Pattern matching

List patterns: new productions for `Pat`

```
Pat ::= '[' ']' | Pat '::' Pat | '[' Pat (';' Pat)* ']'
```

What is pattern matching?

- a powerful mechanism for associating values with variables/parameters by decomposition
- patterns can use constructors, other operators are not allowed
  - constructors guarantee unique decomposition
- all variables in a pattern must be distinct
  - this makes pattern matching more efficient

Examples

Valid patterns: `x` `x::y` `[x;y;z]` `x,y`

Non-valid patterns: `x@y` `x+y` `x&&y` `x,x`

Pattern matching

Examples of use of pattern matching

```
let add (x,y) = x+y;;
add (3,5);; (* does (3,5) match with pattern (x,y)? *)
```

- `(3,5)` and `(x,y)` match with substitution `x=3,y=5`
- if `x=3,y=5`, then `x+y` evaluates to `3+5=8`

Pattern matching

Examples of use of pattern matching

```
let hd (h::t) = h;; (* returns the head of the list *)
hd [3;5];; (* does [3;5] match with pattern h::t? *)
```

- `[3;5]` and `(h::t)` match with substitution `h=3,t=[5]`
- if `h=3,t=[5]`, then `h` evaluates to `3`
- Remarks:
  - `[3;5]` and `[5]` are syntactic abbreviations for `3::5::[]` and `5::[]`
  - variable `t` is unused in the body of `hd`

A different definition of `hd` which does not need variable `t`:

```
let hd (h::_) = h;; (* head of the list, with wildcard '_' *)
```

Remark:
wildcard _ is an anonymous variable with meaning “do not care the value”

Pattern matching

Does a single pattern work for all valid arguments of a function?

```
let hd (h::_) = h;; (* head of the list, with wildcard '_' *)
hd [];; (* error! [] and h::_ do not match *)
```

- `[]` and `h::_` do not match
- this is reasonable, because the head of the empty list is undefined

Remark

- a single variable *x* or wildcard _ is the simplest form of pattern
- match with *x* or _ always succeeds for any kind of value

Examples of functions on lists that need to be defined by cases

- the length of a list
- the sum of all the elements of a list
- the list with the first two elements swapped

Pattern matching

An expression to match values with multiple patterns

```
Exp ::= 'match' Exp 'with' Pat '->' Exp ('|' Pat '->' Exp)*
```

Examples

```
(* functions defined by two cases *)

let rec length l = match l with
    [] -> 0
  | _::t -> 1+length t;; (* t is a local variable for this case *)

let rec sum l = match l with
    [] -> 0
  | h::t -> h+sum t;; (* h and t are local variables for this case *)

(* function defined by three cases *)

let swap l = match l with
    [] -> []
  | [x] -> [x] (* x is a local variable for this case *)
  | x::y::t -> y::x::t;; (* x, y and t are local variables for this case *)
```

Pattern matching

match e **with** p_1 `->` e_1 `| ... |` p_n `->` e_n

Static semantics

- the expression e and all patterns $p_1 \dots p_n$ must have the same type
- all expressions $e_1 \dots e_n$ must have the same type
- each e_i can use the variables in p_i with the inferred types

Dynamic semantics

- e is evaluated
- all patterns $p_1 \dots p_n$ are tried from left to right, top to bottom
- let p_i be the first pattern for which e and p_i match; then, the expression e_i is evaluated, with variables defined by the the successful match
- if there is no match, then exception `Match_failure` is raised

Pattern matching

Static semantics: further checks

A warning is reported if:

- patterns are not exhaustive, that is, some case is missing
- a pattern is unused

Example

```
# let head (_::tl) = hd;;
           ^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]

# let rec length l = match l with
    _::tl -> 1+length tl
  | [x] -> 1
    ^^^
  | [] -> 0;;

Warning 11: this match case is unused.
```

Pattern matching

Unique decomposition

Constructors ensure that if there is a match with *p*, then there are unique values for the variables in *p*

Counter-example

```
# let foo ls = match ls with l1@l2 -> l1;; (* @ not a constructor! *)
                                ^
Error: Syntax error
```

What would be the values of `l1` and `l2` for the application `foo [1;2;3]`?

`[]` and `[1;2;3]`?
`[1]` and `[2;3]`?
`[1;2]` and `[3]`?
`[1;2;3]` and `[]` ?

Pattern matching

Constructors for primitive types

All *literals* (=tokens that represent values) are constant constructors

Example of pattern matching with primitive types

```
let mynot b = match b with false -> true | true -> false;;

let iszero i = match i with 0 -> true | _ -> false;;
```

Remarks

pattern matching with primitive types is seldom used;
conditional expressions and equality test are used more often

Pattern matching

Shorthand notation

- **function** p_1 -> e_1 | ... | p_n -> e_n is a shorthand for
  fun *var* -> **match** *var* **with** p_1 -> e_1 | ... | p_n -> e_n
- p **as** *id*: a pattern (or sub-pattern) p can be associated with variable *id* to refer to the matched value more directly on the right-hand side of ->

Pattern matching

Examples

```
let mynot = function false -> true | _ -> false;;

let iszero = function 0 -> true | _ -> false;;

let rec length = function _::tl -> 1+length tl | _ -> 0;;

let rec sum = function hd::tl -> hd+sum tl | _ -> 0;;

let swap = function x::y::l -> y::x::l | other -> other;;

let ord_swap = function (* ls shorter than x::y::tl *)
    x::y::tl as ls -> if x>y then y::x::tl else ls
  | other -> other;;
```

Strings in OCaml

In a nutshell

- primitive type `string` supported
- standard literals (the only constructors)
  - `""` is the empty string, `"hello world"` is a non-empty string
- concatenation `^`: left-associative, lower precedence than application
- predefined module String

Examples

```
let s = "hello" ^ " " ^ "world";;
val s : string = "hello world"
(^);;
- : string -> string -> string = <fun>
String.length s;;
- : int = 11
String.uppercase_ascii s;;
- : string = "HELLO WORLD"
String.lowercase_ascii "HELLO WORLD";;
- : string = "hello world"
```

Predefined functions on lists in OCaml

Module `List`

- predefined module List
- some examples of functions:
  - `val length : 'a list -> int`

    returns the length (number of elements) of the given list
  - `val nth : 'a list -> int -> 'a`

    returns the n-th element of the given list. The head of the list is at position 0
  - `val init : int -> (int -> 'a)-> 'a list`

    `init len f` is `[f 0; f 1; ...; f (len-1)]` evaluated left to right

Examples

```
# let ls = List.init 10_000 (fun x->x+1);;
val ls : int list = [1; 2; 3; ... ]
# List.length ls;;
- : int = 1000000
# List.nth ls (List.length ls - 1);;
- : int = 1000000
```

Recursion and efficiency

Example 1: sum

```
(* computes the sum of the elements of a list *)
# let rec sum = function
    hd::tl -> hd + sum tl (* inductive case *)
  | _ -> 0;;             (* base case [] *)
val sum : int list -> int = <fun>

# let ls=List.init 1_000 (fun x->x+1)  (* ls = [1;2;...;1_000] *)
in sum ls;;
- : int = 500500

# let ls=List.init 10_000 (fun x->x+1)(* ls = [1;2;...;10_000] *)
in sum ls;;
Stack overflow during evaluation (looping recursion?).
```

Recursion and efficiency

Example 2: reverse

```
# let rec reverse = function
    hd::tl -> reverse tl @ [hd] (* inductive case *)
  | _ -> [];;                   (* base case [] *)
  val reverse : 'a list -> 'a list = <fun>

# let ls=List.init 6_000 (fun x->x+1) (* ls = [1;2;...;6_000] *)
in reverse ls;;                       (* it takes time! *)
- : int list = [6000; 5999; 5998; ...]
```

Time complexity

- `tl @ [hd]` is $O(n)$: linear in the length n of `tl`
- `reverse ls` is $O(n^2)$: quadratic in the length n of `ls`!

Recursion and efficiency

Example 3: fib and bin

```
(* Fibonacci numbers *)
# let rec fib n = if n<=1 then n else fib(n-2)+fib(n-1);;
val fib : int -> int = <fun>

(* binomial coefficients *)
# let rec bin n k = if n=k||k=0 then 1 else bin(n-1)(k-1)+bin(n-1) k;;
val bin : int -> int -> int = <fun>
```

Time complexity

- `fib n` is $O(2^n)$: exponential in `n`!
- `bin n n/2` is $O(2^n)$: exponential in `n`!

Accumulators

A standard loop to accumulate a result

```
(* example with imperative programming, this is not OCaml ! *)
sum(ls){
  acc=0; (* initial value of the accumulator *)
  while(true){
    match ls with
       hd::tl -> {acc=acc+hd; ls=tl;}
    |  [] -> return acc
  }
}
```

Simulation in functional programming with OCaml

```
let acc_sum =                (* acc_sum :  int list -> int *)
  let rec aux acc = function (* aux : int -> int list -> int *)
      hd::tl -> aux (acc+hd) tl
    | _ -> acc
  in aux 0;;
```

Tail recursion

Definition of tail recursion

- the recursive application is always the last performed operation
- it can be implemented with a real loop and no stack

sum is not tail recursive

```
let rec sum = function
    hd::tl -> hd + sum tl (* last operation: addition *)
  | _ -> 0;;
```

aux is tail recursive

```
let rec aux acc = function
    hd::tl -> aux (acc+hd) tl  (* last operation: recursive application *)
  | _ -> acc
in aux 0;;
```

Accumulators and tail recursion

Efficient definition of sum

```
# let acc_sum =
  let rec aux acc = function
      hd::tl -> aux (acc+hd) tl
    | _ -> acc
  in aux 0;;
val acc_sum : int list -> int = <fun>

# let ls=List.init 10_000 (fun x->x+1)(* ls = [1;2;...;10_000] *)
in acc_sum ls;;
- : int = 50005000
```

Remarks

- `aux` is tail recursive with an accumulator
- `aux` hides the implementation details of `acc_sum`
- `acc_sum` calls `aux` and passes the initial value of `acc` (0 in this case)

Accumulators and tail recursion

Efficient definition of reverse

```
# let acc_rev ls = (* parameter ls needed to get a polymorphic function *)
  let rec aux acc = function
      hd::tl -> aux (hd::acc) tl
    | _ -> acc
  in aux [] ls;;
val acc_rev : 'a list -> 'a list = <fun>

# let ls=List.init 10_000 (fun x->x+1)(* creates list [1;2;...;10_000] *)
in acc_rev ls;;
- : int list = [10000; 9999; 9998; ...]
```

Time complexity

- `hd::acc` is $O(1)$: constant time
- `acc_rev ls` is $O(n)$: linear in the length n of `ls`

Remark

Efficient reverse defined in module `List`: `List.rev`