Chapter 4
Linear Structures: Lists

(Version of 24 September 2004)

4.1. Lists

Definition

A *list* is an element collection with the following properties:

- Homogeneity: all elements are of the *same* type
- Variability: the number of elements is *arbitrary*
- Multiplicity: an element may appear *several* times in a list
- Extensionality: all elements must be *explicitly* given
- Linearity: the internal structure is *linear*

In ML, the word 'list' refers to the concrete realisation of the abstract datatype 'linear list' (see Chapter 6), using the (unary, postfix) *type constructor* list

Examples of ML lists

- [18, 12, ˜5, 3+7] is an integer list (this type is denoted int list)
- [2.0, 5.3 / 3.7, Math.sqrt 27.3] is a real-number list (real list)
- ["Pierre", "Esra"] is a list of strings (string list)
- [(1,"A"), (2,"B")] is a list of integer-string couples ((int $*$ string) list)
- [[4,5], [8], [12,˜3,0]] is a list of integer lists (int list list)
- [even, odd] is a list of int $\longrightarrow$ bool functions ((int $\longrightarrow$ bool) list)
- [12, 34.5] is *not* a list

The empty list

The empty list is denoted `[]` or `[ ]` or `nil`

What is the type of the empty list?!

The empty list must be of the type `int list` and `real list` and `string list` and `int list list` and (`int` $\longrightarrow$ `bool`) `list` and ...

The solution is that `[ ]` is a *polymorphic value*: it belongs to several types!

Type expressions

We use *type variables* to express *polymorphic types*:

The type of `[ ]` is α `list` where α is a type variable, denoting an arbitrary type

In ML, type variables are written `'a` `'b` ...

```
-  [ ];
  val 'a it = [] : 'a list
```

4.2. Basic operations

```
-  null [ ] ;
   val it = true : bool

-  hd [1,2,3] ;
   val it = 1 : int

-  tl [1,2,3] ;
   val it = [2,3] : int list

-  tl [1] ;
   val it = [] : int list

-  tl [ ] ;
   ! Uncaught exception: Empty

-  3 :: [8,2] ;
   val it = [3,8,2] : int list

-  3 :: [ ] ;
   val it = [3] : int list

-  [3] :: [8,2] ;
   ! [3] :: [8,2] ;
   !         ^
   ! Type clash: expression of type int
   !                 cannot have type int list

-  3 :: 8 ;
   ! 3 :: 8 ;
   !      ^
   ! Type clash: expression of type int
   !                 cannot have type int list
```

4.3. Constructors and pattern matching

The expression [3,8] is syntactic sugar for 3 :: (8 :: [ ])

The symbol :: is *not* an ML function,
but a *value constructor*:

- *Composition* of a new object from its parts
- *Decomposition* of an object into its parts

Only constructors can be used in patterns
Another value constructor for lists is [ ]

One can indifferently use the *aggregated form* [3,8,5]
and the *constructed form* 3 :: 8 :: 5 :: [ ]
as they represent the same object!

```
-   1 :: 2 :: [ ] = [1,2] ;
  val it = true : bool
```

In ML, the symbol :: is thus a value constructor that is:

- binary
- infix
- *right*-associating: for instance, 3 :: 8 :: [ ] is 3 :: (8 :: [ ])
- of the functional type $\alpha * \alpha$ list $\longrightarrow \alpha$ list

Pattern matching

Use pattern matching for:

- Decomposing an object into its parts
- Accessing the parts of a constructed object

```
-   val (x::xs) = [3,8,5] ;
  val x = 3 : int
  val xs = [8,5] : int list
-   val (x::xs) = [3] ;
  val x = 3 : int
  val xs = [] : int list
-   val (x::xs) = [ ] ;
  ! Uncaught exception: Bind
```

Example: concatenating two lists (`append.sml`)

```
fun append [ ] ys = ys
 |  append (x::xs) ys = x :: (append xs ys)
```

- The two lines of this function declaration are called *clauses*
- A list concatenation function is actually predefined in ML, namely as the (binary, infix, right-associating) operator `@`

- The patterns `[]` and `(x::xs)` are mutually exclusive
- The pattern `[x]` is equivalent to (`x::[ ]`)

Lists vs. tuples

Tuples: example (3, 8.0, 5>8)

- Fixed size
- Heterogeneous (components of possibly different types)
- Direct access to the components via the #i selectors

Lists: example [3, 8, 5]

- Arbitrary length
- Homogenous (elements of the same type)
- Access to the parts via pattern matching with hd and tl
  that is: sequential access to the elements

Constructed form vs. aggregated form

The aggregated form of lists is mostly used for:

- Arguments
- Results (when displayed by ML)

```
-   [3,4,5] @ [6,7] ;
  val it = [3,4,5,6,7] : int list
```

The constructed form is mostly used in function declarations:

- Decomposition of a list by pattern matching
- Composition of a list

4.4. Polymorphism

function hd X
TYPE: α list $\longrightarrow$ α
PRE: X is not the empty list
POST: the head of X

```
fun hd [ ] = error "hd: empty list"
 |  hd (x::xs) = x
```

```
- hd [1,2] ;
  val it = 1 : int
```

```
- hd [true,false,true] ;
  val it = true : bool
```

function first (a,b)
TYPE: $\alpha * \beta \longrightarrow \alpha$
PRE: (none)
POST: the first component of the pair (a,b)

```
fun first (a,b) = a
```

```
- first ([4,5], true) ;
  val it = [4,5] : int list
```

```
- first (hd, 3.5) ;
  val it = fn : 'a list -> 'a
```

The functions hd and first can be used with arguments of varying types, without changing their names or declarations: *polymorphism*

The type of our error *function*

It must be possible to use **error** in *any* situation:
the type of its result is thus some type variable, say α

function error msg
TYPE: string $\rightarrow$ α
PRE, POST: (none)
SIDE-EFFECT: displays msg to the screen and halts the execution

The type of = *(equality)*

Example: membership of an object in a list `(member.sml)`

function member v X
TYPE (tentatively): $\alpha \rightarrow \alpha$ list $\rightarrow$ bool
PRE: (none)
POST: **true** if v is an element of X
false otherwise

```
fun member v [ ] = false
  | member v (x::xs) = (v=x) orelse member v xs
```

The member function is polymorphic:

- It *can* be used with objects
  where α is the type int, real, bool, (int $*$ bool), int list, ...
- It *cannot* be used with objects
  where α is the type (int $\rightarrow$ int), (int $\rightarrow$ bool), (real $\rightarrow$ real), ...
  because the equality test between two functions
  is *not* computable!

The polymorphism of member must be restricted to the types for which the equality test is computable, that is to the types of objects without functions

These *equality types* are denoted by variables of the form $\alpha^=$ $\beta^=$ $\ldots$, or ''a ''b ... in ML

function member v X
TYPE: $\alpha^= \rightarrow \alpha^=$ list $\rightarrow$ bool
PRE: (none)
POST: **true** if v is an element of X
false otherwise

function x = y
TYPE: $\alpha^= * \alpha^= \rightarrow$ bool
PRE: (none)
POST: **true** if x $=$ y
false otherwise

function x <> y
TYPE: $\alpha^= * \alpha^= \rightarrow$ bool
PRE: (none)
POST: **true** if x $\neq$ y
false otherwise

Example:

```
-   fun member ... ;
  val ''a member = fn : ''a -> ''a list -> bool
```

4.5. Simple operations on lists

Reversal of a list (`reverse.sml`)

Specification

function reverse X
TYPE: α list $\rightarrow$ α list
PRE: (none)
POST: the reverse list of X

Construction with the length of X as variant

Base case: X is [ ] : return [ ]

General case: X is of the form (x::xs) : return reverse xs @ [x]

ML program

```
fun reverse [ ] = [ ]
  | reverse (x::xs) = reverse xs @ [x]
```

The list reversal function is actually predefined, as rev

General schema

For most of the simple operations on lists,
the form of the constructed ML program will be:

```
fun  f [ ] ...  = ...
 |   f (x::xs) ...  = ... (f xs) ...
```

Length of a list (`length.sml`)

function length X
TYPE: α list $\longrightarrow$ int
PRE: (none)
POST: the number of elements of X

```
fun length [ ] = 0
 |  length (x::xs) = 1 + length xs
```

The length function is actually predefined in ML

Product of the elements of a list (`prod.sml`)

function prod X
TYPE: int list $\longrightarrow$ int
PRE: (none)
POST: the product of the elements of X

```
fun prod [ ] = 1
 |  prod (x::xs) = x * prod xs
```

List generator (fromTo.sml)

function fromTo i j
TYPE: int → int → int list
PRE: (none)
POST: [ ] if i $>$ j
  [i , i+1 , ... , j] otherwise

Construction with the length of the interval i...j as variant

```
fun fromTo i j =
   if i > j then [ ]
   else i :: fromTo (i+1) j
```

The fromTo and prod functions now allow the *non*-recursive computation of factorials:

```
fun fact n =
   if n < 0 then error "fact: negative argument"
   else prod (fromTo 1 n)
```

Selections

First elements (`take.sml`)

function take (X,k)
TYPE: α list $*$ int $\rightarrow$ α list
PRE: (none)
POST: [ ] if k $\leq$ 0
X if k $>$ length(X)
the list of the first k elements of X, otherwise

```
fun take ([ ],k) = [ ]
  | take (x::xs,k) =
        if k <= 0 then [ ]
        else x :: take (xs,k-1)
```

Last elements (`drop.sml`)

function drop (X,k)
TYPE: α list $*$ int $\rightarrow$ α list
PRE: (none)
POST: [ ] if k $>$ length(X)
X if k $\leq$ 0
the list X without its first k elements, otherwise

```
fun drop ([ ],k) = [ ]
  | drop (x::xs,k) =
        if k <= 0 then x::xs
        else drop (xs,k-1)
```

Last element (`last.sml`)

function last X
TYPE: α list $\rightarrow$ α
PRE: X is not empty
POST: the last element of X

```
fun  last [ ] = error "last: empty list"
 |   last (x::[ ]) = x
 |   last (x::xs) = last xs
```

The complexity is $O(length(\mathsf{X}))$

k^{th} *Element* (`element.sml`)

function element k X
TYPE: int $\rightarrow$ α list $\rightarrow$ α
PRE: $0 < k \leq$ length(X)
POST: the element at position k of X

```
fun  element k [ ] = error "element: pre-condition violated"
 |   element 1 (x::xs) = x
 |   element k (x::xs) = (* k <> 1 *)
           if k <= 0 then error "element: pre-condition violated"
           else (* k > 1 *) element (k−1) xs
```

Note the necessity of defensive programming
in the general case

4.6. Application: polynomials

A simple representation of polynomials

Example: the polynomial $2x^4 + 5x^3 + x^2 + 3$
can be represented by the list [3,0,1,5,2]

In general: the list $[a_0, a_1, \ldots, a_n]$ with $a_n \neq 0$
represents the polynomial

$$P_n(x) = a_n x^n + \cdots + a_1 x + a_0$$

We assume integer coefficients and natural-number powers

Definition of the poly *type*

```
type poly = int list
```

- poly is a type
- poly is another way of naming the int list type:
  see Chapter 5 of this course
- poly and int list can be used interchangeably

Operations on polynomials

Evaluation of a polynomial (`poly.sml`)

function evalPoly P v
TYPE: poly → int → int
PRE: (none)
POST: P(v)

Hörner schema:

$$P_n(v) = a_n v^n + \cdots + a_1 v + a_0$$
$$P_n(v) = (a_n v^{n-1} + \cdots + a_1)v + a_0$$
$$P_n(v) = ((a_n v + a_{n-1})v + \cdots + a_1)v + a_0$$

```
fun  evalPoly [ ] v = 0
 |   evalPoly (a::p) v = (evalPoly p v) * v + a
```

Addition of polynomials (`poly.sml`)

function addPoly P1 P2
TYPE: poly → poly → poly
PRE: (none)
POST: P1 + P2

```
fun  addPoly p1 [ ] = p1
 |   addPoly [ ] p2 = p2
 |   addPoly (a::p1) (b::p2) = (a+b) :: (addPoly p1 p2)
```

Complexity: $O(n)$, with n the min. of the degrees of P1, P2

Sparse polynomials

What if a lot of coefficients are zero?!
Example: $3x^{27} + 4x^5 + 3x^2$

In the preceding representation:

- High memory consumption
- High run time of the operations (many evaluation steps)

We need a better representation!

Representation of sparse polynomials

Example: the polynomial $3x^{27} + 4x^5 + 3x^2$
can be represented by the list [(2,3), (5,4), (27,3)]

In general: the list $[(k_1, c_1), \ldots, (k_m, c_m)]$
with: $c_i \neq 0 \quad \text{for } 1 \leq i \leq m$
$k_i \geq 0 \quad \text{for } 1 \leq i \leq m$
$k_i < k_{i+1} \quad \text{for } 1 \leq i < m$
represents the polynomial

$$c_m x^{k_m} + \cdots + c_1 x^{k_1}$$

Hence the new ML type:

```
type poly = (int * int) list
```

Operations on (sparse) polynomials

Evaluation of a (sparse) polynomial (`polySparse.sml`)

function evalPoly: the *same* specification!

Observation:

$$3v^{27} + 4v^5 + 3v^2 = (3v^{25} + 4v^3)v^2 + 3v^2$$

$$c_m v^{k_m} + \cdots + c_2 v^{k_2} + c_1 v^{k_1} = (c_m v^{k_m - k_1} + \cdots + c_2 v^{k_2 - k_1}) v^{k_1} + c_1 v^{k_1}$$

Specification of a generalised problem:

function evalPolyAux P v k
TYPE: poly $\longrightarrow$ int $\longrightarrow$ int $\longrightarrow$ int
PRE: P represents $c_m x^{k_m} + \cdots + c_1 x^{k_1}$
$k_1 \geq k$
POST: $c_m v^{k_m - k} + \cdots + c_1 v^{k_1 - k}$, that is $P(v)/v^k$

```
fun expo x n = if n=0 then 1 else x * (expo x (n-1))
local
   fun evalPolyAux [ ] v k = 0
    |  evalPolyAux ((k1,c1)::q) v k =
             let val vexp = expo v (k1-k)
             in (evalPolyAux q v k1) * vexp + c1 * vexp
             end
in
   fun evalPoly P v = evalPolyAux P v 0
end
```

Exercises

- Realise the function adding two sparse polynomials
- Realise the function multiplying two sparse polynomials

Summary: an abstract datatype for polynomials

1. Definition of a new class of objects: the polynomials
2. Specification of abstract operations on these objects: creation, evaluation, addition, ...
3. Choice of a concrete representation in ML (two alternatives were studied here)
4. Implementation of the operations

4.7. Tail recursion and iteration

Length of a list, revisited (length.sml)

function length X
TYPE: α list $\rightarrow$ int
PRE: (none)
POST: the number of elements of X

```
fun length [ ] = 0
  | length (x::xs) = 1 + length xs
```

Time complexity: one traversal of the list

length [5,8,4,3]
$\leadsto$ 1 + length [8,4,3]
$\leadsto$ 1 + (1 + length [4,3])
$\leadsto$ 1 + (1 + (1 + length [3]))
$\leadsto$ 1 + (1 + (1 + (1 + length [ ])))
$\leadsto$ 1 + (1 + (1 + (1 + 0)))
$\leadsto$ 1 + (1 + (1 + 1))
$\leadsto$ 1 + (1 + 2)
$\leadsto$ 1 + 3
$\leadsto$ 4

The recursive call of length is nested in an expression: during the evaluation, *all* the terms of the sum are stored, hence the *memory* consumption for expressions & bindings is proportional to the length of the list!

Now take the following ML program:

```
fun  lengthAux [ ] acc = acc
 |   lengthAux (x::xs) acc = lengthAux xs (acc+1)
```

```
lengthAux [5,8,4,3] 0
⇝ lengthAux [8,4,3] (0+1)
⇝ lengthAux [8,4,3] 1
⇝ lengthAux [4,3] (1+1)
⇝ lengthAux [4,3] 2
⇝ lengthAux [3] (2+1)
⇝ lengthAux [3] 3
⇝ lengthAux [ ] (3+1)
⇝ lengthAux [ ] 4
⇝ 4
```

- *Tail recursion*: recursion is the outermost operation
- Space complexity: *constant* memory consumption for expressions & bindings
- Time complexity: (still) one traversal of the list
- The recursive call "behaves" like *iteration* (see: imperative programming)

One can prove that lengthAux X acc = acc + length(X)
This equality is the post-condition of the lengthAux function!

Questions

- How to obtain a tail-recursive program?
- What is the specification of such a program?
- How to write a program for the initial specification?

By *descending generalisation* of the initial specification!

Important: This technique of tail-recursion introduction is *not* the only way of generalising a specification!

Specification of the generalised problem

function lengthAux X acc
TYPE: α list $\rightarrow$ int $\rightarrow$ int
PRE: (none)
POST: acc $+$ length(X)

Program for the initial problem

```
fun length X = lengthAux X 0
```

Factorial, revisited (fact.sml)

function factAux n acc
TYPE: int $\longrightarrow$ int $\longrightarrow$ int
PRE: n $\geq$ 0
POST: acc $*$ n!

```
local
    fun factAux 0 acc = acc
      | factAux n acc = factAux (n−1) (n*acc)
in
    fun fact n =
        if n < 0 then error "fact: negative argument"
        else factAux n 1
end
```

Exercises

- Specify and construct a tail-recursive program for expo
- Specify and construct a tail-recursive program for reverse
  With the program on page 4.11, for a list of length n, $n+1$ evaluation steps build an expression of n calls to @; this expression requires $\frac{n(n+1)}{2}$ evaluation steps, hence the overall *time* complexity is $O(n^2)$
- Specify and construct a tail-recursive program for fib
  There are 10^9 evaluations of base cases for fib 44, and *very* large expressions are built during its evaluation