Keywords:
types; polymorphism; curried functions; nameless functions; lists; pattern matching; case expressions; list manipulation; tail recursion; accumulators; local bindings.

References:
♦ [MLWP, Chapters 2, 3, & 5]
♦ J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I.


---

Types

Every well-formed ML expression has a type. A type denotes a collection of values. All types are determined statically. Given little of no explicit type information, ML can infer all the types involved with a value or function declaration.

Via a mathematical theorem typically known as Subject Reduction, ML guarantees that the value obtained by evaluating an expression coincides with that of the evaluated expression. Thus, type-correct programs cannot suffer run-time type errors.

Polymorphism

In parametric polymorphism, a function (or datatype) is general enough to work with objects of different types.

A polymorphic type is a type scheme constructed from type variables and basic types (like int, real, char, string, bool, unit) using type constructors (like the product type constructor *, the function type constructor ->, etc.).

Polymorphic types represent families of types; viz. the family of instances obtained by substituting types for type variables.

Examples:
1. Swapping

``` ml
fun swap( x , y ) = ( y , x ) ;
fun int_swap( p:int*int ) = swap p ;
fun real_swap( p:real*real ) = swap p ;
fun unit_swap( p:unit*unit ) = swap p ;
```

"poly"

``` ml
> val ('a , 'b) swap = fn : 'a * 'b -> 'b * 'a
> val int_swap = fn : int * int -> int * int
> val real_swap = fn : real * real -> real * real
> val unit_swap = fn : unit * unit -> unit * unit
```
2. Associating

\[
\text{fun assocLR( } (x, y), z ) = ( x, (y, z) );
\]

\[
\text{fun assocRL( } x, (y, z) ) = ( (x, y), z );
\]

~ "poly"

> val ('a, 'b, 'c) assocLR = fn :
   ('a * 'b) * 'c -> 'a * ('b * 'c)

> val ('a, 'b, 'c) assocRL = fn :
   'a * ('b * 'c) -> (a * 'b) * 'c

### Declaring functions

**Curried functions**

Since an ML function can have only one argument, functions taking more than one argument have so far corresponded to ML functions taking tuples.

However, functions admitting multiple arguments can also be realised by the process of **currying** them, to produce another function that takes each of its arguments in turn returning a function as result.

**Examples:**

1. Ternary multiplication

   \[
   \text{fun termult( } a, b, c )
   = a * b * c : \text{int};
   \]

   \[
   \text{fun curried_termult a b c}
   = \text{termult( } a, b, c );
   \]

   ~ "curry"

   > val termult = fn :
      int * int * int -> int
   > val curried_termult = fn :
      int -> int -> int -> int

   **NB:** Curried functions permit **partial evaluation**:

   \[
   \text{fun mult (x,y) = curried_termult 1 x y ;}
   \]

   \[
   \text{fun double x = curried_termult 1 2 x ;}
   \]

   \[
   \text{fun pow3 x = curried_termult x x x ;}
   \]

   ~

   "curry"

   > val mult = fn : int * int -> int
   > val double = fn : int -> int
   > val pow3 = fn : int -> int

   **NB:** Function application associates to the left, whilst the function-type constructor associates to the right.
2. Composition

fun compose f g x
   = f(g x);

fun uncurried_compose (f, g, x)
   = compose f g x;

"curry"

> val ('a, 'b, 'c) compose = fn :
   ( 'a -> 'b ) -> ( 'c -> 'a ) -> 'c -> 'b
> val ('a, 'b, 'c) uncurried_compose = fn :
   ( ( 'a -> 'b ) * ( 'c -> 'a ) ) * 'c -> 'b

fun q x = p 1 x;
val q2 = q 2;
val qtrue = q true;

> val 'a q = fn : 'a -> int * 'a
> val q2 = (1, 2) : int * int
> val qtrue = (1, true) : int * bool

- val q'2 = q' 2 ;

! Warning: the free type variable 'a has been instantiated to int
> val q'2 = (1, 2) : int * int

- val q'true = q' true ;

! Toplevel input:
! val q'true = q' true ;
!^^^^
! Type clash: expression of type
!   bool
! cannot have type
!   int

Polymorphism

Warning

In ML, a polymorphic type can be instantiated in multiple ways, but all type variables for a given instance must be instantiated as a group.

fun p x y = (x, y);
val p12 = p 1 2 ;
val p1true = p 1 true ;

> val ('a, 'b) p = fn : 'a -> 'b -> 'a * 'b
> val p12 = (1, 2) : int * int
> val p1true = (1, true) : int * bool

- val q'2 = q' 2 ;

! Warning: the free type variable 'a has been instantiated to int
> val q'2 = (1, 2) : int * int

- val q'true = q' true ;

! Toplevel input:
! val q'true = q' true ;
!^^^^
! Type clash: expression of type
!   bool
! cannot have type
!   int
Function values

Most functional languages give *function values* full rights, free of arbitrary restrictions. Like other values, functions may be arguments and results of other functions and may belong to other data structures (pairs, etc.).

Nameless functions

An ML function need not have a name. Indeed, the expression

```
fn x => E
```

is a *function value* with formal argument (or parameter) \(x\) and body \(E\). In particular, the declarations

```
fun myfun x = E ; and val myfun = fn x => E ;
```

are equivalent.

Examples:

1. fun Curry f
   
   ```
   = fn x => fn y => f(x,y) ;
   ```

   fun unCurry f
   
   ```
   = fn( x, y ) => f x y ;
   ```

   val ('a,'b,'c) Curry = fn:
   
   ```
   ('a*'b->'c)->'a->'b->'c
   ```

   val ('a,'b,'c) unCurry = fn:
   
   ```
   ('a->'b->'c)->'a*'b->'c
   ```

2. fun split f
   
   ```
   = ( fn x => ( fn(y,z) => y ) ( f x ) ,
   fn x => ( fn(y,z) => z ) ( f x ) ) ;
   ```

   fun pack( f , g )
   
   ```
   = fn x => ( f x , g x ) ;
   ```

   val ('a,'b,'c) split = fn:
   
   ```
   ('a->'b*'c)->('a->'b)*('a->'c)
   ```

   val ('a,'b,'c) pack = fn:
   
   ```
   ('a->'b)*('a->'c)->'a->'b*'c
   ```

Lists

A list is a finite sequence of elements of the same type.

For instance:

* int list is the type of lists of integers,
* string list is the type of lists of strings,
* int list list is the type of lists whose elements are themselves lists of integers.
Examples

- [1, 2, 4, 2, 1]
- ["a", "b", "b", "a"]
- [[3, 6, 9], [5], [7]]
- []

"p01"

> val it = [1, 2, 4, 2, 1] : int list
> val it = ["a", "b", "b", "a"] : string list
> val it = [[3, 6, 9], [5], [7]] : int list list
> val 'a it = [] : 'a list

Lists are either of two kinds:

Empty:

\[
[] : 'a list
\]

Compound:

\[
h :: t
\]

with \( h \) the head of the list and \( t \) the tail

NB:

- :: is an infix operator:
  - op ::;
  - > val 'a it = fn : 'a * 'a list -> 'a list

* The notation \([e_1, e_2, \ldots, e_n]\) is a shorthand for \(e_1 :: e_2 :: \ldots :: e_n :: []\).

Built-in functions

Head: \(-\text{hd};\)

> val 'a it = fn : 'a list -> 'a

Tail: \(-\text{tl};\)

> val 'a it = fn : 'a list -> 'a list

Append:

- op @;

> val 'a it = fn : 'a list * 'a list -> 'a list

Reverse: \(-\text{rev};\)

> val 'a it = fn : 'a list -> 'a list

nil testing: \(-\text{null};\)

> val 'a it = fn : 'a list -> bool

Length: \(-\text{length};\)

> val 'a it = fn : 'a list -> int
Pattern matching

- The use of data constructors as patterns allows to deconstruct data.
- A pattern match takes place between a pattern (= linear expressions built from variables, constants, and data constructors) and data (built from constants and data constructors).

The variables appearing in a pattern are bound to (or unified with) the corresponding parts of the data object. (Constants require an exact match and _ is used to match anything without producing any binding.)

Examples:
1. \( \text{val } a :: b = [0, 1] \);
   \( \text{val } [x, y] = [0, 1] \);
   \( \text{val } [c, _] = [0, 1] \);
   \( \text{val } [_, _] = [0, 1] \);
   "p02"
   > \text{val } a = 0 : \text{int}
   > \text{val } b = [1] : \text{int list}
   > \text{val } x = 0 : \text{int}
   > \text{val } y = 1 : \text{int}
   > \text{val } c = 0 : \text{int}

2. \( \text{val } [a, 2] = [0, 1] \);
   "p03"
   ! Uncaught exception:
   ! Bind

Non-examples
1. \( \text{val } [a, 2] = [1, y] \);
   "p04"
   ! \text{val } [a, 2] = [1, y] ;
   ! Unbound value identifier: y
Deep patterns

Patterns can be as deep or as shallow as required.

They can match a value or the components that make up that value.

val x = [ ( 1 , ( [ (0 ,"F",false) , (1,"T",true) ] ) ) ] ;
val y :: z = x ;
val ( a , ( (b,c,d) :: e ) ) :: [] = x ;

Using patterns

Only one pattern can be used with val, but fun, fn, and case
expressions can include multiple patterns. They are tried in
order until one is successful.

Examples:

1. fun null0 l = l = [] ;
   fun null1 [] = true
   | null1 _ = false ;
   val null2 = fn [] => true | _ => false ;
> val 'a null0 = fn : 'a list -> bool
> val 'a null1 = fn : 'a list -> bool
> val 'a null2 = fn : 'a list -> bool

fun null3 [] = true ;
null3 [] ;
null3 [1,2,3] ;

> val x = [(1, [(0, "F", false), (1, "T", true)])] :
  (int * (int * string * bool) list) list
> val y = (1, [(0, "F", false), (1, "T", true)]) :
  int * (int * string * bool) list
val z = [] : (int * (int * string * bool) list) list
> val a = 1 : int
val b = 0 : int
val c = "F" : string
val d = false : bool
val e = [(1, "T", true)] : (int * string * bool) list

> val null3 [] = true ;
null3 [] ;
null3 [1,2,3] ;

! Warning: pattern matching is not exhaustive
> val 'a null3 = fn : 'a list -> bool
> val it = true : bool

! Uncaught exception:
! Match
2. The conditional expression

```plaintext
if E then E1 else E2
```
abbreviates the function application

```plaintext
(fn true -> E1 | false -> E2)(E)
```

---

**List manipulation**

**Recursive definitions**

**Examples**

1. - fun length [] = 0
   
   | length (h::t) = 1 + length t ;
   
   > val 'a length = fn : 'a list -> int

2. fun append [] l = l
   
   | append (h::t) l = h :: append t l ;

fun longreverse [] = []
   
   | longreverse (h::t) = append (longreverse t) [h] ;
   
   "p09"

> val 'a append = fn : 'a list -> 'a list -> 'a list
> val 'a longreverse = fn : 'a list -> 'a list

```plaintext
Evaluate longreverse [0,1,2,3,4] !
```

---

**List manipulation**

**Tail recursion**

**Example:**

```plaintext
fun lastelem [ e ] = e
   
   | lastelem (h::t) = lastelem t ;

lastelem [ 0, 1, 2, 3, 4 ] ;
lastelem [ ] ;

"p10"

> val 'a lastelem = fn : 'a list -> 'a
> val it = 4 : int
```

! Uncaught exception:

! Match
List manipulation
Tail recursion

A *tail call* is the last thing that happens in a function.
A function is said to be *tail recursive* (or *iterative*) if the
recursive calls are *tail calls*.

No computation is required after the recursive call returns;
so the call could be replaced with a return.
There is no stack of pending operations.

Simple tail recursive functions are the functional
equivalent of *while* loops.

Tail-call optimisation works with mutually recursive
functions.

---

List manipulation
Accumulators

Some functions (like, for instance, *length*, *reverse*, *etc.*) can
be made tail recursive by using an *accumulator* that gathers
the running total of the computation.

1. ```
   fun addlength [] a = a
       | addlength (h::t) a = addlength t (a+1)
   fun length l = addlength l 0;
   ~
   "p11"
   > val 'a addlength = fn : 'a list -> int -> int
   > val 'a length = fn : 'a list -> int
```

List manipulation
Local bindings

`Local functions` are defined and used inside other functions to
present the desired function to the outside world without
exporting its internal implementation.

Local functions and other values are defined by means of the
`let...in...end` construct.

*Remark:* For patterns P₁, ..., Pₙ, the expressions
```f P₁ => E₁ | ... | f Pₙ => Eₙ```
and
```let fun f(P₁) = E₁ | ... | f(Pₙ) => Eₙ in f end```
have the same meaning, provided that the name f is fresh.
Examples

1. fun reverse l = let
   fun auxrev [] l = l
   | auxrev (h::t) l = auxrev t (h::l);
   in
   auxrev l []
   end;

2. fun fact n = let
   fun accfact n x = if n = 0 then x
       else accfact (n-1) (n*x)
   in
   accfact n 1
   end;

3. fun unzip [] = ( [], [] )
   | unzip ( (h1,h2)::t )
     = let val (t1,t2) = unzip t
        in ( h1::t1 , h2::t2 )
        end;
   fun zip ( h1::tl1 , h2::tl2 ) = (h1,h2) :: zip( tl1,tl2 )
   | zip _ = [];

List manipulation
Library functions

Try the following in ML - load"List"; open List;

Examples:
- List.concat;
> val 'a it = fn : 'a list list -> 'a list
- List.take;
> val 'a it = fn : 'a list * int -> 'a list
- List.drop;
> val 'a it = fn : 'a list * int -> 'a list

Recall that patterns are evaluated in the order that they are written.
- List.find;

> val 'a it = fn :
    ('a -> bool) -> 'a list -> 'a option

- List.partition;

> val 'a it = fn :
    ('a -> bool) -> 'a list -> 'a list * 'a list

Investigate what they do and provide your own implementations!