Functional programming
Lecture 03 — Types and classes

Stéphane Vialette
stephane.vialette@univ-eiffel.fr

May 14, 2023

Laboratoire d'Informatique Gaspard-Monge, UMR CNRS 8049,
Université Gustave Eiffel

Type synonyms

Type synonyms

- The simplest way of declaring a new type is to introduce a new name for an existing type.
- In Haskell, you can create new type synonyms by using the `type` keyword.
- The name of the new type must begin with a capital letter.

For example, the standard prelude states that the type `String` is just a synonym for the type `[Char]` of list of characters:

```
type String = [Char]
```

Type synonyms

```
distance1 :: (Int, Int) -> (Int, Int) -> Float
distance1 (x1, y1) (x2, y2) = sqrt (dx^2 + dy^2)
  where
    dx = fromIntegral (x2-x1)
    dy = fromIntegral (y2-y1)

translate1 :: (Int, Int) -> (Int, Int) -> (Int, Int)
translate1 (dx, dy) (x, y) = (x+dx, y+dy)
```

Type synonyms

```haskell
type Point = (Int, Int)
type Vect = (Int, Int)

distance2 :: Point -> Point -> Float
distance2 (x1, y1) (x2, y2) = sqrt (dx^2 + dy^2)
  where
    dx = fromIntegral (x2-x1)
    dy = fromIntegral (y2-y1)

translate2 :: Vect -> Point -> Point
translate2 (dx, dy) (x, y) = (x+dx, y+dy)
```

Type synonyms

Type declarations can be nested (in the sense that one such type can be declared in terms of another).

```
type Point = (Int, Int)

type Polygon = [Point]

type Triangle = (Point, Point, Point)

type Triangulation = [Triangle]

type PolygonTriangulation = Polygon -> Triangulation
```

Type synonyms

Type declarations cannot be recursive

```
type Tree = (Int, [Tree])
```

```
error:
  Cycle in type synonym declarations:
    type Tree = (Int, [Tree])
```

Type synonyms

Type declarations can be parameterized by other types.

```
type Point a = (a, a)
type Vect a = (a, a)

distance3 :: Integral a => Point a -> Point a -> Float
distance3 (x1, y1) (x2, y2) = sqrt (dx^2 + dy^2)
  where
    dx = fromIntegral (x2-x1)
    dy = fromIntegral (y2-y1)

translate3 :: Integral a =>  Vect a -> Point a -> Point a
translate3 (dx, dy) (x, y) = (x+dx, y+dy)
```

Type synonyms

Type synonyms make reading type signature much easier.

```
pInfo :: String -> String -> Int -> Int -> String
pInfo fName lName age height = ...
```

Type synonyms

Type synonyms make reading type signature much easier.

```
type FirstName = String
type LastName  = String
type Age       = Int
type Height    = Int

pInfo :: FirstName -> LastName -> Age -> Height -> String
pInfo fName lName age height = ...
```

Mixing

```
λ > type T = Int
λ > :type (2 :: T) + (3 :: T)
(2 :: T) + (3 :: T) :: T
λ > :type (2 :: T) + (3 :: Int)
(2 :: T) + (3 :: Int) :: T
λ > :type (2 :: Int) + (3 :: T)
(2 :: Int) + (3 :: T) :: Int
λ > :type (2+3)
(2+3) :: Num a => a
λ > :type (2+3) :: T
(2+3) :: T :: T
```

Mixing

```
λ > type T1 = Int
λ > type T2 = Int
λ > :type (2 :: T1) + (3 :: T2)
(2 :: T1) + (3 :: T2) :: T1
λ > :type (2 :: T2) + (3 :: T1)
(2 :: T2) + (3 :: T1) :: T2
λ > :type ((2 :: T1) + (3 :: T1)) :: T2
((2 :: T1) + (3 :: T1)) :: T2 :: T2
λ > :type ((2 :: T2) + (3 :: T2)) :: T1
((2 :: T2) + (3 :: T2)) :: T1 :: T1
λ > :type (1 :: Int) + (2 :: T1) + (3 :: T2)
(1 :: Int) + (2 :: T1) + (3 :: T2) :: Int
```

Type synonyms & functions

```
type Point = (Int, Int)
type Vect = (Int, Int)

type BW = Bool
type ImageBW = Point -> BW

inSquare :: Point -> Point -> ImageBW
inSquare (llx, lly) (urx, ury)
         = \ (x, y) -> x >= llx && x <= urx &&
                       y >= lly && y <= ury

move :: Vect -> ImageBW -> ImageBW
move (dx, dy) i = \ (x, y) -> i (x-dx, y-dy)
```

Type synonyms & functions

```
λ > p = inSquare (0,0) (1,1)

λ > :type p
i :: ImageBW

λ > [((x, y), p (x, y)) |  x <- [-1..2], y <- [-1..2]]
[((-1,-1),False), ((-1, 0),False), ((-1, 1),False),
 ((-1, 2),False), (( 0,-1),False), (( 0, 0), True),
 (( 0, 1), True), (( 0, 2),False), (( 1,-1),False),
 (( 1, 0), True), (( 1, 1), True), (( 1, 2),False),
 (( 2,-1),False), (( 2, 0),False), (( 2, 1),False),
 (( 2, 2),False)]
```

Type synonyms & functions

```
λ > img1 = \ (x, y) -> x > 10 && y > 5

λ > [((x, y), img1 (x, y)) |  x <- [10,11], y <- [5,6]]
[((10,5),False),((10,6),False),((11,5),False),((11,6),True)]

λ > img2 = move (1, 3) img2

λ > [((x, y), img2 (x, y)) |  x <- [11,12], y <- [8,9]]
[((11,8),False),((11,9),False),((12,8),False),((12,9),True)]

λ > img3 = move (1,1) . move (3,-2) . move (-1,1) $ img1

λ > [((x, y), img3 (x, y)) |  x <- [13,14], y <- [5,6]]
```

Algebraic types

Type synonyms

Algebraic types

Record types

`newtype` declaration

Case study: Tautology checker

Data declaration

- A completely new type, as opposed to a synonym for an existing type, can be declared by specifying its values using the `data` mechanism of Haskell.
- Algebraic data type definitions are introduced by the keyword `data` followed by
  - the name of the type,
  - the `=` symbol, and then
  - the constructors of the type being defined.
- The name of the type and the names of the constructors begin with capital letters.

Enumerated types

Enumerated types are defined by enumerating the elements of the type.

```
data Temp = Cold | Hot
```

```
λ> :type Cold
Cold :: Temp
```

```
λ> :type Hot
Hot :: Temp
```

`Cold` and `Hot` are the two constructors of the type `Temp`.

Enumerated types

Enumerated types are defined by enumerating the elements of the type.

```
data Season = Spring | Summer | Autumn | Winter
```

```
λ > :type Spring
Spring :: Season
```

```
λ >  :type Summer
Summer :: Season
```

```
λ >  :type Autumn
Autumn :: Season
```

```
λ > : type Winter
Winter :: Season
```

`Spring`, `Summer`, `Autumn` anr `Winter` are the four constructors of the type `Season`.

Enumerated types

To define functions over enumerated types we use pattern matching.

```
toString :: Season -> String
toString Spring = "spring"
toString Summer = "summer"
toString Autumn = "autumn"
toString Winter = "winter"

britishTemp :: Season -> Temp
britishTemp Winter = Cold
britishTemp _      = Hot
```

Enumerated types

To define functions over enumerated types we use pattern matching.

```
instance Show Season where
  show Spring = "spring"
  show Summer = "summer"
  show Autumn = "autumn"
  show Winter = "winter"
```

Enumerated types

To define functions over enumerated types we use pattern matching.

```
instance Eq Season where
  Spring == Spring = True
  Summer == Summer = True
  Autumn == Autumn = True
  Winter == Winter = True
  _      == _      = False
```

Product types

Product types are algebraic types with a number of typed components

```
data People = Person String Int
```

Product types

Product types are algebraic types with a number of typed components

```
type Name = String
type Age  = Int
data People = Person Name Age

λ >  :type Person "Casimir" 20
Person "Casimir" 20 :: People

λ >  :type Person
Person :: Name -> Age -> People

λ >  :type Person "Casimir"
Person "Casimir" :: Age -> People
```

Product types

The type constructor does not have to be different from the type.

```
type Name = String
type Age  = Int
data People = People Name Age

λ >  :type People "Casimir" 20
People "Casimir" 20 :: People

λ >  :type People
People :: Name -> Age -> People

λ >  :type People "Casimir"
People "Casimir" :: Age -> People
```

Product types

To define functions over product types we use pattern matching.

```
type Name = String
type Age  = Int
data People = People Name Age

showPeople :: People -> String
showPeople (People n a) = n ++ " is " ++ show a ++
                          " year(s) old."
```

```
λ> let casimir = People "Casimir" 20 in showPeople casimir
"Casimir is 20 year(s) old."
```

```
λ> showPeople (People "Casimir" 20)
"Casimir is 20 year(s) old."
```

Product types

To define functions over product types we use pattern matching.

```
type Name = String
type Age  = Int
data People = People Name Age

data Stage = Baby | Child | PTeen | Teen | Adult | Elder

stage (People _ a)
  | a <  2    = Baby
  | a < 12    = PTeen
  | a < 19    = Teen
  | a < 60    = Adult
  | otherwise = Elder
```

Enumerated vs. Product types

Compare

```
type People = (Name, Age)
```

with

```
data People = People Name Age
```

Type synonym vs. Product types

(Some) Advantages of using an algebraic type.

- Explicit labels.
- Reduce errors.
- Better error messages.
- Class instances.

Type synonym vs. Product types

Advantages of using type synonyms.

- More compact definition.
- Polymorphic function reuses (eg. `fst`, `snd`, `zip`, ...).

Type synonym vs. Product types

Type synonym:

```
type Name = String
type Age  = Int
type People = (Name, Age)

name :: People -> Name
name = fst

age :: People -> Age
age = snd
```

Type synonym vs. Product types

Algebraic type:

```haskell
type Name = String
type Age  = Int
data People = People Name Age

name :: People -> Name
name (People n _) = n

age :: People -> Age
age (People _ a) = a
```

Alternatives

```
type FPoint = (Float, Float)
data Shape = Circle FPoint Float
           | Rectangle FPoint FPoint

λ >  :type Circle
Circle :: FPoint -> Float -> Shape

λ >  :type Rectangle
Rectangle :: FPoint -> FPoint -> Shape

λ >  :type Circle (0,0) 1
Circle (0, 0) 1 :: Shape

λ >  :type Rectangle (0,0) (3,5)
Rectangle (0,0) (3,5) :: Shape
```

Alternatives

```
type FPoint = (Float, Float)
data Shape = Circle FPoint Float
           | Rectangle FPoint FPoint

showShape :: Shape -> String
showShape (Circle c r)
          = "circle."          ++
          " center pt="        ++ show  c ++
          " radius="           ++ show r
showShape (Rectangle llp urp)
          = "rectangle."       ++
          " lower left pt="    ++ show llp ++
          " upper right pt="   ++ show urp
```

Alternatives

```
type FPoint = (Float, Float)
data Shape = Circle FPoint Float
           | Rectangle FPoint FPoint

λ> showShape (Circle (0, 0) 1)
"circle. center pt=(0.0,0.0) radius=1.0"

λ>: showShape (Rectangle (0,0) (3,5))
"rectangle. lower left pt=(0.0,0.0) upper right pt=(3.0,5.0)"

λ>: :type [Circle (0,0) 1, Rectangle (0,0) (3,5)]
[Circle (0,0) 1, Rectangle (0,0) (3,5)] :: [Shape]
```

Alternatives

```
type FPoint = (Float, Float)
data Shape = Circle FPoint Float
           | Rectangle FPoint FPoint

mkUnitCircle :: FPoint -> Shape
mkUnitCircle c = Circle c 1

mkUnitCircles1 :: [FPoint] -> [Shape]
mkUnitCircles1 cs = [mkUnitCircle c | c <- cs]

mkUnitCircles2 :: [FPoint] -> [Shape]
mkUnitCircles2 = map mkUnitCircle

mkUnitCircles3 :: [FPoint] -> [Shape]
mkUnitCircles3 = foldr (\c acc -> mkUnitCircle c:acc) []
```

Alternatives

```
type FPoint = (Float, Float)
data Shape = Circle FPoint Float
           | Rectangle FPoint FPoint

mkManyUnitCircles (xMin,xMax) (yMin,yMax) =
     [mkUnitCircle (x,y) | x <- [xMin..xMax]
                         , y <- [yMin..yMax]]

λ > map showShape $ mkManyUnitCircles (0,1) (2,4)
["circle. center pt=(0.0,2.0) radius=1.0",
 "circle. center pt=(0.0,3.0) radius=1.0",
 "circle. center pt=(0.0,4.0) radius=1.0",
 "circle. center pt=(1.0,2.0) radius=1.0",
 "circle. center pt=(1.0,3.0) radius=1.0",
 "circle. center pt=(1.0,4.0) radius=1.0"]
```

Alternatives

```
type FPoint = (Float, Float)
data Shape = Circle FPoint Float
           | Rectangle FPoint FPoint

area :: Shape -> Float
area (Circle _ r) = pi * r^2
area (Rectangle (xll,yll) (xur,yur)) = w*h
  where
    w = xur-xll
    h = yur-yll

λ > map area [Circle (0,0) 1, Rectangle (0,0) (2,3)]
[3.1415927,6.0]
λ > map area $ mkManyUnitCircles (0,1) (0,1)
[3.1415927,3.1415927,3.1415927,3.1415927]
```

class and instance declarations

A new class can de declared using the `class` mechanism.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  x == y = not (x /= y)
  x /= y = not (x == y)
```

- For a type `a` to be an instance of the class `Eq`, it must support equality and inequality operators for the sepcified type.
- Because default definitions have already been included, declaring an instance only requires a definition for the == operator or for the /= operator.

class and instance declarations

The type `Bool` can be made into an equality type as follows: mechanism.

```
instance Eq Bool where
  False == False = True
  True  == True  = True
  _     == _     = False
```

class and instance declarations

- Only types that are declared using the `data` and `newtype` (more on this soon) mechanisms can be made instances of classes.
- Default definitions can be overridden in instance declaration if desired.

class and instance declarations

Class can also be extended to form new class.

For example the class `Ord` is declared in the standard prelude as an extension of the class `Eq`:

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max             :: a -> a -> a


  min x y = if x <= y then x else y
  max x y = if x <= y then y else x
```

For a type to be an instance of `Ord`, it must be an instance of `Eq`, and support six additional operators.

class and instance declarations

Class can also be extended to form new class.

For example the class `Ord` is declared in the standard prelude as an extension of the class `Eq`:

Because default definitions have already been included for `min` and `max`, declaring an equality type (such as `Bool`) as an ordered type only requires defining the four comparison operators:

```
instance Ord Bool where
  False < True = True
  _     < _    = False
  x <= y = x < y || x == y
  x > y  = y < x
  x >= y = y <= x
```

Movable objects

Define a class of types whose members are geometrical objects in two dimensions.

```
data Vect = Vect Float Float

class Movable a where
  move :: Vec -> a -> a
  reflectX :: a -> a
  reflectY :: a -> a
  rotate180 :: a -> a
  rotate180 = reflectX . reflectY
```

Derived instances

- When new types are declared, it is usually approriate to make them into instances of a number of built-in classes.
- Haskell provide a simple facility for automatically making new types into instances of the classes `Eq`, `Ord`, `Show` and `Read` in the form of the `deriving` mechanism.

Derived instances

```
data T = T1 Int | T2 Int | T3 Int
```

Derived instances

```
data T = T1 Int | T2 Int | T3 Int

λ> [T1 1,T2 2,T3 3]
<interactive>: error:
No instance for (Show T) arising from a use of 'print'
```

Derived instances

```
data T = T1 Int | T2 Int | T3 Int

instance Show T where
  show (T1 x) = "T1 " ++ show x
  show (T2 x) = "T2 " ++ show x
  show (T3 x) = "T3 " ++ show x
```

```
λ> T1 1
T1 1
λ> T2 2
T2 2
λ> T3 3
T3 3
```

Derived instances

```
data T = T1 Int | T2 Int | T3 Int

λ> T1 1 == T1 1
<interactive>: error:
No instance for (Ord T) arising from a use of '=='
λ> T1 1 /= T1 1
<interactive>: error:
No instance for (Ord T) arising from a use of '/='
λ> T1 1 == T2 2
<interactive>: error:
No instance for (Ord T) arising from a use of '=='
```

Derived instances

```
data T = T1 Int | T2 Int | T3 Int

instance Eq T where
  T1 x == T1 y = x == y
  T2 x == T2 y = x == y
  T3 x == T3 y = x == y
  _    == _    = False
```

```
λ> T1 1 == T1 1
True
λ> T1 1 == T1 2
False
λ> T1 1 == T2 1
False
λ> T3 1 == T2 1
False
```

```
λ> T1 1 /= T1 1
False
λ> T1 1 /= T1 2
True
λ> T1 1 /= T2 1
True
λ> T3 1 /= T2 1
True
```

Derived instances

```
data T = T1 Int | T2 Int | T3 Int

λ> T1 1 < T1 2
<interactive>: error:
No instance for (Ord T) arising from a use of '<'
λ> T1 1 < T2 2
<interactive>: error:
No instance for (Ord T) arising from a use of '<'
λ> T1 1 < T3 3
<interactive>: error:
No instance for (Ord T) arising from a use of '<'
λ> T1 1 <= T1 2
<interactive>: error:
No instance for (Ord T) arising from a use of '<='
```

Derived instances

```
data T = T1 Int | T2 Int | T3 Int

instance Ord T where
  T1 x < T1 y = x < y
  T1 _ < T2 _ = True
  T1 _ < T3 _ = True
  T2 x < T2 y = x < y
  T2 _ < T3 _ = True
  T3 x < T3 y = x < y
  _    < _    = False

  t    <= t'  = t  <  t' || t == t'
  t    >  t'  = t' <  t
  t    >= t'  = t' <= t
```

Derived instances

```
data T = T1 Int | T2 Int | T3 Int
```

```
λ> T1 1 < T1 1
False
λ> T1 1 <= T1 1
True
λ> T2 3 > T2 2
True
λ> T2 3 >= T2 2
True
λ> T3 2 <= T3 3
True
```

```
λ> T1 1 < T2 2
True
λ> T1 3 < T2 2
True
λ> T1 3 < T3 1
True
λ> T2 2 < T3 1
True
λ> T1 3 > T3 1
False
```

Derived instances

```
data P = data T = T1 Int | T2 Int | T3 Int
         deriving (Show, Eq, Ord)
```

Derived instances

```
data P = data T = T1 Int | T2 Int | T3 Int
         deriving (Show, Eq, Ord)
```

Class `Show`:

```
λ> T1 1
T1 1
λ> T2 2
T2 2
λ> T3 3
T3 3
```

Derived instances

```
data P = data T = T1 Int | T2 Int | T3 Int
         deriving (Show, Eq, Ord)
```

Class Eq:

```
λ> T1 1 == T1 1
True
λ> T1 1 == T1 2
False
λ> T1 1 == T2 1
False
λ> T3 1 == T2 1
False
```

```
λ> T1 1 /= T1 1
False
λ> T1 1 /= T1 2
True
λ> T1 1 /= T2 1
True
λ> T3 1 /= T2 1
True
```

Derived instances

```
data P = data T = T1 Int | T2 Int | T3 Int
         deriving (Show, Eq, Ord)
```

Class `Ord`:

```
λ> T1 1 < T1 1
False
λ> T1 1 <= T1 1
True
λ> T2 3 > T2 2
True
λ> T2 3 >= T2 2
True
λ> T3 2 <= T3 3
True
```

```
λ> T1 1 < T2 2
True
λ> T1 3 < T2 2
True
λ> T1 3 < T3 1
True
λ> T2 2 < T3 1
True
λ> T1 3 > T3 1
False
```

Derived instances

- `deriving` helps to reduce unnecessary duplication and makes code more consistent and clear.
- Keep in mind that foir the purposes of deriving instances of the class `Ord` of ordered types, the orderging of the constructors of the type is determined by their position in its declaration.

  In other words,

  ```
  data T = T1 | T2 deriving (Eq,Ord)
  ```

  and

  ```
  data T = T2 | T1 deriving (Eq,Ord)
  ```

  do behave the same with espect to the class `Ord`.
- You cannot derive a `Show` (resp. `Eq` and `Ord`) instance if your data type contains functions.

Maybe

- Data declaration can be parameterized (1/2).

```
data Maybe a = Nothing | Just a
```

- A value of type `Maybe` is either `Nothing`, or of the form `Just x` for some value `x` of type `a`.
- `Maybe` is defined in module `Data.Maybe`.

Maybe

```haskell
data Maybe a = Nothing | Just a

safeHead :: [a] -> Maybe a
safeHead []     = Nothing
safeHead (x:_) = Just x

safeTail :: [a] -> Maybe [a]
safeTail []     = Nothing
safeTail (_:xs) = Just xs
```

```
λ> safeHead []
Nothing
λ> safeHead [1,2,3,4]
Just 1
```

```
λ> safeTail []
Nothing
λ> safeTail [1,2,3,4]
Just [2,3,4]
```

Maybe

```haskell
data Maybe a = Nothing | Just a

safeDiv :: Integral a => a -> a -> Maybe a
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

safeSqrt :: (Ord a, Floating a) => a -> Maybe a
safeSqrt x
  | x < 0     = Nothing
  | otherwise = Just (sqrt x)
```

```
λ> 10 `safeDiv` 0
Nothing
λ> 10 `safeDiv` 3
Just 3
```

```
λ> safeSqrt (-2)
Nothing
λ> safeSqrt 2
Just 1.4142135623730951
```

Either

- Data declaration can be parameterized (2/2).

```
data Either a b = Left a | Right b
```

- `Either` is just a wrapper around a value that can be either one or the other.
- `Either` is defined in module `Data.Either`.

Maybe

Let's say that we want to parse a `Char` as a decimal digit to an `Int`. This operation could fail if the character is not a digit.

We can represent this error as a data type:

```
data ParseDigitError = NotADigit Char deriving Show
```

And our parsing function can have the type:

```
parseDigit :: Char -> Either ParseDigitError Int
```

Maybe

Let's say that we want to parse a `Char` as a decimal digit to an `Int`. This operation could fail if the character is not a digit.

```
parseDigit :: Char -> Either ParseDigitError Int
parseDigit '0' = Right 0
parseDigit '1' = Right 1
parseDigit '2' = Right 2
parseDigit '3' = Right 3
parseDigit '4' = Right 4
parseDigit '5' = Right 5
parseDigit '6' = Right 5
parseDigit '7' = Right 5
parseDigit '8' = Right 8
parseDigit '9' = Right 9
parseDigit c   = Left (NotADigit c)
```

Maybe

Let's say that we want to parse a `Char` as a decimal digit to an `Int`. This operation could fail if the character is not a digit.

```
parseDigit :: Char -> Either ParseDigitError Int
parseDigit c = case c of
    '0' -> Right 0
    '1' -> Right 1
    '2' -> Right 2
    '3' -> Right 3
    '4' -> Right 4
    '5' -> Right 5
    '6' -> Right 6
    '7' -> Right 7
    '8' -> Right 8
    '9' -> Right 9
    _ -> Left (NotADigit c)
```

Maybe

Let's say that we want to parse a `Char` as a decimal digit to an `Int`. This operation could fail if the character is not a digit.

```
λ> parseDigit '4'
Right 4
λ> parseDigit 'a'
Left (NotADigit 'a')
λ> [parseDigit c | c <- "202303"]
[Right 2,Right 0,Right 2,Right 3,Right 0,Right 3]
```

Recursive types

New types using the `data` and `newtype` mechanisms can also be recursive.

```
data Nat = Zero | Succ Nat
```

A value of type `Nat` is either `Zero`, or of the form `Succ n` for some value `n` of type `Nat`.

This declaration gives rise to an infinite sequence of values, starting with the value `Zero`, and continuing by applying the constructor `Succ` to the previous value in the sequence.

```
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
...
```

Recursive types – Naturals

```
data Nat = Zero | Succ Nat deriving Show

natToInt :: Nat -> Int
natToInt Zero     = 0
natToInt (Succ n) = 1 + natToInt n

λ> natToInt Zero
0
λ> natToInt (Succ (Succ (Succ (Succ (Succ Zero)))))
5
```

Recursive types – Naturals

```
data Nat = Zero | Succ Nat deriving Show

intToNat :: Int -> Nat
intToNat 0 = Zero
intToNat n = Succ (intToNat (n-1))

λ > intToNat 0
Zero
λ > intToNat 5
Succ (Succ (Succ (Succ (Succ Zero))))
```

Recursive types – Naturals

```
data Nat = Zero | Succ Nat deriving Show

addNat1 :: Nat -> Nat -> Nat
addNat1 n1 n2 = intToNat (i1+i2)
  where
    i1 = natToInt n1
    i2 = natToInt n2

λ > addNat1 Zero Zero
Zero
λ > addNat1 (Succ Zero) (Succ (Succ (Succ Zero)))
Succ (Succ (Succ (Succ Zero)))
```

Recursive types – Naturals

```
data Nat = Zero | Succ Nat deriving Show

addNat2 :: Nat -> Nat -> Nat
addNat2 Zero     n2 = n2
addNat2 (Succ n1) n2 = Succ (addNat2 n1 n2)

λ> addNat2 Zero Zero
Zero
λ> addNat2 (Succ Zero) (Succ (Succ (Succ Zero)))
Succ (Succ (Succ (Succ Zero)))
```

Recursive types – Naturals

```
data Nat = Zero | Succ Nat deriving Show

nats :: [Nat]
nats = iterate Succ Zero

λ> take 3 nats
[Zero,Succ Zero,Succ (Succ Zero)]
λ> take 3 (map Succ nats)
[Succ Zero,Succ (Succ Zero),Succ (Succ (Succ Zero))]
λ> head . dropWhile (\n -> natToInt n < 3) $ nats
Succ (Succ (Succ Zero))
λ> head . dropWhile ((< 3)  . natToInt) $ nats
Succ (Succ (Succ Zero))
```

Recursive types – Lists

```
data List a = Nil | Cons a (List a) deriving Show
```

Recursive types – Lists

```haskell
data List a = Nil | Cons a (List a) deriving Show

headL :: List a -> Maybe a
headL Nil         = Nothing
headL (Cons x _) = Just x

tailL :: List a -> Maybe (List a)
tailL Nil         = Nothing
tailL (Cons _ l) = Just l

rangeL :: (Ord a, Enum a) => a -> a -> List a
rangeL from to
  | from > to = Nil
  | otherwise = Cons from (rangeL (succ from) to)
```

Recursive types – Lists

```
data List a = Nil | Cons a (List a) deriving Show

λ > rangeL 1 4
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))

λ > headL Nil
Nothing

λ > headL (rangeL 1 4)
Just 1

λ > tailL Nil
Nothing

λ > tailL (rangeL 1 4)
Just (Cons 2 (Cons 3 (Cons 4 Nil)))
```

Recursive types – Lists

```haskell
data List a = Nil | Cons a (List a) deriving Show

mapL :: (a -> b) -> List a -> List b
mapL _ Nil         = Nil
mapL f (Cons x l) = Cons (f x) (mapL f l)

filterL :: (a -> Bool) -> List a -> List a
filterL _ Nil = Nil
filterL p (Cons x l)
  | p x       = Cons x (filterL p l)
  | otherwise = filterL p l

elemL :: Eq a => a -> List a -> Bool
_ `elemL` Nil       = False
y `elemL`(Cons x l) = x == y || y `elemL` l
```

Recursive types – Lists

```
data List a = Nil | Cons a (List a) deriving Show
```

```
λ> mapL (*3) $ rangeL 1 6
Cons 3 (Cons 6 (Cons 9 (Cons 12 (Cons 15 (Cons 18 Nil)))))
```

```
λ> filterL ((==) 0 . (`mod` 9)) . mapL (*3) $ rangeL 1 6
Cons 9 (Cons 18 Nil)
```

```
λ> elemL 99 . filterL (even) $ rangeL 1 100
False
```

```
λ> elemL 98 . filterL (even) $ rangeL 1 100
True
```

```
λ> mapL (\x -> rangeL 1 x) $ rangeL 1 2
Cons (Cons 1 Nil) (Cons (Cons 1 (Cons 2 Nil)) Nil)
```

Recursive types – Lists

```haskell
data List a = Nil | Cons a (List a) deriving Show

appendL :: List a -> List a -> List a
appendL Nil l2          = l2
appendL (Cons x l1) l2 = Cons x (appendL l1 l2)

reverseL1 :: List a -> List a
reverseL1 Nil          = Nil
reverseL1 (Cons x l) = appendL (reverseL1 l) (Cons x Nil)

reverseL2 :: List a -> List a
reverseL2 = go Nil
  where
    go acc Nil          = acc
    go acc (Cons x l) = go (Cons x acc) l
```

Recursive types – Lists

```
data List a = Nil | Cons a (List a) deriving Show

λ > appendL (rangeL 1 2) (rangeL 1 3)
Cons 1 (Cons 2 (Cons 1 (Cons 2 (Cons 3 Nil))))

λ > reverseL1 (rangeL 1 4)
Cons 4 (Cons 3 (Cons 2 (Cons 1 Nil)))

λ > reverseL1 . reverseL1 $ rangeL 1 4
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))

λ > lastL = headL . reverseL1

λ > :type lastL
lastL :: List a -> Maybe a

λ > lastL (rangeL 1 10000)
Just 10000
```

Recursive types – Lists

```haskell
data List a = Nil | Cons a (List a) deriving Show

zipWithL :: (a -> b -> c) -> List a -> List b -> List c
zipWithL _ Nil _ = Nil
zipWithL _ _    Nil = Nil
zipWithL f (Cons x1 l1) (Cons x2 l2) = Cons x' l'
  where
    x' = f x1 x2
    l' = zipWithL f l1 l2

zipL :: List a -> List b -> List (a, b)
zipL = zipWithL (,)
```

Recursive types – Lists

```
data List a = Nil | Cons a (List a) deriving Show

λ > zipL (rangeL 1 4) (rangeL 100 200)
Cons (1,100) (Cons (2,101) (Cons (3,102) (Cons (4,103) Nil)))

λ > zipWithL (+) (rangeL 1 4) (rangeL 100 200)
Cons 101 (Cons 103 (Cons 105 (Cons 107 Nil)))

λ > fromListL = foldr (\x acc -> Cons x acc) Nil

λ > :type fromListL
fromListL :: Foldable t => t a -> List a

λ > indexL = zipL (fromListL [1..])
λ > :type indexL
indexL :: (Num a, Enum a) => List b -> List (a, b)

λ > indexL (fromListL "abc")
Cons (1,'a') (Cons (2,'b') (Cons (3,'c') Nil))
```

Recursive types – Lists

```haskell
data List a = Nil | Cons a (List a) deriving Show

foldlL :: (b -> a -> b) -> b -> List a -> b
foldlL _ acc Nil         = acc
foldlL f acc (Cons x l) = foldlL f (f acc x) l

foldrL :: (a -> b -> b) -> b -> List a -> b
foldrL _ acc Nil         = acc
foldrL f acc (Cons x l) = f x (foldrL f acc l)
```

Recursive types – Lists

```
data List a = Nil | Cons a (List a) deriving Show
```

```
λ > foldlL (+) 0 (rangeL 1 100)
5050
```

```
λ > foldrL (+) 0 (rangeL 1 100)
5050
```

```
λ > foldlL (\acc _ -> 1+acc) 0 (rangeL 1 100)
100
```

```
λ > foldlL (\acc x -> Cons x acc) Nil (rangeL 1 3)
Cons 3 (Cons 2 (Cons 1 Nil))
```

```
λ > foldrL (\x acc -> Cons (even x) acc) Nil (rangeL 1 4)
Cons False (Cons True (Cons False (Cons True Nil)))
```

Recursive types – Arithmetic expressions

Define a model of the simple numeric expressions using arithmetic operators + and −.

```
data Expr a = Lit a
            | Add (Expr a) (Expr a)
            | Sub (Expr a) (Expr a)
            deriving (Show)
```

Recursive types – Arithmetic expressions

Define a model of the simple numeric expressions using arithmetic operators $+$ and $-$.

```
data Expr a = Lit a
            | Add (Expr a) (Expr a)
            | Sub (Expr a) (Expr a)
            deriving (Show)
```

Some examples are:

2	`Lit 2`
$2+3$	`Add (Lit 2) (Lit 3)`
$2-3$	`Sub (Lit 2) (Lit 3)`
$(3-1)+2$	`Add (Sub (Lit 3) (Lit 1)) (Lit 2)`

Recursive types – Arithmetic expressions

Define a model of the simple numeric expressions using arithmetic operators + and −.

```
data Expr a = Lit a
            | Add (Expr a) (Expr a)
            | Sub (Expr a) (Expr a)
            deriving (Show)
```

```
λ >  :type Lit
Lit :: a -> Expr a
λ >  :type Add
Add :: Expr a -> Expr a -> Expr a
λ >  :type Sub
Sub :: Expr a -> Expr a -> Expr a
```

Recursive types – Arithmetic expressions

Given an expression, we might want to :

- evaluate it;
- turn it into a string, which can then be printed;
- count the operators;
- ...

Recursive types – Arithmetic expressions

```
evalExpr :: (Num a) => Expr a -> a
evalExpr (Lit x)      = x
evalExpr (Add e1 e2) = evalExpr e1 + evalExpr e2
evalExpr (Sub e1 e2) = evalExpr e1 - evalExpr e2
```

Recursive types – Arithmetic expressions

```
evalExpr :: (Num a) => Expr a -> a
evalExpr (Lit x)      = x
evalExpr (Add e1 e2) = evalExpr e1 + evalExpr e2
evalExpr (Sub e1 e2) = evalExpr e1 - evalExpr e2

λ > e1 = Add (Lit 2) (Lit 3)
λ > e2 = Add (Sub (Lit 3) (Lit 1)) (Lit 3)

λ > eval e1
5

λ > eval e2
5

λ > eval (Add e1 e2)
10

λ > eval (Sub e1 e2)
0
```

Recursive types – Arithmetic expressions

```
showExpr :: (Show a) => Expr a -> String
showExpr (Lit x)      = show x
showExpr (Add e1 e2) = "(" ++ showExpr e1 ++ "+" ++
                              showExpr e2        ++ ")"
showExpr (Sub e1 e2) = "(" ++ showExpr e1 ++ "-" ++
                              showExpr e2        ++ ")"
```

Recursive types – Arithmetic expressions

```
showExpr :: (Show a) => Expr a -> String
showExpr (Lit x)      = show x
showExpr (Add e1 e2) = "(" ++ showExpr e1 ++ "+" ++
                              showExpr e2        ++ ")"
showExpr (Sub e1 e2) = "(" ++ showExpr e1 ++ "-" ++
                              showExpr e2        ++ ")"
```

```
λ > e1 = Add (Lit 2) (Lit 3)
λ > e2 = Add (Sub (Lit 3) (Lit 1)) (Lit 3)

λ > showExpr e1
"(2+3)"

λ > showExpr e2
"((3-1)+3)"

λ > showExpr $ Add (Add e1 e2) (Sub e2 e1)
"(((2+3)+((3-1)+3))+(((3-1)+3)-(2+3)))"
```

Recursive types – Arithmetic expressions

```
countExpr :: Num b => Expr a -> b
countExpr (Lit _)      = 0
countExpr (Add e1 e2) = 1 + countExpr e1 + countExpr e2
countExpr (Sub e1 e2) = 1 + countExpr e1 + countExpr e2
```

Recursive types – Arithmetic expressions

```
countExpr :: Num b => Expr a -> b
countExpr (Lit _)      = 0
countExpr (Add e1 e2) = 1 + countExpr e1 + countExpr e2
countExpr (Sub e1 e2) = 1 + countExpr e1 + countExpr e2

λ >  e1 = Add (Lit 2) (Lit 3)
λ >  e2 = Add (Sub (Lit 3) (Lit 1)) (Lit 3)

λ >  countExpr e1
1

λ >  countExpr e2
2

λ >  showExpr $ Add (Add e1 e2) (Sub e2 e1)
"(((2+3)+((3-1)+3))+(((3-1)+3)-(2+3)))"
λ >  countExpr $ Add (Add e1 e2) (Sub e2 e1)
9
```

Recursive types – Arithmetic expressions – Infix constructors

```
data Expr a = Lit a
            | (Expr a) :+: (Expr a)
            | (Expr a) :-: (Expr a)
            deriving (Show)
```

Recursive types – Arithmetic expressions – Infix constructors

```
data Expr a = Lit a
            | (Expr a) :+: (Expr a)
            | (Expr a) :-: (Expr a)
            deriving (Show)
```

```
λ> Lit 1
Lit 1
λ> Lit 1 :+: Lit 2
Lit 1 :+: Lit 2
λ> (Lit 1 :+: Lit 2) :-: (Lit 5 :+: Lit 6)
λ> Lit 1 :+: Lit 2 :+: Lit 3 :+: Lit 4
((Lit 1 :+: Lit 2) :+: Lit 3) :+: Lit 4
```

Recursive types – Arithmetic expressions – Infix constructors

```
data Expr a = Lit a
            | (Expr a) :+: (Expr a)
            | (Expr a) :-: (Expr a)
            deriving (Show)

evalExpr :: Num a => Expr a -> a
evalExpr (Lit x) = x
evalExpr (e1 :+: e2) = evalExpr e1 + evalExpr e2
evalExpr (e1 :-: e2) = evalExpr e1 - evalExpr e2

showExpr :: (Show a) => Expr a -> String
showExpr (Lit x)     = show x
showExpr (e1 :+: e2) = "(" ++ showExpr e1 ++ "+" ++
                              showExpr e2 ++ ")"
showExpr (e1 :-: e2) = "(" ++ showExpr e1 ++ "-" ++
                              showExpr e2 ++ ")"
```

Recursive types – Arithmetic expressions – Fact. constructors

```
data AOp = Add | Sub | Mul | Div deriving (Show)
data Expr a = Lit a
            | Op AOp (Expr a) (Expr a)
            deriving (Show)
```

Recursive types – Arithmetic expressions – Fact. constructors

```haskell
data AOp = Add | Sub | Mul | Div deriving (Show)
data Expr a = Lit a
            | Op AOp (Expr a) (Expr a)
            deriving (Show)

evalExpr1 :: (Fractional a) => Expr a -> a
evalExpr1 (Lit x)          = x
evalExpr1 (Op Add e1 e2) = evalExpr1 e1 + evalExpr1 e2
evalExpr1 (Op Sub e1 e2) = evalExpr1 e1 - evalExpr1 e2
evalExpr1 (Op Mul e1 e2) = evalExpr1 e1 * evalExpr1 e2
evalExpr1 (Op Div e1 e2) = evalExpr1 e1 / evalExpr1 e2
```

Recursive types – Arithmetic expressions – Fact. constructors

```haskell
data AOp = Add | Sub | Mul | Div deriving (Show)
data Expr a = Lit a
            | Op AOp (Expr a) (Expr a)
            deriving (Show)

evalExpr2 :: (Fractional a) => Expr a -> a
evalExpr2 (Lit x)      = x
evalExpr2 (Op o e1 e2) = f o (evalExpr2 e1) (evalExpr2 e2)
  where
    f Add = (+)
    f Sub = (-)
    f Mul = (*)
    f Div = (/)
```

Record types

Type synonyms

Algebraic types

Record types

`newtype` declaration

Case study: Tautology checker

Record

- Record syntax (aka. named fields) is an alternative way to write data types.
- Defining a new data type by using record syntax makes it much easier to understand which types represent which properties of the data type.
- You don't have to write your getters; each field in the record syntax automatically creates a function to access that value from the record.

Record – Syntax

Consider a datatype whose purpose is to hold configuration settings:

```
data Conf = Conf
    String    -- User name
    String    -- Local host
    String    -- Remote host
    Bool      -- Is guest?
    Bool      -- Is superuser?
    String    -- Current directory
    String    -- Home directory
    Integer   -- Time connected
    deriving (Eq, Show)
```

Record – Syntax

```
getUserName :: Conf -> String
getUserName   (Conf un _  _  _  _ _ _ _) = un

getLocalHost :: Conf -> String
getLocalHost  (Conf _  lh _  _  _ _ _ _) = lh

getRemoteHost :: Conf -> String
getRemoteHost (Conf _  _  rh _  _ _ _ _) = rh

getIsGuest :: Conf-> Bool
getIsGuest    (Conf _  _  _  ig _ _ _ _) = ig

...
```

Record – Syntax

```
data Conf = Conf
    { username      :: String
    , localHost     :: String
    , remoteHost    :: String
    , isGuest       :: Bool
    , isSuperuser   :: Bool
    , currentDir    :: String
    , homeDir       :: String
    , timeConnected :: Integer
    }
    deriving (Eq, Show)
```

Record – Syntax

The record declaration automatically generates the following accessor functions :

```
username :: Conf -> String

localHost :: Conf -> String

remoteHost :: Conf -> String

isGuest :: Conf -> Bool

isSuperUser :: Conf -> Bool

currentDir :: Conf -> String

homeDir :: Conf -> String

timeConnected :: Conf -> Integer
```

Record – Pattern matching

You can pattern match against named fields.

```
-- gathering host data
getHostData :: Conf -> (String, String)
getHostData Conf { localHost = lh, remoteHost = rh }
            = (lh, rh)

-- gathering dir data
getDirData :: Conf -> (String, String)
getDirData Conf { currentDir = cd, homeDir = hd }
           = (cd, hd)

-- home dir predicate
isHomeDir :: Conf -> Bool
isHomeDir Conf { currentDir = cd, homeDir = hd }
          = cd == hd
```

Record – Creating

Preferred way:

```
initCFG = Conf
    { username      = "nobody"
    , localHost     = "nowhere"
    , remoteHost    = "nowhere"
    , isguest       = False
    , issuperuser   = False
    , currentdir    = "/"
    , homedir       = "/"
    , timeConnected = 0
    }
```

Record – Creating

Shorter – but prone to errors – way:

```
initCFG :: Conf
initCFG = Conf  "nobody" "nowhere" "nowhere"
                False False "/" "/" 0
```

Highly discouraged!

Record – Creating

Incomplete – and prone to errors – way:

```
cfgFoo :: Conf
cfgFoo = Conf { username = "Foo" }
```

Highly discouraged!

```
λ> username cfgFoo
"Foo"
λ> localHost cfgFoo
*** Exception: Configuration.hs:
      Missing field in record construction localHost
```

Record – Updating

- To update the field `x` in a value `r` to `y`, you write `r { x = y }`.
- You can change more than one ; each should be separated by commas, for instance, `r {x = y, a = b, c = d }`.

Record – Updating

- To update the field `x` in a value `r` to `y`, you write `r { x = y }`.
- You can change more than one ; each should be separated by commas, for instance, `r {x = y, a = b, c = d }`.

```
changeDir :: Conf -> String -> Maybe Conf
changeDir cfg newDir =
    if directoryExists newDir
    then Just (cfg { currentDir = newDir })
    else Nothing
```

Record – Example

```haskell
data Person = Person { name :: String, age:: Int }
              deriving (Show, Eq)

alice = Person { name = "Alice", age = 25 }
bob   = Person { name = "Bob",   age = 30 }
```

Record – Example

```
data Person = Person { name :: String, age:: Int }
              deriving (Show, Eq)

alice = Person { name = "Alice", age = 25 }
bob   = Person { name = "Bob",   age = 30 }

λ > anotherLice = Person { name = name alice, age = 35 }
λ > anotherAlice
Person {name = "Alice", age = 35}
```

Record – Example

```
data Person = Person { name :: String, age:: Int }
              deriving (Show, Eq)

alice = Person { name = "Alice", age = 25 }
bob   = Person { name = "Bob",   age = 30 }

λ > anotherAlice = alice { age = 35 }
λ > anotherAlice
Person {name = "Alice", age = 35}
λ > alice
Person {name = "Alice", age = 25}
```

Record – Example

```haskell
data Person = Person { name :: String, age:: Int }
              deriving (Show, Eq)

alice = Person { name = "Alice", age = 25 }
bob   = Person { name = "Bob",   age = 30 }

timeFlies :: Person -> Person
timeFlies p = p { age = 1 + age p}
```

```
λ> timeFlies alice
Person {name = "Alice", age = 26}
λ> L.head . L.drop 10 $ L.iterate timeFlies bob
Person {name = "Bob", age = 40}
```

Record – Example

```
data Person = Person { name :: String, age:: Int }
              deriving (Show, Eq)

alice = Person { name = "Alice", age = 25 }
bob   = Person { name = "Bob",   age = 30 }

capitalName :: Person -> Person
capitalName p@Person { name = x:xs }
                  = p { name = L.map Data.Char.toUpper xs }
```

```
λ >  capitalName alice
Person {name = "ALICE", age = 25}
λ >  capitalName bob
Person {name = "BOB", age = 30}
```

Record – Example

```
data Person = Person { name :: String, age:: Int }
              deriving (Show, Eq)

alice = Person { name = "Alice", age = 25 }
bob   = Person { name = "Bob",   age = 30 }

shortName :: Person -> Person
shortName  p@Person { name = x:_ } = p { name = x:"." }
```

```
λ >  shortName alice
Person {name = "A.", age = 25}
λ >  shortName bob
Person {name = "B.", age = 30}
```

`newtype` declaration

Type synonyms

Algebraic types

Record types

`newtype` declaration

Case study: Tautology checker

`newtype`

- A `newtype` declaration creates a new type in much the same way as `data`.
- The syntax and usage of `newtype` is virtually identical to that of data declarations – in fact, you can replace the `newtype` keyword with `data` and it'll still compile, indeed there's even a good chance your program will still work.
- The converse is not true, however – `data` can only be replaced with `newtype` if the type has exactly one constructor with exactly one field inside it.

newtype

- Basic use case:

```
newtype Temp = Temp Float
```

- The following would also be valid:

```
data Temp = Temp Float
```

newtype

- Newtypes can have deriving clauses just like normal types:

```
newtype Temp = Temp Float
               deriving (Eq, Ord, Read, Show)
```

- Record syntax is still allowed, but only for one field:

```
newtype Temp = Temp { getTemp :: Float }
```

includind one-field records with deriving clauses

```
newtype Temp = Temp { getTemp :: Float }
               deriving (Eq, Ord, Read, Show)
```

newtype

- 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 Pair a b = Pair (a, b)
```

`newtype`

- The restriction to one constructor with one field means that the new type and the type of the field are in direct correspondence (they are isomorphic).
- After the type is checked at compile time, at run time the two types can be treated essentially the same, without the overhead or indirection normally associated with a data constructor (*i.e.*, no work done when pattern maching).

`newtype` in short

- `newtype` is faster than `data`.
- When using `newtype`, you're restricted to just one constructor with one field. That's exactly what we need to make a new type from an existing type.
- Use `newtype` to make type class instances.

Case study: Tautology checker

Type synonyms

Algebraic types

Record types

`newtype` declaration

Case study: Tautology checker

Language of propositions

Consider a language of propositions built up from basic values (*False*, *True*) and variables ($A, B, \ldots, Z$) using negation ($\neg$), conjunction ($\wedge$), implication ($\Rightarrow$) and parentheses.

Language of propositions

Consider a language of propositions built up from basic values (*False*, *True*) and variables ($A, B, \ldots, Z$) using negation ($\neg$), conjunction ($\wedge$), implication ($\Rightarrow$) and parentheses.

$$A \wedge \neg A$$

$$(A \wedge B) \Rightarrow A$$

$$A \Rightarrow (A \wedge B)$$

$$(A \wedge (A \Rightarrow B)) \Rightarrow B$$

Truth tables

A	$\neg A$
F	T
T	F

A	B	$A \wedge B$
F	F	F
F	T	F
T	F	F
T	T	T

A	B	$A \Rightarrow B$
F	F	T
F	T	T
T	F	F
T	T	T

Truth tables

A	$A \wedge \neg A$
F	F
T	F

A	B	$(A \wedge B) \Rightarrow A$
F	F	T
F	T	T
T	F	T
T	T	T

Truth tables

A	B	$A \Rightarrow (A \wedge B)$
F	F	T
F	T	T
T	F	F
T	T	T

A	B	$(A \wedge (A \Rightarrow B)) \Rightarrow B$
F	F	T
F	T	T
T	F	T
T	T	T

Truth tables

```
data Prop a = Const Bool
            | Var   a
            | Not   (Prop a)
            | And   (Prop a) (Prop a)
            | Imply (Prop a) (Prop a)
```

Truth tables

```haskell
data Prop a = Const Bool
            | Var   a
            | Not   (Prop a)
            | And   (Prop a) (Prop a)
            | Imply (Prop a) (Prop a)
```

$A \wedge \neg A$

```haskell
p1 :: Prop Char
p1 = And (Var 'A') (Not (Var 'A'))
```

Truth tables

```
data Prop a = Const Bool
            | Var   a
            | Not   (Prop a)
            | And   (Prop a) (Prop a)
            | Imply (Prop a) (Prop a)
```

$(A \wedge B) \Rightarrow A$

```
p2 :: Prop Char
p2 = Imply (And (Var 'A') (Var 'B'))
           (Var 'A')
```

Truth tables

```
data Prop a = Const Bool
            | Var   a
            | Not   (Prop a)
            | And   (Prop a) (Prop a)
            | Imply (Prop a) (Prop a)
```

$A \Rightarrow (A \wedge B)$

```
p3 :: Prop Char
p3 = Imply (Var 'A')
           (And (Var 'A') (Var 'B'))
```

Truth tables

```
data Prop a = Const Bool
            | Var   a
            | Not   (Prop a)
            | And   (Prop a) (Prop a)
            | Imply (Prop a) (Prop a)
```

$(A \wedge (A \Rightarrow B)) \Rightarrow B$

```
p4 :: Prop Char
p4 = Imply (And (Var 'A')
                (Imply (Var 'A') (Var 'B')))
           (Var 'B')
```

Show propositions

```
instance Show a => Show (Prop a) where
  show (Const b)     = show b
  show (Var   x)     = show x
  show (Not   p)     = "-(" ++ show p  ++ ")"
  show (And p1 p2)   = "("  ++ show p1 ++ " /\\ "
                            ++ show p2 ++ ")"
  show (Imply p1 p2) = "("  ++ show p1 ++ " => "
                            ++ show p2 ++ ")"
```

Show propositions

```
λ > p1
('A' /\ -('A'))
λ > p2
(('A' /\ 'B') => 'A')
λ > p3
('A' => ('A' /\ 'B'))
λ > p4
(('A' /\ ('A' => 'B')) => 'B')
```

Evaluate propositions

In order to evaluate a proposition to a logical value, we need to know the value of each of its variables.

```
-- list of key-value pairs
type Assoc k v = [(k,v)]

lookupAssoc :: Eq k => k -> Assoc k v -> v
lookupAssoc :: Eq k => k -> Assoc k v -> v
lookupAssoc k kvs = L.head [v | (k',v) <- kvs, k == k']
```

Evaluate propositions

In order to evaluate a proposition to a logical value, we need to know the value of each of its variables.

```
-- substitution
type Subst a = Assoc a Bool

eval :: Eq k => Assoc k Bool -> Prop k -> Bool
eval _ (Const b)   = b
eval s (Var   x)   = lookupAssoc x s
eval s (Not   p)   = not (eval s p)
eval s (And   p q) = eval s p && eval s q
eval s (Imply p q) = eval s p <= eval s q
```

Evaluate propositions

In order to evaluate a proposition to a logical value, we need to know the value of each of its variables.

```
λ > p3
('A' => ('A' /\ 'B'))
λ > eval [('A',False),('B',False)] p3
True
λ > eval [('A',False),('B',True)] p3
True
λ > eval [('A',True),('B',False)] p3
False
λ > eval [('A',True),('B',True)] p3
True
```

Testing

```haskell
-- all distinct variables in a proposition
vars :: (Eq a) => Prop a -> [a]
vars = L.nub . go
  where
    go (Const _)   = []
    go (Var   x)   = [x]
    go (Not   p)   = go p
    go (And   p q) = go p ++ go q
    go (Imply p q) = go p ++ go q
```

Testing

```
λ > (p1, vars p1)
(('A' /\ -('A')),"A")
λ > (p2, vars p2)
((('A' /\ 'B') => 'A'),"AB")
λ > (p3, vars p3)
(('A' => ('A' /\ 'B')),"AB")
λ > (p4, vars p4)
((('A' /\ ('A' => 'B')) => 'B'),"AB")
```

Testing

```haskell
-- all lists of n logical values
bools :: Int -> [[Bool]]
bools 0 = []
bools 1 = [[False],[True]]
bools n = L.map (False:) bs ++ L.map (True:) bs
  where
    bs = bools (n-1)
```

Testing

```
λ > mapM_ print $ bools 3
[False,False,False]
[False,False,True]
[False,True,False]
[False,True,True]
[True,False,False]
[True,False,True]
[True,True,False]
[True,True,True]
```

Testing

```haskell
-- all substitutions of proposition p
substs :: Eq a => Prop a -> [Subst a]
substs p = L.map (L.zip vs) (bools (L.length vs))
  where
    vs = vars p
```

or

```haskell
-- all substitutions of proposition p
substs :: Eq a => Prop a -> [Subst a]
substs p = [L.zip vs bs | bs <- bools (L.length vs)]
  where
    vs = vars p
```

Testing

```
λ >  mapM_ print $ substs p1
[('A',False)]
[('A',True)]
λ > mapM_ print $ substs p3
[('A',False),('B',False)]
[('A',False),('B',True)]
[('A',True),('B',False)]
[('A',True),('B',True)]
```

Testing

```haskell
-- proposition tautology predicate
isTautology :: (Eq a) => Prop a -> Bool
isTautology p = F.and [eval s p | s <- substs p]
```

or

```haskell
-- proposition tautology predicate
isTautology :: (Eq a) => Prop a -> Bool
isTautology p = F.all (\s -> eval s p) $ substs p
```

or (even better)

```haskell
-- proposition tautology predicate
isTautology :: (Eq a) => Prop a -> Bool
isTautology p = F.all (flip eval p) $ substs p
```

Testing

```
λ> (p1, isTautology p1)
(('A' /\ -('A')),False)
λ> (p2, isTautology p2)
((('A' /\ 'B') => 'A'),True)
λ> (p3, isTautology p3)
(('A' => ('A' /\ 'B')),False)
λ> (p4, isTautology p4)
((('A' /\ ('A' => 'B')) => 'B'),True)
```

Improved lookups

Association lists in `Data.List`:

```
λ >  :type L.lookup
L.lookup :: Eq a => a -> [(a, b)] -> Maybe b

λ > L.lookup 'b' [('a',1),('b',2),('c',3)]
Just 2

λ > L.lookup 'd' [('a',1),('b',2),('c',3)]
Nothing
```