

INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

- ▶ Numerical Type Classes
- ▶ Semigroup
- ▶ Newtype
- ▶ Record
- ▶ Monoid
- ▶ Functor

NUMERICAL DATA TYPES

Type	Description
Double	Double-precision floating point. A common choice for floating-point data.
Float	Single-precision floating point. Often used when interfacing with C.
Int	Fixed-precision signed integer; minimum range [-2^29..2^29-1]. Commonly used.
Int8	8-bit signed integer
Int16	16-bit signed integer
Int32	32-bit signed integer
Int64	64-bit signed integer
Integer	Arbitrary-precision signed integer; range limited only by machine resources. Commonly used.
Rational	Arbitrary-precision rational numbers. Stored as a ratio of two Integers.
Word	Fixed-precision unsigned integer; storage size same as Int
Word8	8-bit unsigned integer
Word16	16-bit unsigned integer
Word32	32-bit unsigned integer
Word64	64-bit unsigned integer

SOME NUMERICAL OPERATIONS

Function	Type	Description
(+), (-), (*)	Num a => a -> a -> a	Addition, subtraction, multiplication
(/)	Fractional a => a -> a -> a	Fractional division
(^)	(Num a, Integral b) => a -> b -> a	Raise a number to a non-negative, integral power
(^^)	(Fractional a, Integral b) => a -> b -> a	Raise a fractional number to any integral power
(**), logBase	Floating a => a -> a -> a	Raise to the power of, log with explicit base
abs	Num a => a -> a	Absolute value
maxBound, minBound	Bounded a => a	The maximum/minimum value of a bounded type
pi	Floating a => a	Constant pi
sin, cos, tan, log, sqrt	Floating a => a -> a	Sine, cosine, tangent, natural logarithm, square root
div, mod, quot, rem	Integral a => a -> a -> a	Integer division, modulus, remainder
fromInteger	Num a => Integer -> a	Conversion from Integer into a numeric value
toInteger	Integral a => a -> Integer	Conversion of any Integral to Integer
fromRational	Fractional a => Rational -> a	Possibly lossy conversion from a Rational
toRational	Real a => a -> Rational	Convert losslessly to Rational
approxRational	RealFrac a => a -> a -> Rational	Ratio composition
fromIntegral	(Integral a, Num b) => a -> b	Convert from any Integral to any numeric type
round, truncate	(RealFrac a, Integral b) => a -> b	Rounds to nearest integer; truncates towards zero

TYPE CLASS INSTANCES FOR NUMERIC TYPES

Type	Bits	Bounded	Floating	Fractional	Integral	Num	Real	RealFrac
Double			X	X		X	X	X
Float			X	X		X	X	X
Int	X	X			X	X	X	
Int16	X	X			X	X	X	
Int32	X	X			X	X	X	
Int64	X	X			X	X	X	
Integer	X				X	X	X	
Rational or any Ratio				X		X	X	X
Word	X	X			X	X	X	
Word16	X	X			X	X	X	
Word32	X	X			X	X	X	
Word64	X	X			X	X	X	

NUMERIC TYPE CLASSES

NUM-BERS

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.V...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> :info Num
type Num :: * -> Constraint
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
  -- Defined in 'GHC.Num'
instance Num Double -- Defined in 'GHC.Float'
instance Num Float -- Defined in 'GHC.Float'
instance Num Int -- Defined in 'GHC.Num'
instance Num Integer -- Defined in 'GHC.Num'
instance Num Word -- Defined in 'GHC.Num'
ghci>
```

class Num a where

Source

Basic numeric class.

The Haskell Report defines no laws for Num. However, (+) and (*) are customarily expected to define a ring and have the following properties:

Associativity of (+)

$$(x + y) + z = x + (y + z)$$

Commutativity of (+)

$$x + y = y + x$$

fromInteger 0 is the additive identity

$$x + \text{fromInteger } 0 = x$$

negate gives the additive inverse

$$x + \text{negate } x = \text{fromInteger } 0$$

Associativity of (*)

$$(x * y) * z = x * (y * z)$$

fromInteger 1 is the multiplicative identity

$$x * \text{fromInteger } 1 = x \text{ and } \text{fromInteger } 1 * x = x$$

Distributivity of (*) with respect to (+)

$$a * (b + c) = (a * b) + (a * c) \text{ and } (b + c) * a = (b * a) + (c * a)$$

Coherence with toInteger

if the type also implements Integral, then fromInteger is a left inverse for toInteger, i.e. $\text{fromInteger}(\text{toInteger } i) == i$

Note that it isn't customarily expected that a type instance of both Num and Ord implement an ordered ring. Indeed, in base only Integer and Rational do.

Minimal complete definition

(+), (*), abs, signum, fromInteger, (negate | (-))

NUMERIC TYPE CLASSES

INTEGRAL

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.V...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> :info Integral
type Integral :: * -> Constraint
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
{-# MINIMAL quotRem, toInteger #-}
  -- Defined in 'GHC.Real'
instance Integral Int -- Defined in 'GHC.Real'
instance Integral Integer -- Defined in 'GHC.Real'
instance Integral Word -- Defined in 'GHC.Real'
ghci>
```

class (Real a, Enum a) => Integral a where # Source

Integral numbers, supporting integer division.

The Haskell Report defines no laws for `Integral`. However, `Integral` instances are customarily expected to define a Euclidean domain and have the following properties for the `div/mod` and `quot/rem` pairs, given suitable Euclidean functions `f` and `g`:

- $x = y * \text{quot } x \text{ } y + \text{rem } x \text{ } y$ with $\text{rem } x \text{ } y = \text{fromInteger } 0$ or $g(\text{rem } x \text{ } y) < g \text{ } y$
- $x = y * \text{div } x \text{ } y + \text{mod } x \text{ } y$ with $\text{mod } x \text{ } y = \text{fromInteger } 0$ or $f(\text{mod } x \text{ } y) < f \text{ } y$

An example of a suitable Euclidean function, for `Integer`'s instance, is `abs`.

In addition, `toInteger` should be total, and `fromInteger` should be a left inverse for it, i.e. `fromInteger (toInteger i) = i`.

Minimal complete definition

`quotRem, toInteger`

FRACTIONAL

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.V...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> :info Fractional
type Fractional :: * -> Constraint
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
      -- Defined in 'GHC.Real'
instance Fractional Double -- Defined in 'GHC.Float'
instance Fractional Float -- Defined in 'GHC.Float'
ghci>
```

class Num a => Fractional a where # Source

Fractional numbers, supporting real division.

The Haskell Report defines no laws for **Fractional**. However, (+) and (*) are customarily expected to define a division ring and have the following properties:

- recip gives the multiplicative inverse**
 $x * recip x = recip x * x = fromInteger 1$
- Totality of **toRational****
toRational is total
- Coherence with **toRational****
if the type also implements **Real**, then **fromRational** is a left inverse for **toRational**, i.e. $fromRational (toRational i) = i$

Note that it *isn't* customarily expected that a type instance of **Fractional** implement a field. However, all instances in base do.

Minimal complete definition

fromRational, (recip | (/))

NUMERIC TYPE CLASSES

FLOATING

```
[ghci] > :info Floating
type Floating :: * -> Constraint
class Fractional a => Floating a where
    pi :: a
    exp :: a -> a
    log :: a -> a
    sqrt :: a -> a
    (**) :: a -> a -> a
    logBase :: a -> a -> a
    sin :: a -> a
    cos :: a -> a
    tan :: a -> a
    asin :: a -> a
    acos :: a -> a
    atan :: a -> a
    sinh :: a -> a
    cosh :: a -> a
    tanh :: a -> a
    asinh :: a -> a
    acosh :: a -> a
    atanh :: a -> a
    GHC.Float.log1p :: a -> a
    GHC.Float.expm1 :: a -> a
    GHC.Float.log1pexp :: a -> a
    GHC.Float.log1mexp :: a -> a
{-# MINIMAL pi, exp, log, sin, cos, asin, acos, atan, sinh, co
sh,
    asinh, acosh, atanh #-}
-- Defined in 'GHC.Float'
instance Floating Double -- Defined in 'GHC.Float'
instance Floating Float -- Defined in 'GHC.Float'
```

class Fractional a => Floating a where # Source

Trigonometric and hyperbolic functions and related functions.

The Haskell Report defines no laws for `Floating`. However, `(+)`, `(*)` and `exp` are customarily expected to define an exponential field and have the following properties:

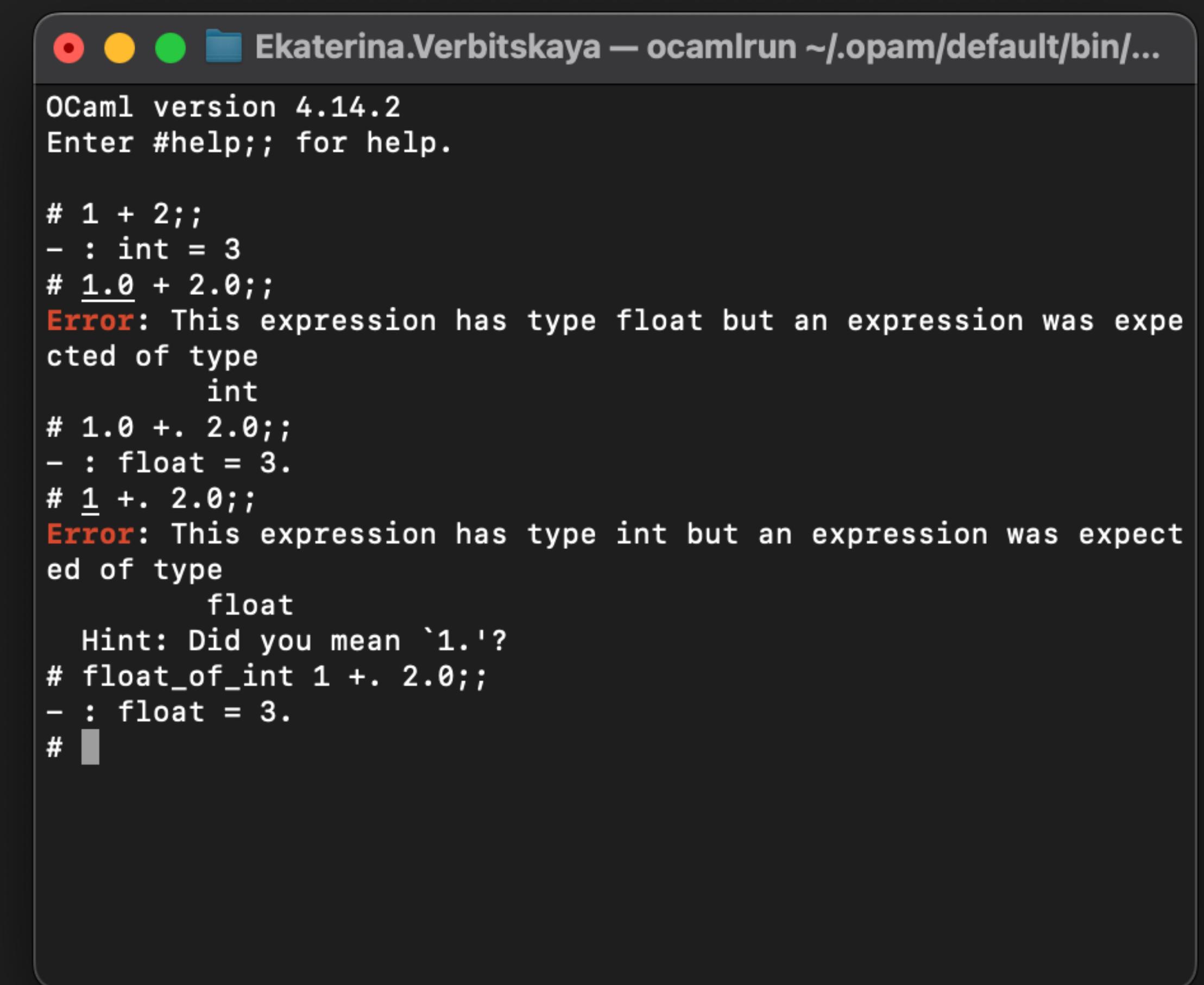
- $\exp(a + b) = \exp a * \exp b$
- $\exp(\text{fromInteger } 0) = \text{fromInteger } 1$

Minimal complete definition

`pi, exp, log, sin, cos, asin, acos, atan, sinh, cosh, asinh,`
`acosh, atanh`

SEEMS STUPID? IT COULD HAVE BEEN WORSE!

- ▶ OCaml – another statically typed functional programming language
- ▶ It doesn't have type classes
- ▶ `1.0` and `1` are not the same
- ▶ Functions over numbers don't convert any type on their own
- ▶ Specific operations for specific types



The screenshot shows a terminal window titled "Ekaterina.Verbitskaya — ocamlrun ~/.opam/default/bin/...". The OCaml version is 4.14.2, and it prompts for help with "#help;;". The user attempts several arithmetic operations:

- # `1 + 2;;` - : int = 3
- # `1.0 + 2.0;;` Error: This expression has type float but an expression was expected of type int
- # `1.0 +. 2.0;;` - : float = 3.
- # `1 +. 2.0;;` Error: This expression has type int but an expression was expected of type float
Hint: Did you mean `1.`?
- # `float_of_int 1 +. 2.0;;` - : float = 3.
- #

EXERCISE

- ▶ Make `Expr` an instance of `Num`
- ▶ Ignore those functions that aren't described by the `Expr` data type

OUTLINE

- ▶ Numerical Type Classes
- ▶ Semigroup
- ▶ Newtype
- ▶ Record
- ▶ Monoid
- ▶ Functor

SEMIGROUP

- ▶ Abstracts a type with an associative binary operation

▶ [doc](#)

- ▶ Which operations are semigroups?

▶ Integer addition

▶ Integer subtraction

▶ List concatenation

`class Semigroup a where`

Source

The class of semigroups (types with an associative binary operation).

Instances should satisfy the following:

Associativity

$$x \text{ } \langle\!\rangle \text{ } (y \text{ } \langle\!\rangle \text{ } z) = (x \text{ } \langle\!\rangle \text{ } y) \text{ } \langle\!\rangle \text{ } z$$

You can alternatively define `sconcat` instead of `(⟨⟩)`, in which case the laws are:

Unit

$$\text{sconcat} (\text{pure } x) = x$$

Multiplication

$$\text{sconcat} (\text{join } \text{xss}) = \text{sconcat} (\text{fmap sconcat } \text{xss})$$

Since: base-4.9.0.0

Minimal complete definition

`(⟨⟩) | sconcat`

EXAMPLES OF SEMI GROUP

- ▶ List concatenation
- ▶ String concatenation
- ▶ Merging two **Maybe** values
- ▶ What's the result of **Nothing** \diamond **Nothing**?

```
● ● ● fp — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib --interactive — 65x24
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> [1,2,3] <> [4,5]
[1,2,3,4,5]
[ghci> "Hello, " <> "world"
"Hello, world"
[ghci> Just [1,2,3] <> Just [4,5]
Just [1,2,3,4,5]
[ghci> Just [1,2,3] <> Nothing
Just [1,2,3]
[ghci> Nothing <> Just [4,5]
Just [4,5]
ghci> Nothing <> Nothing
```

EXAMPLES OF SEMI GROUP

- ▶ List concatenation
- ▶ String concatenation
- ▶ Merging two **Maybe** values
 - ▶ What's the result of **Nothing** \diamond **Nothing**?

```
● ● ● fp — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib --interactive — 65x24
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> [1,2,3] <> [4,5]
[1,2,3,4,5]
[ghci> "Hello, " <> "world"
"Hello, world"
[ghci> Just [1,2,3] <> Just [4,5]
Just [1,2,3,4,5]
[ghci> Just [1,2,3] <> Nothing
Just [1,2,3]
[ghci> Nothing <> Just [4,5]
Just [4,5]
[ghci> Nothing <> Nothing
Nothing
ghci> ]
```

IMPLEMENTATIONS OF SEMIGROUPS

- ▶ For `List`, we just concatenate
- ▶ For `Maybe a`, we concatenate the underlying values of type `a`, if there are any
- ▶ For `Int?`

```
instance Semigroup [a] where
  ( $\diamond$ ) = (++)

instance Semigroup a  $\Rightarrow$  Semigroup (Maybe a) where
  Just x  $\diamond$  Just y = Just (x  $\diamond$  y)
  x  $\diamond$  Nothing = x
  Nothing  $\diamond$  y = y
```

OUTLINE

- ▶ Numerical Type Classes
- ▶ Semigroup
- ▶ Newtype
- ▶ Record
- ▶ Monoid
- ▶ Functor

NUMBERS: WHICH SEMIGROUP TO CHOOSE?

- ▶ There are two associative operations over `Num`

- ▶ `(+)`

- ▶ `(*)`

```
● ● ● fp — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib --interactive — 65x24
GHCi, version 9.6.6: https://www.haskell.org/ghc/  ?: for help
ghci> :set -XUndecidableInstances
ghci> :{
ghci| instance Num a => Semigroup a where
ghci|   x <> y = x + y
ghci| :}
ghci> 2 <> 3
5
ghci>
ghci> :{
ghci| instance Num a => Semigroup a where
ghci|   x <> y = x * y
ghci| :}
ghci> 2 <> 3
6
ghci>
```

NUMBERS: WHICH SEMIGROUP TO CHOOSE?

- ▶ There are two associative operations over `Num`
- ▶ `(+)`
- ▶ `(*)`
- ▶ Two instances cannot coexist easily
- ▶ Solution: `newtype`

```
fp - ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib --interactive — 65x24
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :set -XUndecidableInstances
[ghci]> :{
[ghci| instance Num a => Semigroup a where
[ghci|   x <> y = x + y
[ghci|
[ghci| instance Num a => Semigroup a where
[ghci|   x <> y = x * y
[ghci| :}
[ghci]>
[ghci]> 2 <> 3

<interactive>:10:1: error: [GHC-3999]
  • Ambiguous type variable 'a0' arising from a use of 'print'
    prevents the constraint '(Show a0)' from being solved.
    Probable fix: use a type annotation to specify what 'a0' sh
ould be.
      Potentially matching instances:
        instance Show Ordering -- Defined in 'GHC.Show'
        instance Show a => Show (Maybe a) -- Defined in 'GHC.Show'
,
  ...plus 25 others
  ...plus 21 instances involving out-of-scope types
  (use -fprint-potential-instances to see them all)
```

NEWTYPE

- ▶ Zero-weight wrapper
- ▶ Allows to define multiple behaviours for the same underlying data type
- ▶ Similar to **data**, but
 - ▶ Only one constructor
 - ▶ Only one field

```
fp — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib --interactive — 65x24
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> :{
[ghci| newtype Sum a = Sum a deriving (Show)
[ghci| instance Num a => Semigroup (Sum a) where
[ghci|   Sum x <> Sum y = Sum (x + y)
[ghci|
[ghci| newtype Prod a = Prod a deriving (Show)
[ghci| instance Num a => Semigroup (Prod a) where
[ghci|   Prod x <> Prod y = Prod (x * y)
[ghci| :}
[ghci> Sum 2 <> Sum 3
Sum 5
[ghci> Prod 2 <> Prod 3
Prod 6
[ghci> :m GHC.Base
[ghci> stimes 5 $ Sum 2
Sum 10
[ghci> stimes 5 $ Prod 2
Prod 32
ghci> ]
```

OUTLINE

- ▶ Numerical Type Classes
- ▶ Semigroup
- ▶ Newtype
- ▶ Record
- ▶ Monoid
- ▶ Functor

RECORD SYNTAX IN NEWTYPES

- ▶ You can name fields of a constructor
- ▶ Most useful when you only have one constructor
- ▶ Be careful, names are global

```
fp — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib --interactive — 65x24
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> :{
[ghci| newtype Sum a = Sum { getSum :: a } deriving (Show)
[ghci| instance Num a => Semigroup (Sum a) where
[ghci|   x <> y = Sum (getSum x + getSum y)
[ghci|
[ghci| newtype Prod a = Prod { getProd :: a } deriving (Show)
[ghci| instance Num a => Semigroup (Prod a) where
[ghci|   x <> y = Prod (getProd x * getProd y)
[ghci| :}
[ghci> :t getSum
getSum :: Sum a -> a
[ghci> getSum $ Sum 2 <> Sum 3
5
[ghci> sum = getSum . foldr1 (<>) . map Sum
[ghci> :t sum
sum :: Num c => [c] -> c
[ghci> sum [1,2,3,4]
10
ghci> ]
```

RECORD SYNTAX IN DATA

- ▶ When you have multiple fields in a constructor all of which have the same non-descriptive type
- ▶ Each field name is a function

```
module Record where

data Person = Person String String Int String
            deriving (Show)

getAge :: Person → Int
getAge (Person _ _ age _) = age

me = Person "Ekaterina" "Verbitskaia" 31 "Netherlands"

data Person' = Person' { firstName :: String
                       , lastName :: String
                       , age :: Int
                       , residence :: String
                     }
            deriving (Show)

me' = Person "Ekaterina" "Verbitskaia" 31 "Netherlands"

getAge' = age
```

PATTERN MATCHING AND VALUE CREATION

- ▶ No need to use wildcards and remember what the order of the field is
- ▶ You can bind variables in a pattern match
- ▶ Order of the fields is not important when they are referenced by name

```
module Record where

  data Person = Person String String Int String
               deriving (Show)

  me = Person "Ekaterina" "Verbitskaia" 31 "Netherlands"
  notMe = Person "Verbitskaia" "Ekaterina" 31 "Netherlands"

  data Person' = Person' { firstName :: String
                         , lastName :: String
                         , age :: Int
                         , residence :: String
                         }
               deriving (Show)

  me' = Person' "Ekaterina" "Verbitskaia" 31 "Netherlands"

  me'' = Person' { lastName = "Verbitskaia"
                  , firstName = "Ekaterina"
                  , residence = "Netherlands"
                  , age = 30
                  }

  fullName (Person' { firstName = fn, lastName } ) =
    fn ++ ' ' : lastName
```

EASY UPDATES

- ▶ No mutation happens!
- ▶ A new value is created, with all old fields the same, and only the modified one changed

```
module Record where

data Person = Person String String Int String
            deriving (Show)

me = Person "Ekaterina" "Verbitskaia" 31 "Netherlands"

ageUp :: Person → Person
ageUp (Person fN lN age r) = Person fN lN (age+1) r

data Person' = Person' { firstName :: String
                       , lastName :: String
                       , age :: Int
                       , residence :: String
                     }
            deriving (Show)

me' = Person' "Ekaterina" "Verbitskaia" 31 "Netherlands"

ageUp' :: Person' → Person'
ageUp' person = person { age = age person + 1 }
```

OUTLINE

- ▶ Numerical Type Classes
- ▶ Semigroup
- ▶ Newtype
- ▶ Record
- ▶ Monoid
- ▶ Functor

MONOID

- ▶ When your binary associative operation has a neutral element

▶ `mempty ◊ x = x ◊ mempty = x`

- ▶ What are `mempty` of the types?

▶ `[a]`

▶ `Maybe a`

▶ `Int`

▶ `Bool`

```
class Semigroup a => Monoid a where
```

Source

The class of monoids (types with an associative binary operation that has an identity). Instances should satisfy the following:

Right identity

`x ◊ mempty = x`

Left identity

`mempty ◊ x = x`

Associativity

`x ◊ (y ◊ z) = (x ◊ y) ◊ z` (Semigroup law)

Concatenation

`mconcat = foldr (◊) mempty`

You can alternatively define `mconcat` instead of `mempty`, in which case the laws are:

Unit

`mconcat (pure x) = x`

Multiplication

`mconcat (join xss) = mconcat (fmap mconcat xss)`

Subclass

`mconcat (toList xs) = sconcat xs`

The method names refer to the monoid of lists under concatenation, but there are many other instances.

Some types can be viewed as a monoid in more than one way, e.g. both addition and multiplication on numbers. In such cases we often define newtypes and make those instances of `Monoid`, e.g. `Sum` and `Product`.

NOTE: `Semigroup` is a superclass of `Monoid` since `base-4.11.0.0`.

Minimal complete definition

`mempty | mconcat`

EXAMPLES

- ▶ `or [True, False, True] = True`
- ▶ `and [True, False, True] = False`
- ▶ `all even [1,2,3] = False`
- ▶ `any even [1,2,3] = True`
- ▶ `(Nothing, [1,2,3]) ◊ (Just "hi", [4,5]) = (Just "hi", [1,2,3,4,5])`

`Any x ◊ Any y = Any (x || y)`

`All x ◊ All y = All (x && y)`

`Sum x ◊ Sum y = Sum (x + y)`

`Product x ◊ Product y = Product (x * y)`

`Endo f ◊ Endo g = Endo (f . g)`

`(x, y) ◊ (z, t) = (x ◊ z, y ◊ t)`

OUTLINE

- ▶ Numerical Type Classes
- ▶ Semigroup
- ▶ Newtype
- ▶ Record
- ▶ Monoid
- ▶ Functor

FUNCTION

- ▶ Generalisation of list map
- ▶ Transforms content, doesn't change shape

```
class Functor (f :: Type -> Type) where
```

Source

A type `f` is a Functor if it provides a function `fmap` which, given any types `a` and `b` lets you apply any function from `(a -> b)` to turn an `f a` into an `f b`, preserving the structure of `f`. Furthermore `f` needs to adhere to the following:

Identity

`fmap id == id`

Composition

`fmap (f . g) == fmap f . fmap g`

Note, that the second law follows from the free theorem of the type `fmap` and the first law, so you need only check that the former condition holds. See these articles by School of Haskell or David Loposchansky for an explanation.

Minimal complete definition

`fmap`

FUNCTOR

EXAMPLE

```
data Tree a
= E
| N (Tree a) a (Tree a)
deriving (Show, Eq, Read)

instance Functor Tree where
  fmap :: (a → b) → Tree a → Tree b
  fmap f E = E
  fmap f (N l x r) = N (fmap f l) (f x) (fmap f r)
```

```
● ● ● L07 — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib --interactive Tree.hs — 6...
GHCi, version 9.6.6: https://www.haskell.org/ghc/ :? for help
[1 of 2] Compiling Main           ( Tree.hs, interpreted )
Ok, one module loaded.
ghci> tree = N (N E 1 E) 2 (N E 3 E)
ghci> fmap (const 0) tree
N (N E 0 E) 0 (N E 0 E)
ghci> fmap (+1) tree
N (N E 2 E) 3 (N E 4 E)
ghci> fmap show tree
N (N E "1" E) "2" (N E "3" E)
ghci>
```

FUNCTOR

EXAMPLE

```
data Tree a
= E
| N (Tree a) a (Tree a)
deriving (Show, Eq, Read)

instance Functor Tree where
  fmap :: (a → b) → Tree a → Tree b
  fmap f E = E
  fmap f (N l x r) = N (fmap f l) (f x) (fmap f r)
```

```
● ● ● L07 — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib --interactive Tree.hs — 6...
GHCi, version 9.6.6: https://www.haskell.org/ghc/ :? for help
[1 of 2] Compiling Main           ( Tree.hs, interpreted )
Ok, one module loaded.
ghci> tree = N (N E 1 E) 2 (N E 3 E)
ghci> fmap (const 0) tree
N (N E 0 E) 0 (N E 0 E)
ghci> fmap (+1) tree
N (N E 2 E) 3 (N E 4 E)
ghci> fmap show tree
N (N E "1" E) "2" (N E "3" E)
ghci> const 0 <$> tree
N (N E 0 E) 0 (N E 0 E)
ghci> (+1) <$> tree
N (N E 2 E) 3 (N E 4 E)
ghci> show <$> tree
N (N E "1" E) "2" (N E "3" E)
ghci>
```