Monad P3 : Existential Types (1C)

Young Won Lim
7/29/20

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

This document was produced by using LibreOffice.

Based on

Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

Overloading

The **literals 1, 2**, etc. are often used to represent
both fixed and arbitrary precision integers.
Numeric operators such as + are often defined to work
on many different kinds of numbers.
the **equality operator** (**==** in Haskell) usually works on
numbers and many other (but not all) types.

the **overloaded behaviors** are
different for each type
in fact sometimes **undefined**, or **error**

type classes provide a structured way to control
ad hoc polymorphism, or **overloading**.

In the **parametric polymorphism**
the type truly does **not matter**

(Eq a) =>
Type class
Ad hoc polymorphism

https://www.haskell.org/tutorial/classes.html

Quantification

parametric polymorphism is useful in

defining families of types

by universally quantifying over all types.

Sometimes, however, it is necessary

to quantify over some smaller set of types,

eg. those types whose elements can be compared for equality.

ad hoc polymorphism

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

elem :: (Eq a) => a -> [a] -> Bool

https://www.haskell.org/tutorial/classes.html

Type class and parametric polymorphism

type classes can be seen as providing a structured way
to quantify over a constrained set of types

the **parametric polymorphism** can be viewed
as a kind of **overloading** too!

parametric polymorphism

an **overloading** occurs implicitly over all types

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

ad hoc polymorphism

a **type class** for a constrained set of types

elem :: (Eq a) => a -> [a] -> Bool

https://www.haskell.org/tutorial/classes.html

Parametric polymorphism (1) definition

Parametric polymorphism refers to

when the **type** of a **value** contains

one or more (**unconstrained**) **type variables**,

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

that results from substituting those variables with **concrete types**.

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

https://wiki.haskell.org/Polymorphism

Parametric polymorphism (2) unconstrained type variable

In Haskell, this means any type in which a **type variable**,

denoted by a name in a type

beginning with a **lowercase letter**,

appears **without constraints**

(i.e. does not appear to the left of a **=>**).

In **Java** and some similar languages,

generics (roughly speaking) fill this role.

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

https://wiki.haskell.org/Polymorphism

Parametric polymorphism (3) examples

For example, the function **id :: a -> a** contains
an **unconstrained type variable a** in its type,
and so can be used in a context requiring

Char -> Char or

Integer -> Integer or

(Bool -> Maybe Bool) -> (Bool -> Maybe Bool) or

any of a literally infinite list of other possibilities.

Likewise, the empty **list [] :: [a]** belongs to every list type,

and the polymorphic function **map :: (a -> b) -> [a] -> [b]**
may operate on any function type.

https://wiki.haskell.org/Polymorphism

Parametric polymorphism (4) multiple appearance

Note, however, that if a single **type variable** appears multiple times,
it must take the same type everywhere it appears,
so e.g. the result type of **id** must be the same as the argument type,
and the input and output types of the function
given to **map** must match up with the list types.

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

https://wiki.haskell.org/Polymorphism

Parametric polymorphism (5) parametricity

Since a parametrically polymorphic value does not "know"
anything about the **unconstrained type variables**,
it must behave the same regardless of its type.

This is a somewhat limiting
but extremely useful property
known as **parametricity**

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

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (1)

Ad-hoc polymorphism refers to

when a **value** is able to adopt any one of several **types**

because it, or a value it uses, has been given

a separate definition for each of those **types**.

the **+ operator** essentially does something entirely different

when applied to floating-point values

as compared to when applied to integers

elem :: (Eq a) => a -> [a] -> Bool

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (2)

in languages like C, **polymorphism** is restricted to
only *built-in* **functions** and **types**.

Other languages like C++ allow programmers
to provide their own **overloading**,
supplying **multiple definitions** of a **single function**,
to be disambiguated by the **types** of the **arguments**

In Haskell, this is achieved via the system of
type classes and **class instances**.

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (3)

Despite the similarity of the name,
Haskell's **type classes** are quite different from
the **classes** of most object-oriented languages.

They have more in common with **interfaces**,
in that they specify a series of **methods** or **values**
by their **type signature**,
to be implemented by an **instance declaration**.

```
class Eq a where
  (==)        :: a -> a -> Bool

instance Eq Integer where
  x == y     =  x `integerEq` y

instance Eq Float where
  x == y     =  x `floatEq` y
```

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (4)

So, for example, if **my type** can be compared for **equality**
(most types can, but some, particularly function types, cannot)
then I can give **an instance declaration** of the **Eq class**

All I have to do is specify
the behaviour of the **== operator** on **my type**,
and I gain the ability to use all sorts of functions
defined using **== operator**, e.g.
checking if a value of **my type** is present in a list,
or looking up a corresponding value in a list of pairs.

```
class Eq a where
  (==)        :: a -> a -> Bool

instance Eq Integer where
  x == y    =  x `integerEq` y

instance Eq Float where
  x == y    =  x `floatEq` y
```

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (5)

Unlike the **overloading** in some languages,

overloading in Haskell is not limited to **functions**

– **minBound** is an example of an **overloaded value**,

as a **Char**, it will have value **'\NUL**'

as an **Int** it might be **-2147483648**

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (6)

Haskell even allows **class instances** to be <u>defined</u> for **types** which are themselves **polymorphic** (either ad-hoc or parametrically).

So for example, an **instance** can be defined of **Eq** that says "if **a** has an **equality operation**, then **[a]** has one".

Then, of course, **[[a]]** will automatically also have an instance, and so **complex compound types** can have instances built for them out of the instances of their components.

https://wiki.haskell.org/Polymorphism

Ad hoc polymorphism (7)

You can recognise the presence of **ad-hoc polymorphism**
by looking for **constrained type variables**:
that is, variables that appear <u>to the left of **=>**</u>,
like in **elem :: (Eq a) => a -> [a] -> Bool**.

Note that **lookup :: (Eq a) => a -> [(a,b)] -> Maybe b**
exhibits both **parametric** (in **b**) and **ad-hoc** (in **a**) **polymorphism**.

https://wiki.haskell.org/Polymorphism

Parametric and ad hoc polymorphism

Parametric polymorphism	**ad hoc polymorphism**
Type variables	**Type calsses**
(a, b, etc)	**(Eq, Num, etc)**
Universal	**Existential?**
Compile time	**Runtime (also)**
C++ templates	**Classical**
Java generics	**(ordinary OO)**

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

Polymorphic data types and functions

```
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)
data Either a b = Left a | Right b


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

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

Polymorphic Types

types that are universally quantified in some way over all types.

polymorphic type expressions essentially describe families of types.

For example, **(forall a) [a]** is the family of types
consisting of, for every **type a**, the **type of lists of a**.

- lists of integers (e.g. **[1,2,3]**),
- lists of characters (**['a','b','c']**),
- even lists of lists of integers, etc.,

(Note, however, that [2,'b'] is not a valid example,
since there is *no single type* that contains both 2 and 'b'.)

https://www.haskell.org/tutorial/goodies.html

Polymorphic Types

Identifiers such as **a** above are called **type variables**,
and are uncapitalized to distinguish them
from specific types such as **Int**.

since Haskell has only universally quantified **types**,
there is no need to explicitly write out the symbol
for **universal quantification**,
and thus we simply write **[a]** in the example above.

In other words, all **type variables** are implicitly universally quantified

https://www.haskell.org/tutorial/goodies.html

Polymorphic Types

Lists are a commonly used data structure in functional languages,
and are a good vehicle for explaining the principles of polymorphism.

The list **[1,2,3]** in Haskell is actually shorthand for the list **1:(2:(3:[]))**,
where **[]** is the empty list and **:** is the **infix operator**
that adds its first argument to the front of its second argument (a list).

Since **:** is right associative, we can also write this list as **1:2:3:[]**.

https://www.haskell.org/tutorial/goodies.html

Polymorphic Types

```
length              :: [a] -> Integer
length []            =  0
length (x:xs)        =  1 + length xs

length [1,2,3]     =>    3
length ['a','b','c']       =>    3
length [[1],[2],[3]]       =>    3
```

an example of a polymorphic function.

It can be applied to a list containing elements of any type,

for example **[Integer]**, **[Char]**, or **[[Integer]]**.

https://www.haskell.org/tutorial/goodies.html

Polymorphic Types

```
length          :: [a] -> Integer
length []        =  0
length (x:xs)    =  1 + length xs
```

The left-hand sides of the equations contain patterns such as [] and x:xs.

In a function application these patterns are matched against actual parameters in a fairly intuitive way

https://www.haskell.org/tutorial/goodies.html

Polymorphic Types

```
length              :: [a] -> Integer
length []           =  0
length (x:xs)       =  1 + length xs
```

[] only matches the empty list,

x:xs will successfully match any list with at least one element,

binding x to the first element and xs to the rest of the list

If the match succeeds,

the right-hand side is evaluated

and returned as the result of the application.

If it fails, the next equation is tried,

and if all equations fail, an error results.

https://www.haskell.org/tutorial/goodies.html

Polymorphic Types

Function head returns the first element of a list,
function tail returns all but the first.

```
head           :: [a] -> a
head (x:xs)    =  x

tail           :: [a] -> [a]
tail (x:xs)    =  xs
```

Unlike length, these functions are not defined
for all possible values of their argument.
A runtime error occurs when these functions
are applied to an empty list.

https://www.haskell.org/tutorial/goodies.html

Polymorphic Types

With polymorphic types, we find that
some types are in a sense strictly more general than others
in the sense that the set of values they define is larger.

For example, the type **[a]** is more general than **[Char]**.
In other words, the latter type can be derived from the former
by a suitable substitution for **a**.

https://www.haskell.org/tutorial/goodies.html

Polymorphic Types

With regard to this **generalization ordering**,
Haskell's type system possesses two important properties:

First, every well-typed expression is guaranteed
to have a **unique principal type** (explained below),

and second, the **principal type** can be inferred automatically.

In comparison to a monomorphically typed language such as C,
the reader will find that polymorphism improves expressiveness,
and **type inference** lessens the burden of types on the programmer.

https://www.haskell.org/tutorial/goodies.html

Polymorphic Types

An expression's or function's **principal type** is
the least general type that, intuitively,
"contains all instances of the expression".

For example, the principal type of head is **[a]->a**; **[b]->a**, **a->a**,
or even **a** are correct types, but too general,
whereas something like **[Integer]->Integer** is too specific.
The existence of unique **principal types** is the hallmark feature
of the **Hindley-Milner type system**,
which forms the basis of the type systems of Haskell,
ML, Miranda, ("Miranda" is a trademark of Research Software, Ltd.)
and several other (mostly functional) languages.

https://www.haskell.org/tutorial/goodies.html

Explicitly Quantifying Type Variables

to explicitly bring fresh **type variables** into **scope**.

Example: **Explicitly quantifying** the **type variables**

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

for any combination of types **a** and **b**

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

then it's valid to say that map has the type

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

Here we are **instantiating** the general type of **map**

to a more specific type.

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

Implicit **forall**

any introduction of a **lowercase type parameter**

<u>implicitly</u> begins with a **forall** keyword,

Example: Two equivalent type statements

id :: a -> a

id :: forall a . a -> a

We can apply <u>additional</u> **constraints**

on the quantified **type variables**

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

Existential Types

Normally when creating a new type

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

every **type variable** that appears on the right-hand side

must also appear on the left-hand side.

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

Existential types are a way of escaping

Existential types can be used for several different purposes.

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

https://wiki.haskell.org/Existential_type

Type Variable Example – (1) error

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

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

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

In Haskell98, you would have to write

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

https://wiki.haskell.org/Existential_type

Type Variable Example – (2) explicit type signature

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

so long as it belongs to some particular class.

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

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

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

will invoke the dreaded monomorphism restriction.

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

https://wiki.haskell.org/Existential_type

Type Variable Example – (3) existential type

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

foo :: Worker Int Int
```

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

https://wiki.haskell.org/Existential_type

Type Variable Example – (4) characteristics

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

- 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.)

https://wiki.haskell.org/Existential_type

Type Variable Example – (4) characteristics

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

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

https://wiki.haskell.org/Existential_type

Hiding a type

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

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

https://wiki.haskell.org/Existential_type

Conversion to less a specific type

Note: You can use **existential types**

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

into a **less specific one**.

There is no way to perform the reverse conversion!

https://wiki.haskell.org/Existential_type

A heterogeneous list example

This illustrates **creating a heterogeneous list**,

all of whose members implement "**Show**",

and progressing through that list to show these items:

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

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

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

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

https://wiki.haskell.org/Existential_type

References

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

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