# Programming Languages

ML

CSCI-GA.2110-001

Summer 2011

# ML overview

- originally developed for use in writing theorem provers
- functional: functions are first-class values
- garbage collection
- strict
- strong and static typing; powerful type system
  - parametric polymorphism
  - structural equivalence
  - all with type inference!
- advanced module system
- exceptions
- miscellaneous features:
  - datatypes (merge of enumerated literals and variant records)
  - pattern matching
  - `ref` type constructor (like "const pointers" ("not pointers to const"))

# A sample SML/NJ interactive session

| | |
|---|---|
| `- val k = 5;` | user input |
| *val k = 5 : int* | system response |
| `- k * k * k;` | |
| *val it = 125 : int* | 'it'– last computation |
| `- [1, 2, 3];` | |
| *val it = [1,2,3] : int list* | |
| `- ["hello", "world"];` | |
| *val it = ["hello","world"] : string list* | |
| `- 1 :: [ 2, 3 ];` | |
| *val it = [1,2,3] : int list* `- [ 1, "hello"];` | |
| *error* | |

# Operations on lists

```
- null [1, 2];
val it = false : bool
- null [ ];
val it = true : bool
- hd [1, 2, 3];
val it = 1 : int
- tl [1, 2, 3];
val it = [ 2, 3 ] : int list
- [ ];
val it = [ ] : 'a list
```

this list is polymorphic

# Simple functions

A function *declaration*:

```
- fun abs x = if x >= 0.0 then x else ~x
val abs = fn : real -> real
```

A function *expression*:

```
- fn x => if x >= 0.0 then x else ~x
val it = fn : real -> real
```

# Functions, II

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

*val length = fn : 'a list -> int*

`'a` denotes a type variable; `length` can be applied to lists of *any* element type

The same function, written in pattern-matching style:

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

*val length = fn : 'a list -> int*

# Type inference and polymorphism

Advantages of type inference and polymorphism:

- frees you from having to write types.
  A type can be more complex than the expression whose type it is, e.g., `flip`
- with type inference, you get polymorphism for free:
  - no need to specify that a function is polymorphic
  - no need to "instantiate" a polymorphic function when it is applied

# Multiple arguments?

- All functions in ML take exactly one argument
- If a function needs multiple arguments, we can

1. pass a tuple:
   ```
   - (53, "hello");  (* a tuple *)
   val it = (53, "hello") : int * string
   ```
   We can also use tuples to return multiple results.
2. use currying (named after Haskell Curry, a logician)

# The tuple solution

Another function; takes two lists and returns their concatenation

```
- fun append1 ([ ],    ys) = ys
    | append1 (x::xs, ys) = x :: append1 (xs, ys);
val append1 = fn: 'a list * 'a list -> 'a list

- append1 ([1,2,3], [8,9]);
val it = [1,2,3,8,9] : int list
```

# Currying

The same function, written in curried style:

```
- fun append2 [ ]      ys = ys
    | append2 (x::xs) ys = x :: (append2 xs ys);
val append2 = fn: 'a list -> 'a list -> 'a list

- append2 [1,2,3] [8,9];
val it = [1,2,3,8,9] : int list

- val app123 = append2 [1,2,3];
val app123 = fn : int list -> int list

- app123 [8,9];
val it = [1,2,3,8,9] : int list
```

# More partial application

But what if we want to provide the other argument instead, i.e., append `[8,9]` to its argument?

- here is one way: (the Ada/C/C++/Java way)

```
fun appTo89 xs = append2 xs [8,9]
```

- here is another: (using a higher-order function)

```
val appTo89 = flip append2 [8,9]
```

`flip` is a function which takes a curried function `f` and returns a function that works like `f` but takes its arguments in the reverse order.
In other words, it "flips" `f`'s two arguments.
We define it on the next slide...

# Type inference example

```
fun flip f y x = f x y
```

The type of `flip` is $(\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$. Why?

- Consider (`f x`). `f` is a function; its parameter must have the same type as `x`.

  `f` : $A \to B$ `x` : $A$ `(f x)` : $B$

- Now consider (`f x y`). Because function application is left-associative, `f x y` $\equiv$ `(f x) y`. Therefore, `(f x)` must be a function, and its parameter must have the same type as `y`:

  `(f x)` : $C \to D$ `y` : $C$ `(f x y)` : $D$

- Note that $B$ must be the same as $C \to D$. We say that $B$ must *unify* with $C \to D$.
- The return type of `flip` is whatever the type of `f x y` is. After renaming the types, we have the type given at the top.

# Type rules

The type system is defined in terms of inference rules. For example, here is the rule for variables:

$$\frac{(x : \tau) \in E}{E \vdash x : \tau}$$

and the one for function calls:

$$\frac{E \vdash e_1 : \tau' \to \tau \quad E \vdash e_2 : \tau'}{E \vdash e_1\ e_2 : \tau}$$

and here is the rule for `if` expressions:

$$\frac{E \vdash e : \texttt{bool} \quad E \vdash e_1 : \tau \quad E \vdash e_2 : \tau}{E \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

# Passing functions

```
- fun exists pred [ ]      = false
    | exists pred (x::xs) = pred x orelse
                            exists pred xs;
val exists = fn : ('a -> bool) -> 'a list -> bool
```

- `pred` is a predicate : a function that returns a boolean
- `exists` checks whether `pred` returns true for any member of the list

```
- exists (fn i => i = 1) [2, 3, 4];
val it = false : bool
```

# Applying functionals

```
- exists (fn i => i = 1) [2, 3, 4];
val it = false : bool
```

Now partially apply `exists`:

```
- val hasOne = exists (fn i => i = 1);
val hasOne = fn : int list -> bool
- hasOne [3,2,1];
val it = true : bool
```

# Functionals 2

```
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

`let` provides local scope:

```
(* 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
```

# A classic in functional form: mergesort

```
fun mrgSort op< []       = []
  | mrgSort op< [x]      = [x]
  | mrgSort 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
        mrgSort op< left @ mrgSort op< right
    end
```

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

# Another variant of mergesort

```
fun mrgSort op< []        = []
  | mrgSort op< [x]       = [x]
  | mrgSort 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
        mrgSort op< left @ mrgSort op< right
    end
```

$$\mathtt{mrgSort} : (\alpha * \alpha \to \mathtt{bool}) \to \alpha\,\mathtt{list} \to \alpha\,\mathtt{list}$$

# The type system

- primitive types: `bool`, `int`, `char`, `real`, `string`, `unit`
- constructors: `list`, array, product (tuple), function, record
- "datatypes": a way to make new types
- structural equivalence (except for datatypes)
  - as opposed to name equivalence in e.g., Ada
- an expression has a corresponding type expression
- the interpreter builds the type expression for each input
- type checking requires that type of functions' parameters match the type of their arguments, and that the type of the context matches the the type of the function's result

# ML records

Records in ML obey structural equivalence (unlike records in many other languages).

A type declaration: *only needed if you want to refer to this type by name*

```
type vec = { x : real, y : real }
```

A variable declaration:

```
val v = { x = 2.3, y = 4.1 }
```

Field selection:

```
#x v
```

Pattern matching in a function:

```
fun dist {x,y} =
    sqrt (pow (x, 2.0) + pow (y, 2.0))
```

# Datatypes

A `datatype` declaration:

- defines a new type *that is not equivalent to any other type* (name equivalence)
- introduces *data constructors*
  - *data constructors* can be used in patterns
  - they are also values themselves

# Datatype example

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

`Leaf` and `Node` are *data constructors*:

- `Leaf : int` $\rightarrow$ `tree`
- `Node : tree * tree` $\rightarrow$ `tree`

We can define functions by pattern matching:

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

# Parameterized datatypes

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

$$\texttt{flatten} : \texttt{tree} \to \texttt{int list}$$

---

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

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

$$\texttt{names} : \texttt{string gentree}$$

# The rules of pattern matching

Pattern elements:

- integer literals: `4`, `19`
- character literals: `#'a'`
- string literals: `"hello"`
- data constructors: `Node (···)`
  - 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, ⋯, pn]`
  means `(p1 :: (p2 :: ⋯ (pn :: [])⋯))`
- `(p1, p2, ⋯, pn)` – a tuple
- `{field1, field2, ⋯ fieldn}` – a record
- `{field1, field2, ⋯ 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`:

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

No! It's slightly more general:

$$(\alpha_1 * \alpha_2 \to \texttt{bool}) \to \alpha_1 \to (\alpha_2 * \beta)\,\texttt{list} \to \beta\,\texttt{option}$$

# 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:

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

# Useful library functions

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

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

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

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

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

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

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

# 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
```