

Functional Programming

WS 2016/17

Cezary Kaliszyk (VO+PS)

Bertram Felgenhauer (PS)

Sebastiaan Joosten (PS)

Akihisa Yamada (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 02



Overview

- Week 2 - Lists
 - Summary of Week 1
 - List Basics
 - List Functions
 - Modules



Overview

- Week 2 - Lists
 - Summary of Week 1
 - List Basics
 - List Functions
 - Modules



User-defined Types

```
type 'a mylist = Nil | Cons of ('a * 'a mylist)
```

User-defined Types

```
type 'a mylist = Nil | Cons of ('a * 'a mylist)
```

Diagram illustrating the type definition `type 'a mylist = Nil | Cons of ('a * 'a mylist)` with annotations:

- `'a` is annotated as a **type variable**.
- `'a` (the first one in the tuple) is annotated as a **type variable**.
- `'a mylist` (the second part of the tuple) is annotated as a **type variable**.

User-defined Types

user-defined type constructor

```
type 'a mylist = Nil | Cons of ('a * 'a mylist)
```

User-defined Types

user-defined (data) constructor with argument

```
type 'a mylist = Nil | Cons of ('a * 'a mylist)
```



user-defined (data) constructor without argument

User-defined Types

```
type 'a mylist = Nil | Cons of ( 'a * 'a mylist )
```

recursive definition

Recursion, Pattern Matching, and Currying

Example

```
let rec map (f,ls) = match ls with
| Nil          -> Nil
| Cons(x,xs)  -> Cons(f x,map(f,xs))
```

Recursion, Pattern Matching, and Currying (cont'd)

Currying

```
let rec map f ls = match ls with
| Nil          -> Nil
| Cons(x,xs)  -> Cons(f x,map f xs)
```

Recursion, Pattern Matching, and Currying (cont'd)

Currying

```
let rec map f ls = match ls with
| Nil          -> Nil
| Cons(x,xs)  -> Cons(f x,map f xs)
```

Syntactic Sugar

```
let rec map f = function
| []          -> []
| x::xs      -> f x::map f xs
```

Overview

- Week 2 - Lists
 - Summary of Week 1
 - List Basics
 - List Functions
 - Modules



This Week

Practice I

OCaml introduction, **lists**, strings, trees

Theory I

lambda-calculus, evaluation strategies, induction, reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing,

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, dependent types, monads

Overview

- Week 2 - Lists
 - Summary of Week 1
 - **List Basics**
 - List Functions
 - Modules



The Type of Lists

Polymorphic Lists

```
type 'a list = [] | (:::) of ('a * 'a list)
```

The Type of Lists

Polymorphic Lists

```
type 'a list = [] | (:::) of ('a * 'a list)
```

predefined type

The Type of Lists

Polymorphic Lists

```
type 'a list = [] | (:::) of ('a * 'a list)
```

└──┬──┘
infix 'cons'

The Type of Lists

Polymorphic Lists

```
type 'a list = [] | (::) of 'a * 'a list
```

Example

<code>[true;false]</code>	bool list	<code>[(3,2)]</code>	<code>(int * int)list</code>
<code>[1;3;5;7]</code>	int list	<code>[]</code>	<code>'a list</code>
<code>['a';'b']</code>	char list	<code>2::3::[]</code>	int list
<code>["str";"w"]</code>	string list	<code>'a'::['b']</code>	char list

Overview

- Week 2 - Lists
 - Summary of Week 1
 - List Basics
 - List Functions
 - Modules



Accessing List Elements - Selectors

```
let hd = function x::_ -> x
          | _      -> failwith "empty_list"

let tl = function _::xs -> xs
          | _          -> failwith "empty_list"
```

Polymorphic List Functions

Example (Append)

```
let rec (@) xs ys = match xs with []      -> ys
                       | x::xs -> x::(xs @ ys)
```

- this function has type 'a list -> 'a list -> 'a list
- hence it is polymorphic (in `xs` and `ys`)

```
[1;2] @ [3;4]
```

Polymorphic List Functions

Example (Append)

```
let rec (@) xs ys = match xs with []      -> ys
                    | x::xs  -> x::(xs @ ys)
```

- this function has type 'a list -> 'a list -> 'a list
- hence it is polymorphic (in `xs` and `ys`)

```
[1;2] @ [3;4]
→ 1::([2] @ [3;4])
```

Polymorphic List Functions

Example (Append)

```
let rec (@) xs ys = match xs with []      -> ys
                       | x::xs -> x::(xs @ ys)
```

- this function has type 'a list -> 'a list -> 'a list
- hence it is polymorphic (in `xs` and `ys`)

```
[1;2] @ [3;4]
→ 1::([2] @ [3;4])
→ 1::2::([] @ [3;4])
```

Polymorphic List Functions

Example (Append)

```
let rec (@) xs ys = match xs with []      -> ys
                       | x::xs  -> x::(xs @ ys)
```

- this function has type 'a list -> 'a list -> 'a list
- hence it is polymorphic (in `xs` and `ys`)

```
[1;2] @ [3;4]
→ 1::([2] @ [3;4])
→ 1::2::([ ] @ [3;4])
→ 1::2::[3;4]
```


Polymorphic List Functions

Example (Append)

```
let rec (@) xs ys = match xs with []      -> ys
                    | x::xs  -> x::(xs @ ys)
```

- this function has type 'a list -> 'a list -> 'a list
- hence it is polymorphic (in `xs` and `ys`)

```
[1;2] @ [3;4]
→ 1::([2] @ [3;4])
→ 1::2::([ ] @ [3;4])
→ 1::2::[3;4]
= [1;2;3;4]
```

Polymorphic List Functions (cont'd)

Example (Replicate)

```
let rec replicate n x =  
  if n < 1 then [] else x::replicate (n-1) x
```

- this function has type `int -> 'a -> 'a list`
- hence it is polymorphic (in `x`)

```
replicate 2 'c'
```

Polymorphic List Functions (cont'd)

Example (Replicate)

```
let rec replicate n x =  
  if n < 1 then [] else x::replicate (n-1) x
```

- this function has type `int -> 'a -> 'a list`
- hence it is polymorphic (in `x`)

```
replicate 2 'c'  
→ if 2 < 1 then [] else 'c'::replicate (2-1) 'c'
```

Polymorphic List Functions (cont'd)

Example (Replicate)

```
let rec replicate n x =  
  if n < 1 then [] else x::replicate (n-1) x
```

- this function has type `int -> 'a -> 'a list`
- hence it is polymorphic (in `x`)

```
replicate 2 'c'  
→ if 2 < 1 then [] else 'c'::replicate (2-1) 'c'  
→+ 'c'::replicate 1 'c'
```

Polymorphic List Functions (cont'd)

Example (Replicate)

```
let rec replicate n x =  
  if n < 1 then [] else x::replicate (n-1) x
```

- this function has type `int -> 'a -> 'a list`
- hence it is polymorphic (in `x`)

```
replicate 2 'c'  
→ if 2 < 1 then [] else 'c'::replicate (2-1) 'c'  
→+ 'c'::replicate 1 'c'  
→ 'c'::if 1 < 1 then [] else 'c'::replicate (1-1) 'c'
```

Polymorphic List Functions (cont'd)

Example (Replicate)

```
let rec replicate n x =  
  if n < 1 then [] else x::replicate (n-1) x
```

- this function has type `int -> 'a -> 'a list`
- hence it is polymorphic (in `x`)

```
replicate 2 'c'  
→ if 2 < 1 then [] else 'c'::replicate (2-1) 'c'  
→+ 'c'::replicate 1 'c'  
→ 'c'::if 1 < 1 then [] else 'c'::replicate (1-1) 'c'  
→+ 'c'::'c'::replicate 0 'c'
```

Polymorphic List Functions (cont'd)

Example (Replicate)

```
let rec replicate n x =
  if n < 1 then [] else x::replicate (n-1) x
```

- this function has type `int -> 'a -> 'a list`
- hence it is polymorphic (in `x`)

```
replicate 2 'c'
→ if 2 < 1 then [] else 'c'::replicate (2-1) 'c'
→+ 'c'::replicate 1 'c'
→ 'c'::if 1 < 1 then [] else 'c'::replicate (1-1) 'c'
→+ 'c'::'c'::replicate 0 'c'
→ 'c'::'c'::if 0 < 1 then []
                        else 'c'::replicate (0-1) 'c'
```

Polymorphic List Functions (cont'd)

Example (Replicate)

```
let rec replicate n x =
  if n < 1 then [] else x::replicate (n-1) x
```

- this function has type `int -> 'a -> 'a list`
- hence it is polymorphic (in `x`)

```
replicate 2 'c'
→ if 2 < 1 then [] else 'c'::replicate (2-1) 'c'
→+ 'c'::replicate 1 'c'
→ 'c'::if 1 < 1 then [] else 'c'::replicate (1-1) 'c'
→+ 'c'::'c'::replicate 0 'c'
→ 'c'::'c'::if 0 < 1 then []
                        else 'c'::replicate (0-1) 'c'
→+ ['c';'c']
```


Functions on Integer Lists

Example (Range, Sum, Prod)

```
let rec range m n = if m >= n then []  
                    else m::range (m+1) n  
  
let rec sum = function []    -> 0  
                    | x::xs -> x + sum xs  
  
let rec prod = function []    -> 1  
                    | x::xs -> x * prod xs
```

Functions on Integer Lists (cont'd)

```
range 1 3 = [1;2]
```

```
range 3 2 = []
```

```
sum [1;2;3] = 1 + 2 + 3
```

```
sum [] = 0
```

```
prod [1;2;3] = 1 * 2 * 3
```

```
prod [] = 1
```

$$\text{sum}(\text{range } 1 \ n) = \sum_{i=1}^{n-1} i$$

Higher-Order Functions (map)

- functions taking functions as arguments

Example (Map)

```
let rec map f = function []    -> []  
                    | x::xs -> f x::map f xs  
  
    map succ [1;2;3]
```

Higher-Order Functions (map)

- functions taking functions as arguments

Example (Map)

```
let rec map f = function []    -> []  
                    | x::xs -> f x::map f xs
```

```
map succ [1;2;3]  
→ succ 1::map succ [2;3]
```

Higher-Order Functions (map)

- functions taking functions as arguments

Example (Map)

```
let rec map f = function []    -> []  
                    | x::xs -> f x::map f xs
```

```
map succ [1;2;3]  
→ succ 1::map succ [2;3]  
→ 2::map succ [2;3]
```

Higher-Order Functions (map)

- functions taking functions as arguments

Example (Map)

```
let rec map f = function []    -> []  
                    | x::xs -> f x::map f xs
```

```
map succ [1;2;3]  
→ succ 1::map succ [2;3]  
→ 2::map succ [2;3]  
→ 2::succ 2::map succ [3]
```

Higher-Order Functions (map)

- functions taking functions as arguments

Example (Map)

```
let rec map f = function []    -> []  
                    | x::xs -> f x::map f xs
```

```
map succ [1;2;3]  
→ succ 1::map succ [2;3]  
→ 2::map succ [2;3]  
→ 2::succ 2::map succ [3]  
→ 2::3::map succ [3]
```

Higher-Order Functions (map)

- functions taking functions as arguments

Example (Map)

```
let rec map f = function []    -> []  
                    | x::xs -> f x::map f xs
```

```
map succ [1;2;3]  
→ succ 1::map succ [2;3]  
→ 2::map succ [2;3]  
→ 2::succ 2::map succ [3]  
→ 2::3::map succ [3]  
→ 2::3::succ 3::map succ []
```


Higher-Order Functions (map)

- functions taking functions as arguments

Example (Map)

```
let rec map f = function []    -> []  
                    | x::xs -> f x::map f xs
```

```
map succ [1;2;3]  
→ succ 1::map succ [2;3]  
→ 2::map succ [2;3]  
→ 2::succ 2::map succ [3]  
→ 2::3::map succ [3]  
→ 2::3::succ 3::map succ []  
→ 2::3::4::map succ []
```

Higher-Order Functions (map)

- functions taking functions as arguments

Example (Map)

```
let rec map f = function []    -> []  
                    | x::xs -> f x::map f xs
```

```
map succ [1;2;3]  
→ succ 1::map succ [2;3]  
→ 2::map succ [2;3]  
→ 2::succ 2::map succ [3]  
→ 2::3::map succ [3]  
→ 2::3::succ 3::map succ []  
→ 2::3::4::map succ []  
→ 2::3::4::[] = [2;3;4]
```

Higher-Order Functions (foldr)

```
let rec foldr f b = function []      -> b
                        | x::xs -> f x (foldr f b xs)
```

- this function has type
(`'a -> 'b -> 'b`) -> `'b -> 'a list -> 'b`
- this function computes

```
foldr f b [e1;e2;...;eN] = f e1 (f e2 (...(f eN b)...))
```

- this function is useful to implement other functions

```
let sum xs = Lst.foldr ( + ) 0 xs
```

```
let prod xs = Lst.foldr ( * ) 1 xs
```

- here `Lst` refers to a module (see next section)

Overview

- Week 2 - Lists
 - Summary of Week 1
 - List Basics
 - List Functions
 - Modules



Structuring Code

Modules are used to . . .

- split source code into several files
- separate namespaces for functions and types
- abstract from concrete representations

Module Basics - Split Source Code

- for each module *Module* create **implementation** file *module.ml*
- code of each module goes into corresponding *.ml file

Example

```
let hd = ...
let tl = ...
let rec map f = ...
let rec foldr f b = ...
```

Lst.ml

```
let rec range m n = ...
let sum = ...
let prod = ...
```

IntLst.ml

Module Basics - Separate Namespaces

- refer to function *fun* from module *Module* by *Module.fun*
- no problem to have same function names in different modules

Example

Compute greatest number that can be encoded in binary using n bits

```
let pow2 n = IntLst.prod(Lst.replicate n 2)
```

Int.ml

```
let main() =  
  let n = read_int() in  
  let r =  
    IntLst.sum (Lst.map Int.pow2 (IntLst.range 0 n)) in  
  Printf.printf "%i\n" r  
in main()
```

maxbin.ml

Module Basics - Abstraction

- create **interface** file `module.mli` for module `Module`
- **signature** (i.e., names and types) of module goes into corresponding `*.mli` file

Example

```
type 'a t
val empty : 'a t
val push : 'a -> 'a t -> 'a t
val pop : 'a t -> ('a * 'a t)
```

`stck.mli`

```
open Lst
type 'a t = 'a list
let empty = []
let push e s = e::s
let pop s = (hd s,tl s)
```

`stck.ml`

```
type 'a t =
| Empty
| Full of ('a * 'a t)
...
```

`stck.ml`