Functional Programming

http://cl-informatik.uibk.ac.at/teaching/ss07/fp/

Stefan Blom

Computational Logic
Institute of Computer Science
University of Innsbruck

SS 2007

Goals

Learn how to ...

1. use the functional programming language OCaml
2. implement a functional programming language
3. prove properties about a functional program

Evaluation

- 50 points for small programming projects
- 50 points for written final exam
- need 50 points to pass
- First written exam in last week: Juli 4, 8.00 - 10.00.
- Anrechnung as 'Programming in OCaml' for Masters students

Reading material

- Jason Hickey, An Introduction to the Objective Caml Programming Language
  `http://mojave.caltech.edu/jyh/publications.html`
- The OCaml reference manual
  `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`

programming projects

- In principle everybody works alone
- Examples:
  - Tree data types: Heaps, AVL trees, Red-Black trees, etc.
  - Combinator Parser
  - Type Inference Engine
  - Interpreter for toy-ML
  - Japanese puzzles: Kakuro, Sudoku, etc.
- Current set allows some choice
- but not enough: feel free to suggest your own project(s)

What makes OCaml different?

- everything is an expression
  E.g. $inc : n \mapsto n + 1$ is written as `fun n -> n+1`
  You can bind the name `inc` with:

```
let inc = fun n -> n+1;;
```

- pattern matching
  E.g. factorial

$$n! = \begin{cases} 1 & \text{, if } n = 0 \\ n \cdot (n-1)! & \text{, otherwise} \end{cases}$$

is defined as

```
let rec factorial x = match x with
  | 0 -> 1
  | n -> n * factorial(n-1)
;;
```

How does pattern matching work?

$$
\begin{array}{ll}
 & \text{factorial } 3 \\
\rightarrow & \text{match } 3 \text{ with} \\
 & \quad |0 \rightarrow 1 \\
 & \quad |n \rightarrow n * \text{factorial}(n-1) \\
\rightarrow & \text{match } 3 \text{ with} \\
 & \quad |n \rightarrow n * \text{factorial}(n-1) \\
\rightarrow & 3 * \text{factorial}(3-1) \\
\rightarrow & 3 * \text{factorial}(2) \\
\rightarrow & 3 * (2 * \text{factorial}(1)) \\
\rightarrow & 3 * (2 * (1 * \text{factorial}(0))) \\
\rightarrow & 3 * (2 * (1 * 1)) \\
\rightarrow & 3 * (2 * 1) \\
\rightarrow & 3 * 2 \\
\rightarrow & 6
\end{array}
$$

Lists

- the list of x_1 up to x_n is denoted as $[x_1; \cdots ; x_n]$
- which really stands for $x_1 :: (x_2 :: (\cdots (x_n :: []) \cdots))$
- all elements $x_1, \cdots , x_n$ must have the same type:
  - $[1; 2; 3]$ is a list of integers
  - $1 :: 2 :: 3 :: []$ and $1 :: [2; 3]$ are equivalent
  - $[''X''; ''r'']$ is a list of strings
  - $[[1]; [1; 2]]$ is a list of lists of integers
  - $[1; [1; 2]]$ is illegal

pattern matching on lists

- Length of a list:

```
let rec length x = match x with
  | []      -> 0
  | x :: xs -> 1 + (length xs)
```

  or

```
let rec length = function
  | []      -> 0
  | x :: xs -> 1 + (length xs)
```

- Removing double occurences

```
let rec uniq = function
 | x1::x2::xs when x1 = x2 -> uniq(x2::xs)
 | x :: xs -> x :: uniq(xs)
 | [] -> []
```

- The function map is specified by

$$\text{map } f \; [x_1; \cdots; x_n] = [f\,x_1; \cdots; f\,x_n]$$

- It can be defined as

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

- Note that `map (fun n -> n+1)` is a legal expression. That is `map` has a function as argument and returns a function.
- Other functions are

$$\begin{array}{lcl} \text{fold\_left} \diamond e \, [x_1; \cdots; x_n] & = & ((\cdots((e \diamond x_1) \diamond x_2) \cdots) \diamond x_n) \\ \text{fold\_right} \diamond [x_1; \cdots; x_n] \, e & = & (x_1 \diamond (\cdots (x_n \diamond e) \cdots)) \end{array}$$

Q & A

Q What is the 'greek letter equivalent' of `fun n -> n+1`?

A $\lambda n.n + 1$ which is `\n.n+1` in ASCII.

Q Does C have an equivalent for `fun n -> n+1`?

A No, not as an expression

Q Does Java have an equivalent for `fun n -> n+1`?

A Yes, if you declare

```
interface Function{
  public int call(int x);
}
```

then `fun n -> n+1` can be written as

```
new Function(){public int call(int x){return x+1;}}
```

This feature is called anonymous class.