Standard ML

국민대학교 컴퓨터공학부
강 승 식

ML overview

함수형, 논리형 언어

BNF스타일 = 논리형

+ 명령형 – 타입이 없는 변수 선언가능 타입이 있는 변수 선언가능

- Developed for theorem provers
- Functional language
  - Garbage collection
- Strong and static typing
  - Powerful type system
  - Parametric polymorphism(ADA generics)
  - Structural equivalence
- Exceptions, advanced module system
- Miscellaneous features
  - Data types: merge of enumerated literals and variant records
  - Pattern matching
  - References

Interactive programming

- Compiler: Standard ML of New Jersey,
  - http://www.smlnj.org/
  - http://smlnj.cs.uchicago.edu/dist/working/110.78/index.html

```
- val k=5;
- k*k*k;
- [1, 2, 3];
- ["hello", "world"];   string list
- 1::[1, 2, 3];   [1,1,2,3] : int list
```

```
- null [1, 2];   false
- null [ ];   true
- hd [1, 2, 3];   첫번째꺼만 lisp은 car이였
- tl [1, 2, 3];   첫번째 제외하고 lisp은 cdr이
- [ ];
```

Simple functions

- Function declaration
  - fun abs x = if x >= 0.0 then x else ~x;

.0 소숫점 꼭 지켜야함 ~ = 마이너스 (−)

- Function expression
  - fn x = if x >= 0.0 then x else ~x;
  - fn is like *lambda* in Scheme/Lisp

SML file in “foo.sml”

```
fun double (x:int):int = 2 * x;
fun square (x:int):int = x * x;     square(4); = val it = 16 : int
fun power (x:int,y:int):int =
        if (y=0) then 1 else x * power (x,y-1);

- use "foo.sml";
[opening foo.sml]
val double = fn : int -> int
val square = fn : int -> int
val power = fn : int * int -> int
val it = () : unit
```

Simple functions

```
- fun fac : (fn: int -> int) 0 = 1
          | fac n = n * fac (n-1);
```

```
- fun haha 0 = 1
= | haha n = n * haha(n-1);
```

```
fun fact x = if x <= 1 then 1 else x * fact(x-1);
```

ML이란 언어는 두가지를 다 지원한다.

```
- fun fac 0 = 1
          | fac n = n * fac(n-1);
```

Functions: list의 길이

- Function length

```
- fun length xs =
    if null xs
    then 0
    else 1 + length (tl xs);
```

- Pattern-match style

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

Type inference and Polymorphism

- No need to specify types
- With type inference
  - A function is polymorphic
  - No need to "instantiate" a polymorphic function when it is applied

Multiple arguments: tuple, currying(curried style)

- Append: Tuple argument

```
- fun append1 ([ ],      ys) = ys
    | append1 (x :: xs, ys) = x :: append1 (xs, ys);

- append1 ([1, 2, 3], [4, 5]);
```

- "Currying" is named after Haskell Curry

```
- fun append2 [ ]       ys = ys
    | append2 (x :: xs) ys = x :: (append2 xs ys);

- append2 [1, 2, 3] [4, 5];
- val app123 = append2 [1, 2, 3];
- app123 [4, 5];
```

More partial application

- fun appTo45 xs = append2 xs [4, 5];
- val appTo45 = flip append2 [4, 5];

- *flip* is a function which takes a curried function and "flips" its two arguments.

Passing functions

```
- fun exists pred [ ]          = false
    | exists pred (x :: xs) = pred x orelse
                              exists pred xs;
-  exists (fn i => i = 1) [2, 3, 4];
-  val hasOne = exists (fn i => i = 1);
-  hasOne [3, 1, 2];
```

pred is a predicate : a function that returns a boolean

exists checks whether pred is true for any member of the list

Apply functionals

```
fun all pred [ ]      = true
  | all pred (x::xs) = pred x andalso all pred xs;

fun filter pred [ ]      = [ ]
  | filter pred (x::xs) = if pred x
                          then x :: filter pred xs
                          else filter pred xs;
```

$$\texttt{all} : (\alpha \to \texttt{bool}) \to \alpha\,\texttt{list} \to \texttt{bool}$$

$$\texttt{filter} : (\alpha \to \texttt{bool}) \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$$

Block structure and nesting

```
(* standard Newton-Raphson *)
fun findroot (a, x, acc) =
   let val nextx = (a / x + x) / 2.0
       (* nextx is the next approximation *)
   in
       if abs (x - nextx) < acc * x
       then nextx
       else findroot (a, nextx, acc)
   end;
```

Quick sort

```
fun qSort op< []        = []
  | qSort op< [x]       = [x]
  | qSort op< (a::bs) =
    let fun partition (left, right, []) =
              (left, right)   (* done partitioning *)
          | partition (left, right, x::xs) =
              (* put x to left or right *)
              if x < a
              then partition (x::left, right, xs)
              else partition (left, x::right, xs)
        val (left, right) = partition ([], [a], bs)
    in
        qSort op< left @ qSort op< right
    end;
```

$$\texttt{qSort} : (\alpha * \alpha \rightarrow \texttt{bool}) \rightarrow \alpha\,\texttt{list} \rightarrow \alpha\,\texttt{list}$$

Another variant of merge sort

```
fun qSort op< []        = []
  | qSort op< [x]       = [x]
  | qSort op< (a::bs)  =
    let fun deposit (x, (left, right)) =
            if x < a
            then (x::left, right)
            else (left, x::right)
        val (left, right) = foldr deposit ([], [a]) bs
    in
        qSort op< left @ qSort op< right
    end;
```

in end 사이는 실행파트

$$\mathtt{qSort} : (\alpha * \alpha \rightarrow \mathtt{bool}) \rightarrow \alpha\,\mathtt{list} \rightarrow \alpha\,\mathtt{list}$$

The type system

- Primitive types
  - Bool, int, char, real, string, unit
- Constructors
  - List, array, product(tuple), function, record
- Structural equivalence (except for data types)
- An expression has a corresponding type expression

ML records

- A type declaration
  - type vec = { x: real, y: real };
- A variable declaration
  - val v = { x=2.3, y=4.5 };
- Field selection
  - #x v;
- Pattern matching in a function
  - fun dist {x, y} =

    sqrt (pow (x, 2.0) + pow(y, 2.0));

Datatype example

- A datatype declaration
  - defines a new type that is not equivalent to any other type
  - Introduces data constructors

```
datatype tree = Leaf of int
        | Node of tree * tree;
```

Leaf and Node are data constructors

Leaf : int → tree

Node : tree * tree → tree

Pattern matching

```
fun sum (Leaf t)           = t
  | sum (Node (t1, t2)) = sum t1 + sum t2;

fun flatten (Leaf t)           = [t]
  | flatten (Node (t1, t2)) =
    flatten t1 @ flatten t2;
```

$$\mathtt{flatten} : \mathtt{tree} \longrightarrow \mathtt{int\ list}$$

Parameterized datatypes

```
datatype 'a gentree =
    Leaf of 'a
  | Node of 'a gentree * 'a gentree;

val names = Node (Leaf "this", Leaf "that")
```

names : string gentree

The rules of pattern matching

Pattern elements:

- integer literals: `4`, `19`
- character literals: `#'a'`
- string literals: `"hello"`
- data constructors: `Node ($\cdots$)`
  - depending on type, may have arguments, which would also be patterns
- variables: `x`, `ys`
- wildcard: `_`

Convention is to capitalize data constructors, and start variables with lower-case.

More rules of pattern matching

Special forms:

- `()`, `{}` – the unit value
- `[]` – empty list
- `[p1, p2, $\cdots$, pn]`
  means `(p1 :: (p2 :: $\cdots$ (pn :: [])$\cdots$))`
- `(p1, p2, $\cdots$, pn)` – a tuple
- `{field1, field2, $\cdots$ fieldn}` – a record
- `{field1, field2, $\cdots$ fieldn, ...}`
  – a partially specified record
- `v as p`
  – `v` is a name for the entire pattern `p`

Common idiom: option

`option` is a built-in datatype:

```
datatype 'a option = NONE | SOME of 'a;
```

Defining a simple lookup function:

```
fun lookup eq key []                = NONE
  | lookup eq key ((k,v)::kvs) =
    if eq (key, k)
    then SOME v
    else lookup eq key kvs;
```

Is the type of `lookup`:

$$(\alpha * \alpha \rightarrow \texttt{bool}) \rightarrow \alpha \rightarrow (\alpha * \beta)\,\texttt{list} \rightarrow \beta\,\texttt{option}?$$

Another lookup function

We don't need to pass two arguments when one will do:

```
fun lookup _            []              = NONE
  | lookup checkKey ((k,v)::kvs) =
    if checkKey k
    then SOME v
    else lookup checkKey kvs;
```

The type of this lookup:

$$(\alpha \rightarrow \texttt{bool}) \rightarrow (\alpha * \beta)\,\texttt{list} \rightarrow \beta\,\texttt{option}$$

Useful library functions

- $\texttt{map} : (\alpha \rightarrow \beta) \rightarrow \alpha\,\texttt{list} \rightarrow \beta\,\texttt{list}$

```
map (fn i => i + 1) [7, 15, 3]
```
$\Longrightarrow$ `[8, 16, 4]`

- $\texttt{foldl} : (\alpha * \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\,\texttt{list} \rightarrow \beta$

```
foldl (fn (a,b) => "(" ^ a ^ "+" ^ b ^ ")")
      "0"  ["1", "2", "3"]
```
$\Longrightarrow$ `"(3+(2+(1+0)))"`

- $\texttt{foldr} : (\alpha * \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\,\texttt{list} \rightarrow \beta$

```
foldr (fn (a,b) => "(" ^ a ^ "+" ^ b ^ ")")
      "0"  ["1", "2", "3"]
```
$\Longrightarrow$ `"(1+(2+(3+0)))"`

- $\texttt{filter} : (\alpha \rightarrow \texttt{bool}) \rightarrow \alpha\,\texttt{list} \rightarrow \alpha\,\texttt{list}$

Overloading

Ad hoc overloading interferes with type inference:

```
fun plus x y = x + y;
```

Operator '+' is overloaded, but types cannot be resolved from context (defaults to int).

We can use explicit typing to select interpretation:

```
fun mix1 (x, y, z) = x * y + z : real;
fun mix2 (x: real, y, z) = x * y + z;
```

Parametric polymorphism vs. generics

- a function whose type expression has type variables applies to an infinite set of types
- equality of type expressions means structural not name equivalence
- all applications of a polymorphic function use the same body: no need to instantiate

```
let val ints = [1, 2, 3];
    val strs = ["this", "that"];
in
    len ints +   (* int    list -> int *)
    len strs     (* string list -> int *)
end;
```

ML signature

An ML *signature* specifies an interface for a module.

```
signature STACKS =
sig
    type stack
    exception Underflow
    val empty : stack
    val push : char * stack -> stack
    val pop  : stack -> char * stack
    val isEmpty : stack -> bool
end;
```

ML structure

```
structure Stacks : STACKS =
struct
    type stack = char list
    exception Underflow
    val empty = [ ]
    val push = op::
    fun pop (c::cs) = (c, cs)
      | pop []      = raise Underflow
    fun isEmpty [] = true
      | isEmpty _  = false
end;
```