Programming Languages

ML

Spring 2006
- originally developed as for use in writing theorem provers
- functional: functions are first-class values
- garbage collection
- strict
- strong and static typing; powerful type system
  - parametric polymorphism (somewhat like Ada generics)
  - structural equivalence
  - all with type inference!
- advanced module system
- exceptions
- miscellaneous features:
  - datatypes (sort of a combination of enumerated literals and variant records)
  - pattern matching
  - references (like “const pointers”)
A sample SML/NJ interactive session

- `val k = 5;`
  
  `val k = 5 : int`
  
  user input  
  
  system response  

- `k * k * k;`
  
  `val it = 125 : int`
  
  ‘it’ denotes the 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`
- 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
A function *declaration*:

```ml
- fun abs x = if x >= 0.0 then x else -x

val abs = fn : real -> real
```

A function *expression*:

```ml
- fn x => if x >= 0.0 then x else -x

val it = fn : real -> real
```
- fun len xs = if null xs
  then 0
  else 1 + len (tl xs);

val len = fn : 'a list -> int

'a denotes a type variable; len can be applied to lists of any element type

The same function, written in pattern-matching style:

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

val len = fn : 'a list -> int
Why are type inference and polymorphism good for you?

- 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
All functions in ML take exactly one argument

If a function needs multiple arguments, we can

1. use a tuple:
   - (53, "hello"); (*a tuple *)
   ```ml
   val it = (53, "hello") : int * string
   ```
   We can also use tuples to return multiple results.

2. use currying (named after Haskell Curry, a logician)
Another function; takes two lists and yields 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], [10,20]);
val it = [1,2,3,10,20] : int list
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`
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)
  
  \[
  \text{fun appTo89 } xs = \text{append2 } xs \ [8,9]
  \]

- here is another:
  
  \[
  \text{val appTo89 = flip append2 } [8,9]
  \]

\text{flip} is a function which takes a curried function and “flips” its two arguments.
- 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** is 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
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

all : (\alpha \to \text{bool}) \to \alpha \text{list} \to \text{bool}

filter : (\alpha \to \text{bool}) \to \alpha \text{list} \to \alpha \text{list}
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:

\[
\frac{E \vdash e_1 : \tau' \rightarrow \tau \quad E \vdash e_2 : \tau'}{E \vdash e_1 \ e_2 : \tau}
\]

and here is the rule for \texttt{if} expressions:

\[
\frac{E \vdash e : \text{bool} \quad E \vdash e_1 : \tau \quad E \vdash e_2 : \tau}{E \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}
\]
fun flip f y x = f x y

The type of \texttt{flip} is $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \beta \rightarrow \alpha \rightarrow \gamma$. Why?

- Consider $(f \ x)$. $f$ is a function; its argument has the same type as $x$.

$$f : A \rightarrow B \quad x : A \quad (f \ x) : B$$

- 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 argument must have the same type as $y$:

$$(f \ x) : C \rightarrow D \quad y : C \quad (f \ x \ y) : D$$

- Note that $B$ must be the same as $C \rightarrow D$.
- The return type of \texttt{flip} is whatever the type of $f \ x \ y$ is. After renaming the types, we have the type given at the top.
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
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

mrgSort : (α * α → bool) → α list → α list
fun mrgSort op< [] = []
  | mrgSort op< [x] = [x]
  | mrgSort op< (a::bs) =
    let fun deposit (x, (left, right)) =
      if a < x
        then (x::left, right)
        else (left, x::right)
    val (left, right) = foldr deposit ([], [a]) bs
    in
      mrgSort op< left @ mrgSort op< right
    end

mrgSort : (α * α → bool) → α list → α 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 expression of functions and their arguments match, and that type expression of the context matches the result of a function
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*

\[
\text{type vec} = \{ \ x : \text{real}, \ y : \text{real} \ \}
\]

A variable declaration:

\[
\text{val v} = \{ \ x = 2.3, \ y = 4.1 \ \}
\]

Field selection:

\[
\#x \ v
\]

Pattern matching in a function:

\[
\text{fun dist \{x,y\} = sqrt (pow (x, 2.0) + pow (y, 2.0))}
\]
Datatypes

A **datatype** declaration:

- defines a new type *that is not equivalent to any other type*
- introduces *data constructors*
  - *data constructors* can be used in patterns
  - they are also values themselves
datatype tree = Leaf of int
  | Node of tree * tree

Leaf and Node are data constructors:

- Leaf : int → tree
- Node : tree * tree → tree

We can define functions by pattern:

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  

flatten : tree → int list

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

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

names : string gentree
Pattern elements:

- integer literals: 4, 19, etc.
- data constructors: Node (...)
  - depending on type, may have arguments, which would also be patterns
- variables: \( x, y \)
- wildcard: _

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

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

Defining a simple lookup function:

```haskell
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**:

```
(α * α → bool) → α → (α * β)list → β option?
```

No! It’s slightly more general:

```
(α₁ * α₂ → bool) → α₁ → (α₂ * β)list → β option
```
We don’t need to pass two arguments when one will do:

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

The type of this lookup:

\[(\alpha \to \text{bool}) \to (\alpha \times \beta) \text{list} \to \beta \text{ option}\]
Useful library functions

- **map**: \((\alpha \to \beta) \to \alpha \text{list} \to \beta \text{list}\)
  
  \[
  \text{map} \ (\text{fn} \ i \Rightarrow i + 1) \ [7, 15, 3] \\
  \implies [8, 15, 4]
  \]

- **foldl**: \((\alpha \times \beta \to \beta) \to \beta \to \alpha \text{list} \to \beta\)
  
  \[
  \text{foldl} \ (\text{fn} \ (a,b) \Rightarrow "(" \ ^\ a \ ^\ "+" \ ^\ b \ ^\ ")")\) \ "0" \ ["1", "2", "3"] \\
  \implies "(3+(2+(1+0)))"
  \]

- **foldr**: \((\alpha \times \beta \to \beta) \to \beta \to \alpha \text{list} \to \beta\)
  
  \[
  \text{foldr} \ (\text{fn} \ (a,b) \Rightarrow "(" \ ^\ a \ ^\ "+" \ ^\ b \ ^\ ")")\) \ "0" \ ["1", "2", "3"] \\
  \implies "(1+(2+(3+0)))"
  \]

- **filter**: \((\alpha \to \text{bool}) \to \alpha \text{list} \to \alpha \text{list}\)
Ad hoc overloading interferes with type inference:

```haskell
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:

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

```ml
let val ints = [1, 2, 3];
val strs = ["this", "that"];
in
  len ints + (* int list -> int *)
  len strs  (* string list -> int *)
end;
```
An ML signature specifies an interface for a module.

```ml
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
```
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