Monad P3 : Types (1A)

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, newtype, type

- data is for making new, complicated **types**,

  data Person = Bob | Cindy | Sue

- newtype is for “decorating” or making a copy of an existing type,

  newtype Dollar = Dollar Double

- type is for renaming a type,

  type Polygon = [Point]

just makes **Dollar** be equivalent to **Double** and

is mostly only used for making certain code easier to read.

https://webcache.googleusercontent.com/search?q=cache:_5Dl-cKznPcJ:https://andre.tips/wmh/newtype/
+&cd=12&hl=en&ct=clnk&gl=us

data, newtype, type

data: zero or more **constructors**,

each can contain zero or more **values**.

newtype: similar to **data**

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

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 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

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

Newtype – wrap

wrap one type in another type and

A new **type** is *almost the same* as an original **type**

represented the same as the original type in memory,

zero runtime penalty for using a **newtype**

newtype Dollars = Dollars Int

to convert the *uninformative type* **Int**

into a more *descriptive type*, **Dollars**.

to make a value of **Dollars**,

Dollars 3

https://webcache.googleusercontent.com/search?q=cache:_5Dl-cKznPcJ:https://andre.tips/wmh/newtype/+&cd=12&hl=en&ct=clnk&gl=us

Newtype – examples using **data**

a **Dollar** type, a **Yen** type, and a **Euro** type

all just **wrappers** around **Double**

```
data Dollar  = Dollar  Double     deriving (Read, Show)
data Euro    = Euro    Double     deriving (Read, Show)
data Yen   = Yen       Double     deriving (Read, Show)
```

Let a **Currency** **typeclass** has

a **convertToDollars**

and **convertFromDollars** function.

Then, let’s **add**, **subtract**, and **multiply** the **currency**

https://webcache.googleusercontent.com/search?q=cache:_5Dl-cKznPcJ:https://andre.tips/wmh/newtype/+&cd=12&hl=en&ct=clnk&gl=us

Newtype – inferring typeclasses

To **add**, **subtract**, and **multiply** the **currency**

1) use a **data** definition and
   an **instance** of the **Num** **typeclass**

2) use a **newtype** definition and
   automatic derivation of the **Num** **typeclass**
   {-# LANGUAGE GeneralizedNewtypeDeriving #-}

https://webcache.googleusercontent.com/search?q=cache:_5Dl-cKznPcJ:https://andre.tips/wmh/newtype/
+&cd=12&hl=en&ct=clnk&gl=us

Newtype – using **instance**

In order to **add** or **subtract** two Dollars.

use an **instance** of the **Num** **typeclass**

```
instance Num Dollar where
    (Dollar a) + (Dollar b) = Dollar (a + b)
    (Dollar a) - (Dollar b) = Dollar (a - b)
    (Dollar a) * (Dollar b) = Dollar (a * b)
    negate (Dollar a) = Dollar (-a)
    abs (Dollar a) = Dollar (abs a)

instance Num Euro where …
instance Num Yen where …
```

https://webcache.googleusercontent.com/search?q=cache:_5Dl-cKznPcJ:https://andre.tips/wmh/newtype/+&cd=12&hl=en&ct=clnk&gl=us

Newtype – examples using **newtype**

Wrapping one type (**Double**) in another (**Dollar**)
is needed so frequently, that there is a special syntax for it
newtype.

```
newtype Dollar  = Dollar    Double     deriving (Read, Show)
newtype Euro    = Euro      Double     deriving (Read, Show)
newtype Yen     = Yen       Double     deriving (Read, Show)

data Dollar  = Dollar  Double     deriving (Read, Show)
data Euro    = Euro    Double     deriving (Read, Show)
data Yen   = Yen       Double     deriving (Read, Show)
```

https://webcache.googleusercontent.com/search?q=cache:_5Dl-cKznPcJ:https://andre.tips/wmh/newtype/
+&cd=12&hl=en&ct=clnk&gl=us

Newtype – to derive typeclass automatically

The main difference between using **newtype** and **data** is that

newtype only works with the very simple cases

of wrapping one type in one other type.

Still, we cannot sum types or have multiple types wrapped up in one.

a special GHC feature

derives automatically ncessary **typeclasses**

Enabled by **{-# LANGUAGE GeneralizedNewtypeDeriving #-}**

at the top of your code

(a pragma to turn on a language extension)

https://webcache.googleusercontent.com/search?q=cache:_5Dl-cKznPcJ:https://andre.tips/wmh/newtype/+&cd=12&hl=en&ct=clnk&gl=us

Newtype – examples using a directive

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Dollar  = Dollar    Double     deriving (Read, Show)
newtype Euro    = Euro      Double     deriving (Read, Show)
newtype Yen     = Yen       Double     deriving (Read, Sho
```

the Num type class is derived automatically

```
(Dollar 3) + (Dollar 4)
        Dollar 7.0.
```

https://webcache.googleusercontent.com/search?q=cache:_5Dl-cKznPcJ:https://andre.tips/wmh/newtype/+&cd=12&hl=en&ct=clnk&gl=us

Single constructors of **newtype** and **data**

Both **newtype** and the single-constructor **data**

introduce a single **data constructor**,

but the **data constructor**

introduced by **newtype** is **strict**

and the single **data constructor**

introduced by **data** is **lazy**.

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

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

Strict evaluation of **undefined**

Haskell tries to only evaluate things

only when they are really necessary,

if you write **1+2** it won’t actually evaluate

that until it needs to. (**lazy by default**)

a **special value** named **undefined** (**bottom**)

If **undefined (bottom)** is pass to any function

then your program instantly crash when it is evaluated.

https://webcache.googleusercontent.com/search?q=cache:_5Dl-cKznPcJ:https://andre.tips/wmh/newtype/+&cd=12&hl=en&ct=clnk&gl=us

data (lazy), newtype (strict)

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

But **D** **undefined** is not equivalent to **undefined**, lazy

and it can be **evaluated** as long as

you do not try to peek inside.

Then **N** **undefined** is equivalent to **undefined** strict

and causes an **error** when **evaluated**.

undefined is evaluated strictly

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

Algebraic type – sum and product

This is a type where we specify the **shape** of each of the **elements**.

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 | 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 **product (combination)**

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 **sum (alternation)**

just **one** number, either an **Int** or else a **Double**.

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 – ADT and GADT

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

Type classes

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 definition

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 constraint

Eq a expresses a **constraint** that

a **type a** *must be* an **instance** of the **class Eq**

Eq a

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

context : a constraint on a type

type

(Eq a) =>

class name

class instance

for every **type a** that is
an **instance** of the **class Eq**

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

Type class constraint examples

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 `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

type

instance Eq integer

class name

class instance

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

Type class instance examples

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

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

the definition of the **method ==**

```
instance Eq Float where
x == y        =  x `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 analogy with OOPs

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 is not an object

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**.

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

Type class method is type safe

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**.

> Haskell functions can take functions as **parameters**
> and return functions as **return values**.
> A function that does either of those is called a **higher order function**.

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

Bottom Value

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 '⊥'.

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, *undefined*

then **strict f x** evaluates to "not computable". *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 ≠ ⊥ then f x else ⊥.

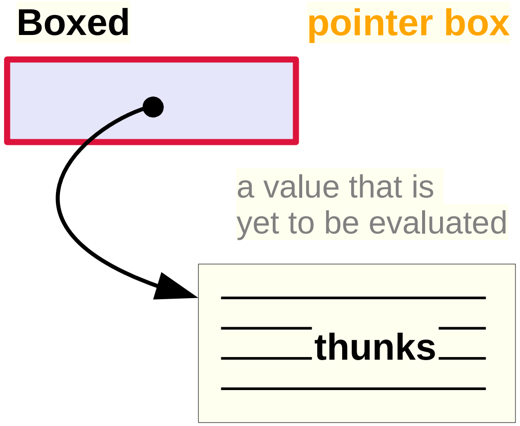
Operationally, **strict f x** is reduced by
first reducing **x** to **weak head normal form** (**WHNF**)
and then reducing the application **f x**.

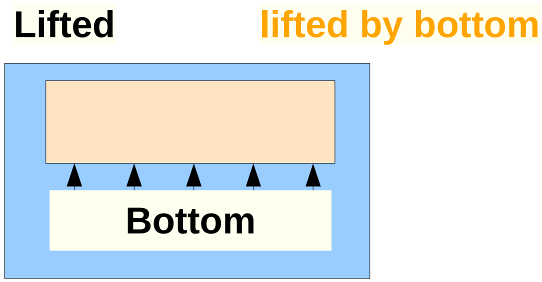
Alternatively, it is safe to reduce **x** and **f x** in parallel,
but not allow access to the result until **x** is in **WHNF**.

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	one or more **constructors**,
Primitive	a **built-in type**



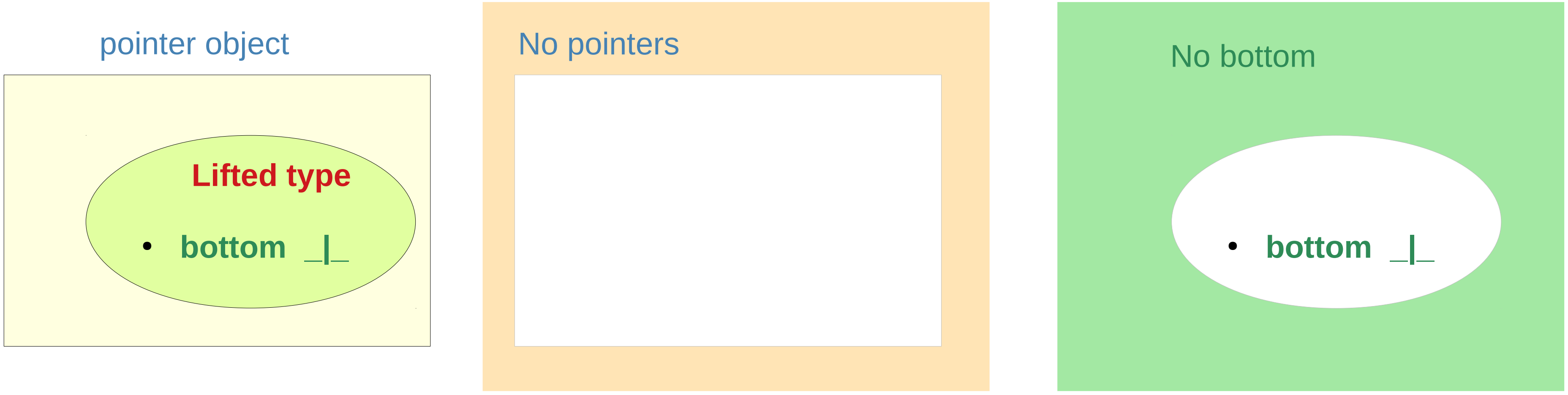


Undefined
Infinite loop
Exception

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

(Un)Lifted and (Un)Boxed types

Boxed type **Unboxed type** **Unlifted type**



pointer object

Lifted type

- **bottom _|_**

No pointers

No bottom

- **bottom _|_**

Lifted type → **Boxed type**

kind *

Unboxed type → **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 ⊥ 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 ⊤

(**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, ⊥ 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")
\x -> (x + 1)
```

not in normal form:

```
1 + 2            -- we could evaluate this to 3
(\x -> 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.

Normal form (NF)

Weak Head Normal Form (WHNF)

No unevaluated subexpressions

No unevaluated head expression

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