

Monad P3 : Types (1A)

Copyright (c) 2016 - 2020 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

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

This document was produced by using LibreOffice.

Based on

Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

data

data - creates new **algebraic type** with **value constructors**

- Can have several value constructors
- **Value constructors** are lazy
- **Values** can have several fields
- Affects both compilation and runtime, have runtime overhead
- Created type is a distinct new type
- Can have its own **type class instances**
- When pattern matching against **value constructors**,
WILL be evaluated at least to weak head normal form (**WHNF**) *
- Used to create *new data type*
(example: `Address { zip :: String, street :: String }`)

https://www.reddit.com/r/haskell/comments/6xri4d/whats_the_difference_between_newtype_type_and_data/

newtype

newtype - creates new “decorating” type with **value constructor**

- Can have only one value constructor
- **Value constructor** is strict
- **Value** can have only one field
- Affects only compilation, no runtime overhead
- Created type is a distinct new type
- Can have its own **type class instances**
- When pattern matching against **value constructor**,
CAN be not evaluated at all *
- Used to create *higher level concept* based on existing type
with distinct set of supported operations or
that is not interchangeable with original type
(example: Meter, Cm, Feet is Double)

https://www.reddit.com/r/haskell/comments/6xri4d/whats_the_difference_between_newtype_type_and_data/

type

type - creates an **alternative name** (synonym) for a **type** (typedef in C)

- No value constructors
- No fields
- Affects only compilation, no runtime overhead
- No new type is created (only a new name for existing type)
- Can NOT have its own **type class instances**
- When pattern matching against **data constructor**, behaves the same as original type
- Used to create higher level concept based on existing type with the same set of supported operations (example: String is [Char])

https://www.reddit.com/r/haskell/comments/6xri4d/whats_the_difference_between_newtype_type_and_data/

data, newtype, type

data: zero or more **constructors**,
each can contain zero or more **values**.

newtype: similar to above
but exactly one **constructor**
and only one **value** in that constructor,
and has the exact **same runtime representation**
as the value that it stores.

type: **type synonym**, compiler more or less
forgets about it once it is expanded.

https://www.reddit.com/r/haskell/comments/6xri4d/whats_the_difference_between_newtype_type_and_data/

data (lazy), newtype (strict)

Both **newtype** and the single-constructor data introduce a single data constructor, but the **data constructor** introduced by **newtype** is **strict** and the **data constructor** introduced by **data** is **lazy**.

```
data    D = D Int    -- lazy
newtype N = N Int    -- strict
```

Then **N undefined** is equivalent to **undefined** and causes an **error** when **evaluated**.

But **D undefined** is not equivalent to **undefined**, and it can be **evaluated** as long as you don't try to peek inside.

https://www.reddit.com/r/haskell/comments/6xri4d/whats_the_difference_between_newtype_type_and_data/

Data definition without data constructors (1)

a **data definition** without **data constructors**

cannot be **instantiated**

data B

a new **type constructor B**,

but no **data constructors** to produce **values** of **type B**

In fact, such a data type is declared in the Haskell base: **Void**

```
ghci> import Data.Void
```

```
ghci> :i Void
```

```
data Void -- Defined in 'Data.Void'
```

<https://stackoverflow.com/questions/45385621/data-declaration-with-no-data-constructor-can-it-be-instantiated-why-does-it-c>

Data definition without data constructors (2)

Being able to have **uninhabited types** turns out to be useful in some areas

passing an **uninhabited type** as a **type parameter** to another **type constructor**

<https://stackoverflow.com/questions/45385621/data-declaration-with-no-data-constructor-can-it-be-instantiated-why-does-it-c>

Data definition with data constructors

```
data B = String
```

defines

a **type constructor B** and
a **data constructor String**,
both taking no arguments.

Note that the **String** you define is in the **value namespace**,
so is different from the usual **String type constructor**.

```
ghci> data B = String
```

```
ghci> x = String
```

```
ghci> :t x
```

```
x :: B
```

<https://stackoverflow.com/questions/45385621/data-declaration-with-no-data-constructor-can-it-be-instantiated-why-does-it-c>

Algebraic type

This is a type where we specify the shape of each of the elements. Wikipedia has a thorough discussion. "Algebraic" refers to the property that an Algebraic Data Type is created by "algebraic" operations. The "algebra" here is "sums" and "products":

"sum" is alternation ($A \mid B$, meaning A or B but not both)

"product" is combination ($A B$, meaning A and B together)

http://wiki.haskell.org/Algebraic_data_type

Algebraic type

Examples:

`data Pair = P Int Double` is a pair of numbers, an `Int` and a `Double` together. The tag `P` is used (in constructors and pattern matching) to combine the contained values into a single structure that can be assigned to a variable.

`data Pair = I Int | D Double` is just one number, either an `Int` or else a `Double`. In this case, the tags `I` and `D` are used (in constructors and pattern matching) to distinguish between the two alternatives.

http://wiki.haskell.org/Algebraic_data_type

Algebraic type

Sums and products can be repeatedly combined into an arbitrarily large structures.

Algebraic Data Type is not to be confused with *Abstract* Data Type, which (ironically) is its opposite, in some sense. The initialism "ADT" usually means *Abstract* Data Type, but GADT usually means Generalized *Algebraic* Data Type.

http://wiki.haskell.org/Algebraic_data_type

Decorate type

Tip: types, functions and values

Type variables in Haskell are typically named starting at a, b, etc. They are sometimes (but not often) decorated with numbers like a1 or b3.

http://wiki.haskell.org/Algebraic_data_type

Decorate type

Functions used as higher-order arguments are typically named starting at f, g, etc. They will sometimes be decorated with numbers like type variables and will also be decorated with the ' character like g'. You would read this latter example as "Jee-prime" and it is typically a function that is in some way related to g used as a helper or the like. Occasionally functions may be given names that are not on this continuum as an aide memoir, for example a function parameter used internally as a predicate may be given the name p.

http://wiki.haskell.org/Algebraic_data_type

Decorate type

Arguments to functions, or variables used exclusively inside short functions, are often given names starting at x, y, etc., again occasionally decorated by numbers. Other single-letter variable names may be chosen if they can act as a mnemonic for their role such as using a variable named p for a value known to be prime.

Note that these are guidelines and not rules. Any of them can and will be ignored, modified and/or abused in any given piece of Haskell code. (A quick look at the Standard Prelude as provided in the Haskell 98 Report should be convincing enough for this.)

http://wiki.haskell.org/Algebraic_data_type

Type class instances

Type classes allow us

to declare which types are **instances** of which class, and

to provide **definitions** of the overloaded operations
associated with a **class**.

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

Type class instances

For example, let's define a **type class** containing an **equality operator**:

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

Eq is the **name** of the **class** being defined,
== is the single **operation** in the **class**.

a **type a** is an **instance** of the **class Eq**
if there is an (**overloaded**) **operation ==**,
of the appropriate **type**, defined on it.

(Note that == is only defined on pairs of objects of the same type.)

class	Eq	type
		a
	class name	class instance

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

Type class instances

Eq a expresses a **constraint** that
a **type a** must be an **instance** of the **class Eq**

Eq a
is not a **type expression**
expresses a **constraint** on a **type**
called a **context**
placed at the front of **type expressions**

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

Type class instances

For example, the effect of the above class declaration is to assign the following type to `==`:

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

for every **type a** that is an **instance** of the **class Eq**,
`==` has type **a->a->Bool**

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

for every **type a** that is an **instance** of the **class Eq**,
`elem` has type **a->[a]->Bool**

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

Type class instances

An **instance declaration** specifies

which types are **instances** of the **class Eq**, and
the actual behavior of == on each of those **types**

instance Eq Integer where

$x == y = x \text{ `integerEq` } y$

the **definition** of == is called a **method**.

integerEq happens to be the **primitive function**

in general, any valid expression for a function definition

instance Eq integer

class name type
class instance

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

Type class instances

instance Eq Integer where

$x == y = x \text{ `integerEq` } y$

the **type** Integer is an **instance** of the **class** Eq

the definition of the **method** ==

instance Eq Float where

$x == y = x \text{ `floatEq` } y$

the **type** Float is an **instance** of the **class** Eq

the definition of the **method** ==

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

Type class instances

simply substituting **type class** for **class**, and **type** for **object**, yields a valid summary of Haskell's **type class mechanism**:

"**Classes** capture common sets of operations.

A particular **object** may be an **instance** of a **class**, and will have a **method** corresponding to each **operation**.

Classes may be arranged **hierarchically**, forming notions of **superclasses** and **sub classes**, and permitting **inheritance** of operations/methods.

A **default method** may also be associated with an operation."

Haskell	OOP
type class	class
type	object

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

Type class instances

In contrast to OOP, it should be clear that **types** are not objects, and in particular there is no notion of an object's or type's **internal mutable state**.

An advantage over some OOP languages is that **methods** in Haskell are completely type-safe: any attempt to apply a **method** to a **value** whose **type** is not in the required **class** will be detected at compile time instead of at runtime.

In other words, **methods** are not "looked up" at runtime but are simply passed as **higher-order functions**.

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

Type class instances

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.

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

Type class instances

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

Indeed, we can think of **parametric polymorphism** as a kind of **overloading** too!

an **overloading** occurs implicitly over all types
a **type class** for a constrained set of types

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

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., are all members of this family.

(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 $[a] \rightarrow a$; $[b] \rightarrow a$, $a \rightarrow a$, or even a are correct types, but too general, whereas something like $[Integer] \rightarrow Integer$ 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**
implicitly begins with a **forall** keyword,

Example: Two equivalent type statements

id :: a -> a

id :: forall a . a -> a

We can apply additional 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

Bottom

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

Bottom

bottom in Haskell specifically called **undefined**.

This is only one form of it

though technically **bottom** is also

a non-terminating computation, such as `length [1..]`

bottom is used to represent an expression which is

- not computable
- runs forever
- never returns a value
- throws an exception
- etc.

<https://stackoverflow.com/questions/26428828/what-does-%E2%8A%A5-mean-in-the-strictness-monad-from-p-wadlers-paper>

Bottom represents computations

The term **bottom** refers to

a **computation** which never completes successfully.

a **computation** that fails due to some kind of error,

a **computation** that just goes into an infinite loop

(without returning any data).

The mathematical symbol for bottom is ' \perp '.

In plain ASCII, '_'.

<https://wiki.haskell.org/Bottom>

Bottom – a member of any type

Bottom is a **member** of any type,
even the trivial type **()** or
the equivalent simple type:
data Unary = Unary

<https://wiki.haskell.org/Bottom>

Bottom – definitions

Bottom can be expressed in Haskell thus:

```
bottom = bottom
```

```
bottom = error "Non-terminating computation!"
```

Indeed, the Prelude exports a function

```
undefined = error "Prelude.undefined"
```

Other implementations of Haskell, such as Gofer, defined bottom as:

```
undefined | False = undefined
```

The type of bottom is arbitrary, and defaults to the most general type:

```
undefined :: a
```

<https://wiki.haskell.org/Bottom>

Bottom – Usage

As **bottom** is an **inhabitant** of every **type** *a value of every type*

bottoms can be used wherever a value of that type would be.

This can be useful in a number of circumstances:

-- For leaving a **todo** in your program to come back to later:

```
foo = undefined
```

-- When dispatching to a **type class instance**:

```
print (sizeof (undefined :: Int))
```

-- When using **laziness**:

```
print (head (1 : undefined))
```

<https://wiki.haskell.org/Bottom>

Bottom Rule

if x is computable,

then **strict** $f\ x$ evaluates to $f\ x$,

but if x is not computable,

then **strict** $f\ x$ evaluates to "not computable".

undefined

undefined

for example, $f\ x = 2 * x$.

consider $f\ (1 / 0)$

can't evaluate it because you can't evaluate $(1 / 0)$

$(1 / 0)$ not computable

$f\ (1 / 0)$ not computable

<https://stackoverflow.com/questions/26428828/what-does-%E2%8A%A5-mean-in-the-strictness-monad-from-p-wadlers-paper>

strict f x

Sometimes it is necessary to control order of evaluation in a **lazy** functional program.

Use the computable function **strict**,
strict f x = if x \neq \perp then f x else \perp .

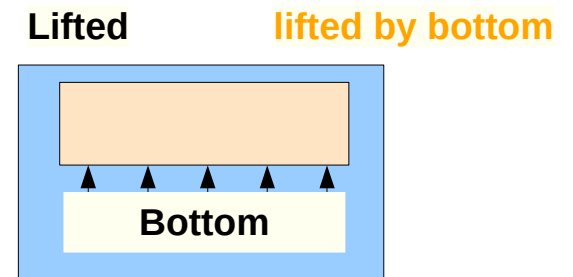
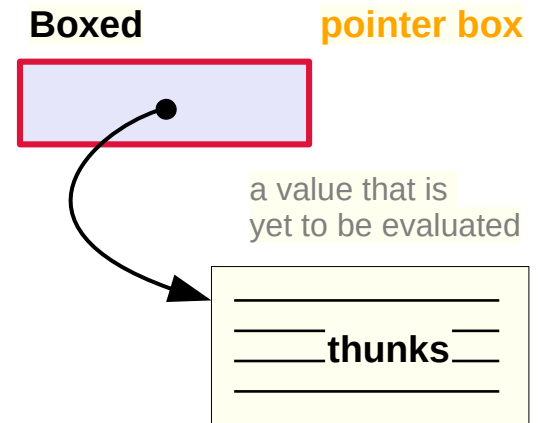
Operationally, **strict f x** is reduced by
first reducing **x** to **weak head normal form (WHNF)**
and then reducing the application **f x**.

Alternatively, it is safe to reduce **x** and **f x** in parallel,
but not allow access to the result until **x** is in **WHNF**.

<https://stackoverflow.com/questions/26428828/what-does-%E2%8A%A5-mean-in-the-strictness-monad-from-p-wadlers-paper>

Classifying types – Summary

Boxed	a pointer to a heap object.
Unboxed	no pointer
Lifted	bottom as an element.
Unlifted	no extra values .
Algebraic	<u>one or more constructors</u> ,
Primitive	a built-in type

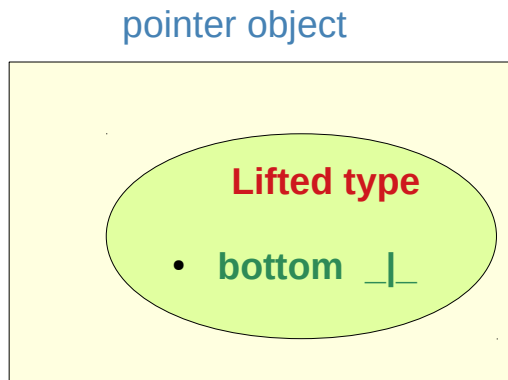


Undefined
Infinite loop
Exception

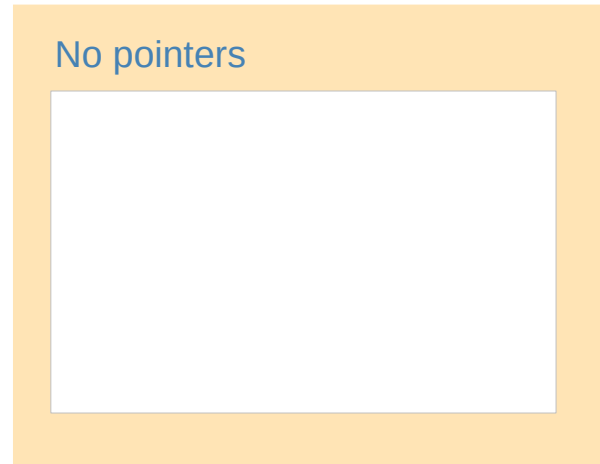
<https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/type-type>

(Un)Lifted and (Un)Boxed types

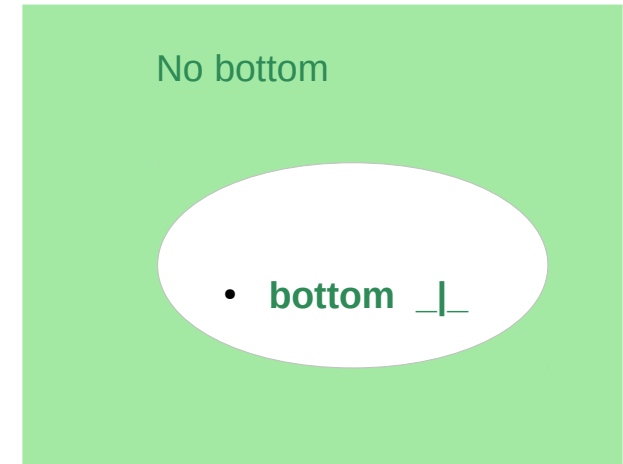
Boxed type



Unboxed type



Unlifted type



Lifted type \longrightarrow Boxed type
kind *

Unboxed type \longrightarrow Unlifted type
kind #

<https://stackoverflow.com/questions/39985296/what-are-lifted-and-unlifted-product-types-in-haskell>

Bottom in a programming language

programming language :

bottom refers to a value that is less defined than any other.

It's common to assign the **bottom value** to every computation that either produces an **error** or **fails to terminate**,

because trying to distinguish these conditions which greatly weakens the mathematics and complicates program analysis.

<https://stackoverflow.com/questions/26428828/what-does-%E2%8A%A5-mean-in-the-strictness-monad-from-p-wadlers-paper>

Bottom in an order theory

order theory (particularly **lattice theory**) :

The **bottom** element of a partially ordered set,
if one exists, is the one that precedes all others.

<https://stackoverflow.com/questions/26428828/what-does-%E2%8A%A5-mean-in-the-strictness-monad-from-p-wadlers-paper>

Bottom in a lattice theory

Lattice theory

the logical **false** value

is the **bottom element** of a **lattice** of **truth values**,
and **true** is the **top element**

classical logic

these are the only two – **true** and **false**

but one can also consider logics

with infinitely many truthfulness values,

such as **intuitionism** and various forms of **constructivism**.

These take the notions in a rather different direction.

<https://stackoverflow.com/questions/26428828/what-does-%E2%8A%A5-mean-in-the-strictness-monad-from-p-wadlers-paper>

Bottom in a standard Boolean logic

standard Boolean logic

the symbol \perp read **falsum** or **bottom**,
is simply a statement which is always false,
the equivalent of the false constant in programming languages.

The form is an inverted (upside-down) version of the symbol \top
(**verum** or **top**), which is the equivalent of true -
and there's mnemonic value in the fact that the symbol looks
like a capital letter T.

<https://stackoverflow.com/questions/26428828/what-does-%E2%8A%A5-mean-in-the-strictness-monad-from-p-wadlers-paper>

Bottom – verum an falsum

The names **verum** and **falsum** are Latin for "**true**" and "**false**"; the names "**top**" and "**bottom**" come from the use of the symbols in the **theory of ordered sets**, where they were chosen based on the location of the horizontal crossbar

<https://stackoverflow.com/questions/26428828/what-does-%E2%8A%A5-mean-in-the-strictness-monad-from-p-wadlers-paper>

Bottom – computability theory

computability theory, \perp is also
the value of an **uncomputable computation**,
so you can also think of it as the **undefined value**.

It doesn't matter why the computation is uncomputable -
whether because it has **undefined inputs**,
or **never terminates**, or whatever.

it defines **strict** as a **function**
that makes any computation (another function) **undefined**
whenever its inputs (arguments) are **undefined**.

<https://stackoverflow.com/questions/26428828/what-does-%E2%8A%A5-mean-in-the-strictness-monad-from-p-wadlers-paper>

WHNF (Weak Head Normal Form)

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

Normal Form

An **expression** in **normal form**

is fully evaluated,

contains no un-evaluated thunks

no sub-expression could be evaluated any further

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

Normal Form Examples

in normal form:

42

(2, "hello")

$\lambda x \rightarrow (x + 1)$

not in normal form:

1 + 2 -- we could evaluate this to 3

$(\lambda x \rightarrow x + 1) 2$ -- we could apply the function

"he" ++ "llo" -- we could apply the (++)

(1 + 1, 2 + 2) -- we could evaluate 1 + 1 and 2 + 2

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

Head – outermost function application

The **head** in **WHNF** (Weak Head Normal Form) does not refer to the **head** of a **list**, but to the **outermost function application**.

thunks

generally refer to **unevaluated expressions**

HNF (Head normal form) is irrelevant for Haskell.

It differs from **WHNF** in that

the **bodies** of lambda expressions are also **evaluated** *to some extent*.

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

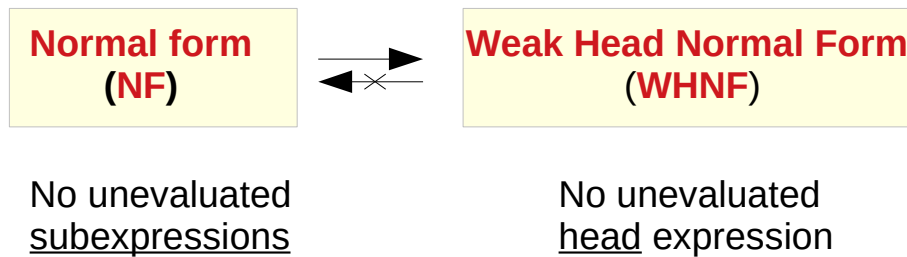
NF is WHNF

An **expression** in **WHNF** (weak head normal form)

has been evaluated to the outermost

data constructor or **lambda abstraction** (the **head**).

sub-expressions may or may not have been evaluated.



<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

Weak Head Normal Form Test

To determine whether an expression is in weak head normal form, we only have to look at the **outermost part** of the expression.

If the **outermost** part of the expression

is a **data constructor** or a **lambda**,

then it is in **weak head normal form**.

is a **function application**,

then it is not in **weak head normal form**.

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

Evaluation Example

outermost application

from left to right;

lazy evaluation.

Example:

take 1 (1:2:3:[]) => { apply take }

1 : take (1-1) (2:3:[]) => { apply (-) }

1 : take 0 (2:3:[]) => { apply take }

1 : []

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

Reduced Normal Form

evaluation stops when there are

no more function applications left to replace.

the result is in **normal form**

(or reduced normal form, **RNF**).

no unevaluated subexpressions

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

Lazy Evaluation

No matter in which **order** you evaluate an expression,
you will always end up with the same normal form
(but only if the evaluation terminates).

There is a slightly different description for **lazy evaluation**.

Namely, it says that you should evaluate everything
to weak head normal form (WHNF) only.

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

The head of the expression

There are precisely **three cases** for an expression to be in **WHNF**:

A **constructor**: `constructor expression_1 expression_2 ...`

A **built-in function** with too few arguments, like `(+) 2` or `sqrt`

A **lambda-expression**: `\x -> expression`

In other words, the **head** of the **expression**
(i.e. the **outermost function application**)
cannot be evaluated any further,
but the function argument may contain
unevaluated expressions.

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

Weak Head Normal Form Test

in weak head normal form:

- `(1 + 1, 2 + 2)` -- the outermost part is the data constructor (`,`)
- `\x -> 2 + 2` -- the outermost part is a lambda abstraction
- `'h' : ("e" ++ "llo")` -- the outermost part is the data constructor (`:`)

As mentioned, all the normal form expressions listed above are also in weak head normal form.

not in weak head normal form:

- `1 + 2` -- the outermost part here is an application of (`+`)
- `(\x -> x + 1) 2` -- the outermost part is an application of (`\x -> x + 1`)
- `"he" ++ "llo"` -- the outermost part is an application of (`++`)

in normal form:

```
42
(2, "hello")
\x -> (x + 1)
```

<https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form>

References

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