Lists

Lists are built-in values in OCaml

Some examples of built-in composite list types:

- `int list`
- `(int * int)list`
- `(int -> int)list`
- `int list list`

List constructors

- Syntax: `Exp ::= '[' ']'| Exp '::'Exp`
- `[]` is the empty list constructor
- `::` is the non-empty list constructor: h`::`t is the list with head h and tail t

Lists

Examples

```
# let l=1::2::3::[];;
val l : int list = [1; 2; 3]
# let l2=0::l;;
val l2 : int list = [0; 1; 2; 3]
# let pl = (1,2)::(3,4)::[];;
val pl : (int * int) list = [(1, 2); (3, 4)]
# let fl=(fun x->x+1)::(fun x->x*2)::[];;
val fl : (int -> int) list = [<fun>; <fun>]
# let ll=(1::[])::(2::3::[])::[];;
val ll : int list list = [[1]; [2; 3]]
```

Syntactic rules for lists

The usual properties of constructors hold

- `[]` $\neq h::t$ $\quad h \neq h::t$ $\quad t \neq h::t$
- $h_1::t_1 = l$ if and only if $l = h_2::t_2$, $h_1 = h_2$ and $t_1 = t_2$

Non-empty list constructor `::`

- right syntactic associativity holds
  $h_1::h_2::t$ is equivalent to $h_1::(h_2::t)$
  this is the only sensible choice (see later on)
- `::` has lower precedence than unary and binary infix operators
- `::` has higher precedence than
  - the tuple constructor
  - anonymous function expression (**`fun ... -> ...`**)
  - conditional expression (**`if ... then ... else ...`**)

Syntactic rules for lists

A useful shorthand notation

`[`e_1`;`e_2`;...;`e_n`]` is equivalent to e_1`::`e_2`::...::`e_n`::[]`

Examples

```
# 1::[];;
- : int list = [1]
# 1::2::3::[];;
- : int list = [1; 2; 3]
# (1,true)::[];;
- : (int * bool) list = [(1, true)]
# (1,true::[]);;
- : int * bool list = (1, [true])
```

Warning

Use parentheses if you mix lists and tuples together!

Lists and types

Examples of list types

Built-in composite types defined with the list type constructor:

- `int list`: the lists of integers
- `(int * int)list`: the lists of pairs of integers
- `(int -> int)list`: the lists of functions from integers to integers
- `int list list`: the lists of lists of integers

Syntax of the list type constructor

- `list` is a unary postfix constructor
  - unary: it has one argument which defines the type of the elements of the list
  - post-fix: the constructor comes after its argument
- `list` has higher precedence than the `->` and `*` constructors
- the usual properties of constructors hold
  - $t \neq t$ `list`
  - t_1 `list` $=$ t if and only if $t = t_2$ `list` and $t_1 = t_2$

Lists and types

Lists versus tuples

1.
   - lists must be homogeneous: all elements must have the same type
   - elements in tuples can have different types
   - Examples:

     `1,2,true` is allowed and has type `int*int*bool`

     `[1;2;true]` is not allowed

2.
   - lists can have different length
   - the size of tuples is fixed
   - Examples:

     `[1]` and `[3;7]` are lists of type `int list` but with different length

     `1,2` and `1,2,3` are tuples of incompatible types:

     all tuples of type `int*int` have size 2

     all tuples of type `int*int*int` have size 3

Lists and types

Static semantics of list constructors

- `[]` has type `'a list`
- e_1`::`e_2 is type correct and has type t `list` if and only if
  e_1 is type correct and has type t
  e_2 is type correct and has type t `list`

Polymorphic types

- `'a list` is a polymorphic type or type scheme
- `'a` is a type variable
- meaning: the set of values which is the intersection of t `list` for all t
  that is, `int list`, `bool list`, `(int*bool)list`, `(int -> int)list`, `int list list`, ...
  Remark: such an intersection contains only the empty list!
- polymorphic types are mostly used with functions (see later)
  examples: `'a*'b->'a`   `'a->'b->'a`   `'a list->int`

Lists and types

Static semantics of list constructors

- `[]` has type `'a list`
- e_1`::`e_2 is type correct and has type t `list` if and only if
  e_1 is type correct and has type t
  e_2 is type correct and has type t `list`

Examples

- `2` has type `int`
- `[]` has type `'a list` therefore `int list`
- `2::[]` has type `int list`
- `true` has type `bool`
- `true::2::[]` is not type correct

List concatenation

Syntax

Binary infix operator

```
Exp ::= Exp '@' Exp
```

- left syntactic associativity
- `@` has lower precedence than the `::` constructor

Static semantics

- e_1 `@` e_2 is type correct and has type t `list` if and only if e_1 and e_2 are type correct and have type t `list`

List concatenation

Concatenation is not a constructor!

Example:

- `[1;2;3]` can be uniquely decomposed into its head `1` and tail `[2;3]`

  `h::t=[1;2;3]` if and only if `h=1` and `t=[2;3]`

- `[1;2;3]` cannot be uniquely decomposed into a concatenation

  `[]@[1;2;3]=[1]@[2;3]=[1;2]@[3]=[1;2;3]@[]=[1;2;3]`

List concatenation

Time complexity of list constructor and concatenation

- time complexity of the non-empty list constructor is $O(1)$
- time complexity of concatenation is $O(n)$, with n the length of the left operand

  Beware: if *ls* is a long list, then `42::`*ls* is much faster than *ls*`@[42]`!

  Remark: `list` values in OCaml are implemented as singly linked lists

Concatenation as curried function

Examples

```
# (@)
- : 'a list -> 'a list -> 'a list = <fun>
# [1]@[2;3]=(@) [1] [2;3]
- : bool = true
# [1]@[2;3]@[4]=(@) ((@) [1] [2;3]) [4]
- : bool = true
```

Arithmetic/logic operators as curried functions

Examples

```
# (+);;
- : int -> int -> int = <fun>
# let inc = (+) 1 ;;
val inc : int -> int = <fun>
# ( * );;
- : int -> int -> int = <fun>
# (-);;
- : int -> int -> int = <fun>
# (/);;
- : int -> int -> int = <fun>
# (&&);;
- : bool -> bool -> bool = <fun>
# (||);;
- : bool -> bool -> bool = <fun>
# (<);;
- : 'a -> 'a -> bool = <fun>
# (=);;
- : 'a -> 'a -> bool = <fun>
```

Pattern matching

List patterns: new productions for `Pat`

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

What is pattern matching?

- a powerful mechanism for defining variables/parameters by value decomposition
- all variables in a pattern must be distinct
  - this makes pattern matching more efficient
- patterns are built with constructors, not with operators
  constructors guarantee unique decomposition

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 we apply substitution `x=3,y=5` to `x+y`, then we obtain `3+5`

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 we apply substitution `h=3,t=[5]` to `h`, then we obtain `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 '_' *)
```

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* is the simplest form of pattern
- match with *x* always succeeds, but with a unique case

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

$\texttt{match}$ e $\texttt{with}$ p_1 `->` e_1 `|` ... `|` p_n `->` e_n

Static semantics

- the expression e and all patterns $p_1 \ldots p_n$ must have the same type
- all expressions $e_1 \ldots e_n$ must have the same type

Dynamic semantics

- e is evaluated
- all patterns $p_1 \ldots p_n$ 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 substitution for the 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 exists a unique substitution for the variables in *p*

Counter-example

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

If the argument is `[1;2;3]` what are the values for `l1` and `l2`???
`[]` 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;;
(* the wildcard '_' is used for the second pattern *)
```

Remarks

- the wildcard `'_'` is useful when all arguments match and no variable is needed
- pattern matching with primitive types is seldom used; the conditional expressions and the 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;;
```