Monad P3 : Primitive Types (1B)

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

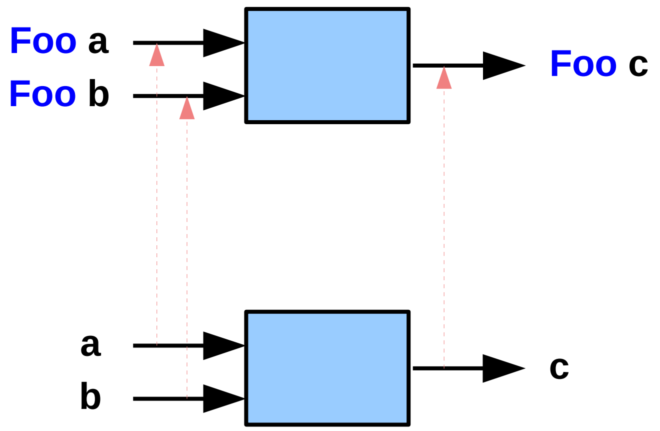
Lifting (1)

Typical data type with a parameter

data Foo a = Foo { ...stuff here ...}

Suppose that a lot of uses of Foo take numeric types (Int, Double etc) and you keep having to write code that *unwraps* these numbers, adds or multiplies them, and then *wraps* them back up.
You can short-circuit this by writing the *unwrap-and-wrap* code once.
This function is traditionally called a "**lift**" because it looks like this:

liftFoo2 :: (a -> b -> c) -> Foo a -> Foo b -> Foo c



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

Lifting (2)

liftFoo2 :: (a -> b -> c) -> Foo a -> Foo b -> Foo c

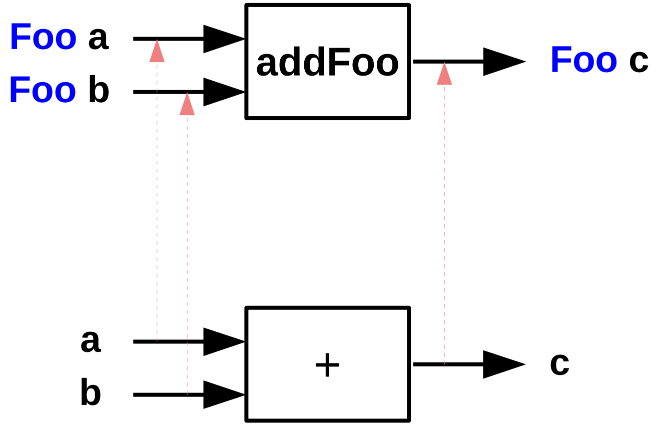
in other words you have a function

which takes a two-argument function

(such as the (+) operator) and

turns it into the equivalent function for **Foo**s.

addFoo = liftFoo2 (+)



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

Bottom

Haskell allows you to use a special value called **undefined**.

This is sometimes also refereed to as **bottom**, ⊥, or _**|**_

Member of all types

```
Prelude> i = undefined
Prelude> i + 1
-- error!
Prelude> l = [1,2,3,4,undefined]
Prelude> l !! 3
4
Prelude> l !! 4
-- error!
```

Laziness

```
Prelude> head [1, undefined]
1
Prelude> head [undefined, 1]
-- error!
```

As a return value

```
Prelude> stupid = sum [1..]
Prelude> stupid
-- infinite loop
```

As an argument to a function

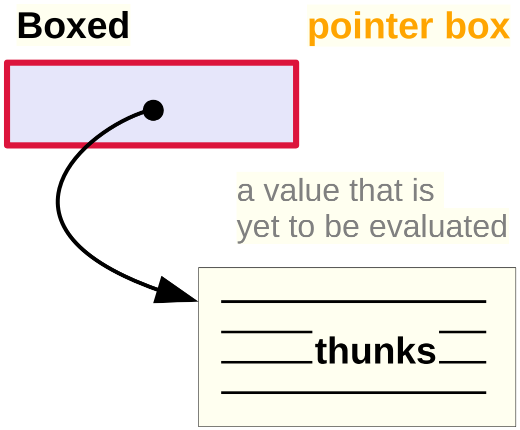
```
Prelude> weird x = 3

Prelude> weird . sum $ [1..]
3

Prelude> weird undefined
3
```

https://andre.tips/wmh/brief-note-undefined/

Box

In most implementations of **lazy evaluation**,
values are represented at runtime as **pointers**
to either their **value**,
or **code** for computing their value.

This extra level of **indirection**,
together with any extra **tags**
needed by the **runtime**,
is known as a **box**.



https://en.wikibooks.org/wiki/Haskell/Libraries/Arrays

Boxed representation

the expressiveness of **non-strict arrays** comes at a price,
especially if the array elements are simple numbers (**values**).

Instead of direct storing those numeric elements,
non-strict arrays require a **boxed representation**

> the elements are **pointers** to **heap objects**
> containing the numeric values.

This **additional indirection** requires extra **memory** and
drastically reduces the **efficiency** of array access,
especially in **tight loops**.

https://www.tweag.io/posts/2017-09-27-array-package.html

Boxed vs Unboxed Kinds

```
> :k Int
Int :: *
> :k Int#
Int# :: #
```

Int# has a different **kind** than normal Haskell datatypes: #.

https://haskell-lang.org/tutorial/primitive-haskell

Boxed vs Unboxed Types

values of **boxed** **type** are represented

by a **pointer** to a **heap object**

The representation of a Haskell **Int** is

a two-word **heap** object

An **unboxed** type is represented

by the **value** itself,

no **pointers** or **heap allocation** are involved.

unboxed types correspond to the "raw machine" types in C

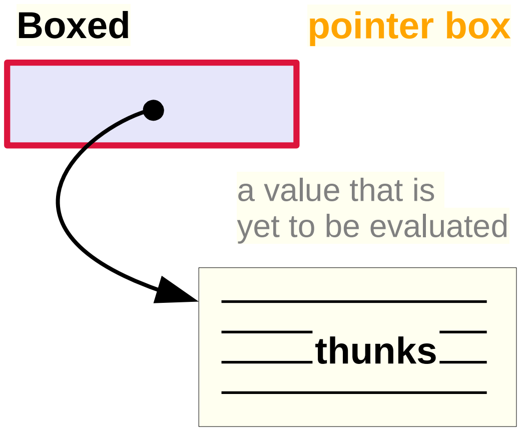
Int#	(long int)
Double#	(double)
Addr#	(void *)

Most **types** in GHC are **boxed**,

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

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**

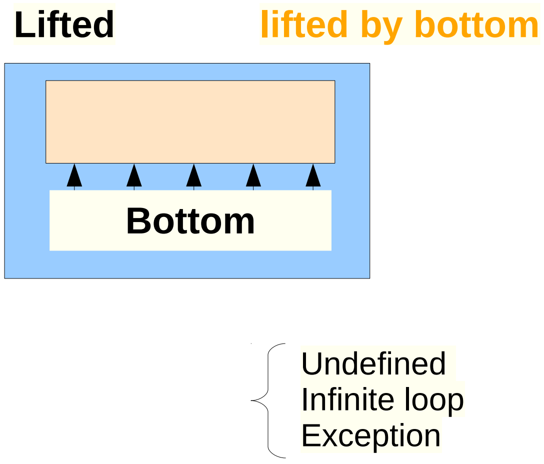


Boxed

pointer box

a value that is
yet to be evaluated

thunks



Lifted

lifted by bottom

Bottom

Undefined
Infinite loop
Exception

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

Classifying types – Boxed, Unboxed

Boxed types

a value is represented by a **pointer** to a **heap** object.

Unboxed types

a type is **unboxed** iff its representation is other than a **pointer**.

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

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

Classifying types – Lifted, Unlifted

Lifted types

A type is **lifted** iff it has **bottom** as an element.

A **value** of a **lifted type** can be **bottom**.

it can be **undefined**, or perhaps a
computation that **never finishes**, or
one that **throws** an **exception**.

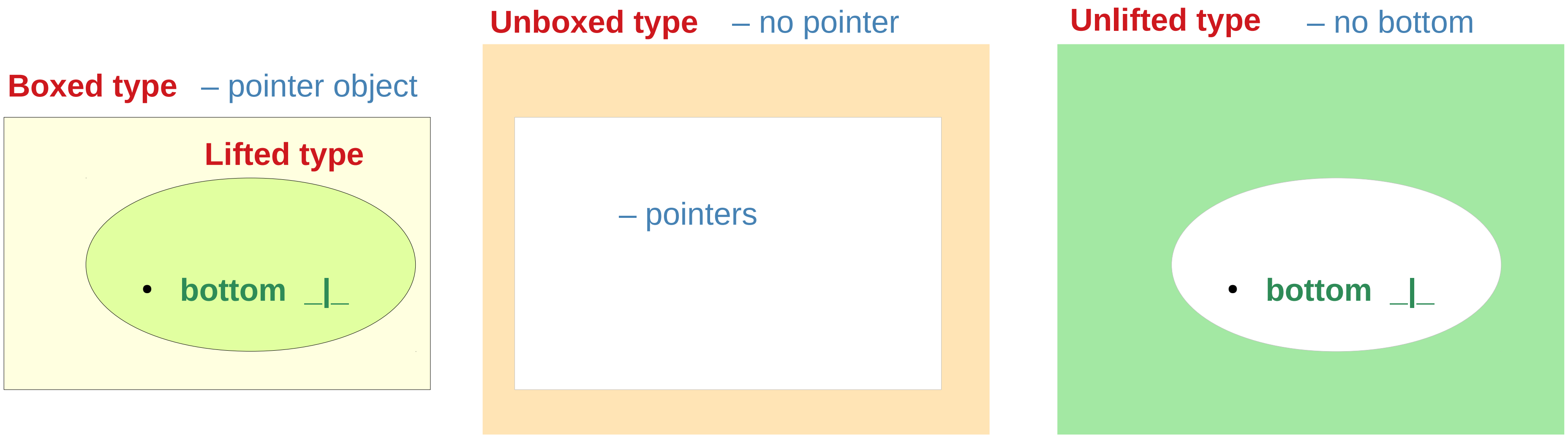
Unlifted types

do not have these potentially troublesome **extra values**.

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

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

(Un)Lifted and (Un)Boxed types



Unboxed type – no pointer
Unlifted type – no bottom
Boxed type – pointer object
Lifted type
• bottom _|_
– pointers
• 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

Applications of Unboxed types

Unboxed types

- cannot have **thunks**
    - since **thunks** are **pointers** to data
    - telling you how to produce the value
- cannot_exploit **laziness**
- really just hold **values**.
- they can be **faster**.

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

Applications of lifted types

Closures always have **lifted** types: i.e.

any **let-bound** identifier in **Core** must have a **lifted** type.

Operationally, a lifted object is one that can be entered.

Only **lifted** types may be unified with a **type variable**.

Polymorphism does not play with **unlifted types**.

parametric type must be **lifted**.

Something like **id 0 :: Int#** does not work.

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

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

Applications of unlifted types

Unlifted types do not have **bottom** as a value

This can be useful in a **purely "semantic" level**
(if you don't want those extra values) and

it can also facilitate more **efficient implementations**
by reducing **costly indirections**.

A GHC optimization called the **worker-wrapper transformation**
exploits this

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

Classifying types – Algebraic

Algebraic

a data type with one or more **constructors**,
whether declared with **data** or **newtype**.

An **algebraic** type is one that can be deconstructed
with a **case** expression.

Algebraic is NOT the same as **lifted**
because **unboxed** (and thus **unlifted**) **tuples** count as "**algebraic**".

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

Classifying types – Primitive

Primitive

a type is **primitive** iff it is a **built-in type**

that can't be expressed in Haskell.

Currently, all **primitive** types are **unlifted**,

but that's not necessarily the case.

(E.g. **Int** could be **primitive**.)

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

Type classification examples

	Primitive	Boxed	Lifted	Algebraic
Int#	Yes	No	No	No
Array#	Yes	Yes	No	No
(# a, b #)	Yes	No	No	Yes
(a, b)	No	Yes	Yes	Yes
[a]	No	Yes	Yes	Yes

Some **primitive types** are **unboxed**, such as **Int#**, whereas some are **boxed** but **unlifted** (such as **Array#**). The only **primitive types** that we classify as **algebraic** are the **unboxed tuples**.

Array#	**Boxed**	**Unlifted**	pointer, no bottom
ByteArray#	**Unboxed**	**Unlifted**	no pointer, no bottom

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

Primitive types ≈

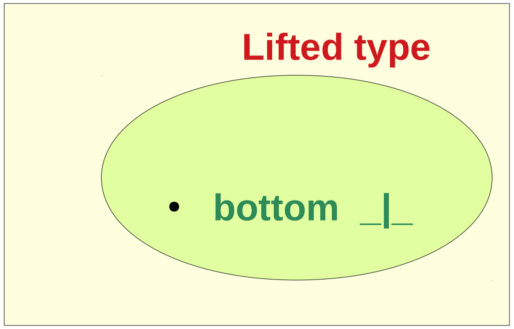
Unboxed types ≈

Unlifted types

Lifted type → **Boxed type**

Unlifted type ← **Unboxed type**

Boxed type – pointer object



Int# not normal data type

The **Int# constructor** is actually

just a normal **data constructor** in Haskell with a **#**

Int# is not a normal **data type**

In **GHC.Prim**, it's implementation is:

data Int#

- like everything else in GHC.Prim is really a **lie**.
- is provided by the **implementation**,
- is in fact a normal **long int** from C

Int#

Normal **data constructor**

Not normal **data type**

https://haskell-lang.org/tutorial/primitive-haskell

Magic hash

By convention, all **unlifted types** end with a **#**,

called the **magic hash**,

enabled by the **MagicHash extension**.

examples include **Char#** and **Int#**.

to distinguish **unboxed operations** – **functions with #**

```
(+#) :: Int# -> Int# -> Int#
(+#) = let x = x in x
```

You can even have

unboxed tuples (# a, b #)

unboxed sums (# a | b #)

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

https://haskell-lang.org/tutorial/primitive-haskell

Primitive Operations

The **primitive operations** (**PrimOps**) on **primitive types**

e.g., **(+#)** is **addition** on **Int#**s

the machine-addition

– usually one instruction.

the **standard +** operator and **Int** data type

are actually themselves defined in normal Haskell code,

which provides many benefits:

standard type class support, **laziness**, etc.

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

PrimOps

primops, short for **primitive operations**,

are core pieces of **functionality** provided by **GHC** itself.

They are the *magical, elegant boundary*

between "things we do in Haskell itself"

and "things which our implementation provides."

https://haskell-lang.org/tutorial/primitive-haskell

Functions in primitive operations

Look at the implementation of other functions in **GHC.Prim**;

they're *all* defined as **let x = x in x**.

```
and# :: Word# -> Word# -> Word#
and# = let x = x in x
```

When GHC reaches a call to one of these **primops**,

it automatically replaces it with the **real implementation**,

- an **assembly code**, an **LLVM code**, or something else

dummy implementation to give **Haddock** documentation

GHC.Prim is processed by **Haddock** more or less

like any other module; but is effectively ignored by GHC itself.

let x = x in x

https://haskell-lang.org/tutorial/primitive-haskell

let x = z in y

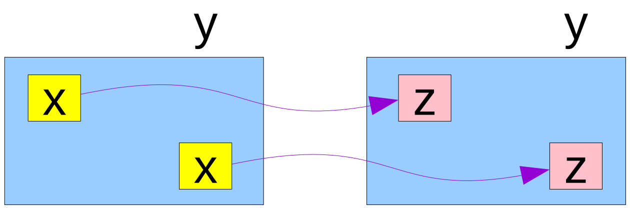
let x = z in y

change the variable **x** to the expression **z**

wherever **x** occurs in the expression **y**

Considered as the **reduction rule** for the **application**

of the lambda abstraction **\x -> y** to the **term z**



https://haskell-lang.org/tutorial/primitive-haskell

let x = x in x

let x = x in x

these data declarations/functions are

to provide access to the raw compiler internals.

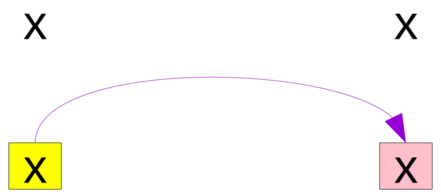
GHC.Prim exists to export these primitives,

it doesn't actually implement them or anything

(eg its code isn't actually useful).

All of that is done in the **compiler**.

It's meant for code that needs to be extremely optimized.



let x = x in x

https://haskell-lang.org/tutorial/primitive-haskell

Primitive Types

Primitive (**unlifted**, **unboxed**) **types**

cannot be defined in Haskell, and thus

are built into the language and compiler.

Primitive types are always **unlifted**; that is,

bottom cannot be a **value** of a **primitive type**

We use the convention

that **primitive types**, **values**, and **operations**

have a **# suffix**.

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

Primitive Values

Primitive values are often represented by a simple bit-pattern, such as **Int#**, **Float#**, **Double#**.

But **Array#** is not necessarily the case:
a **primitive value** might be represented
by a **pointer** to a **heap-allocated object**. .... (Boxed)

Examples include **Array#**, the type of **primitive arrays**.

	Primitive	Boxed	Lifted	Algebraic
Int#	**Yes**	**No**	**No**	**No**
Array#	**Yes**	**Yes**	**No**	**No**
(# a, b #)	**Yes**	**No**	**No**	**Yes**

Primitive types ≈
Unboxed types ≈
Unlifted types

Int# → **Primitive**
Array# → **Boxed Arrays**
(# a, b #) → **Unboxed Tuples**

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

Primitive types are faster

```
--boxed.hs
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)

main = print (fac 10)

--unboxed.hs
import GHC.Exts
fac :: Int# -> Int#
fac 0# = 1#
fac n   = n *# fac (n -# 1#)

main = print (I# (fac 10#))
```

can’t compute fac(500) … overflow

```
$ ghc boxed.hs
$ ghc -XMagicHash unboxed.hs
$ time ./boxed
$ time ./unboxed
```

The language extension -XMagicHash allows "#" as a postfix modifier to identifiers.

in **GHC.Exts**

data Int

A fixed-precision integer type with at least the range [-2^29 .. 2^29-1]. The exact range by using **minBound** and **maxBound**

Constructors

I# Int# **I#(500#)**

500# :: Int#

I#(500#) :: Int

https://moserei.de/2012/04/03/haskell-boxed-vs-unboxed.html

Restrictions on Primitive Types (1)

cannot pass a **primitive value** to a **polymorphic function** or

cannot store a **primitive value** in a **polymorphic data type**.

cannot use **a primitive value in a list type.**

lists of **primitive integers** are not possible : **[Int#]**

polymorphic arguments and **constructor fields**

are assumed to be **pointers**:

Nevertheless, A **numerically-intensive** program

using **unboxed types** can go a lot faster

than its “standard” counterpart

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

Restrictions on Primitive Types (2)

polymorphic **arguments** and **constructor fields**

are assumed to be **pointers**:

If an **unboxed** integer is used in such fields

the **garbage collector** would attempt

to follow an **unboxed** integer, *dereference*

leading to unpredictable **space leaks**.

a **seq** operation on the **polymorphic component** may attempt

to **dereference** the pointer, with disastrous results.

Even worse, the **unboxed value** might be larger than a **pointer**

(Double# for instance).

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

Primitive Arrays

A **primitive array** is **heap-allocated**

because it is too big a value to fit in a **register**,

and would be too expensive to copy around;

in a sense, it is accidental that it is represented by a **pointer**.

If a **primitive value** is represented by a **pointer** … Array#

then the pointer really does point to that value

– no **unevaluated thunks**, no **indirections**…

– nothing can be at the other end of the **pointer**

but the **primitive value**.

Array#

Primitive

Boxed ..… using a pointer

Unlifted … no bottom

primitive values
Int#, Float#, Double#

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

Primitive Arrays – **Array# obj** and **ByteArray#** (1)

A primitive array is **heap-allocated**

type Array# obj

primitive arrays of (**boxed**) Haskell objects obj

Primitive

Boxed ….. use a pointer

Unlifted … no bottom

type ByteArray#

primitive arrays of bytes … similar to C arrays

Primitive

Unboxed …. no pointer

Unlifted …... no bottom

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

Primitive Arrays – **Array# obj** and **ByteArray#** (2)

A **primitive array** is **heap-allocated**

type Array# obj

primitive arrays of (**boxed**) Haskell objects **obj**

type ByteArray#

primitive arrays of **bytes** (no pointer)

	Primitive	Boxed	Lifted	Algebraic
Int#	**Yes**	**No**	**No**	**No**
ByteArray#	**Yes**	**No**	**No**	**No**
Array#	**Yes**	**Yes**	**No**	**No**

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

Primitive Arrays – **Array# obj** and **ByteArray#** (3)

A **primitive array** is **heap-allocated**

type Array# obj **….** **boxed** Haskell objects **obj**

type ByteArray# **….** **unboxed bytes** (no pointer)

Array# obj	**ByteArray#**
Primitive	**Primitive**
Boxed ….. use a pointer	**Unboxed** ….. no pointer
Unlifted … no bottom	**Unlifted** … no bottom

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

Boxed Arrays

GHC **heap** contains two kinds of objects

some are just byte sequences,

other contains pointers to other objects (so called "boxes").

These segregation allows to find chains of references

when performing **garbage collection** and **update** these pointers

when memory used by heap is compacted and

objects are moved to new places.

Internal (**raw**) GHC's type **Array#** represents

a sequence of object **pointers** (**boxes**).

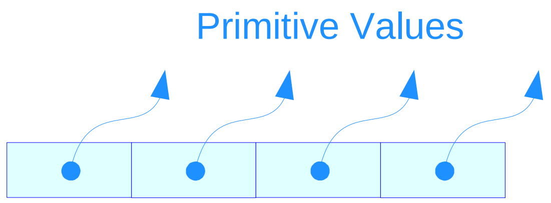
The **Array#** type is used inside **Array** type

which represents **boxed immutable arrays**.

Therefore,

Unboxed Arrays	**ByteArray#**
Boxed Arrays	**Array#**

Array# : sequence of pointers (boxes)



https://en.wikibooks.org/wiki/Haskell/Libraries/Arrays

Unboxed Arrays

Unboxed arrays (defined in **Data.Array.Unboxed**)
are more like arrays in C

they contain just the **plain values**
without the extra level of **indirection**, … no pointer (box)

for example, an array of 1024 values of type Int32
will use only 4 kb of memory.

- **indexing** of such arrays can be significantly **faster**.
- only of plain values having a **fixed size**
- must be **evaluated** when the array is evaluated

Unboxed arrays are represented by the **ByteArray#** type

https://en.wikibooks.org/wiki/Haskell/Libraries/Arrays. https://wiki.haskell.org/Arrays

Array#

Array# is more primitive than a Haskell array

– an **Array#** is indexed only by **Int#**s, starting at zero.

– **unboxed** but is a **heap allocated** object

– **unboxed** but is represented by

a **pointer** to the array itself

not to a **thunk** or to **bottom**

– the components of an **Array#** are themselves are **boxed**

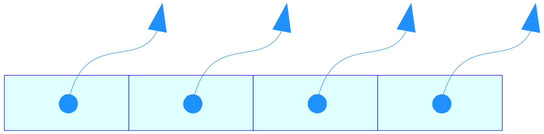
the Haskell Array interface is implemented using Array#

The type **Array# obj** is

the type of **primitive**, **unpointed arrays** of **values of type obj.**

Array# : sequence of pointers (**boxes**)

Points to value itself
No thunks no bottom
Unboxed elements



Boxed arrays
sequence of pointers

https://downloads.haskell.org/~ghc/5.04.1/docs/html/users_guide/primitives.html

ByteArray#

Unboxed arrays are represented by the **ByteArray#** type.

It's just a plain memory area in the heap, like the C's array.

ByteArray# is **unboxed** but **unlifted**

There are two primitive **operations**

that creates a **ByteArray#** of specified size

newByteArray

newPinnedByteArray

https://en.wikibooks.org/wiki/Haskell/Libraries/Arrays

ByteArray# – normal heap

One **primitive operation** allocates memory in **normal heap**
and so this **byte array** can be moved each time
when garbage collection occurs.

This prevents converting of **ByteArray#**
to **plain memory pointer**
that can be used in C procedures

although it's still possible to pass
current **ByteArray#** **pointer**
to "unsafe foreign" procedure
if it don't try to store this **pointer** somewhere

newByteArray

https://en.wikibooks.org/wiki/Haskell/Libraries/Arrays

ByteArray# – pinned heap area

The second **primitive operation**
allocates **ByteArray#** of specified size
in **pinned heap** area,
which contains objects with fixed place.

Such byte array will never be moved by garbage collection
so it's address can be used as plain **Ptr** and
shared with C world.

newPinnedByteArray

https://en.wikibooks.org/wiki/Haskell/Libraries/Arrays

Unboxed tuple – multiple return value

(# e_1, ..., e_n #)

e_1 .. e_n are **expressions** of any type (**primitive** or **non-primitive**).

Unboxed tuples are used for

functions that need to *return multiple values*,

but they avoid the **heap allocation** of

fully-fledged **tuples** (**boxed real tuple**)

when an **unboxed tuple** is returned,

the **components** are put directly

into **registers** or on the **stack**;

https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/primitives.html

Unboxed tuple – no heap allocation

the **unboxed tuple** itself

does not have a **composite representation**.

no tuples within tuples representation

Many of the **primitive operations**

return **unboxed tuples**.

In particular, the **IO** and **ST monads**

use **unboxed tuples**

to avoid unnecessary **allocation**

during sequences of operations.

https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/primitives.html

Unboxed tuple examples

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

The first primitive is the **unboxed tuple**, seen in code as **(# x, y #)**.

1. **State# RealWorld**
2. **a**

a **multiple value return** syntax

But not actual **real tuples** and

can't be put in **variables** as such.

Boxed real tuple incurs **heap allocation**

whenever an **IO action** is performed,

http://blog.ezyang.com/2011/05/unraveling-the-mystery-of-the-io-monad/

Types and Kinds

Just like values / terms can be classified into **types**,

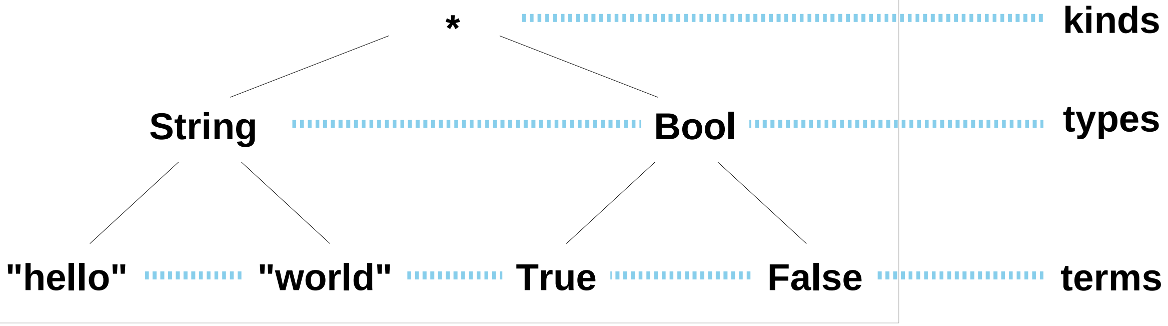
types can be classified into **kinds**.

The values **"hello"** and **"world"** are of type **String**.

The values **True** and **False** are of type **Bool**.

Similarly, the types **String** and **Bool** are

of kind *****, pronounced “star”.



https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

* kind

*, pronounced "**star**", is

the **kind** of **all data types**

all lifted inhabited type

seen as nullary type constructors, and

also called proper types in this context.

this normally includes function types

Inhabitable Lifted type	Inhabitable Unlifted type
bottom	no bottom
kind *	**kind #**

https://en.wikipedia.org/wiki/Kind_(type_theory)

:type and :kind

:t or**:type**   to check the **type** of a term

:k or **:kind** to check the kin**d** of a type.

λ> :t True

True :: Bool

Term :: Type
(value)

λ> :k Bool

Bool :: *

Type ;; Kind

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

Kind encode type representation

Kinds are like types for types

lifted inhabitable types have the **kind ***

’c’ :: Char :: *

Just 1 :: Maybe Int :: *

Type constructors, on the other kind, contain the **arrow symbol**

Maybe :: * -> *

Either :: * -> * -> *

Unlifted types are of the **# kind**

’c’# :: Char # :: #

Inhabitable Lifted type	**Inhabitable Unlifted type**
bottom	no bottom
kind *	**kind #**

Haskell High Performance Programming,, Samuli Tomason, 2016

Inhabited types

In standard Haskell, all **inhabited types**

(types that have at least 1 value) are of **kind ***

Int

Int -> String

[Int]

Maybe Int

Either Int Int

each of these types has at least one **term**

therefore all these types are of **kind ***

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

Uninhabited types

Maybe and **Either** are **uninhabited**.

But they are **type constructors**

There is **no term** of **type Maybe**, not even the **infinite loop**!

```
λ> x = undefined :: Maybe
<interactive>:9:18: error
    • Expecting one more argument to ‘Maybe’

λ> f x = f x :: Maybe
<interactive>:10:14: error:
    • Expecting one more argument to ‘Maybe’
```

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

Terms

Just as **expressions** denote values,

type expressions are **syntactic terms**

that denote **type values** (or just **types**).

Examples of **type expressions** include

the **atomic types**

Integer (infinite-precision integers), **Char** (characters),

Integer->Integer (functions mapping Integer to Integer),

the **structured types**

[Integer] (homogeneous lists of integers) and

(Char,Integer) (character, integer pairs).

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

Type Constructors with type arguments

A **type constructor** takes one or more **type arguments**,
and produces a **data** **type** when enough arguments are supplied,
i.e. it supports **partial application** thanks to **currying**.

This is how Haskell achieves **parametric types**.

For instance, the **type** **[]** is a **type constructor** -
it takes a **single argument** to specify
the type of the elements of the list.

Hence, **[Int]**, **[Float]** and even **[[Int]]** are
valid applications of the **[]** **type constructor**.

https://en.wikipedia.org/wiki/Kind_(type_theory)

Type Constructors and data constructors

a **nullary / unary type constructor** (or simply a **type**).

has zero / one argument

data Bool = True | False

a nullary type constructor **...** **Bool**

two nullary data constructors **...** **True** and **False**

data Tree a = Tip | Node a (Tree a) (Tree a)

a unary type constructor **...** **Tree**

a nullary data constructors **...** **Tip**

a unary data constructors **...** **Node**

https://wiki.haskell.org/Constructor

Data constructors - first class values

Data constructors

are first class values in Haskell and

actually have a type.

For instance, the type of the **Left constructor**

of the **Either data type** is:

data Either a b = Left a | Right b

Left :: a -> Either a b

As first class values, they may be

- passed to functions,
- held in a list,
- data elements of other algebraic data types, and so forth.

https://wiki.haskell.org/Constructor

Data constructors – not types

Data constructors are not **types**

they denote values.

Node a (~~Node~~ a) (~~Node~~ a)

It is illegal because the **type** is **Tree**, not **Node**.

data Tree a = Tip | Node a (Tree a) (Tree a)

https://wiki.haskell.org/Constructor

Type constructors and Kinds

∗ -> ∗ is the **kind** of a unary **type constructor**,

e.g. of a **list type constructor**.

∗ -> ∗ -> ∗ is the **kind** of a binary **type constructor** (via currying),

e.g. of **a pair type constructor**, and also

that of **a function type constructor**

(not to be confused with the result of its **application**,

which itself is a function type, thus of kind ∗

(∗ -> ∗) -> ∗ is the **kind** of a higher-order **type operator**

from unary **type constructors** to proper **types**.

https://en.wikipedia.org/wiki/Kind_(type_theory)

Kind examples (1)

Haskell's kind system has just two rules:

∗ pronounced "**type**" is the **kind** of **all *lifted* data types**.

k1 -> k2 is the kind of a **unary type constructor**,
which takes a type of kind k1 and
produces a type of kind k2

https://en.wikipedia.org/wiki/Kind_(type_theory)

Kind examples (2)

[] is a **type** of **kind ∗ -> ∗** .

Because **Int** has **kind ∗** ,
applying **type constructor []** to it
results in **[Int]**, of **kind ∗** .

The **2-tuple constructor (,)** has kind ∗ -> ∗ -> ∗,

the **3-tuple constructor (, ,)** has kind ∗ -> ∗ -> ∗ -> ∗ and so on.

https://en.wikipedia.org/wiki/Kind_(type_theory)

Inhabited types with kind *

An **inhabited type**

a type which has **values**.

a so called proper types in Haskell)

For instance, ignoring type classes

4 is a **value** of type **Int**,

[1, 2, 3] is a **value** of type **[Int]** (list of Ints).

all **inhabited *lifted* types** are of **kind ***

Int and **[Int]** have **kind ∗**

any **function type** has **kind ∗**

for instance **Int -> Bool** or even **Int -> Int -> Bool**.

https://en.wikipedia.org/wiki/Kind_(type_theory)

Inhabited types with kind

all **inhabited *lifted* types** are of **kind ***

* is the **kind** of all **inhabited boxed** (or **lifted**) **types**.

However, in GHC’s version of Haskell,

there are also some **inhabited types**

that are not of **kind ***

unboxed / **unlifted / primitive types**

are of **kind #**

Q these are defined in the **GHC.Prim** module

from the ghc-prim package.

Inhabitable Lifted type	**Inhabitable Unlifted type**
bottom	no bottom
kind *	**kind #**

Primitive ≈

Unboxed ≈

Unlifted types

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

* kind and # kind

So **ByteArray#**, the type of raw blocks of memory, is

~~**boxed** because it is represented as a **pointer**,~~ unboxed

but **unlifted** because **bottom** is not an element.

```
> undefined :: ByteArray#
Error: Kind incompatibility when matching types:
   a0 :: *
   ByteArray# :: #
```

Therefore it appears that the old User's Guide definition is more accurate than the GHC Commentary one:

***** is the **kind** of **lifted types**.

(And, conversely, **#** is the **kind** of **unlifted types**.)

Unboxed Arrays	**ByteArray#**
Boxed Arrays	**Array#**

https://stackoverflow.com/questions/27095011/what-exactly-is-the-kind-in-haskell

Kind and runtime representation

Each **unlifted type** has a **kind**

that describes its runtime representation.

*Is this a **pointer** to something in the **heap**?*

*Is it a signed/unsigned **word**-**sized value**?*

The compiler then uses that **type's kind**

to decide which **machine code** it needs to produce -

this is called "**kind-directed compilation**".

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

Runtime representation of values

GHC maintains a property that

the **kind** of all **inhabited types** tells us

the **runtime representation** of **values** of that **type**.

(as distinct from **type constructors** or **type-level data**)

Inhabited types – instance

kind tells the runtime representation

of values of that type

http://hackage.haskell.org/package/base-4.12.0.0/docs/GHC-Exts.html#t:MutVar-35-

Unified types and kinds

Starting with GHC8, **types** and **kinds** have been **unified**.

a single indexed type of types

```
data TYPE a                        :: RuntimeRep -> *

data TYPE (a :: RuntimeRep) :: RuntimeRep -> Type
```

$\underbrace{\texttt{TYPE (a :: RuntimeRep)}}_{\text{a single type indexed by a}}$ :: $\underbrace{\texttt{RuntimeRep -> Type}}_{\text{a kind}}$

a single type

indexed by a

a kind

*** -> ***

Haskell High Performance Programming,, Samuli Tomason, 2016

The **kind *** - the synonym Type

Recently, the **kind *** is often referred to as **Type**

(do not confuse with **TYPE r**).

these are **synonyms** for now,

and the plan is to gradually phase out * in favour of **Type**.

```
data TYPE a                     :: RuntimeRep -> *
data TYPE (a :: RuntimeRep) :: RuntimeRep -> Type
```

here, all **inhabited types** are of **kind ***

not just **inhabited lifted types**

Old usage

Kind *

for all **lifted inhabitable types**

Kind #

for all **unlifted inhabitable types**

Recent usage

Kind * or **Type**

for **all inhabitable types**

either **lifted** or **unlifted**

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

Kind **TYPE r**

TYPE IntRep has the **kind** of **unlifted integers**,

TYPE FloatRep has the **kind** of **unlifted floats**, etc.

TYPE LiftedRep has the **kind** for **all lifted types** -

in fact, the * **kind** is nothing more than a synonym for **TYPE LiftedRep**

TYPE r enables us to abstract

not only over all **unlifted** types,

but also over **lifted** ones.

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

Kind **TYPE LiftedRep**

True :: Bool	**:t True**	to check the **type** of a term
Bool :: *	**:k Bool**	to check the **kind** of a type.

data TYPE (a :: RuntimeRep) :: RuntimeRep -> Type

type Type = TYPE LiftedRep

The **kind** of **types** with **lifted values**. For example

Int :: Type

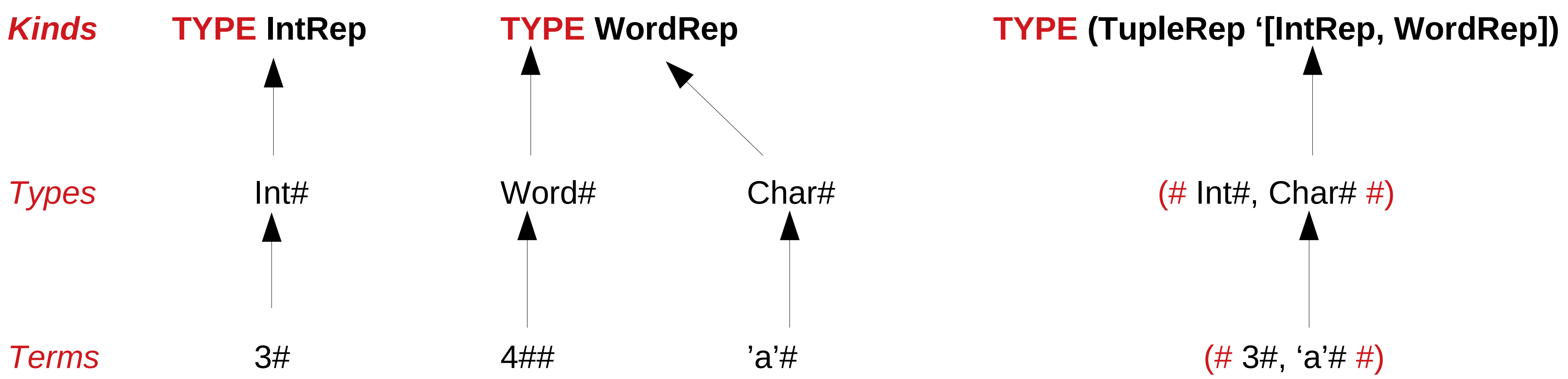
Int :: TYPE LiftedRep

type :: kind
term :: type

https://hackage.haskell.org/package/ghc-prim-0.6.1/docs/GHC-Types.html#v:LiftedRep

Inhabited types with kind **TYPE r**

Here are some examples:



Kinds **TYPE IntRep** **TYPE WordRep** **TYPE (TupleRep '[IntRep, WordRep])**

Types Int# Word# Char# (# Int#, Char# #)

Terms 3# 4## 'a'# (# 3#, 'a'# #)

```
TupleRep [RuntimeRep]     -- ^ An unboxed tuple of the given reps
IntRep                    -- ^ signed, word-sized value
WordRep                   -- ^ unsigned, word-sized value
```

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

RuntimeRep – TYPE

This datatype encodes the **choice** of **runtime value**.

Note that **TYPE** is parameterised by **RuntimeRep**;

```
data TYPE a                    :: RuntimeRep -> *
data TYPE (a :: RuntimeRep) :: RuntimeRep -> Type
type Type = TYPE LiftedRep
```

this is precisely what we mean by the fact

that a **type's** **kind** encodes the **runtime representation**.

A **type synonym** is a new name for an existing type.

type MyChar = Char

http://hackage.haskell.org/package/base-4.12.0.0/docs/GHC-Exts.html#t:MutVar-35-

The data type Type

The single data type **Type** is used to represent

- **types** (possibly of **higher kind**);
  e.g. **[Int]**, **Maybe**
- **kinds** (which classify **types** and **coercions**);
  e.g. **(* -> *)**, **T :=: [Int]**.
- **sorts** (which classify types);
  e.g. **TY**, **CO**

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

Kind Rationale

Haskell has a very powerful and expressive **static type system**.

The Haskell kind system has been extended to overcome an unsatisfactorily inexpressiveness in **programming at the type level**,

https://gitlab.haskell.org/ghc/ghc/-/wikis/kind-system

Note: As of June 2013, this page is rather out of date. This page is currently a WIP ..

Tools for programming in type level

• **Data constructors**	**Type constructors**
• **Type signatures**	**Kind signatures**
• **High Order Functions**	**Higher Kinded Types**

- **Other kinds except ***
  - **Unboxed / Unlifted types**
  - **Constraints**
  - **Datatype Promotion**
  - **GHC.TypeList**
- **Kind polymorphism**
- **Levity polymorphism**

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

RuntimeRep – constructors

VecRep VecCount VecElem	a SIMD vector type
TupleRep [RuntimeRep]	An unboxed tuple of the given reps
SumRep [RuntimeRep]	An unboxed sum of the given reps
LiftedRep	lifted; represented by a pointer
UnliftedRep	unlifted; represented by a pointer
IntRep	signed, word-sized value
WordRep	unsigned, word-sized value
Int64Rep	signed, 64-bit value (on 32-bit only)
Word64Rep	unsigned, 64-bit value (on 32-bit only)
AddrRep	A pointer, but not to a Haskell value
FloatRep	a 32-bit floating point number
DoubleRep	a 64-bit floating point numbe

http://hackage.haskell.org/package/base-4.12.0.0/docs/GHC-Exts.html#t:MutVar-35-

Kind **TYPE r**

the **kind TYPE r**

this **kind** is *parameterised* over **r :: RuntimeRep**,

RuntimeRep

describes a type’s runtime representation

can be one of the following:

```
data RuntimeRep = VecRep VecCount VecElem        --  a SIMD vector type
                  | TupleRep [RuntimeRep]   --  An unboxed tuple of the given reps
                  | SumRep [RuntimeRep]     --  An unboxed sum of the given reps
                  | LiftedRep               --  lifted; represented by a pointer
                  | UnliftedRep             --  unlifted; represented by a pointer
                  | IntRep                  --  signed, word-sized value
                 | WordRep                  --  unsigned, word-sized value
                 | Int64Rep                 --  signed, 64-bit value (on 32-bit only)
                 | Word64Rep                --  unsigned, 64-bit value (on 32-bit only)
                 | AddrRep                  --  A pointer, but /not/ to a Haskell value
                 | FloatRep                 --  a 32-bit floating point number
                 | DoubleRep                --  a 64-bit floating point number
```

https://diogocastro.com/blog/2018/10/17/haskells-kind-system-a-primer/

RuntimeRep – boxed values

```
data Array# (a :: Type)      :: Type -> TYPE UnliftedRep
data ByteArray#              :: TYPE UnliftedRep
data Char#                   :: TYPE WordRep
data Double#                 :: TYPE DoubleRep
data Float#                  :: TYPE FloatRep
data Int#                    :: TYPE IntRep
data Int32#                  :: TYPE IntRep
data Int64#                  :: TYPE Int64Rep


data TYPE (a :: RuntimeRep) :: RuntimeRep -> Type
```

http://hackage.haskell.org/package/base-4.12.0.0/docs/GHC-Exts.html#t:MutVar-35-

Closure (1)

A **closure**, the opposite of a **combinator**,
is a function that makes use of **free variables** in its **definition**.
It 'closes' around some portion of its environment. for example

```
f x = (\y -> x + y)
```

f returns a **closure**, because the variable **x**,
which is bound outside of the lambda abstraction
is used inside its definition.

An interesting side note: the context in which **x** was bound
shouldn't even exist anymore, and wouldn't,
had the lambda abstraction not closed around x.

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

Closure (2)

```
mkAdder :: Int -> (Int -> Int)
mkAdder y = \x -> x + y
```

mkAdder takes an **Int** as an **argument**, and
returns a function (**Int -> Int**) as a result.

the returned function **\x -> x + y**
has a free variable (**y**) which refers to its environment.

calling **mkAdder** with a particular argument (say, 3),
returns a closure, containing the function **\x -> x + y**
together with the environment (y = 3).

https://mail.haskell.org/pipermail/beginners/2009-July/002067.html

Closure (3)

mkAdder is really just (+), written in a funny way!

So this isn't a contrived example;

closures are quite fundamental in Haskell.

https://mail.haskell.org/pipermail/beginners/2009-July/002067.html

Combinator (1)

There are two distinct meanings of the word "combinator"

The first is a narrow, technical meaning, namely:
A function or definition with no free variables.

A "function with no free variables" is a **pure lambda-expression** that refers only to its **arguments**, like

```
\a -> a
\a -> \b -> a
\f -> \a -> \b -> f b a
```

and so on. The study of such things is called combinatory logic.

https://wiki.haskell.org/Combinator

Combinator (1)

The second meaning of "combinator" is a more informal sense referring to the combinator pattern, a style of organizing libraries centered around the idea of combining things.

This is the meaning of "combinator" which is more frequently encountered in the Haskell community.

Usually there is some type T, some functions for constructing "primitive" values of type T, and some "combinators" which can combine values of type T in various ways to build up more complex values of type T.

https://wiki.haskell.org/Combinator

Let binding

A **let binding** is very similar to a **where binding**.

A **where binding** is a syntactic construct that binds variables at the end of a function and the whole function (or a whole pattern-matching subpart) can see these variables, including all the guards

A **let binding** binds variables anywhere and is an expression itself, but its scope is tied to where the let expression appears. So if it’s defined within a guard, its scope is local and it will not be available for another guard. But it can also take global scope over all pattern-matching clauses of a function definition if it is defined at that level.

https://chercher.tech/haskell/let-bindings

Case expression

A case expression must have at least one alternative

and each alternative must have at least one body.

Each body must have the same type,

and the type of the whole expression is that type.

```
aaa x = case x of
            1 ->  "A"
            2 ->  "B"
            3 ->  "C"
```

Input: aaa 3

Output: "C"

http://zvon.org/other/haskell/Outputsyntax/caseQexpressions_reference.html

Polymorphism

A **value** is **polymorphic** if there is more than one type it can have.
Polymorphism is widespread in Haskell and
is a key feature of its type system.

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**.

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.

https://wiki.haskell.org/Polymorphism

Parametric Polymorphism

the function **id :: a -> a**

- contains an unconstrained type variable **a**

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

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

if a **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**,
the **input** and **output types** of the **function** given to **map**
must match up with the **list types**.

```
id :: a -> a
Char -> Char
Integer -> Integer
(Bool -> Maybe Bool) ->
        (Bool -> Maybe Bool)
```

https://wiki.haskell.org/Polymorphism

Ad-hoc Polymorphism

For example, the **+ operator** essentially
does something entirely different
when applied to **floating-point values** as compared to
when applied to **integer values**

Most languages support at least some ad-hoc polymorphism,

if a **type** can be compared for equality
then an **instance declaration** of the **Eq** class is given
if the behaviour of the **== operator** on the given **type** is specified,
all sorts of functions defined using that operator can be accessed

- checking if a value of the type is *present* in a list, or
- *looking* up a corresponding value in a list of pairs.

https://wiki.haskell.org/Polymorphism

References

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

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