Monad P3 : Inhabitedness and Formal Logic (1E)

Young Won Lim
3/19/22

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Inhabitedness and formal logic

Void data type

The **Void** datatype is part of the Haskell standard library

Void has the following declaration

data Void

it's a **datatype**, with an empty collection of **constructors**
(this is a valid declaration).

cannot construct any value with type **Void**,
a fact that both programmers and the compiler can exploit.

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Void data type

Though a **Void** value is unconstructable,

it is still possible to write a valid Haskell **term**

which has the **Void** type.

```
aVoidTerm :: Void
aVoidTerm = aVoidTerm

-- Alternatively:
aVoidTerm = undefined
-- Or even:
aVoidTerm = error "Tried to evaluate a `Void` term"
```

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Void data type

aVoidTerm :: Void

- **aVoidTerm = aVoidTerm**
- **aVoidTerm = undefined**
- **aVoidTerm = error "Tried to evaluate a `Void` term"**

RHS is to be evaluated recursively, infinitely

All these terms are **non-terminating**.
While lazy evaluation allows them
to appear in programs without any problem,

But, any attempt to evaluate these terms will fail:
either because of an infinite loop or a runtime error.

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Inhabited types

Types with inhabitants are said to be inhabited.

Void has the property of being uninhabited,
because it has no "inhabitants"

Note that valid terminating terms can have the **Void** type.

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Inhabited types and formal logic

this **reasoning** about inhabited types looks a lot like **formal logic**.

If inhabitedness is "**truth**",

then uninhabitedness is "**falsehood**".

inhabited	⟺	**truth**
uninhabited	⟺	**false**

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Types and logic

a -> Void is uninhabited if and only if **a** is inhabited, and vice versa;

Either a b is inhabited if and only if at least one of **a**, **b** is inhabited

(a, b) is inhabited if and only if both **a** and **b** are inhabited

a -> b is uninhabited (**false**)

if and only if **a** is inhabited (**true**) and **b** is uninhabited (**false**).

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Types and logic

a -> Void	⟺	not a
Either a b	⟺	a or b
(a, b)	⟺	a and b
a -> b	⟺	a → b

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

a -> b type

For a (terminating) function with type **a -> b**,

b can be uninhabited only if **a** is uninhabited

otherwise the function could evaluate the argument of type **a**,

have it terminated, and be forced to produce

a terminating **value** of type **b**

: an impossibility.

b must be inhabited if **a** is inhabited

a	**->**	**b**
inhabited		inhabited
uninhabited		uninhabited

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Void -> a type

Void -> a

is inhabited for any choice of **a**, even uninhabited choices of **a**

a can be uninhabited only because **Void** is uninhabited.

tautology

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

a -> Void type

a -> Void

is inhabited only for choices of **a** which are uninhabited.

if **a** is uninhabited, then we can write

a **terminating term** with type **a -> Void**

a **terminating term** with type **Void -> a**

The result is: **a -> Void** is inhabited

if and only if **a** is uninhabited, and vice versa.

a	➡	a -> Void
inhabited		uninhabited
uninhabited		inhabited
True		False
False		True

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Maybe a type

We can extend this reasoning about inhabitants

to many other basic Haskell types.

Maybe a, for example, is always inhabited

by the **terminating term Nothing**,

even for uninhabited choices of **a**.

tautology

a		Maybe a
inhabited		nhabited
uninhabited		inhabited
True		True
False		True

for all a, always true : tautology

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Either a b type

Either a b is inhabited provided one of **a** or **b** is inhabited,
because you could wrap the **terminating term**
with type **a** (or **b**) in a **Left** (or **Right**) **constructor**
to give a **terminating term** of type **Either a b**.

Conversely, if **Either a b** is inhabited,
then at least one of **a** or **b** must be inhabited
(though the proof is much more difficult to summarize).

In a similar vein, the tuple type **(a, b)** is inhabited
if and only if both a, b are inhabited.

a	b	Either a b
uninhabited	uninhabited	uninhabited
uninhabited	inhabited	inhabited
inhabited	uninhabited	inhabited
inhabited	inhabited	inhabited

False	False	False
False	True	True
True	False	True
True	True	True

logical or

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

(a, b) type

In a similar vein, the tuple type **(a, b)** is inhabited if and only if both a, b are inhabited.

a	b	(a, b)
uninhabited	uninhabited	uninhabited
uninhabited	inhabited	uninhabited
inhabited	uninhabited	uninhabited
inhabited	inhabited	inhabited

False	False	False
False	True	False
True	False	False
True	True	True

logical and

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

Continuation a

Probably the most classic use of **Void** is in **CPS**.

type Continuation a = a -> Void

that is, a **Continuation** is

a **function** which never returns.

non-terminating … uninhabited

Continuation is the type version of "**not**."

https://stackoverflow.com/questions/14131856/whats-the-absurd-function-in-data-void-useful-for

Logical Not

a	➡	a -> Void		
inhabited		uninhabited	… never returning	… false
uninhabited		inhabited	… returning	… true
True		False		
False		True		

a	➡	forall r. a -> r		
inhabited		never returning	… uninhabited	… false
uninhabited		returning	… inhabited	… true

https://ivanbakel.github.io/posts/intuitionistic-logic-in-haskell/

CPS a

```
type Continuation a = a -> Void
```

From this we get a **monad** of **CPS** (corresponding to classical logic)

```
newtype CPS a = Continuation (Continuation a)
```

since Haskell is **pure**, we can't get anything out of this type.

can’t get the value **a** back

https://stackoverflow.com/questions/14131856/whats-the-absurd-function-in-data-void-useful-for

Logical double negation – **Not Not**

technically "**false**" should correspond to

an **uninhabited data type** (often called **Void**)

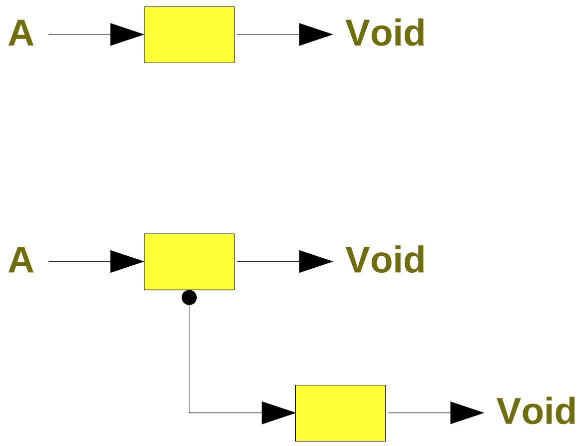
so "**not (not A)**" would be

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

Assume **forall r. r** stands for "**false**"

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

using **r** instead of **Void** lets us get value **A** back out.



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

A pure function

A **function** is called **pure** if it corresponds to a function
in the mathematical sense:
it associates each possible input value with an output value,
and does nothing else.

In particular, it has no side effects, that is to say,
invoking it produces no observable effect
other than the result it returns;
it cannot also e.g. write to disk, or print to a screen.

https://wiki.haskell.org/Pure

Referentially transparent

A **pure function** is trivially referentially transparent -

it does not depend on anything other than its parameters,

so when invoked

in a different **context** or

at a different **time**

but with the same **arguments**,

it will produce the same **result**.

A **programming language** may be called **purely functional**

if **evaluation of expressions** is **pure**.

https://wiki.haskell.org/Pure

A universally quantified type

A **universally quantified type** is a type of the form **forall a. f a**.

A **value** of that **type** can be thought of

as a **function** that takes a **type a** as its **argument**

and returns a **value** of **type f a**.

Except that in Haskell these **type arguments**

are passed implicitly by the **type system**.

This **function f** has to give you the same value

no matter which type it receives,

so the **value** is **polymorphic**.

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

Universally quantified type example I

For example, consider the type **forall a. [a]**.
A **value** of that type takes another **type a** and gives you back
a **list** of elements of that same **type a**.

There is only one possible implementation, of course.
It would have to give you the **empty list [ ]**
because **a** could be absolutely any type.

The **empty list** is the only **list value**
that is **polymorphic** in its element type
(since it has no elements).

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

Universally quantified type example II

Next, consider the type **forall a. a -> a**.

The **caller** of such a **function** provides

both a **type a** and a **value** of **type a**.

The implementation then has to return a **value** of that same **type a**.

There's only one possible implementation again.

It would have to return **the same value** that it was given.

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

Uninhabited type

Void or **(forall a. a)**

This is somewhere between a style question and a sanity check.

So I think these two types are isomorphic:

```
runRight :: Either Void b -> b
runRight' :: Either (forall a. a) b -> b
```

https://www.reddit.com/r/haskell/comments/30iq0x/void_or_forall_a_a/

Logical **or** using De Morgan's law and forall

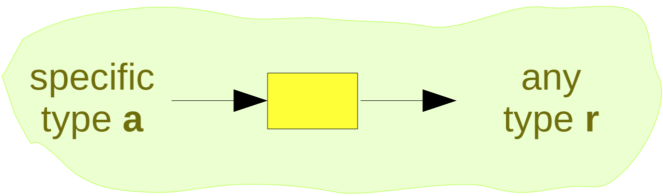
When applying **De Morgan's laws** to **quantifiers**;
function inputs are **negated**

There's an 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)

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

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

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

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

Logical **and** using De Morgan's law and forall

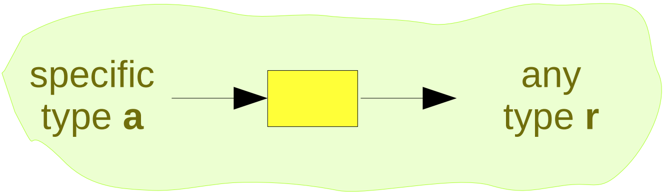
When applying **De Morgan's laws** to **quantifiers**;
function inputs are **negated**

There's an equivalence between

(a, b) … implicit universal quantification

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

which corresponds to "**A and B**"
being the same as "**not ((not A) or (not B))**".



(Not a) or (Not b)

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

Not ((Not a) or (Not b))

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

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

Existentially quantified type (1)

exists a. a

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

for all result types **r**,
given a function that
for all types **a**
takes an argument of type **a**
and returns a value of type **r**,
we can get a result of type **r**.

forall r.
forall a.
a
-> r
-> r

a function with
a specific type
must be *given*

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

Existentially quantified type (2)

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

the overall type is not **universally quantified** for **a**

it takes an **argument** that itself is **universally quantified** for **a**

(forall a. a -> r)

it can then use with whatever specific type it chooses

eg) **Int -> r**

thus, it is **existentially quantified** for **a**

exists a. a

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

Existentially quantified type (3)

The relations between **logical double-negation** and **continuation-passing style**

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

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

Existentially quantified type (4)

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

exists a. a

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

forall  a. a
```

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

exists a. a -> a (1)

An **existentially quantified type** like **exists a. a -> a** means that,

for some particular type "**a**", we can implement a **function**

whose type is **a -> a**.

for example, let's choose **Boolean** as a particular type:

func :: exists a. a -> a

func True = False

func False = True

func :: Boolean -> Boolean

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

exists a. a -> a (2)

```
func :: exists a. a -> a
func True = False
func False = True
```

the "**not**" function on **booleans**.

But we can't use it as a “**not**” **function**,

because all we know about the type "**a**" is that it exists.

any information about which type “**a**” might be has been discarded,

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

This is not very useful.

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

exists a. a -> a (3)

```
func :: exists a. a -> a
func True = False
func False = True
```

So what can we do with **func**?

we know that it's a **function**

with the same type for its **input** and **output**,

so we could compose it with itself, for example.

Essentially, the only things 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.

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

exists a. [a]

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.

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

De Morgan’s law and forall

That last bit brings us back around to universal quantifiers,

and the reason why Haskell doesn't have existential types directly

since things with existentially quantified types

can only be used with operations

that have universally quantified types,

we can write the type **exists a. a**

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

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

Haskell quantification

- the things being quantified over are **types** (ignoring certain language extensions, at least),
- logical statements are also **types**
- a "**true**" logical statement as "can be implemented".
- technically "**false**" should correspond to an **uninhabited data type** (often called **Void**)

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

Existential types and forall

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

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

exists a. a

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

forall a. a -> Int
forall a. a -> String
forall a. a -> Double

a caller chooses **type r**

The **caller** of the overall function
(a -> r) -> r
chooses any type **r**

The **body** of the overall function
(a -> r) -> r
chooses any type **a**

the **body** of the callback function
must handle for all type **a**

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

id function example

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

for any possible type **a**, *quantified over types*

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

can be implemented *a true logical statement*

id works for all **a**.

a will unify with (or will be fixed to) any type

that caller of **id** may choose.

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

A type signature and a function body

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

will be fixed when the corresponding **function**

is used (called)

in a type signature, **a** is universally quantified

but in the **body** of the function

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

we cannot inspect the **argument a**

(**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

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

Fictitious syntax ***exists a.***

An **existentially quantified type** could be better explained
using the fictitious ***exists a.*** syntax

exists a. **a -> a**

for a certain **type a**,
we can implement a **function** whose type is **a -> a**.

any function will do,
then the "**not**" function on **Bool** satisfies the type **a -> a**

```
func :: exists a. a -> a
func True = False
func False = True
```

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

Function implementations and applications

the function implementation on booleans

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

func True = False

func False = True

but we cannot use (apply) it as the "**not**" function

because all we know about the **type a** is

that it exists.

Any information about which type it might be

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

this means we can't apply **func** to any values

Existentials are always about

throwing type information away.

sometimes we want to work with **types**

that we don't know at compile time.

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 ≅ forall x. p x x -> c

a function **p** that takes 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 any one of the types **x**
that may be encoded in the **existential type**. **exists x.**

Haskell does not need an existential quantifier

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

Existential types and forall

a function that accepts 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.

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 doesn't have **existential types** directly
(***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

Parametric polymorphism (1)

universal quantification is the default

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

Parametric polymorphism (2)

Parametric polymorphism refers to
when **the type of a value** contains
one or more (unconstrained) **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 substituting those **type variables**
with **concrete types**.

data Maybe a = Just a | Nothing

Just 2.0 :: Maybe Double
Just 'a' :: Maybe Char
Just True :: Maybe Boolean

https://wiki.haskell.org/Polymorphism

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
```

```
Just 2.0   :: Maybe Double
Just 'a'   :: Maybe Char
Just True  :: Maybe Boolean
```

Polymorphic functions

```
reverse :: [a] -> [a]
fst :: (a, b) -> a
id :: a -> a
```

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

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**.

data Maybe a = Nothing | Just a

reverse :: [a] -> [a]

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 consumer of the variable's expression

can choose what it will be.

A **variable** is existentially quantified

when the consumer of the variable's expression

has to deal with the fact that the choice was made for him.

consumers of **a function**

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

Universally quantified variable:

the consumer chooses a value

Existentially quantified variable:

the choice is made for the consumer

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

Making existentials – hiding type variables

```
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

Existential wrappers – data and type constructors

```
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

type constructor data constructor

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

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 – pattern matching

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

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

X X

The **constructor** accepts any **a** we like,
but after construction we
lose the type information
and pattern matching afterwards only reveals
that there is some **a**,
but nothing regarding what it is.

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

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

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

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

Existential wrappers – constructing and using a value

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

the constructor function Something return

existentially quantified data of type Something

Something a	:: Something
a data value is *constructed*	a data value is *used*
universally quantified **a**	existentially quantified **a**

a function parameter, pattern matching

```
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)

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

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

  existential wrappers
  - return **existentially quantified data** from a **function**.
  - avoid unification of **existentials** with *outer context*
  - avoid escaping of **type variables**.

id 1 :: Int
id ‘a’ :: Char
Id 2.0 :: Double

Something 1 :: Something
Something ’a’ :: Something
Something 2.0 :: Something

findx (Something x) -> x
not possible !!!
cannot extract type variable a

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

example consumer function

foo :: Something -> Int

foo x = …

x :: Something

type variable **a** is already chosen

could be one of these

Something 1 :: Something

Something 'a' :: Something

Something 2.0 :: Something

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

Existential wrappers – similar forms

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

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

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

Assume the callback function name is r

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

• • •

```
Something 1    :: Something
Something 'a'  :: Something
Something 2.0  :: Something
  • • •
```

```
r  1    :: r
r  'a'  :: r
r  2.0  :: r
  • • •
```

```
r  1    :: Int
r  'a'  :: Int
r  2.0  :: Int
  • • •
r  1    :: Char
r  'a'  :: Char
r  2.0  :: Char
  • • •
r  1    :: Double
r  'a'  :: Double
r  2.0  :: Double
  • • •
```

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

Existential wrappers – similar forms

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

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

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

Assume the callback function name is r

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

r a	:: r

a data value is *constructed*	a data value is *used*
universally quantified **a**	existentially quantified **a**

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

• • •

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

Existential wrappers – rank-2 type

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

forall r.	argument callback *exponentially quantified **a***	-> r
	(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 hidden in the **type r**

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

Existential types and forall

we can write the type

***exists a.* a**

as

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

for all **result types r**,

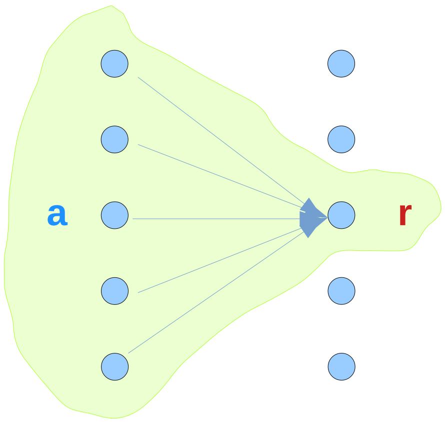
given a function **a -> r**

that takes an argument of **type a**, for all **types a**

and returns 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**

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

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**

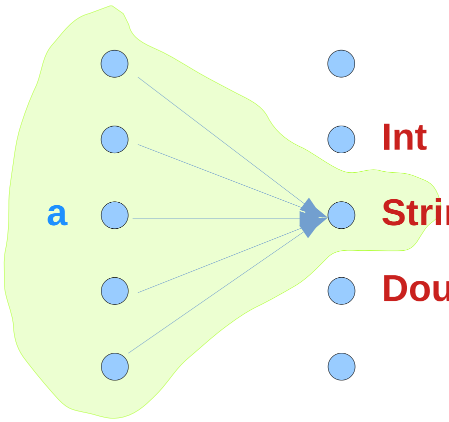
for a given **type r**

forall a. a -> Int

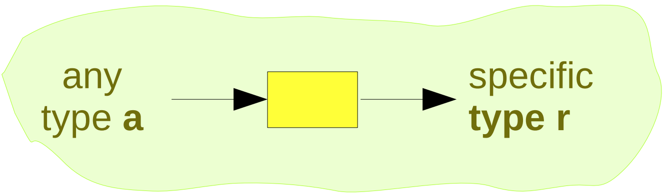
forall a. a -> String

forall a. a -> Double

a caller chooses **type r**



a caller of the overall type
determines the specific type r



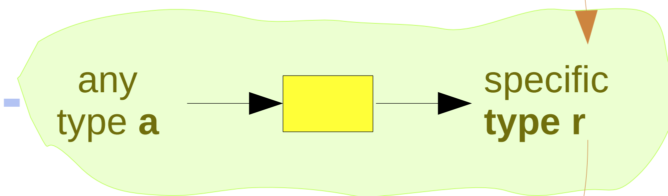
https://stackoverflow.com/questions/14299638/existential-vs-universally-quantified-types-in-haskell

Existential types and forall

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

a caller of the overall type function
chooses the specific **type r**

universally quantified	**r**



The body of the overall 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**

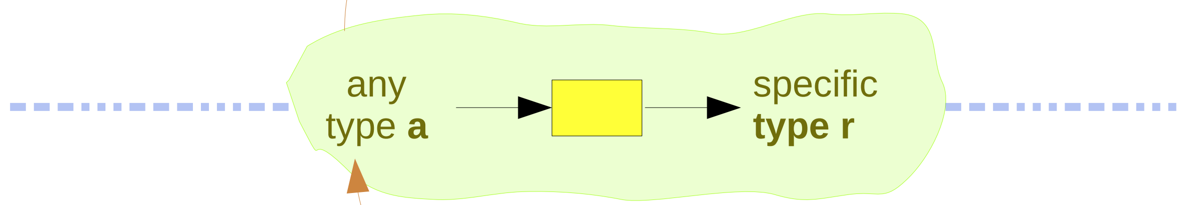
https://stackoverflow.com/questions/14299638/existential-vs-universally-quantified-types-in-haskell

Existential types and forall

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

existentially quantified **a**

a caller of the overall type function
must handle any **type a**



universally quantified **a**

The body of the overall type function
chooses the specific **type a**

The body of the callback function
must also handle any **type a**

for the **callers** of the **function**	in the **body** 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

overall function type

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

universally quantified **r**

specific type **a**

existentially quantified **a**

Body

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

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 _not_ **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 specific **type r** it chooses.

for the **callers** of the **function**	in the **body** 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

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

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** does not appear

in the type of **MkFoo** and

in the **data type** itself, Foo

Hidden

```
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**) packages an **integer** with a function

(**MkFoo 'c' isUpper**) packages a **character** with a function

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

even :: Integer -> Bool

isUpper :: Char -> Bool

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

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**.

cannot extract **val** and **fn**

```
f :: Foo -> Bool
fn ::  a -> Bool
f (MkFoo val fn) = fn val
```

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

Existentially quantified data constructors (4)

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

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

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

References

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

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