

A datatype declaration defines a type constructor and data constructors.

↓
values of a particular type ; they are also **functions**.
that let us create data, or **values**, of a particular
type.

The need for types: Type systems in logic & mathematics have been designed to impose constraints that **enforce correctness**.

A type system defines the association between different parts of a program and checks that all parts fit together in a logically consistent, provable correct way.

Static type system enable many errors to be caught at compile time.

Good type systems also enable **compiler optimizations**, because the compiler can know and predict certain things about the execution of a program based on the types

→ Reading type signatures

(*) When we query the type of numeric values, we see type class information instead of a concrete type., because the compiler doesn't know which specific numeric type a value is until the type is either declared or the compiler is forced to infer a specific type based on a function.

values → fully applied
functions

$: t \ 13$ \rightarrow constrained polymorphic
 $13 :: \text{Num } a \Rightarrow a$

→ Understanding the function type

(*) ' \rightarrow ' is the type constructor for functions. It's a type constructor, very much like Bool , except the ' \rightarrow ' type constructor takes arguments and has no data constructors.

$: \text{info} (\rightarrow)$	$: \text{info} (,)$
$\text{data } (\rightarrow) \ a\ b$	$\text{data } (,) \ a,b = (,) \ a\ b$

Like the tuple type constructor which needs to be applied to two arguments, a fn. must have two arguments - one input & a result. But the fn. type constructor does not have data constructors.

↓

The value that shows up at term level is the function, hence **function are values**.

↓

Arrow is an infix operator that has two parameters and associates to the right. (although the fn. appn. is left associative)

$\xrightarrow{\text{two parameters}}$
 $\text{fst} :: (a,b) \xrightarrow{\text{a}} \text{the function type}$

→ Typeclass - constrained type variables

: t (f)

(f) :: Num a $\Rightarrow a \rightarrow a \rightarrow a$

: t (r)

(r) :: Fractional a $\Rightarrow a \rightarrow a \rightarrow a$

Compiler gives the least specific & more general type it can.

* Instead of limiting these fns to a concrete type, we get a typeclass - constrained polymorphic type variable.

* When a typeclass constraints a typeclass this way, the variable could represent any of the concrete types that have instances of that typeclass so that specific operations on which the fn. depends are defined for that type.

↓

Using typeclasses allows us to generalise an operation over multiple types.

* Multiple type constraints: (Ord a, Num a) $\Rightarrow a \rightarrow a \rightarrow$ ordering

→ Currying: As in lambda calculus, all fns. in Haskell take one argument and return one result. There is no built-in support for multiple arguments in Haskell.

Instead there are syntactic conveniences that construct curried fns. by default. Currying refers to the nesting of multiple fns., each accepting one argument & returning one

result, to allow the illusion of multiple-parameter fns.

Clearly evident from the defn. of the 'function type'



Each arrow in a type signature represents one argument and a result, with the final type being the final result.



for example: $(+): \text{Num} a \Rightarrow a \rightarrow a \rightarrow a$ final result

↖
type class constraint,
Addition is defined in
that typeclass

↓
represents successive
function applications

↓
The function on the top
is returning another function
that accepts the next
argument.

This is equivalent to $(+): \text{Num} a \Rightarrow a \rightarrow (a \rightarrow a)$



this kind of definition makes
varying the default behaviour
in Haskell.

*) Like in lambda expression, applying the expression to one argument returns a partially applied fn. which awaits application to a second argument. After we apply it to the second argument, we have the final result.

* one argument, one result (no matter how many nested lambdas)

* explicit parenthesization, as when an input parameter is itself a fn., may be used to indicate order of evaluation.

→ Partial Application : The ability to apply only some of a function's arguments is called partial application.

`addStuff :: Integer → (Integer → Integer)`



Applying `addStuff` to one argument gives us the function `addTen`, which is the return function of `addStuff`. Now applying `addTen` to an argument gives us a return value, and no more function arrows left.

→ Manual currying and uncurrying



$f \ a \ b = f(a, b)$ means un-nesting the functions

$f(a, b) = f \ a \ b.$ and replacing the two functions with a tuple of two values



$(+): \text{Num } a \Rightarrow (a, a) \rightarrow a$ (better fits the description of addition)



an example of manual currying using lambda fns.

`anonNested :: Integer → Bool → Integer`

anonNested = $\lambda i \rightarrow \lambda b \rightarrow i + \text{nonsense } b$.

anonNested 10 $\rightarrow \lambda b \rightarrow 10 + \text{nonsense } b$.

→ Sectioning : partial application of infix operators.

let $x = 5$

let $y = (2^)$

let $z = (^2)$

$yx \rightarrow 32$

$zx \rightarrow 25$

[.] → shorthand to
create a list of
all elements between
the first & last
values.

↓
Allows us to choose whether
the argument we're partially
applying the operator to, is
the first or second argument.

↓
argument order plays an important
role with noncommutative operators.

↓
not limited to arithmetic only

for example let celebrate = $(++ \text{ ``woot!''})$

↑
 $\gg \text{celebrate ``naptime''}$
 $\Rightarrow ``\text{naptime woot!}''$

↓
This syntax can be used with prefix based fns.
by making them infix.

↑
9 `elem` [1..10]

↓
let c = (^ elem` [1..10])

c 9 $\rightarrow \text{True}$.

Sectioning syntax
exists to allow
some freedom in
which arg. of a
binary op. we apply
a func. to.

Exercises : type arguments

let $f :: a \rightarrow a \rightarrow a \rightarrow a$; $f = \text{undefined}$

let $x :: \text{Char}$; $x = \text{undefined}$

: $f\ f\ x$

$\Rightarrow \text{Char} \rightarrow \text{Char} \rightarrow \text{Char}$

Y
↓

we can check the type
of things that haven't
been implement yet,
as long as we give
it a undefined value in
ghci.

→ polymorphism