Monad P3 : Existential Types (1D)

Young Won Lim
7/26/21

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

Haskell quantification

- the things being <u>quantified</u> <u>over</u> are **types**
  (ignoring certain language extensions, at least),

- <u>logical statements</u> are also **types**

- a "**<u>true</u>**" <u>logical statement</u> as "<u>can be implemented</u>".

- technically "**false**" should correspond to
  an **uninhabited data type** (often called **Void**)

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

Universally quantified type

id :: forall a. a -> a

id x = x

    for <u>any</u> possible type **a**,   *quantified over types*

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

        <u>can</u> <u>be</u> <u>implemented</u>   *a true logical statement*

    id works for <u>all</u> **a**.

        a will unify with (or will be fixed to) <u>any type</u>

        that <u>caller</u> of **id** may <u>choose</u>.

universally quantified type variables

in a type signature are

existentially quantified

in a function body

https://markkarpov.com/post/existential-quantification.html

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

Universally quantified type variables

universally quantified **type variables** in a type signature

      will be fixed when the corresponding **function**

      is used (called)

Since **a** is universally quantified

      in the **body** of the function

            we know nothing about the **argument a**,

            we cannot inspect the **argument a**

      (because **a** *is fixed when the function is used*)

id :: forall a. a -> a

id x = x

universally quantified type variables

existentially quantified in a function body

https://markkarpov.com/post/existential-quantification.html

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

Young Won Lim
7/26/21

Lack of information in a function body

universally quantified **type variables** in a type signature

    callers can pass (choose) anything to **id**

    but due to the lack of information
    about the **argument** in the body of **id**

    a caller can only pass a value to **id**
    without doing anything meaningful

So, **id x = x** is the only possible function of the type **a -> a**

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

a **caller** chooses values for
universally quantified variables

in the **body** of a such function,
must handle any type values
which is given by a caller :
existentially quantified variable

https://markkarpov.com/post/existential-quantification.html

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

Parametric polymorphism (1)

universal quantification is the <u>default</u>

      any **type variables** in a **type signature** are

      <u>implicitly</u> universally quantified,

      id :: a -> a

      id :: forall a. a -> a

also known as **parametric polymorphism**

in some other languages (e.g., C#) known as **generics**.

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

Young Won Lim
7/26/21

Parametric polymorphism (2)

Parametric polymorphism refers to

when **the <u>type</u> of a value** contains

one or more (<u>unconstrained</u>) **type variables**,

      beginning with a lowercase letter

      without constraints (nothing to the **left** of a =>)

so that **the value** may adopt **any type**

      that results from <u>substituting</u> those **type variables**

      with **concrete types**.

Just 2.0   **:: Maybe Double**

Just 'a'   **:: Maybe Char**

Just True **:: Maybe Boolean**

https://wiki.haskell.org/Polymorphism

Young Won Lim
7/26/21

Parametric polymorphism (3)

Polymorphic datatypes

data Maybe a = Nothing | Just a

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

data Either a b = Left a Right b

Polymorphic functions

reverse :: [a] -> [a]

fst :: (a, b) -> a

id :: a -> a

Just 2.0   :: Maybe Double

Just 'a'   :: Maybe Char

Just True :: Maybe Boolean

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

Young Won Lim
7/26/21

Parametric polymorphism (4)

Since a **parametrically polymorphic value** does not know

anything about the unconstrained **type variables**,

it must behave identically **for all type** (regardless of its **type**)

(related to universally quantification)

This is a somewhat limiting but extremely useful property

known as **parametricity**.

https://wiki.haskell.org/Polymorphism

Parametric polymorphism (5)

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.

if a single **type variable** appears multiple times,

it must take the same **type** everywhere it appears

→ the **result type** of **id** must be the same as the **argument type**

https://wiki.haskell.org/Polymorphism

Quantified variable choice

A **variable** is universally quantified

    when the <u>consumer</u> of the variable's expression

    can choose what it will be.

A **variable** is existentially quantified

    when the <u>consumer</u> of the variable's expression

    has to deal with the fact that the choice <u>was made</u> for him.

<u>consumers</u> of **a function**

callers of a function	the **body** of such a **function**

Universally quantified variable

the <u>consumer</u> <u>chooses</u>

Existentially quantified variable

the choice is <u>made</u> for the <u>consumer</u>

https://markkarpov.com/post/existential-quantification.html

Quantified variables with forall

Both universally and existentially quantified variables

     are introduced with **forall**.

There is no **exists** in Haskell.

In fact, it's not necessary.

https://markkarpov.com/post/existential-quantification.html

Young Won Lim
7/26/21

Making existentials

data Something where

  Something :: forall a. a -> Something

one way to have **existentials** –

      by putting **values** in **wrappers**

      that "hide" **type variables** from **signatures**.

  Something a   **:: Something**

      the **type variable a** is hidden in the **type Something**

https://markkarpov.com/post/existential-quantification.html

Young Won Lim
7/26/21

Existential wrappers

```
data Something where

  Something :: forall a.  a -> Something



  Something  a         :: Something



  Something 2.0        :: Something

  Something 'a'        :: Something

  Something True       :: Something


        the constructor function Something return

        data value of type Something
```

constructor

```
data Point a     = Pt a a
```

constructor   polymorphic type

```
Pt  2.0  3.0      :: Point Float

Pt  'a'  'b'      :: Point Char

Pt True False    :: Point Bool
```

type constructor +
bounded type parameter
: a concrete type

https://markkarpov.com/post/existential-quantification.html

Existential wrappers

```
data Something where
  Something :: forall a.  a -> Something
```

```
data Point a      = Pt a a
```

```
findx :: Something -> Float         ✕  ✕
findx (Something x) -> x
```

```
pointx :: Point Float -> Float
pointx (Pt x _) = x
```

The **constructor** accepts <u>any</u> **a** we like,

       but <u>after construction</u> we

           lose the type information

      and <u>pattern matching</u> afterwards only reveals

           that <u>there is some **a**</u>,

           but <u>nothing</u> regarding <u>what it is</u>.

```
pointy :: Point Float -> Float
pointy (Pt  _ y) = y
```

Young Won Lim
7/26/21

Existential wrappers

data Something where

  Something :: forall a.   **a -> Something**

the constructor function Something return

     existentially quantified data of type **Something**

Something a	:: Something
When constructing a data value	When data value is used
universally quantified **a**	existentially quantified **a**

Something 1 :: Something

Something 'a' :: Something

Something 2.0 :: Something

https://markkarpov.com/post/existential-quantification.html

Existential wrappers

```
data Something where

  Something :: forall a.   a -> Something


data r where

  r :: forall a.   a -> r


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

Something	a	:: Something

When constructing a data value	When data value is used
universally quantified **a**	existentially quantified **a**

the **type variable a** is <u>hidden</u> in the **type r**

Existential wrappers

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

forall **r.** | *argument callback* | -> r
exponentially quantified a

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

Outer level

Inner level

Inner level	Outer level
callback function body	callback function as an argument
universally quantified **a**	existentially quantified **a**

the **type variable a** is <u>hidden</u> in the **type r**

Returning existentially quantified data

- passing a value to **id**: (universally quantified)

  we can <u>pass</u> anything to **id** but we <u>lack any information</u>
  about the **argument** <u>in the **body** of **id**</u>.

- passing a value to **Something** (existentially quantified)

  existential wrappers

  → return **existentially quantified data** from a **function**.

  → <u>avoid</u> <u>unification</u> of **existentials** with *outer context*

  → <u>escape</u> **type variables**.

id 1 :: Int

id 'a':: Char

Id 2.0:: Double

Something 1 :: Something

Something 'a' :: Something

Something 2.0 :: Something

https://markkarpov.com/post/existential-quantification.html

Returning existentially quantified data

- passing a value to **id**:   (universally quantified)

  universally quantified variable

  the consumer chooses

  id :: forall a. a -> a

- passing a value to **Something**   (existentially quantified)

  existentially quantified variable

  the choice is made for the consumer

  data Something where

        Something :: forall a.   a -> Something

id Int :: Int

id Char:: Char

id Double:: Double

foo :: Something -> Int

foo Something a :: Int

forall r. foo :: Something -> r

foo Something a :: Int

foo Something a :: Char

foo Something a :: Double

https://markkarpov.com/post/existential-quantification.html

Existentially quantified data constructors (1)

data **Foo**   = forall a. **MkFoo** a (a -> Bool) | **Nil**

the **data type Foo** has *two* **constructors** with types:

MkFoo :: forall a. a -> (a -> Bool) -> Foo

Nil   :: Foo

Notice that the **type variable a** in the type of **MkFoo**

does <u>not</u> <u>appear</u> in the **data type** itself, which is plain **Foo**.

MkFoo 3 even :: Foo

MkFoo 'c' isUpper :: Foo

even :: Integer -> Bool

isUpper :: Char -> Bool

https://downloads.haskell.org/~ghc/6.6/docs/html/users_guide/type-extensions.html

Existentially quantified data constructors (2)

MkFoo :: forall a. a -> (a -> Bool) -> Foo

a valid expression example

  [MkFoo 3 even, MkFoo 'c' isUpper] :: [Foo]

      (MkFoo 3 even) <u>packages</u> an **integer** with a function   even :: Integer -> Bool

      (MkFoo 'c' isUpper) packages a **character** with a function   isUpper :: Char -> Bool

      Each of these are of type **Foo** and can be put in a list.

Existentially quantified data constructors (3)

What can we do with a **value** of **type Foo**?.

In particular, what happens when we pattern-match on **MkFoo**?

f (MkFoo val fn) = ???

Since all we know about **val** and **fn** is that they are compatible,

the only (useful) thing we can do with them is

to apply **fn** to **val** to get a **boolean**.

https://downloads.haskell.org/~ghc/6.6/docs/html/users_guide/type-extensions.html

Young Won Lim
7/26/21

Existentially quantified data constructors (4)

f :: **Foo** -> **Bool**

fn :: **a** -> **Bool**

f (**MkFoo** **val** **fn**) = **fn** **val**

What this allows us to do is

to package heterogenous **values** together

with a bunch of **functions** that manipulate them,

and then treat that collection of packages in a uniform manner.

In this way, you can express **object-oriented-like** programming

fn :: **a** -> **Bool**

even :: **Integer** -> **Bool**

isUpper :: **Char** -> **Bool**

https://downloads.haskell.org/~ghc/6.6/docs/html/users_guide/type-extensions.html

Unknown types at compile time

Existentials have always to do with

throwing type information away.

sometimes we want to work with **types**

that we don't know at compile time.

the **types** typically depend on the **state** of **external world**:

the **types** could depend on user's input,

on contents of a file to be parsed, etc.

Haskell's type system is powerful enough in these cases

https://markkarpov.com/post/existential-quantification.html

Young Won Lim
7/26/21

Preserving information about existentials

We want to <u>work</u> with **values** of **types**

　　　that we <u>don't know</u> at compile time,

　　　but at run time there are **no types** at all:

　　　　　they have been erased!

then we <u>have to</u> *preserve* some information

　　　about existentially quantified type to make use of it,

otherwise we'll be in the same position as implementers of **id**

　　　<u>having</u> a **value** and <u>only being able to pass</u> it around

　　　<u>never doing anything</u> meaningful with it.

There are various degrees of how much we might want to *preserve*:

https://markkarpov.com/post/existential-quantification.html

Young Won Lim
7/26/21

Parameterizing another type

We could have **a** in the type **[a]** existentially quantified.

There are still some things we could do with a **value** of this type.

    we could compute length of the list.

So knowing nothing about **a** type is also an option sometimes

    when it parameterizes **another type** and

    we have parametrically-polymorphic functions

        that work on that type.

In this case the set of possible types for **a** is open i.e. it can grow.

https://markkarpov.com/post/existential-quantification.html

Young Won Lim
7/26/21

Existentially quantified type with **constraints**

data Showable where

  Showable :: forall a. ==**Show a =>**== **a -> Showable**

We could assume that the existentially quantified type

has *certain properties* (instances):

- pattern-matching on **Showable** will give us

  the corresponding dictionary back.
- can do as much as the knowledge about the attached **constraint**
- the set of possible types for **a** is open

  (additional new **instances** of **Show** can be defined).

data Something where

  Something :: forall a. a -> Something

simple existentially quantified type variable

https://markkarpov.com/post/existential-quantification.html

The first **forall** at the type signature

myPrettyPrinter

  :: forall a. **Show a =>**

     (forall b. **Show b =>** **b -> String)**

     -> Int

     -> Bool

     -> a

     -> String

Only **variables** with **forall**s at the beginning of **type signature**

will be fixed when the corresponding **function** is used

Other **forall**s deal with independent **type variables**:

---

forall a. * (forall b. ***)**

when **myPrettyPrinter** is used

    a will be *fixed*

    but not **b**

the 1st argument is

    a call back function

    b -> String

https://markkarpov.com/post/existential-quantification.html

Two levels of **forall**s

myPrettyPrinter

  :: forall a. Show a =>

      (forall b. Show b => b -> String)   -- call back function

              -> Int

              -> Bool

              -> a

              -> String

two levels of **forall**s (**rank-2 type**)

  forall a. * (forall b. ***)**

in general such constructions
are called **rank-N types**.

https://markkarpov.com/post/existential-quantification.html

For consumers of a function

Both universally and existentially quantified variables

are introduced with **forall**.

for callers of **myPrettyPrinter**

- **a** is universally quantified

  we can choose what the type will be

- **b** is existentially quantified

  the callback function has to prepare to deal with any **b**

  that will be given to the callback **b -> String**

myPrettyPrinter

  :: forall a. Show a =>

     (forall b. Show b => b -> String)

     -> Int

     -> Bool

     -> a

     -> String

callers of **myPrettyPrinter** provide

the call back **b -> String**

which must handle any **b**

https://markkarpov.com/post/existential-quantification.html

For consumers of a function

print (myPrettyPrinter callback 123 True)

Consumers of the expression 1

myPrettyPrinter **fn** i t x =

... **fn** 0.8 ...

Consumers of the expression 2

return str

fn :: b -> String

i :: Int

t :: Bool

x :: a

str :: String

myPrettyPrinter

:: forall **a**. Show **a** =>

(forall **b**. Show **b** => **b** -> String)

-> Int

-> Bool

-> **a**

-> String

https://markkarpov.com/post/existential-quantification.html

Young Won Lim
7/26/21

In the body of a function

- for the **callers** of **myPrettyPrinter**, **a** is universally quantified
- in the **body** of **myPrettyPrinter**, **a** is existentially quantified
  - ➔ the caller of **myPrettyPrinter** _already_ _has_ _chosen_ the **type**
  - ➔ A specific return type of the callback function **b -> String**

- for the **callers** of **myPrettyPrinter**, **b** is existentially quantified
- in the **body** of **myPrettyPrinter**, **b** is universally quantified
  - ➔ **b** is the first **argument** of the call back function **b -> String**
  - ➔ <u>when the call back function</u> is applied with **b**
    the body of **myPrettyPrinter** _can_ _choose_ its **concrete type**

b -> String -> Int -> Bool -> a -> String

myPrettyPrinter

  :: forall a. Show a =>

    (forall b. Show b => b -> String)

    -> Int

    -> Bool

    -> a

    -> String

Universally quantified variable
the <u>consumer</u> <u>choose</u>

Existentially quantified variable
the choice is <u>made</u> for the <u>consumer</u>

https://markkarpov.com/post/existential-quantification.html

Fictitious syntax *exists a.*

An **existentially quantified type** <u>could</u> be better <u>explained</u>

     using the fictitious *exists a.* syntax

     exists a. **a -> a**

     for <u>a certain</u> **type a**,

     we <u>can implement</u> a **function** whose type is **a -> a**.

     <u>any function</u> will do,

     then the "**not**" function on **Bool** satisfies the type **a -> a**

func :: *exists a.* a -> a

func True = False

func False = True

Function implementations and applications

the function <u>implementation</u> on booleans

func :: *exists a.* a -> a

func True = False

func False = True

but we cannot <u>use</u> (apply) it as the "**not**" function

because <u>all we know</u> about the **type a** is

<u>that it exists</u>.

Any <u>information</u> about which type it might be

has been discarded (i.e, is not used),

this means <u>we can't apply **func** to any values</u>

Existentials are always about

throwing type information away.

sometimes we want to work with **types**

that we <u>don't know</u> at compile time.

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

No direct existential types

This fact brings us back to **universal quantifiers**,

and the reason why Haskell <u>doesn't</u> have **existential types** <u>directly</u>

(***exists a.*** above is entirely fictitious)

since things with **existentially quantified types**

      can only be used with **operations**

      that have **universally quantified types**,

- for the **callers** of **myPrettyPrinter**

    b is existentially quantified

- in the **body** of **myPrettyPrinter**

    b is universally quantified

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

Subtyping

subtyping (also **subtype polymorphism**)

is a form of type polymorphism in which a subtype is a datatype

that is related to another datatype (the supertype)

by some notion of substitutability,

meaning that program elements,

typically subroutines or functions,

written to operate on elements of the supertype

can also operate on elements of the subtype.

https://en.wikipedia.org/wiki/Subtyping

Young Won Lim
7/26/21

Existential types and forall

Haskell doesn't have a notion of **subtyping**

Quantifiers can be considered as a tool for **subtyping**,

with a **hierarchy** going from **universal** to **concrete** to **existential**.

type forall a. a could be <u>converted</u> to **any other type**,

so it could be seen as a **subtype** of <u>everything</u>;

any type could be <u>converted</u> to the **type exists a. a**,

making that a **supertype** of everything.

forall a. a   universal

↓

any type   concrete

↓

exists a. a   existential
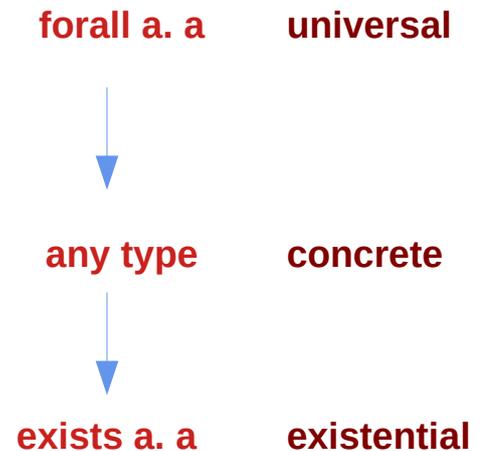
Existential types and forall

forall a. a is impossible

> there are <u>no</u> <u>values</u> of type **forall a. a** except errors

exists a. a is useless

> you <u>canot</u> <u>do</u> anything with the type **exists a. a**

but the analogy works on paper at least.

So, the basic idea is roughly that

universally quantified types describe

> things that work the same for **any type**,

existentially quantified types describe

> things that work with a **specific** but **unknown** type.

forall a. a   subtype of everything

    ↓ impossible –
    no such value

any type

    ↓ useless –
    cannot do anything

exists a. a   supertype of everything

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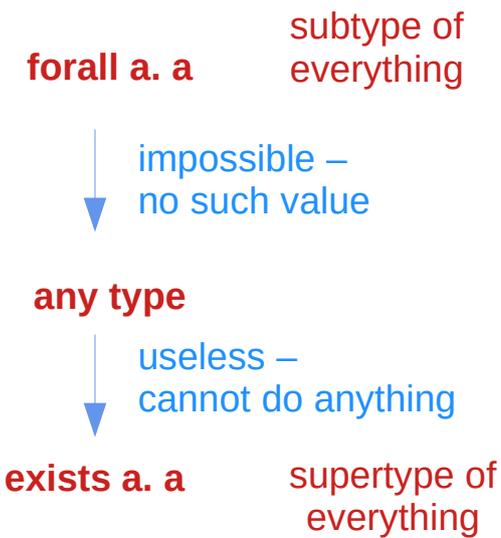
Existential types and forall

Assume **r** stands for "**false**"

technically "**false**" should correspond to
an **uninhabited data type** (often called **Void**),
but using **r** instead lets us get values back out.

So "**not (not A)**" would be

      (A -> Void) -> Void   -- useless

      forall r. (A -> r) -> r   -- can extract the **A** value, i.e.
                                        -- double-negation elimination.
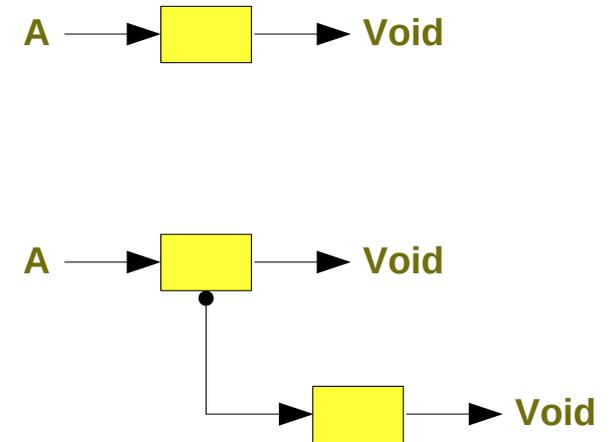
Existential types and forall

De Morgan's laws as applied to **quantifiers**;

function inputs are **negated**, logically speaking.

any
type **a** → ☐ → specific
type **r**

There's a similar equivalence between

Either a b   … implicit universal quantification

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

which corresponds to "**A or B**"

being the same as "**not (not A) and (not B)**".
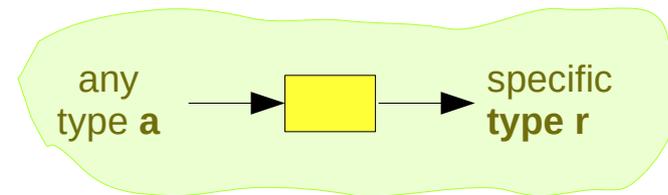
(Not a) and (Not b)

(a -> r , b -> r)

Not ((Not a) and (Not b))

((a -> r , b -> r)) ->   r

Existential types and forall

Look up the connection between **logical double-negation**

and continuation-passing style if you want to know more

Due to duality, **exists a. a** can be expressed as

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

Due to duality, **forall a. a** can be expressed as

exists r. (exists a. a -> r) -> r

Existential types and forall

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

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

exists a. **a**

think a **callback function** **a** -> **r**

      forall **a.** **a** -> **Int**
      forall **a.** **a** -> **String**   a caller chooses **type r**
      forall **a.** **a** -> **Double**

The **caller** of the <u>overall</u> function

        (**a** -> **r**) -> **r**

        chooses any type **r**

The **body** of the <u>overall</u> function

        (**a** -> **r**) -> **r**

        chooses any type **a**

the **body** of the <u>callback</u> function

        must handle for all type **a**

Existential types and forall

we can write the type

> ***exists a. a***

as

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

      for <u>all</u> **result types r**,

          given a function   **a -> r**

              that takes an <u>argument</u> of **type a**, for <u>all</u> **types a**

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

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

      a caller supplies the callback function of the type **a -> r**
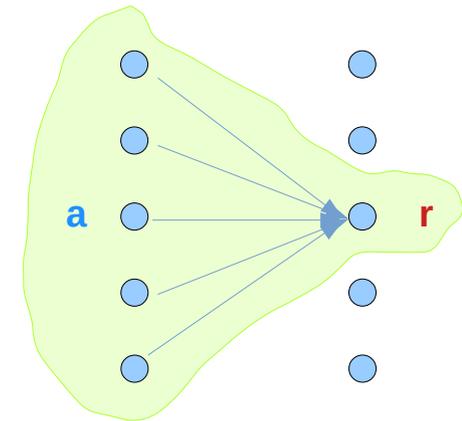
A caller supplies the callback function with the type **a -> r**

Existential types and forall

we can write the type

> ***exists a. a***

as

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

    a caller supplies the callback function of the type **a -> r**

    for a given **type r**

        forall a. a -> Int
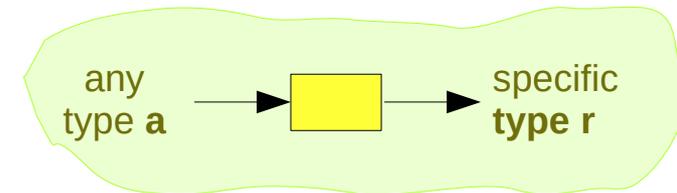
        forall a. a -> String   a caller chooses **type r**

        forall a. a -> Double
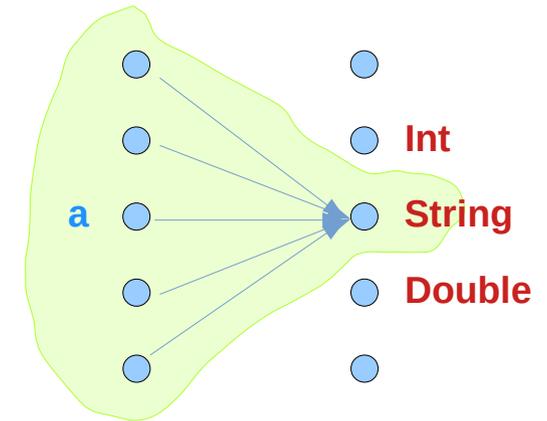
a caller of the overall type
determines the specific type r

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

Existential types and forall

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

a caller of the <u>overall</u> type function
chooses the specific **type r**

universally
quantified **r**

any
type **a** → ▮ → specific
type r
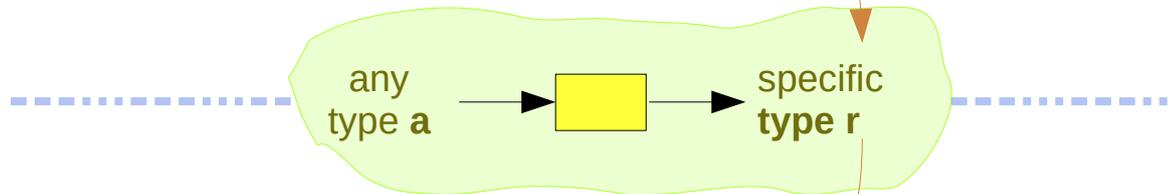
The body of the <u>overall</u> type function
must handle any **type r**

existentially
quantified **r**

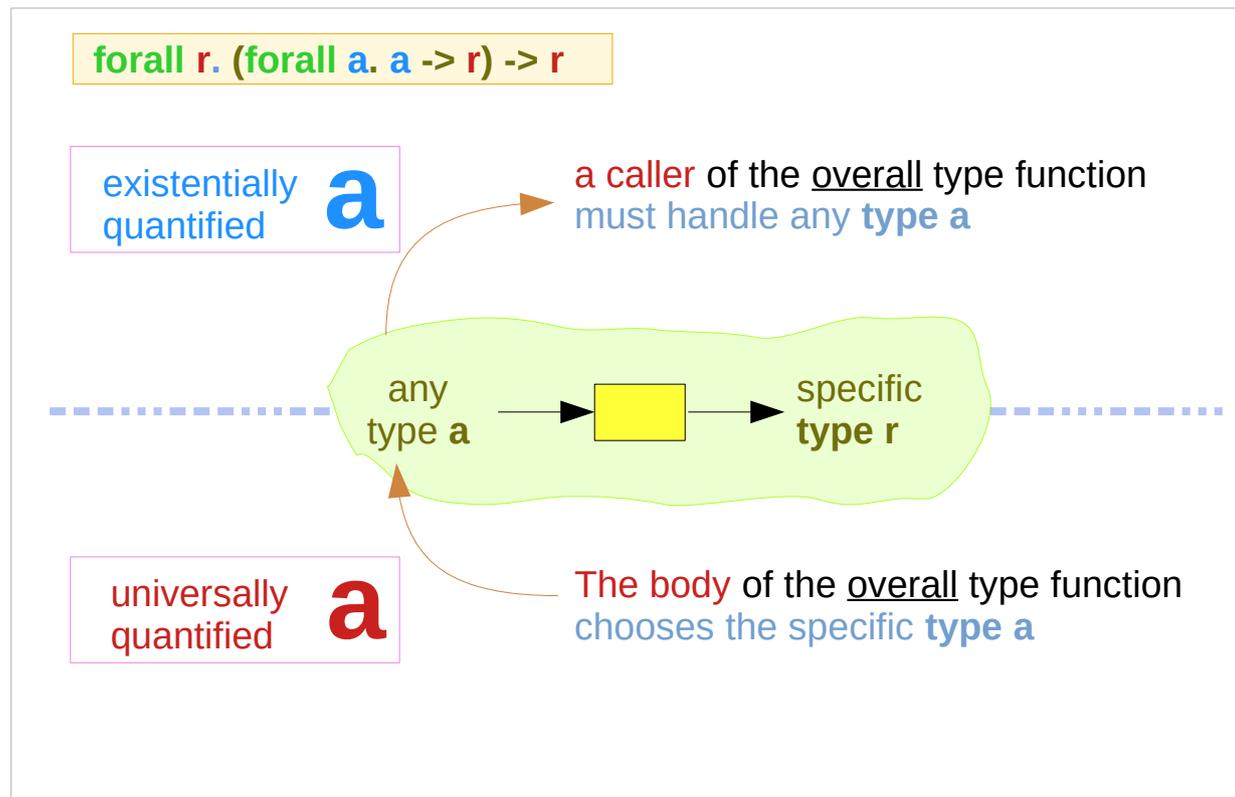| for the **callers** of the **function**	in the **body** of the **function**
universally quantified **r**	existentially quantified **r**
existentially quantified **a**	universally quantified **a**

Existential types and forall

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

existentially quantified **a**

a caller of the <u>overall</u> type function must handle any **type a**

The body of the <u>callback</u> function must also handle any **type a**

any type **a** → specific **type r**

universally quantified **a**

The body of the <u>overall</u> type function chooses the specific **type a**

<u>for the **callers**</u> of the **function**		<u>in the **body**</u> of the **function**	
universally quantified	**r**	existentially quantified	**r**
existentially quantified	**a**	universally quantified	**a**

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

Existential types and forall

forall **r**.

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

    -> **r**

for the **callers** of the **function**	in the **body** of the **function**
universally quantified **r**	existentially quantified **r**
existentially quantified **a**	universally quantified **a**

myPrettyPrinter

  :: **forall a.** Show a =>

     (forall **b.** Show b => **b -> String**)

     -> Int

     -> Bool

     -> **a**

     -> String

for the **callers** of the **function**	in the **body** of the **function**
universally quantified **a**	existentially quantified **a**
existentially quantified **b**	universally quantified **b**

callers of **myPrettyPrinter** provide

the call back function **b -> String**

which must handle any **b**

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

Existential types and forall

overall function type

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

universally
quantified **r**

specific
type a

existentially
quantified **a**

the overall type can choose
whatever specific type **r**

Body

the 1st argument of the **overall** type
is a **callback** function
its 1st argument **a** is selected somehow
in the **body** of the **overall** function

callback function type

Caller

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

universally
quantified **a**

specific
type r

existentially
quantified **r**

the callback function type can
choose whatever specific type **a**

the **caller** of the **overall** function
supplies a **callback** function for a
specific return type **r**

For the caller of the function

for the **callers** of the **function**
universally quantified **r**
existentially quantified **a**

For the body of the function

in the **body** of the **function**
existentially quantified **r**
universally quantified **a**

Existential types and forall

we can write the type

     exists a. a

as

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

the overall type is <u>*not*</u> **universally quantified** for **a**

     only its argument **(forall a. a -> r) is universally quantified** for **a**

The overall type takes an **argument**   …   **(forall a. a -> r)**

     that itself is **universally quantified** for **a**,

The overall type can then use

     with whatever <u>specific</u> **type r** it <u>chooses</u>.

<u>for the **callers**</u> of the **function**		<u>in the **body**</u> of the **function**	
universally quantified	**r**	existentially quantified	**r**
existentially quantified	**a**	universally quantified	**a**

The overall type can choose whatever specific type r
Universally quantified

Restoring exact types

```
data EType a where
    ETypeWord8      :: EType Word8
    ETypeInt        :: EType Int
    ETypeFloat      :: EType Float
    ETypeDouble     :: EType Double
    ETypeString     :: EType String


data Something where
    Something :: EType a -> a -> Something
```

We could use GADTs to restore exact types of
existentially quantified variables later:

https://markkarpov.com/post/existential-quantification.html

Young Won Lim
7/26/21

How to make use of existentials

Matching on one of the **data constructors** of **EType**

reveals **a** and after that we are free to do anything

with the **value** of corresponding **type**

because we know it.

With this approach the set of possible types for **a**

    is limited and closed.

It can be expanded

    by changing the **definition** of **EType** though.

```
data EType a where
    ETypeWord8      :: EType Word8
    ETypeInt        :: EType Int
    ETypeFloat      :: EType Float
    ETypeDouble     :: EType Double
    ETypeString     :: EType String

data Something where
    Something
        :: EType a -> a -> Something
```

https://markkarpov.com/post/existential-quantification.html

Young Won Lim
7/26/21

Generalized Algebraic Data Type (1)

Generalised Algebraic Data Types

generalise ordinary algebraic data types

by allowing you to give the **type signatures** of **constructors** explicitly.

```
data Term a where
    Lit     :: Int              -> Term Int
    Succ    :: Term Int         -> Term Int
    IsZero  :: Term Int         -> Term Bool
    If      :: Term Bool        -> Term a -> Term a -> Term a
    Pair    :: Term a -> Term b -> Term (a,b)
```

https://downloads.haskell.org/~ghc/6.6/docs/html/users_guide/gadt.html

Young Won Lim
7/26/21

Generalized Algebraic Data Type (2)

Notice that the **return type** of the constructors is <u>not</u> <u>always</u> **Term a**,

as is the case with ordinary vanilla data types.

Now we can write a well-typed **eval** function for these Terms:

```
eval :: Term a -> a
eval (Lit i)          = i
eval (Succ t)         = 1 + eval t
eval (IsZero t)       = eval t == 0
eval (If b e1 e2)     = if eval b then eval e1 else eval e2
eval (Pair e1 e2)     = (eval e1, eval e2)
```

https://downloads.haskell.org/~ghc/6.6/docs/html/users_guide/gadt.html

Existential Quantification

Young Won Lim
7/26/21

Existentials

Existential types, or

Existentials for short,

provide a way of

squashing a group of types

into one, single type.

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 and type variables

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

type variables

       those variables that begin with a **lowercase** 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

Young Won Lim
7/26/21

Type variables in a polymorphic function

Example: A polymorphic function

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

a **lowercase type parameter**

    <u>implicitly</u> <u>begins</u> 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

Instantiating 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 <u>general type</u> of **map**

to a more <u>specific type</u>

a = Int

b = String

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

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

Hiding a type variable

Young Won Lim
7/26/21

A rule for creating a new type

Normally when creating a new type

using **type**, **newtype**, **data**, etc.,

      every **type variable** that appears on the <u>right-hand side</u>

      <u>must</u> also <u>appear</u> on the <u>left-hand side</u>.

      newtype ST s a = ST (State# s -> (# State# s, a #))
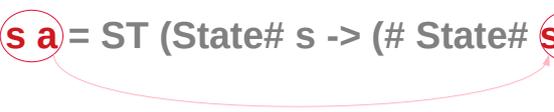
Existential types are a way of <u>escaping</u> this rule

Existential types can be used for several different purposes.

But what they do is to **<u>hide</u>** a **type variable** on the <u>right-hand side</u>.

https://wiki.haskell.org/Existential_type

Not specifying a type variable

Normally, any **type variable** appearing <u>on the right</u>

      must also appear <u>on the left</u>:

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

      This is an **error**, since the **type b** of the **buffer**

      is <u>not</u> <u>specified</u> on the <u>right</u>

      (**b** is a **type variable** rather than a **type**)

      but also is <u>not</u> <u>specified</u> on the <u>left</u>

      (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}

Record Access Functions
buffer   **:: Worker x y -> b**
input   **:: Worker x y -> x**
output   **:: Worker x y -> y**

https://wiki.haskell.org/Existential_type

A type variable and a class

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

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

Monomorphism restriction

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

Young Won Lim
7/26/21

Monomorphism restriction example

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

<u>Without</u> an explicit signature for **plus**,

the compiler will <u>not</u> <u>infer</u> 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'.

Young Won Lim
7/26/21

Explicit types and Existential types

Explicit type signature :

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

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

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 <u>not</u> <u>appear</u>

in the **Worker** type at all.   **Worker x y**

https://wiki.haskell.org/Existential_type

Existential types and forall

func is a function with the <u>same</u> **type** for its **input** and **output**

so we could compose it with itself, for example.

<u>the only things</u> you can do with something

that has an **existential type** are

the things you can do based on the **non-existential parts** of the **type**.

Similarly, given something of type **exists a. [a]**

we can find its length, or concatenate it to itself,

or drop some elements, or anything else we can do to **any list**.

func :: exists a. a -> a

func True = False

func False = True

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

Existential types and forall

in *pseudo*-Haskell:

   (exists x. p x x) -> c $\cong$ **forall x. p x x -> c**

It tells us that a <u>function</u> **p** that <u>takes</u> an **existential type x**

is equivalent to a **polymorphic function**

using a **universal quantifier forall x**

because the **function p** must be prepared

to handle <u>any one</u> of <u>the types</u> **x**

that may be encoded in the **existential type**.

Haskell does not need an existential quantifier

Existential types and forall

a function that <u>accepts</u> a **sum type** must be implemented as

a **case** statement, with a **tuple of handlers**,

one for every type present in the sum.

Here, the sum type is replaced by a coend,

and a family of handlers becomes an end,

or a polymorphic function.

Existential types and forall

an example of an **existentially quantified type**

data Sum = forall a. Constructor a

forall a. (Constructor_a:: a -> Sum) $\cong$ **Constructor:: (exists a. a) -> Sum**

data Sum = int | char | bool | ....

an example of a **universally quantified type**

data Product = Constructor (forall a. a)

data Product = int char bool ....

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

Young Won Lim
7/26/21

Hiding a type variable (6)

you will usually want a **hidden type** to belong to a **specific class**,

or you will want to **pass some functions** along

that can <u>work</u> <u>on that type</u>.

Otherwise you'll have some value belonging

to a **random unknown type**,

and you won't be able to do anything 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 (7)

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

Hiding a type variable (7)

In Haskell, an existential data type is one

that is defined in terms not of a concrete type,

but in terms of a quantified type variable,

introduced on the right-hand side of the data declaration.

https://blog.sumtypeofway.com/posts/existential-haskell.html

Hiding a type variable (7)

an existential type provides

a well-typed "box" around an unspecified type.

The box does "hide" the type in a sense,

which allows you to make a heterogeneous list of such boxes,

ignoring the types they contain.

It turns out that an unconstrained existential pretty useless,

but a constrained type allows you to pattern match

to peek inside the "box" and make the type class facilities available:

https://blog.sumtypeofway.com/posts/existential-haskell.html

Less specific types

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

Young Won Lim
7/26/21

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

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

82

Young Won Lim
7/26/21

Type hider

Suppose we have a group of values.

    they may <u>not</u> be all the <u>same</u> **type**,

    but they are all members of <u>some</u> **class**

    thus, they have a certain **property**

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

    normally this is <u>impossible</u> because lists elements

    <u>must</u> be of the same type

    (homogeneous with respect to types).

existential types allow us to <u>loosen</u> 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

Young Won Lim
7/26/21

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 <u>different types</u>,

      to place them all into <u>a single list</u>

      virtually the same type for each one.

Use the **forall** in the **constructor**

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

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

Young Won Lim
7/26/21

instance **Show ShowBox** where

show **(SB s) =** show s

In the definition of **show** for **ShowBox**

we <u>don't</u> know the **type** of **s**.

But we do <u>know</u> that the **type** is an **instance** of **Show**

due to the **constraint** on the **SB constructor**.

Therefore, it's legal to use the function **show** on **s**,

as seen in the right-hand side of the function definition.

ShowBox data type made into

an instance of the **Show** class

by this **instance declaration**:

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

Heterogeneous list example (4)

```
instance Show ShowBox where
  show (SB s) = show s


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


main = f heteroList


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

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

Heterogeneous list example (5)

Example: Using our heterogeneous list

instance Show ShowBox where

  show (SB s) = show s

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 ()

mapM, mapM_, and map (1)

mapM maps an "**action**" (ie function of type **a -> m b**)

over a **list [a]** 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

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 <u>useless</u>

      but waste some memory.

If you use **mapM_**, you would just get a **()**,

      but it would still print everything.

Young Won Lim
7/26/21

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 <u>cannot</u> 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_ <u>doesn't</u> <u>exist</u> because, since you <u>don't</u> have <u>any effects</u>,

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

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 <u>construct</u> a **value** of such **type**, we have

to <u>provide</u> <u>all</u> the **components** of the **tuple** <u>at once</u>.

-- one formula generating an infinity of functions

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

Young Won Lim
7/26/21

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 <u>construct</u> a **value** of such **type**, we have

     to <u>provide</u> <u>all</u> the **components** of the **tuple** <u>at once</u>.

in case of **numeric types**, <u>one</u> **numeric constant**

may be used to <u>initialize</u> **many types** <u>at once</u>.

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.

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   -- type hider

may be conceptualized as a **sum**:

Example: Sum type

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

Young Won Lim
7/26/21

Quantified Types as Products and Sums

Example: Existential type

  data ShowBox = forall s. Show s => SB s   -- type hider

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

(**SBUnit**, **SBInt**, **SBBool**, **SBIntList** ...)

A **polymorphic constructor SB**

combines all those constructors into one.

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

Young Won Lim
7/26/21

Quantification as a primitive

Young Won Lim
7/26/21

Pair type example (1)

Existential quantification is useful

for <u>defining</u> **data types** that aren't already defined.

Suppose there was <u>no</u> such thing as **pairs** built into haskell.

Existential quantification could be used to <u>define</u> them.

Young Won Lim
7/26/21

Pair type example (2)

```
{-# 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



Defining a data type c that is not already defined
```

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)

Young Won Lim
7/26/21

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

every type variable that appears on the right-hand side

must also appear on the left-hand side.

Existential type hides a type variable c on the right-hand side.

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

Pair type example (4)

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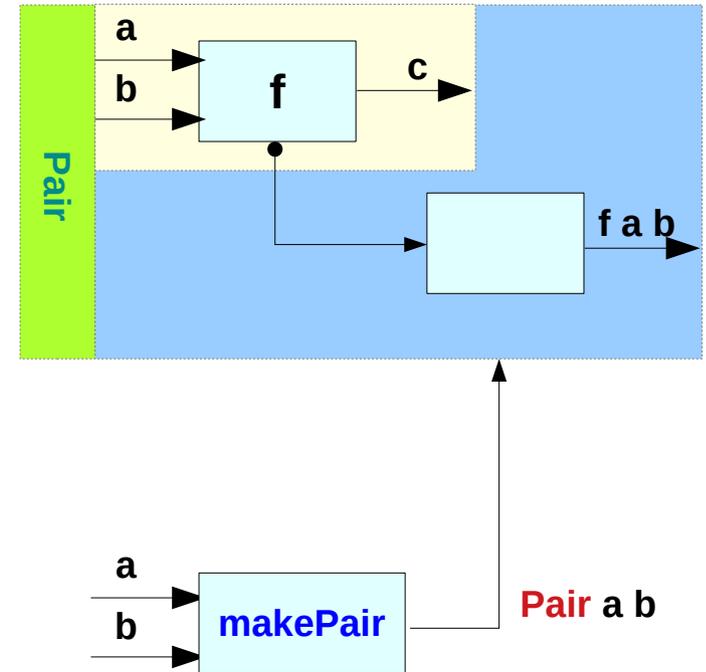
https://en.wikibooks.org/wiki/Haskell/Existentially_quantified_types

Pair type example (5)

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**

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

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
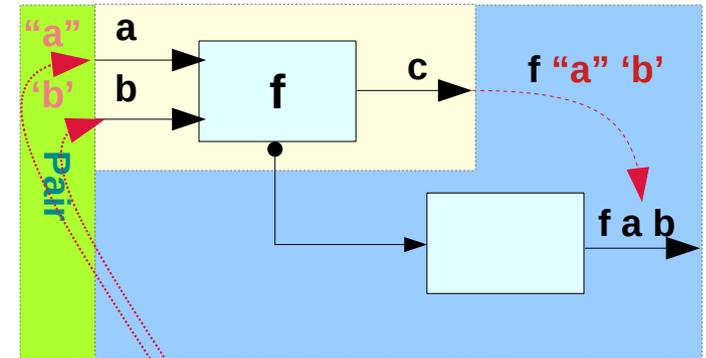
f "a" 'b'

f 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 (7)

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
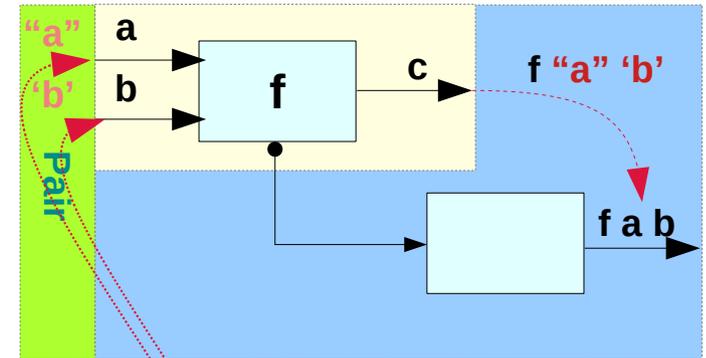
Pair a b

makePair "a" 'b'

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

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

Young Won Lim
7/26/21

Pair type example (8)

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   (\x y -> x) = (\x y -> x) "a" 'b'
runPair pair   (\x y -> y) = (\x y -> y) "a" 'b'
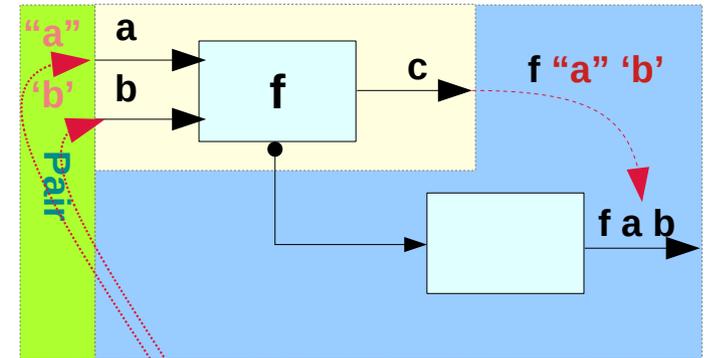
https://en.wikibooks.org/wiki/Haskell/Existentially_quantified_types

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

makePair "a" 'b'

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

runPair pair **(\x y -> x) = (\x y -> x) "a" 'b'**

runPair pair **(\x y -> y) = (\x y -> y) "a" 'b'**

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

(\x y -> x) "a" 'b'

 "a"

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

(\x y -> y) "a" 'b'

 'b'

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

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

(\x y -> x)

"a" → a
'b' → b
f
c → "a"

c → "a"

"a" → a
'b' → b
makePair
Pair a b

pair (\x y -> x)

makePair "a" 'b' (\x y -> x)

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

(\x y -> y)

"a" → a
'b' → b
f
c → "a"

c → 'b'

"a" → a
'b' → b
makePair
Pair a b

pair (\x y -> y)
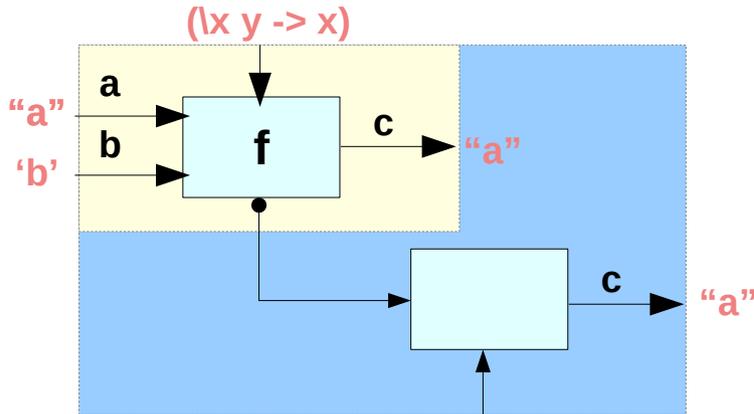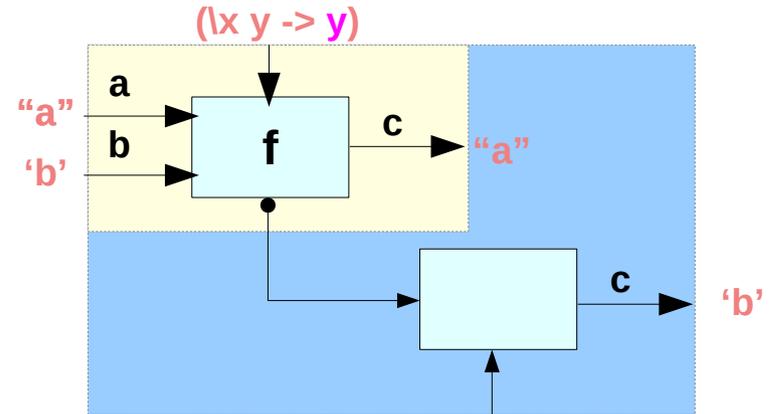
makePair "a" 'b' (\x y -> y)

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

newtype and an <u>access</u> <u>function</u>

newtype can have a named 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**

newtype can have a named 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**

newtype – constructor and unwrap functions (1)

```
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)
```

Young Won Lim
7/26/21

newtype – constructor and unwrap functions (2)

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)** }

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**)

newtype – instantiation

---

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

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

2) can <u>instantiate</u> **Parser** by providing a **parser** function

    p = Parser (\s -> Nothing)

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

  it is capable of <u>*running Parser's*</u>.

    unwrap the function

    then apply the function

Young Won Lim
7/26/21

newtype – unwrapping

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

parser :: **Parser** a -> <mark>**String -> Maybe (a, String)**</mark>

parser (**Parser** **(\s -> Nothing))** "my input"

<mark>**(\s -> Nothing))**</mark> "my input"

Nothing

You are unwrapping the function using **parse** and

then calling the unwrapped function with "myInput".

newtype – without record syntax (1)

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

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

it <u>stores</u> a **function** `String -> Maybe (a,String)`.

To <u>run</u> this parser, we will need to make an **extra function:**

runParser' **:: `Parser'` a ->** `String -> Maybe (a,String)`
runParser' **(Parser' f) i = f i**

newtype – without record syntax (2)

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 <u>curried</u>,

we can simply <u>remove</u> the reference to the <u>input</u> **i** to get:

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

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

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 <u>takes</u> a parser and <u>extracts</u> the parser function from it;

(Parser' f') -> f'

however, thanks to **currying**, **runParser''**

can still be used with two arguments.

newtype – with record syntax (1)

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**.

Young Won Lim
7/26/21

newtype – with record syntax (2)

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

parse can be used with <u>two</u> <u>arguments</u> 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

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

Young Won Lim
7/26/21

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

References

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

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