Functional Programming

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

Stefan Blom

Computational Logic
Institute of Computer Science
University of Innsbruck

SS 2007

Basic types

unit
```
# ();;
- : unit = ()
```

booleans
```
# not true;;
- : bool = false
# true || false;;
- : bool = true
# true && false;;
- : bool = false
```

Basic types

chars
```
# 'x';;
- : char = 'x'
# '\065';;
- : char = 'A'
```

strings
```
# "hello";;
- : string = "hello"
# "\065\n";;
- : string = "A\n"
```

Basic types

integers
```
# 1+1;;
- : int = 2
# 2 * 4;;
- : int = 8
```

float
```
# 2.0 +. 3.0 ;;
- : float = 5.
# 2.0 *. 3.0 ;;
- : float = 6.
```

Basic types

tuples
```
# (1,2);;
- : int * int = (1, 2)
# 1,2;;
- : int * int = (1, 2)
# (1,2,3)=(1,(2,3));;
This expression has type int * (int * int)
but is here used with type  int * int * int
# ('x',"x",(2.5,3),());;
- : char * string * (float * int) * unit
                 = ('x', "x", (2.5, 3), ())
```

Overloading

impossible

```
# (+) ;;
- : int -> int -> int = <fun>
# ( * ) (* remember comments *);;
- : int -> int -> int = <fun>
# (+.);;
- : float -> float -> float = <fun>
# ( *. );;
- : float -> float -> float = <fun>
```

faked

```
# (<);;
- : 'a -> 'a -> bool = <fun>
```

Same for the other comparisons: `<=`, `!=`, `==`, `=`, `>=`, `>`

Problem with overloading

- Not every data type supports $+, *, \cdots$.
- The usual rule for overloading says:

  given the argument types,
  the result type must be unique

  That is enough for type checking.
- But not for type inference. For example,

```
let double x = x + x;;
```

  If $+$ were overloaded then what is the type `double`?

Faking of overloading

- Comparison is defined for every data type
- No problem with type inference
- The compiler inserts the correct code for each data type

Equality

pointer
```
# "x" == "x" ;;
- : bool = false
# "x" != "x";;
- : bool = true
```

structural
```
# "x" = "x" ;;
- : bool = true
# "x" <> "x";;
- : bool = false
```

Equality

lists
```
# [] == [] ;;
- : bool = true
# [] = [];;
- : bool = true
# [1] == [1];;
- : bool = false
# [1] = [1];;
- : bool = true
```

Printf

```
# open Printf;;
# printf;;
- : ('a, out_channel, unit) format -> 'a = <fun>
# printf "x=%d\n";;
- : int -> unit = <fun>
# printf "x=%d\n" 3;;
x=3
- : unit = ()
# printf "%s%c%f";;
- : string -> char -> float -> unit = <fun>
```

Please note that the compiler translates strings to formats

A list data type in C

```
typedef struct cons* list_t;
struct cons {
        void* elem;
        list_t next;
};
```

Length with a loop in C

```
int length(list_t list){
        int len=0;
        while(list!=NULL){
                list=list->next;
                len++;
        }
        return len;
}
```

Length with recursion in C

```
int length(list_t list){
        if(list!=NULL){
                return 1+length(list->next);
        } else {
                return 0;
        }
}
```

List data types in OCaml

- The pre-defined type `'a list'` (`[1;2]`, `1::2::[]`, etc.)
- Your own list type:

```
type 'a mylist = Nil | Cons of 'a * 'a mylist;;
```

Why the explicit `Nil`?

It solves a problem in C/Java:

- Our C data type used null as empty list.
- The get(key) method of java.util.Map returns

  the value to which this map maps the specified key,
  or null if the map contains no mapping for this key.

- These are conflicting uses of null:

  you can never map an object to an empty list

Solutions in OCaml

empty list `Nil`

map.get the pre-defined

```
type 'a option = None | Some of 'a;;
```

allows

```
get : ('a,'b) hashtable -> 'a -> 'b option
```

> *get map key returns Some e if key maps to e and None the map contains no mapping for this key.*

Length with recursion OCaml

```
let rec length = function
  | Nil -> 0
  | Cons(_,xs) -> 1+(length xs)
;;

length(Cons(1,Cons(3,Cons(5,Nil))))
 = 1+(length(Cons(3,Cons(5,Nil))))
 = 1+(1+(length(Cons(5,Nil))))
 = 1+(1+(1+(length(Nil))))
 = 1+(1+(1+0))) = ... = 3
```

- builds an expression during evaluation
- this expression has to be stored (on the stack)

Length with tail recursion OCaml

```
let rec len n x = match x with
  | Nil -> n
  | Cons(_,xs) -> len (n+1) xs
;;
let length x = len 0 x;;
```

Length with tail recursion OCaml

```
length(Cons(1,Cons(3,Cons(5,Nil))))
 = len 0 (Cons(1,Cons(3,Cons(5,Nil))))
 = len 1 (Cons(3,Cons(5,Nil)))
 = len 2 (Cons(5,Nil))
 = len 3 (Nil)
 = 3
```

- does not build an expression during evaluation
- easy to avoid memory use in compiler

Optimization

It's not nice that `len` is visible, so we make len local:

```
let length x =
  let rec len n x = match x with
    | Nil -> n
    | Cons(_,xs) -> len (n+1) xs
  in len 0 x
;;
```

Two functions

Consider

```
# let f(x,y)=x+y;;
val f : int * int -> int = <fun>
# let g x y = x + y;;
val g : int -> int -> int = <fun>
```

- `let f(x,y)=x+y` is short for
  `let f p = match p with (x,y) -> x+y`

Two functions

- Different syntax, also for calls:

```
# f(1,2);;
- : int = 3
# g 1 2;;
- : int = 3
```

- Only for g, we have

```
# g 1;;
- : int -> int = <fun>
```

Two functions

- Different syntax, also for calls:

```
# f(1,2);;
- : int = 3
# g 1 2;;
- : int = 3
```

- ```
  # let curry f = fun x y -> f(x,y);;
  val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
  # let uncurry f = fun (x,y) -> f x y ;;
  val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
  ```
  ```

Head and tail of a non-empty list.

We can define the first element of a list as:

```
# let hd(Cons(a,_)) = a;;
...
# hd(Cons(1,Cons(3,Cons(5,Nil))));;
- : int = 1
```

Head and tail of a non-empty list.

However,

```
# let hd(Cons(a,_)) = a;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Nil
val hd : 'a mylist -> 'a = <fun>
# hd(Nil);;
Exception: Match_failure ("", 15, -88).
```

Head and tail of a non-empty list.

So, we write

```
let hd = function
 | Cons(a,_) -> a
 | Nil -> failwith "hd must be called on non-empty list"
;;
```

```
# hd(Cons(1,Cons(3,Cons(5,Nil))));;
- : int = 1
# hd(Nil);;
Exception: Failure "hd must be called on non-empty list".
```

Head and tail of a non-empty list.

Similarly

```
let tl = function
 | Cons(_,x) -> x
 | Nil -> failwith "tl must be called on non-empty list"
;;
```

Assignment in OCaml

- Does not work on let (rec) defined variables.
- Works on references:

```
# ref;;
- : 'a -> 'a ref = <fun>
# (:=);;
- : 'a ref -> 'a -> unit = <fun>
# (!);;
- : 'a ref -> 'a = <fun>
```

Example

```
# let x=ref 2;;
val x : int ref = {contents = 2}
# !x;;
- : int = 2
# x:=3;;
- : unit = ()
# x;;
- : int ref = {contents = 3}
```

Length with a loop in OCaml

```
let length x =
  let xs  = ref x in
  let len = ref 0 in
  (while !xs != Nil do
     len := !len+1;
     xs  := tl(!xs)
   done;
   !len
  )
;;
```

More examples

- ```
  let rec filter p x = match x with
    | [] -> []
    | a::x when p a -> a::(filter p x)
    | a::x -> filter p x
  ;;
  ```
- ```
  let rec split p x = match x with
   | [] -> ([],[])
   | a::x -> let (xs,ys)=split p x in
               if p a then (a::xs,ys) else (xs,a::ys)
  ;;
  ```

More examples

- ```
  let rec concat = function
   | [] -> []
   | a::x -> a@(concat x)
  ;;
  ```
- `let concat = fold_left (@) [];;`
- `let concat = fold_right (@) [];;`
  ```

Binary Trees

- A labeled binary tree is a tree, whose nodes have either two children and a label or no children and no label.
- In OCaml a tree labeled with elements from `'a` is defined as

  ```
  type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
  ```
- ```
  # Leaf;;
  - : 'a tree = Leaf
  # Node(1,Leaf,Leaf);;
  - : int tree = Node (1, Leaf, Leaf)
  ```
  ```

Examples

```
let rec size = function
 | Leaf -> 0
 | Node(a,t1,t2) -> 1+size(t1)+size(t2)
;;

let rec depth = function
 | Leaf -> 0
 | Node(a,t1,t2) -> 1 + max (depth t1) (depth t2)
;;
```

Examples

```
let rec map f = function
 | Leaf ->
 | Node(a,t1,t2) ->
;;
```

```
val map : ('a -> 'b) -> 'a tree -> 'b tree = <fun>
# map even (Node(1,Node(1,Leaf,Leaf),Leaf));;
- : bool tree = Node (false, Node (false, Leaf, Leaf), Leaf
# map ((+)1) (Node(1,Node(1,Leaf,Leaf),Leaf));;
- : int tree = Node (2, Node (2, Leaf, Leaf), Leaf)
```

Examples

```
let rec map f = function
 | Leaf -> Leaf
 | Node(a,t1,t2) ->
;;

val map : ('a -> 'b) -> 'a tree -> 'b tree = <fun>
# map even (Node(1,Node(1,Leaf,Leaf),Leaf));;
- : bool tree = Node (false, Node (false, Leaf, Leaf), Leaf
# map ((+)1) (Node(1,Node(1,Leaf,Leaf),Leaf));;
- : int tree = Node (2, Node (2, Leaf, Leaf), Leaf)
```

Examples

```
let rec map f = function
 | Leaf -> Leaf
 | Node(a,t1,t2) -> Node(f a,map f t1,map f t2)
;;

val map : ('a -> 'b) -> 'a tree -> 'b tree = <fun>
# map even (Node(1,Node(1,Leaf,Leaf),Leaf));;
- : bool tree = Node (false, Node (false, Leaf, Leaf), Leaf
# map ((+)1) (Node(1,Node(1,Leaf,Leaf),Leaf));;
- : int tree = Node (2, Node (2, Leaf, Leaf), Leaf)
```