Background – Constructors (1A)

Young Won Lim
11/30/17

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

This document was produced by using OpenOffice.

Based on

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Data Constructor

data **Color** = **Red** | **Green** | **Blue**

Color	is a type
Red	is a *constructor* that contains a *value* of type **Color**.
Green	is a *constructor* that contains a *value* of type **Color**.
Blue	is a *constructor* that contains a *value* of type **Color**.

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Data Constructor with Parameters

```
data Color = RGB Int Int Int
```

Color is a type

RGB is not a value but a *function* taking three Int's and *returning a value*

```
RGB :: Int -> Int -> Int -> Color
```

RGB is a **data constructor** that is a *function*
taking three Int values as its arguments,
and then uses them to construct a new value.

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Type Constructor

Consider a binary tree to store Strings

```
data SBTree = Leaf String  |  Branch String SBTree SBTree
```

a type

SBTree	is a **type**
Leaf	is a **data constructor** (a function)
Branch	is a **data constructor** (a function)

```
Leaf :: String -> SBTree
Branch :: String -> SBTree -> SBTree -> SBTree
```

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Similar Type Constructors

Consider a binary tree to store Strings

```
data SBTree = Leaf String  |   Branch String SBTree SBTree
```

Consider a binary tree to store Bool

```
data BBTree = Leaf Bool  |  Branch Bool BBTree BBTree
```

Consider a binary tree to store a parameter type

```
data BTree a = Leaf a  |   Branch a (BTree a) (BTree a)
```

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Type Constructor with a Parameter

Type constructors

Both **SBTree** and **BBTree** are type constructors

```
data SBTree = Leaf String  |   Branch String SBTree SBTree
data BBTree = Leaf Bool   |  Branch Bool BBTree BBTree
```

```
data BTree a = Leaf a  |   Branch a (BTree a) (BTree a)
```

Now we introduce a type variable a as a parameter to the type constructor.

BTree has become a function.
It takes a type as its argument and it returns a new type.

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Type Constructors and Data Constructors

A **type constructor**

- a "function" that takes 0 or more types
- gives you back a new **type**.

Type constructors with parameters
allows slight variations in types

type **SBTree** = **BTree** String
type **BBTree** = **BTree** Bool

A **data constructor**

- a "function" that takes 0 or more values
- gives you back a new **value**.

Data constructors with parameters
allows slight variations in values

RGB 12 92 27
#0c5c1b
RGB 255 0 0
RGB 0 255 0
RGB 0 0 255

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

( )

( ) is both a **type** and a **value**.

( ) is a special **type**, pronounced “unit”,
has one **value** **( )**, sometimes pronounced “void”

the **unit type** has only one **value** which is called **unit**.

( ) :: ( )

Type :: Expression

It is the same as the void type **void** in Java or C/C++.

https://stackoverflow.com/questions/20380465/what-do-parentheses-used-on-their-own-mean

Unit Type

a **unit type** is a type that allows *only one value* (and thus can hold *no information*).

It is the same as the void type **void** in Java or C/C++.

```
:t
Expression :: Type
```

```
data Unit = Unit

Prelude> :t Unit
Unit :: Unit
```

```
Prelude> :t ()
() :: ()
```

https://stackoverflow.com/questions/20380465/what-do-parentheses-used-on-their-own-mean

Type Language and Expression Language

data Tconst **T**var … **T**var = **V**const type … type **|** …
Vconst type … type

A new datatype declaration

Tconst (Type Constructor)	is added to *the type language*
Vconst (Value Constructor)	is added to *the expression language* and *its pattern sublanguage*
	must not appear in types

Argument types in **V**const type … type
are the types given to the arguments (**T**const **T**var … **T**var)
are used in expressions

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Datatype Declaration Examples

```
data Tree a =     Leaf |  Node (Tree a) (Tree a)
```

Tree	(Type Constructor)
Leaf or Node	(Value Constructor)

```
data Type = Value
```

```
data ( ) =   ( )
```

( )	(Type Constructor)
( )	(Value Constructor)

the type (), often pronounced "Unit"
the value (), sometimes pronounced "void"

the type () containing only one value ()

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Type Synonyms

```
type String = [Char]

phoneBook :: [(String,String)]
```

```
type PhoneBook = [(String,String)]

phoneBook :: PhoneBook
```

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]

phoneBook :: PhoneBook
```

```
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

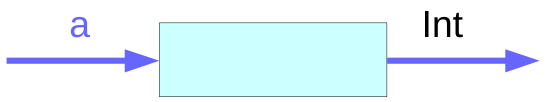
http://learnyouahaskell.com/making-our-own-types-and-typeclasses

Type Synonyms for Functions

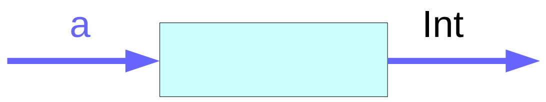
```
type Bag a = a -> Int

data Gems = Sapphire | Emerald | Diamond deriving (Show)
```

a -> Int



Bag a



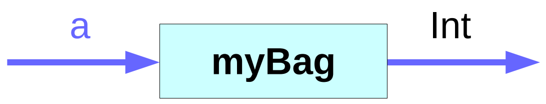
https://stackoverflow.com/questions/14166641/haskell-type-synonyms-for-functions

Type Synonyms for Functions

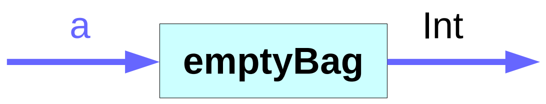
```
type Bag a = a -> Int

data Gems = Sapphire | Emerald | Diamond deriving (Show)
```

myBag :: Bag Gems



emptyBag :: Bag Gems



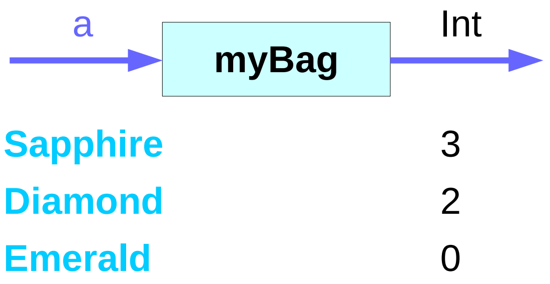
https://stackoverflow.com/questions/14166641/haskell-type-synonyms-for-functions

Type Synonyms for Functions

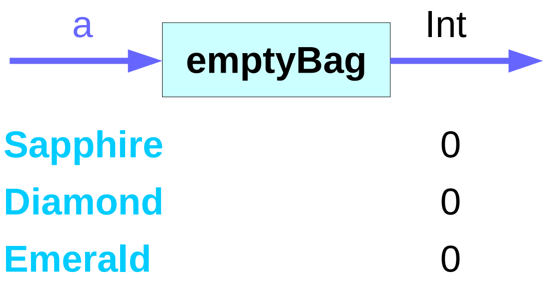
```
type Bag a = a -> Int

data Gems = Sapphire | Emerald | Diamond deriving (Show)
```

```
myBag :: Bag Gems
myBag Sapphire = 3
myBag Diamond = 2
myBag Emerald = 0
```



```
emptyBag :: Bag Gems
emptyBag Sapphire = 0
emptyBag Diamond = 0
emptyBag Emerald = 0
```



https://stackoverflow.com/questions/14166641/haskell-type-synonyms-for-functions

Record Syntax (named field)

```
data Configuration = Configuration
    { username          :: String
    , localHost         :: String
    , currentDir        :: String
    , homeDir           :: String
    , timeConnected     :: Integer
    }
```

```
username :: Configuration -> String                    -- accessor function  (automatic)
localHost :: Configuration -> String
-- etc.

changeDir :: Configuration -> String -> Configuration    -- update function
changeDir cfg newDir =
    if directoryExists newDir              -- make sure the directory exists
        then cfg { currentDir = newDir }
        else error "Directory does not exist"
```

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

newtype and **data**

$$\begin{tikzcd} \textbf{data} \arrow[r, shift left, "\times" marking] & \textbf{newtype} \arrow[l, shift left] \end{tikzcd}$$

Data can <u>only</u> be replaced with newtype
if the type has exactly *<u>one constructor</u>* with exactly *<u>one field</u>* inside it.

It ensures that the trivial **wrapping** and **unwrapping**
of the single field is eliminated by the **compiler**.

simple wrapper types such as **State** are usually defined with **newtype**.

type : used for type synonyms

```
newtype State s a = State { runState :: s -> (s, a) }
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

newtype examples

```
newtype Fd = Fd CInt
-- data Fd = Fd CInt would also be valid

-- newtypes can have deriving clauses just like normal types
newtype Identity a = Identity a
  deriving (Eq, Ord, Read, Show)

-- record syntax is still allowed, but only for one field
newtype State s a = State { runState :: s -> (s, a) }

-- this is *not* allowed:
-- newtype Pair a b = Pair { pairFst :: a, pairSnd :: b }
-- but this is:
– data Pair a b = Pair { pairFst :: a, pairSnd :: b }
-- and so is this:
newtype NPair a b = NPair (a, b)
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

References

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