

Functional Programming

WS 2007/08

Christian Sternagel¹ (VO + PS)
 Friedrich Neurauder² (PS)
 Harald Zankl³ (PS)

Computational Logic
 Institute of Computer Science
 University of Innsbruck

12 October 2007

¹christian.sternagel@uibk.ac.at

²friedrich.neurauder@uibk.ac.at

³harald.zankl@uibk.ac.at

Overview

Week 2 - Lists

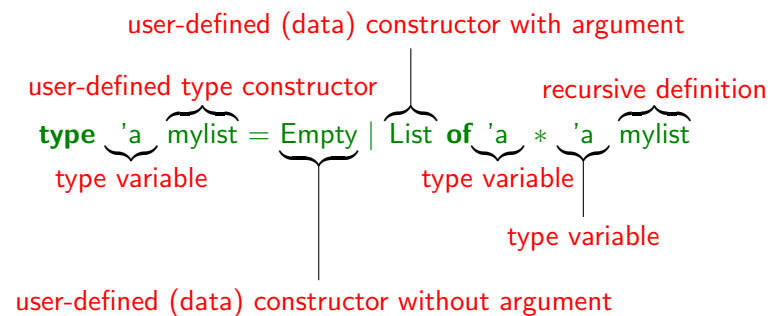
Summary of Week 1
 List Basics
 List Functions
 Modules

Overview

Week 2 - Lists

Summary of Week 1
 List Basics
 List Functions
 Modules

User-defined Types



Recursion, Pattern Matching, and Currying

Recursion and Pattern Matching

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

Currying

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

Syntactic Sugar

```
let rec map f = function
| Empty -> Empty
| List (x, xs) -> List (f x, map f xs)
```

Overview

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

The Type of Lists

Polymorphic Lists

```
type 'a list = Nil | Cons 'a * 'a list
```

'[]' as syntactic sugar
 predefined type
 infix '::' as syntactic sugar

Example

	type		type
[true; false]	bool list	[(3, 2); (4, 3)]	(int * int) list
[1;3;5;7]	int list	[]	'a list
['a'; 'b'; 'c']	char list	1::2::3::[]	int list
["hello"; "world"]	string list	'a'::['b']	char list

Overview

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

Accessing List Elements - Selectors

```
let hd = function
| [] -> failwith "hd: empty list"
| x :: _ -> x
;;
```

```
let tl = function
| [] -> failwith "tl: empty list"
| _ :: xs -> xs
;;
```

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
;;
```

```
range 1 3 = [1; 2; 3]
range 3 2 = []
sum [1; 2; 3] = 1 + 2 + 3
sum [] = 0
prod [1; 2; 3] = 1 * 2 * 3
prod [] = 1
```

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

A Polymorphic List Function

Example (Init)

```
let rec init i l = if l < 1 then [] else i :: init i (l - 1);;
```

- ▶ this function has type 'a -> int -> 'a list
- ▶ hence it is polymorphic in i

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

Higher-Order Functions

- ▶ 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]
```

Fold - A Very Expressive Function

```

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

sum ls = fold (+) 0 ls
prod ls = fold ( * ) 1 ls

fold f b [e1; e2; ...; eN] = f e1 (f e2 (f e3 ( ... (f eN b) ... )))

```

Structuring Code

Modules provide ...

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

Overview

Week 2 - Lists

Summary of Week 1

List Basics

List Functions

Modules

Module Basics - Split Source Code

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

Example

```

let hd = ...
let tl = ...
let rec init i l = ...
let rec map f = ...
let rec fold 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 the greatest number that can be encoded in binary using n bits.

```
let pow2 n = IntLst.prod (Lst.init 2 n);;
let maxbin n =
  IntLst.sum (Lst.map pow2 (IntLst.range 0 (n - 1)))
;;
Format.printf "%i\n" (maxbin (read_int ()));;
maxbin.ml
```

Module Basics - Abstraction

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

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

stck.ml

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

stck.ml