Datatypes and functions

Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOG
Robertino Martinez, IOG
Karina Lopez, IOG
Antonio Ibarra, IOG

February, 2024

Goals

- Introduce `Bool`, `Maybe`, `[a]`, binary trees, and one or two other types.
- Introduce the standard design pattern (*catamorphism*) by example for all these types.

Bool

Booleans

```haskell
data Bool = False | True
```

Booleans

```haskell
data Bool = False | True
```

Constructors:

```haskell
False :: Bool
True  :: Bool
```

Booleans

```haskell
data Bool = False | True
```

Constructors:

```haskell
False :: Bool
True  :: Bool
```

Functions on Booleans:

```haskell
fun :: Bool -> TDOTS
fun False = ...
fun True  = ...
```

Patterns

Allowed patterns are:

- Saturated constructor applications to other patterns.
- Variables (match anything).
- Underscore / wildcard pattern (matches anything).
- Literals (numbers, characters, strings, lists).

Exhaustive patterns and overlap

- Patterns are matched in order.
- In particular, catch-all cases must come last.
- Best practice: split programming problems by expanding variables into all possible constructors for the type of the variable.
- For non-overlapping cases, the order does not matter.

Step-by-step function definition

```haskell
not :: Bool -> Bool
not = _
```

Start with the type signature. Use typed holes.

Type of hole: `Bool -> Bool`.

Step-by-step function definition

```haskell
not :: Bool -> Bool
not x = _
```

Introduce function arguments.

Type of hole: `Bool`.
Available locally: `x :: Bool`.

Step-by-step function definition

```
not :: Bool -> Bool
not False = _
not True  = _
```

Split cases (if you cannot solve directly).

Type of holes: `Bool` and `Bool`.

Step-by-step function definition

```
not :: Bool -> Bool
not False = True
not True  = False
```

Solve. Easy in this case.

Step-by-step function definition

```haskell
not :: Bool -> Bool
not False = True
not True  = False
```

Reflect. Everything looks good.

Disjunction (or)

```haskell
(||) :: Bool -> Bool -> Bool
(||) = _
```

Start with the type signature.

Type of hole: `Bool -> Bool -> Bool`.

Disjunction (or)

```haskell
(||) :: Bool -> Bool -> Bool
(||) x y = _
```

Introduce function arguments.

Type of hole: `Bool`.
Available locally: `x :: Bool` and `y :: Bool`.

Disjunction (or)

```haskell
(||) :: Bool -> Bool -> Bool
x || y = _
```

(If preferred, we can use infix notation.)

Type of hole: `Bool`.
Available locally: `x :: Bool` and `y :: Bool`.

Disjunction (or)

```
(||) :: Bool -> Bool -> Bool
False || y = _
True  || y = _
```

Split on a suitable argument (let's take the first).

Type of holes: `Bool` and `Bool`.
Available locally: `y :: Bool` (in each case).

Disjunction (or)

```
(||) :: Bool -> Bool -> Bool
False || y = y
True  || y = True
```

We actually can solve at this point.

Disjunction (or)

```
(||) :: Bool -> Bool -> Bool
False || y = y
True  || y = True
```

Reflect. Everything looks good.

Different options

```haskell
(||) :: Bool -> Bool -> Bool
False || y = y
True  || y = True
```

```haskell
(||) :: Bool -> Bool -> Bool
False || False = False
False || True  = True
True  || False = True
True  || True  = True
```

Different options

```haskell
(||) :: Bool -> Bool -> Bool
False || y = y
True  || y = True
```

```haskell
(||) :: Bool -> Bool -> Bool
False || False = False
False || True  = True
True  || False = True
True  || True  = True
```

```haskell
loop :: a
loop = loop
```

What about `True || loop`?

Undefined / run-time crashes

Haskell does not prevent looping or crashing:

```haskell
undefined :: a
error :: String -> a
```

Undefined / run-time crashes

Haskell does not prevent looping or crashing:

```
undefined :: a
error :: String -> a
```

First (library) version:

```
GHCi> True || undefined
True
```

Second (fully expanded) version:

```
GHCi> True || undefined
*** Exception: Prelude.undefined
```

Undefined / run-time crashes

Haskell does not prevent looping or crashing:

```haskell
undefined :: a
error :: String -> a
```

First (library) version:

```
GHCi> True || undefined
True
```

Second (fully expanded) version:

```
GHCi> True || undefined
*** Exception: Prelude.undefined
```

Also relevant for possibly infinite or just very large terms.

If-then-else

```haskell
ifthenelse :: Bool -> a -> a -> a
ifthenelse False t e = e
ifthenelse True  t e = t
```

Works as expected: only one branch is evaluated.

Syntax for if-then-else

```haskell
ifthenelse :: Bool -> a -> a -> a
ifthenelse c t e = if c then t else e
```

Guards

```haskell
ifthenelse :: Bool -> a -> a -> a
ifthenelse c t e
  | c         = t
  | otherwise = e
```

Guards

```haskell
ifthenelse :: Bool -> a -> a -> a
ifthenelse c t e
  | c         = t
  | otherwise = e
```

- Guards are tried one by one.
- Conditions all of type `Bool`.
- First guard that evaluates to `True` is chosen.

Guards

```haskell
ifthenelse :: Bool -> a -> a -> a
ifthenelse c t e
   | c         = t
   | otherwise = e
```

- Guards are tried one by one.
- Conditions all of type `Bool`.
- First guard that evaluates to `True` is chosen.

```haskell
otherwise :: Bool
otherwise = True
```

Maybe

Optional values

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

Optional values

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

Constructors:

```haskell
Nothing :: Maybe a
Just    :: a -> Maybe a
```

Optional values

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

Constructors:

```haskell
Nothing :: Maybe a
Just    :: a -> Maybe a
```

Functions on `Maybe`:

```haskell
fun :: Maybe a -> ...
fun Nothing  = ...
fun (Just x) = ...
```

Using a default Value

```haskell
fromMaybe :: a -> Maybe a -> a
fromMaybe def Nothing  = def
fromMaybe _   (Just x) = x
```

Using a default Value

```haskell
fromMaybe :: a -> Maybe a -> a
fromMaybe def Nothing  = def
fromMaybe _   (Just x) = x
```

```
GHCi> fromMaybe 3 Nothing
3
GHCi> fromMaybe 3 (Just 5)
5
```

Chaining optional values

```haskell
orelse :: Maybe a -> Maybe a -> Maybe a
orelse Nothing  y = y
orelse (Just x) _ = Just x
```

Chaining optional values

```haskell
orelse :: Maybe a -> Maybe a -> Maybe a
orelse Nothing  y = y
orelse (Just x) _ = Just x
```

```haskell
GHCi> Nothing `orelse` Just 5
Just 5
GHCi> Just 3 `orelse` Just 5
Just 3
GHCi> Just 3 `orelse` Nothing
Just 3
GHCi> Nothing `orelse` Nothing
Nothing
```

Modifying an optional value

```haskell
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing  = Nothing
mapMaybe f (Just x) = Just (f x)
```

Modifying an optional value

```haskell
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing  = Nothing
mapMaybe f (Just x) = Just (f x)
```

```
GHCi> mapMaybe (+ 1) Nothing
Nothing
GHCi> mapMaybe (+ 1) (Just 5)
Just 6
GHCi> mapMaybe not (Just True)
Just False
```

Adding two optional values

```haskell
addMaybes :: Maybe Int -> Maybe Int -> Maybe Int
addMaybes (Just x) (Just y) = Just (x + y)
addMaybes _        _        = Nothing
```

Adding two optional values

```haskell
addMaybes :: Maybe Int -> Maybe Int -> Maybe Int
addMaybes (Just x) (Just y) = Just (x + y)
addMaybes _        _        = Nothing
```

```haskell
liftMaybe ::
  (a -> b -> c)
   -> Maybe a -> Maybe b -> Maybe c
liftMaybe f (Just x) (Just y) = Just (f x y)
liftMaybe _ _        _        = Nothing
```

Pairs

Pairs

```
data (a, b) = (a, b)  -- special syntax
```

Pairs

```haskell
data (a, b) = (a, b)  -- special syntax
```

Constructor:

```haskell
(, ) :: a -> b -> (a, b)
```

Pairs

```haskell
data (a, b) = (a, b)  -- special syntax
```

Constructor:

```haskell
(, ) :: a -> b -> (a, b)
```

Functions on pairs:

```haskell
fun :: (a, b) -> ...
fun (a, b) =  ...
```

Extracting components

```haskell
fst :: (a, b) -> a
fst (a, b) = a
```

```haskell
snd :: (a, b) -> b
snd (a, b) = b
```

Swapping components

```haskell
swap :: (a, b) -> (b, a)
swap (a, b) = (b, a)
```

Currying and uncurrying

```haskell
curry :: ((a, b) -> c) -> a -> b -> c
curry f a b = f (a, b)
```

Currying and uncurrying

```haskell
curry :: ((a, b) -> c) -> a -> b -> c
curry f a b = f (a, b)
```

```haskell
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (a, b) = f a b
```

Lists

Lists

```
data [a] = [] | a : [a]  -- special syntax
```

Lists

```haskell
data [a] = [] | a : [a]  -- special syntax
```

Constructors:

```haskell
[]  :: [a]
(:) :: a -> [a] -> [a]
```

Lists

```haskell
data [a] = [] | a : [a]  -- special syntax
```

Constructors:

```haskell
[]  :: [a]
(:) :: a -> [a] -> [a]
```

Functions on lists:

```haskell
fun :: [a] -> ...
fun []       = ...
fun (x : xs) = ... fun xs ...
```

Recursion in types and functions are connected!

Length of a list

```haskell
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Finding an element in a list

```haskell
elem :: Eq a => a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

Appending two lists

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Reversing a list

```
reverse :: [a] -> [a]
reverse []         = []
reverse (x : xs) = reverse xs ++ [x]
```

Filtering a list

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x : xs)
  | p x         = x : filter p xs
  | otherwise   =     filter p xs
```

Simple look-up tables

Modelling a look-up table

```haskell
type Table k v = [(k, v)]
```

Modelling a look-up table

```haskell
type Table k v = [(k, v)]
```

Interface:

```haskell
empty  :: Table k v
insert :: k -> v -> Table k v -> Table k v
delete :: Eq k => k -> Table k v -> Table k v
lookup :: Eq k => k -> Table k v -> Maybe v
```

Note: functional data structures are *persistent*!

An empty table

```haskell
empty :: Table k v
empty = []
```

Inserting a new key-value pair

```haskell
insert :: k -> v -> Table k v -> Table k v
insert k v t = (k, v) : t
```

Deleting a key

```haskell
delete :: Eq k => k -> Table k v -> Table k v
delete k []     = []
delete k ((k', v) : t)
   | k == k'    = delete k t
   | otherwise = (k', v) : delete k t
```

Deleting a key

```haskell
delete :: Eq k => k -> Table k v -> Table k v
delete k []     = []
delete k ((k', v) : t)
   | k == k'    = delete k t
   | otherwise = (k', v) : delete k t
```

```haskell
delete :: Eq k => k -> Table k v -> Table k v
delete k = filter (\ (k', _) -> not (k == k'))
```

Finding a value for a given key

```haskell
lookup :: Eq k => k -> Table k v -> Maybe v
lookup k []     = Nothing
lookup k ((k', v) : xs)
   | k == k'     = Just v
   | otherwise = lookup k xs
```

A new datatype for tables

```haskell
newtype Table k v = Table [(k, v)]
```

Constructor:

```haskell
Table :: [(k, v)] -> Table k v
```

Functions on tables:

```haskell
fun :: Table k v -> ...
fun (Table table) = ...
```

newtype vs. data

`newtype`:

- restricted to one constructor with one argument,
- otherwise like `data`,
- guaranteed to have the same representation as the wrapped type.

`data`:

- arbitrary number of constructors and argument,
- introduces an additional indirection.

Making tables abstract

- Functions on tables have to be adapted to (un)wrap the `Table` constructor applications.
- The `Table` constructor can then be hidden from the module exports.
- Pattern matching and accessing the internal representation becomes unavailable outside of the defining module, only the interface defined by the functions remains.

Transactions

Modelling a transaction

```haskell
data Transaction =
  Transaction Amount Account Account
  deriving (Eq, Show)

type Amount  = Int
type Account = String
```

Modelling a transaction

```haskell
data Transaction =
  Transaction Amount Account Account
  deriving (Eq, Show)

type Amount  = Int
type Account = String
```

Constructor (same name as type):

```haskell
Transaction ::
  Amount -> Account -> Account -> Transaction
```

Functions on transactions:

```haskell
fun :: Transaction -> ...
fun (Transaction amount from to) = ...
```

Accessing components

```haskell
trAmount :: Transaction -> Amount
trAmount (Transaction a _ _) = a

trFrom :: Transaction -> Account
trFrom (Transaction _ f _) = f

trTo :: Transaction -> Account
trTo (Transaction _ _ t) = t
```

Record syntax

```haskell
data Transaction =
  Transaction
    { trAmount :: Amount
    , trFrom   :: Account
    , trTo     :: Account
    }
  deriving (Eq, Show)
```

- Constructor can still be used with positional arguments.
- Provides selector functions for free.
- Also provides record construction and update syntax, and record patterns.

Updating an account table with a transaction

```haskell
type Accounts = Table Account Amount

processTransaction ::
  Transaction -> Accounts -> Accounts
processTransaction (Transaction amount f t) as =
  let
    fOld = fromMaybe 0 (lookup f as)
    tOld = fromMaybe 0 (lookup t as)
  in
    insert f (fOld - amount)
      (insert t (tOld + amount) as)
```

(It would be better to define a dedicated `update` on tables.)

Binary trees

Binary trees

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

Constructors:

```
Leaf :: a -> Tree a
Node :: Tree a -> Tree a -> Tree a
```

Functions on trees:

```
fun :: Tree a -> ...
fun (Leaf x)   = ...
fun (Node l r) = ... fun l ... fun r ...
```

Functions usually recurse twice!

Flattening a tree into a list

```haskell
flatten :: Tree a -> [a]
flatten (Leaf x)   = [x]
flatten (Node l r) = flatten l ++ flatten r
```

Computing the height of a tree

```haskell
height :: Tree a -> Int
height (Leaf x)   = 0
height (Node l r) = 1 + max (height l) (height r)
```

Expressions

Abstract syntax trees of an expression language

```haskell
data Expr =
      Lit Int
    | Add Expr Expr
    | Neg Expr
    | IfZero Expr Expr Expr
```

Constructors:

```haskell
Lit    :: Int -> Expr
Add    :: Expr -> Expr -> Expr
Neg    :: Expr -> Expr
IfZero :: Expr -> Expr -> Expr -> Expr
```

Functions on expressions

```
fun :: Expr -> ...
fun (Lit n)             = ...
fun (Add e1 e2)         = ... fun e1 ... fun e2
fun (Neg e)             = ... fun e ...
fun (IfZero e1 e2 e3) =
  ... fun e1 ... fun e2 ... fun e3 ...
```

Evaluating an expression

```haskell
eval :: Expr -> Int
eval (Lit n)              = n
eval (Add e1 e2)          = eval e1 + eval e2
eval (Neg e)              = -(eval e)
eval (IfZero e1 e2 e3) =
  if eval e1 == 0 then eval e2 else eval e3
```