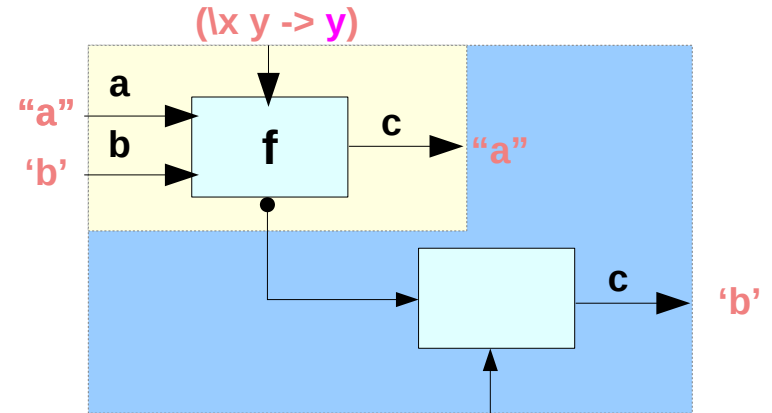
Pair \$ \lambda f \rightarrow f a b :: \mathbf{Pair} a b



pair $(\lambda x y \rightarrow x)$

$\mathbf{makePair}$ "a" 'b' $(\lambda x y \rightarrow x)$

Pair \$ \lambda f \rightarrow f a b :: \mathbf{Pair} a b



pair $(\lambda x y \rightarrow y)$

$\mathbf{makePair}$ "a" 'b' $(\lambda x y \rightarrow y)$

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

References

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