ITI0212 Functional programming

Lecture 2

Pawel Sobocinski

Evaluation stategies

- When do arguments to functions get evaluated?
    - `function_1(a+b, function_2(c))`
- Eager (aka strict, aka call-by-value)
    - most languages
    - arguments get evaluated before executing the function body
- Lazy (aka call-by-name)
    - Haskell
    - arguments get evaluated only when actually used
- Idris has eager evaluation

Question

- Can you think of a piece of code that behaves differently with call-by-name and call-by-value?

Lists

- A list can be any size
- Every element must be the same type

```
Idris> [1,3,5,6,9]
[1, 3, 5, 6, 9] : List Integer
Idris> ["Akadeemia","tee","21B"]
["Akadeemia", "tee", "21B"] : List String
```

Basic operations

Concatenation

```
Idris> [1,3,5] ++ [7,9,11]
[1, 3, 5, 7, 9, 11] : List Integer
```

Cons

```
Idris> 'p' :: ['a','w','e','l']
['p', 'a', 'w', 'e', 'l'] : List Char
```

Length

```
Idris> length [1,2,3,4]
4 : Nat
```

What can we say about **any** list?

- It is either empty

```
Idris> []
(input):Can't infer argument elem to []
Idris> the (List Int) []
[] : List Int
```

- Or it contains at least one element, it is of the form `x :: xs`

```
Idris> :let x = 1
Idris> :let xs = [3,5,7]
Idris> x :: xs
[1, 3, 5, 7] : List Integer
```

Inductive data types

- List is an example of something called an **inductive data type**

- Every value of an inductive data type can be constructed from some **constructors**

- Every list can be constructed from `[]` and `x :: xs`

```
Idris> 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: []
[1, 2, 3, 4, 5, 6] : List Integer
```

Pattern matching

- Functions that take as argument an inductive data type benefit from **pattern matching**

- Idris has some cool features that help us write such functions

  - this is our first example of **type driven development**

Writing a length function 1

- Take a list and output its length

```
len : (List Int) -> Int
```

Ctrl-Alt-A — add definition -
adds a skeleton definition for the name under the cursor

```
len : (List Int) -> Int
len xs = ?len_rhs
```

Idris allows you to replace parts of your programs with **holes** to be filled later

Writing a length function 2

Ctrl-Alt-T — type-check name —
— displays the type of the name under the cursor

```
1   len : (List Int) -> Int
2   len xs = ?len_rhs
3
```

Idris: Type of `len_rhs`

```
  xs : List Int
--------------------------------------
len_rhs : Int
```

Writing a length function 3

Ctrl-Alt-C — case split -
Splits a definition into pattern-matching clauses for the name under the cursor

```
1   len : (List Int) -> Int
2   len [] = ?len_rhs_1
3   len (x :: xs) = ?len_rhs_2
```

```
1   len : (List Int) -> Int
2   len [] = ?len_rhs_1
3   len (x :: xs) = ?len_rhs_2
4
```

Idris: Type of `len_rhs_1`

```
--------------------------------------
len_rhs_1 : Int
```

```
1   len : (List Int) -> Int
2   len [] = ?len_rhs_1
3   len (x :: xs) = ?len_rhs_2
4
```

Idris: Type of `len_rhs_2`

```
  x : Int
  xs : List Int
--------------------------------------
len_rhs_2 : Int
```

Writing a length function 3

```
1   len : (List Int) -> Int
2   len [] = 0
3   len (x :: xs) = 1 + len xs
```

```
*Len> len [1,2,3]
3 : Int
```

Syntactic sugar

```
Idris> :doc List
Data type Prelude.List.List : (elem : Type) -> Type
    Generic lists

    The function is: public export
Constructors:
    Nil : List elem
        Empty list

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

        The function is: public export
```

- **Syntactic sugar** - features in syntax that are convenient but do not add expressivity
- `[]` is syntactic sugar for `Nil`
- `[1..5]` expands to `[1,2,3,4,5]`
- `[1,3..7]` expands to `[1,3,5,7]`
- `[5,4..1]` expands to `[5,4,3,2,1]`

Natural numbers

- `Nat` is an inductive data type

```
Data type Prelude.Nat.Nat : Type
    Natural numbers: unbounded, unsigned integers which can be pattern matched.

    The function is: public export
Constructors:
    Z : Nat
        Zero

        The function is: public export
    S : Nat -> Nat
        Successor

        The function is: public export
```

Defining addition

```
1  addnats : Nat -> Nat -> Nat
2
```

Ctrl-Alt-A

```
1  addnats : Nat -> Nat -> Nat
2  addnats k j = ?addnats_rhs
```

Ctrl-Alt-C

```
1  addnats : Nat -> Nat -> Nat
2  addnats Z j = ?addnats_rhs_1
3  addnats (S k) j = ?addnats_rhs_2
```

Ctrl-Alt-T

```
1  addnats : Nat -> Nat -> Nat
2  addnats Z j = ?addnats_rhs_1
3  addnats (S k) j = ?addnats_rhs_2
4
```

```
idris: Type of addnats_rhs_1

  j : Nat
------------------------------------
addnats_rhs_1 : Nat
```

Ctrl-Alt-S - search -searches for a term that has the type of the hole under the cursor

```
addnats : Nat -> Nat -> Nat
addnats Z j = j
addnats (S k) j = ?addnats_rhs_2
```

Defining addition 2

Ctrl-Alt-S

```
addnats : Nat -> Nat -> Nat
addnats Z j = j
addnats (S k) j = j
```

Unfortunately, Idris is not magic…

```
addnats : Nat -> Nat -> Nat
addnats Z j = j
addnats (S k) j = S (addnats k j)
```

Defining multiplication

```
multnats : Nat -> Nat -> Nat
multnats Z j = Z
multnats (S k) j = j + (multnats k j)
```

Repeat

- Take something and create a list containing k copies of it

```
repeat : ty -> Nat -> (List ty)
```

```
repeat : ty -> Nat -> (List ty)
repeat x Z = []
repeat x (S k) = x :: (repeat x k)
```

Generic types

```
repeat : ty -> Nat -> (List ty)
```


lowercase name in type declaration = variable

- here the variable stands for a type
- In Idris, variables in types can also stand for values, as we will see soon

Stutter

- Take a List and a Nat k and output a list where every element appears k times

```
stutter: (List ty) -> Nat -> (List ty)
```

Tuples

- A tuple is a fixed size collection where each element can have a different type

```
("hello", "world") : (String, String)
Idris> ("hello",1,'c')
("hello", 1, 'c') : (String, Integer, Char)
Idris> (1,2,3,'z')
(1, 2, 3, 'z') : (Integer, Integer, Integer, Char)
```

Basic operations on tuples

- `fst`, `snd`

```
Idris> :let x = (3,"three")
Idris> :t x
x : (Integer, String)
Idris> fst x
3 : Integer
Idris> snd x
"three" : String
```

Zip

- Write a function
- `zip_it: List ty -> List ty' -> List (ty, ty')`
- Takes two lists and “zips” them into one

Records

```
record Address where
    constructor MkAddress
    number : Int
    street, city : String
    postcode : Int
    country : string
```

Operations on records

```
*Address> MkAddress 15 "Akadeemia tee" "Tallinn" 12618 "Estonia"
MkAddress 15 "Akadeemia tee" "Tallinn" 12618 "Estonia" : Address
*Address> :let x = MkAddress 15 "Akadeemia tee" "Tallinn" 12618 "Estonia"
*Address> number x
15 : Int
*Address> city x
"Tallinn" : String
*Address> :t city
city : Address -> String
*Address>
```