ITI0212 Functional programming

Lecture 4

Pawel Sobocinski

1st Homework Assignment submission instructions

- **No lab tomorrow**
- By midday 12pm tomorrow (19/02/2020) send
  - Your solutions `Solutions.idr` + digitally signed statement `statement.txt`
    - I **your name** certify that the solutions submitted are my own
  - to `pawel@cs.ioc.ee`

Either

```
Idris> :doc Either
Data type Prelude.Either.Either : (a : Type) -> (b : Type) -> Type
    A sum type

    The function is: public export
Constructors:
    Left : (l : a) -> Either a b
        One possibility of the sum, conventionally used to represent errors

        The function is: public export
    Right : (r : b) -> Either a b
        The other possibility, conventionally used to represent success

        The function is: public export
```

```
Idris> :t Left
Left : a -> Either a b
Idris> :t Right
Right : b -> Either a b
```

Simple example

```
show : Either Bool Int -> String
show (Left l) = "Bool: " ++ show l
show (Right r) = "Int: " ++ show r
```

```
*Either> show (Left True)
"Bool: True" : String
*Either> show (Right 4)
"Int: 4" : String
```

Exercises

- `pair: (c -> a) -> (c -> b) -> (c -> (a,b))`
- `copair : (a -> c) -> (b -> c) -> (Either a b -> c)`

Maybe

```
Data type Prelude.Maybe.Maybe : (a : Type) -> Type
    An optional value. This can be used to represent the possibility of failure, where a function
    may return a value, or not.

    The function is: public export
Constructors:
    Nothing : Maybe a
        No value stored

        The function is: public export
    Just : (x : a) -> Maybe a
        A value of type a is stored

        The function is: public export
```

Simple example

```elm
head : List ty -> Maybe ty
head [] = Nothing
head (x :: xs) = Just x
```

```
*Head> Main.head ["Hello","world"]
Just "Hello" : Maybe String
*Head> Main.head (the (List String) [])
Nothing : Maybe String
*Head>
```

Vectors - more precise lists

```
Data type Data.Vect.Vect : (len : Nat) -> (elem : Type) -> Type
    Vectors: Generic lists with explicit length in the type
    Arguments:
        len : Nat  -- the length of the list

        elem : Type  -- the type of elements

    The function is: public export
Constructors:
    Nil : Vect 0 elem
        Empty vector

        The function is: public export
    (::) : (x : elem) -> (xs : Vect len elem) -> Vect (S len) elem
        A non-empty vector of length S len, consisting of a head element and the rest of the list, of
        length len.
        infixr 7
```

```
import Data.Vect

fourints : Vect 4 Int
fourints = [0,1,2,3]

sixints : Vect 6 Int
sixints = [4,5,6,7,8,9]

tenints : Vect 10 Int
tenints = fourints ++ sixints
```

Example: refine `allLengths`

```
allLengths : List String -> List Nat
allLengths strs = map length strs
```

Exercise

- Define `add: Vect m Int -> Vect m Int -> Vect m Int`
  - Take a look at `Prelude` function `zipWith`

Explicit vs implicit arguments

- `reverse : (elem: Type) -> List elem -> List elem`
  - `elem` is **explicit**: it needs to be provided when `reverse` is called
- `reverse: {elem: Type} -> List elem -> List elem`
  - `elem` is **implicit** and **bound**
- `reverse: List elem -> List elem`
  - `elem` is **implicit** and **unbound**, internally rewritten to
  - `reverse: {elem : _} -> List elem -> List elem`

Sorting a vector using insertion sort

```
import Data.Vect

inssort : Vect n ty -> Vect n ty
```

Matrix functions

- A matrix is a rectangular array of numbers arranged in rows and columns
- e.g. a 3×4 matrix has 3 rows and 4 columns
- matrices of equal sizes can be added
- matrices of A and B can be multiplied if the number of columns of A is the same as the number of rows of B