Functional Programming

WS 2013/14

Harald Zankl (VO+PS)
Cezary Kaliszyk (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 3

Lists

Syntax

- `[]` ‘nil’, the empty list
- `::` ‘cons’, add element
- `[1;2;3]` syntactic sugar

Functions

- `Lst.hd` first element
- `Lst.tl` all but first
- `Lst.replicate` create list
- `Lst.map` apply function to list elements
- `Lst.foldr` combine list elements by function

Modules

Using Files

- implementation files (`.ml`)
- signature files (`.mli`)
- ADTs - abstract data types (e.g., `Stck`)

Inline

- **`module`** *Module* `:` *Sig* `=` *Imp*
- **`module`** *Imp* `=` **`struct`** . . . **`end`**
- **`module type`** *Sig* `=` **`sig`** . . . **`end`**

Modules (cont'd)

Signature (.mli)

- types, values
- '**`type`** *type* [= . . .]' for types (possibly abstract)
- '**`val`** *name* `:` *type*' for values

Implementation (.ml)

- type declarations, function definitions, constants
- '**`type`** *type* = . . . ' for types
- '**`let`** *name* = . . . ' for values

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 dynamic programming

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, monads, ...

Built-In Type for Strings

Syntax

- constructed using double quotes ‘"’
- concatenation: `( ^ ) : string -> string -> string`

Example

`"Hello" ^ "␣" ^ "World" = "Hello␣World"`

A String Implementation Using Lists

Strng.ml

- install type abbreviation **type** t = char list
- advantage: all list functions can be used for l-strings
- of_string : string -> t
- to_string : t -> string
- of_int : int -> t
- print : t -> unit

Nice Interpreter Output

Toplevel directives

- always start with # and end with ;;
- #cd "*dir*";; change directory
- #install_printer *name*;; change output function for certain type
- #load "*file.cmo*";; load bytecode
- #quit;; exit the interpreter
- #remove_printer *name*;; remove output function for certain type
- #trace *fun*;; trace computation of function
- #untrace *fun*;; stop tracing of function
- #use "*file*";; execute file content

Nice Interpreter Output (cont'd)

.ocamlinit

```
#cd "_build/"
#install_printer Strng.toplevel_printer
#install_printer Picture.toplevel_printer
open PictureData
```

Implementation of Strng

```
(* W01 *)
(* W03 *)
(* type t *)
type t = char list
(* of_string : string -> char list *)
let of_string s =
 let rec of_string i acc =
  if i < 0 then acc else of_string (i-1) (s.[i]::acc)
 in
 of_string (String.length s - 1) []
(* to_string : char list -> string *)
let to_string xs =
 let buffer = Buffer.create 128 in
 List.iter (Buffer.add_char buffer) xs;
 Buffer.contents buffer
(* of_int : int -> char list *)
let of_int i = of_string(string_of_int i)
(* print : char list -> unit *)
let print s = Printf.printf "%s" (to_string s)
(* toplevel_printer : Format.formatter -> char list -> unit *)
let toplevel_printer fmt s =
 Format.fprintf fmt "\"%s\"" (String.escaped(to_string s))
(* blanks : int -> t *)
let blanks i = Lst.replicate i ' '
```

The Picture Analogon

Picture

- atomic part: pixel
- height and width
- white pixel

L-String

- atomic part: character
- rows and columns
- blank character (space)

The Type of Pictures

```
type width = int

type height = int

type t = (width * height * Strng.t list)
```

Representing Pictures via L-Strings

Example

Picture:

```
*******
*hello*
*******
```

L-String:

```
(7,3,["*******";"*hello*";"*******"])
```

w/o pretty printer:

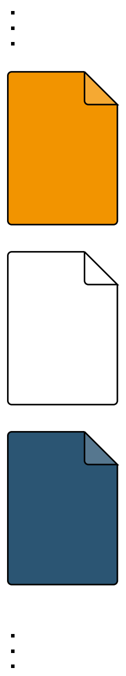
```
(7,3,[['*';'*';'*';'*';'*';'*';'*'];
      ['*';'h';'e';'l';'l';'o';'*'];
      ['*';'*';'*';'*';'*';'*';'*']])
```

Combining Pictures - Stack Above Each Other

Above

```
let above (w1,h1,p1) (w2,h2,p2) =
 if w1 = w2 then (w1,h1+h2,p1@p2)
            else failwith "different␣widths"
```

Combining Pictures - Stack Above Each Other (cont'd)



stack

```
let stack ps = Lst.foldr1 above ps
```

Fold Lists Containing At Least One Element

Fold Right One

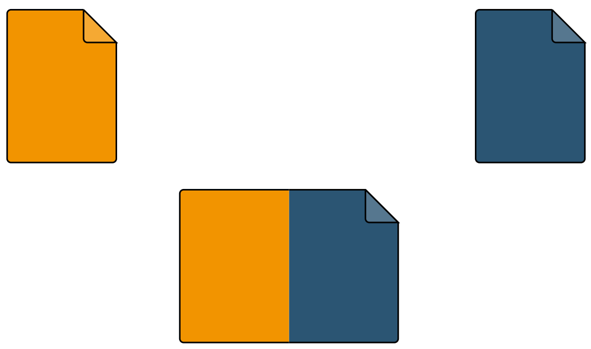
```
Lst.foldr1 : ('a -> 'a -> 'a) -> 'a list -> 'a
```

$$\texttt{Lst.foldr1} \circ \texttt{[}x_1\texttt{;}\dots\texttt{;}x_{n-1}\texttt{;}x_n\texttt{]} = (x_1 \circ (\cdots (x_{n-1} \circ x_n) \cdots))$$

Example

```
      foldr1 (+) [1;2;3] =   1+(2+3)   = 6
 foldr1 (^) ["Hell";"o"] = "Hell"^"o" = "Hello"
```

Combining Pictures - Spread Side By Side



Beside

```
let beside (w1,h1,p1) (w2,h2,p2) =
  if h1 = h2 then (w1+w2,h1,Lst.zip_with (@) p1 p2)
             else failwith "different␣heights"
```

Combine Two Lists Via Function

Zip with Function

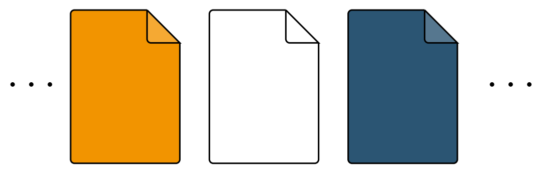
```
zip_with :
  ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

$$\texttt{Lst.zip\_with} \circ [x_1; \ldots ; x_m]\ [y_1; \ldots ; y_n] = [x_1 \circ y_1; \ldots ; x_{\min\{m,n\}} \circ y_{\min\{m,n\}}]$$

Example

```
zip_with ( * ) [1;2] [3;4;5]
= [1*3;2*4]
= [3;8]
zip_with Lst.drop [1;0] [['a'];['b']]
= [Lst.drop 1 ['a']; Lst.drop 0 ['b']]
= [[];['b']]
```

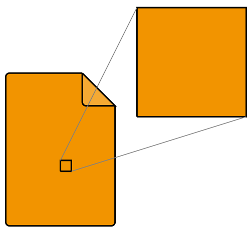
Combining Pictures - Spread Side By Side (cont'd)



Spread

```
let spread ps = Lst.foldr1 beside ps
```

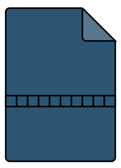
Creating Pictures - Pixels



Pixel

```
let pixel c = (1,1,[[c]])
```

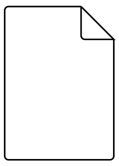
Creating Pictures - Rows



Row

```
let row s = spread(Lst.map pixel s)
```

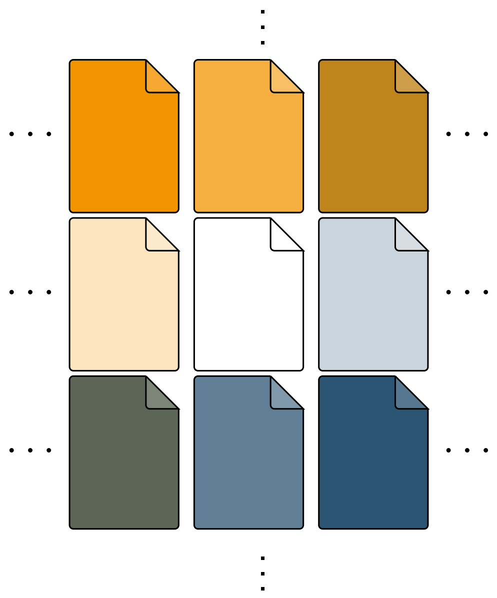
Creating Pictures - Empty Pictures



Empty

```
let empty w h =
 let line = Lst.replicate w ' ' in
 let rows = Lst.replicate h line in
 stack(Lst.map row rows)
```

Combining Pictures - Tiling



Tile

```
let tile pss = stack(Lst.map spread pss)
```

Margins

Signatures

- `stack_with : height -> t list -> t`
- `spread_with : width -> t list -> t`
- `tile_with : height -> width -> t list list -> t`

Functions

```
let stack_with h ps = Lst.foldr1 (fun p q ->
 above (above p (empty (width q) h)) q) ps

let spread_with w ps = Lst.foldr1 (fun p q ->
 beside (beside p (empty w (height q))) q) ps

let tile_with w h pss =
 stack_with h (Lst.map (spread_with w) pss)
```

Printing Pictures

Idea

- convert to `Strng.t` and use `Strng.print`

Realization

- `Picture`:

```
let to_strng (_,_,p) = Lst.join ['\n'] p
```

- `Strng`:

```
let print s = Printf.printf "%s" (to_string s)
```

Join Function

`join : 'a list -> 'a list list -> 'a list`

$$\texttt{Lst.join}\ d\ [x_1; \ldots; x_n] = x_1 @ d @ x_2 @ \cdots @ x_{n-1} @ d @ x_n$$