- **Types.** Static, and resolved at compile time (before runtime).
- **Data.** At runtime (the real execution).

Data constructor arities

- **Arity.** Number of `args`.
- **Nullary.** Takes no arguments.
- **Unary.** Takes 1 argument.
- **Binary.** Takes 2 arguments.

```
data Example2 =
  Example2 Int String           -- product of Int and String
  deriving (Eq, Show)
```

Algebraic and Cardinality

Algebraic datatypes are algebraic since the patterns of argument structures using 2 basic operations: *sum* and *product*. And even **distributive**!

The **cardinality** a datatype is the number of possible values it defines.

1 newtype

A newtype has **no runtime overhead**, as it reuses the representation of the type it contains.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}           -- with this pragma!

class TooMany a where
  tooMany :: a -> Bool

instance TooMany Int where
  tooMany n = n > 42

newtype Goats =                                  -- newtype allows us to get it free!
  Goats Int deriving (Eq, Show, TooMany)          -- but we can actually define it!
```

If you want to do anything other than `TypeConstructor a1 a2 a3` ... (and as have to be type variables) than you need **Flexible Instances**.

```
{-# LANGUAGE FlexibleInstances          #-}

instance TooMany (Int, String) where -- require FlexibleInstances
  tooMany (x, _) = tooMany x
```

About the **bounds**, well.

```
Prelude> let n = Numba (-128)
--          Literal 128 is out of the
--          Int8 range -128..127 blah blah blah (complaining before negate)
Prelude> let n = (-128)
Prelude> let x = Numba n -- or use :set -XNegativeLiterals (not prevent the warnings)
```

Record syntax

The whole record must be declared instead of **partially** do it.

```
Prelude> let partialAf = Programmer Mac                --  OK
Prelude> let partialAf' = Programmer { os = Mac}       -- bottom
Prelude> partialAf'
Programmer {os = Mac, lang =
*** Exception: <interactive>:5:18-39: Missing field in record construction lang
```

And, better to use Maybe, rather than a data constructor `Null`. Better to split out the **product type** with the *type constructor*.

```
-- Split out the record/product
data Car = Car { make :: String
               , model :: String
               , year :: Integer }
               deriving (Eq, Show)
-- The Null is still not great, but
-- we're leaving it in to make a point
```

```
data Automobile = Null
                | Automobile Car
                deriving (Eq, Show)
Prelude> make Null -- the typechecker catch us b4 runtime
-- Blah blah complaining about type
```

Function type is exponential

The number of inhabitants of `a -> b` is b^a.

```
-- 3 ^ 3
quantFlip1 :: Quantum -> Quantum   -- Yes | No | Both
quantFlip1 Yes = Yes
quantFlip1 No = Yes
quantFlip1 Both = Yes

quantFlip2 :: Quantum -> Quantum          -- f a1 = b options
quantFlip2 Yes= Yes                       -- f a2 = b options
quantFlip2 No= Yes                        -- f a3 = b options
quantFlip2 Both = No                      --  b^a

quantFlip3 :: Quantum -> Quantum
quantFlip3 Yes= Yes
quantFlip3 No= Yes
quantFlip3 Both = Both
-- blah blah blah
```

2 Higher-kinded datatypes

Those that need to be fully applied are `* -> * -> *`.

```
data Silly a b c d =                  -- * -> * -> * -> * -> *
  MkSilly a b c d deriving Show       -- Silly Int String Int Int Int, or (,,,,)
```

Infix type and data constructors

All infix data constructors (type) start with a **colon** `(:)`.