Review

Last week

• Types

Outline

• ML

ML overview

• originally developed for use in writing theorem provers
• functional: functions are first-class values
• garbage collection
• 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 (merge of enumerated literals and variant records)
  – pattern matching
  – references (like “const pointers”)
A sample SML/NJ interactive session

- `val k = 5;`  
  user input

- `val k = 5 : int`  
  system response

- `k * k * k;`  
  system response

  `val it = 125 : int`  
  'it' denotes the last computation

- `[1, 2, 3];`  
  system response

  `val it = [1,2,3] : int list`

- `null [1, 2];`  
  system response

  `val it = false : bool`

- `val it = true : bool`

- `["hello", "world"];`  
  system response

  `val it = ["hello","world"] : string list`

- `1 :: [ 2, 3 ];`  
  system response

  `val it = [1,2,3] : int list`

Operations on lists

- `null [ ];`  
  system response

  `val it = true : bool`

- `hd [1, 2, 3];`  
  system response

  `val it = 1 : int`

- `tl [1, 2, 3];`  
  system response

  `val it = [ 2, 3 ] : int list`

- `[ ];`  
  system response

  `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`

`fn` is like `lambda` in SCHEME.

Functions

- `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:
     ```
     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 yields their concatenation

- `fun append1 (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 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)
  \[
  \text{fun } \text{appTo89 } xs = \text{append2 } xs \ [8,9];
  \]
- here is another: (using a higher-order function)
  \[
  \text{val } \text{appTo89} = \text{flip append2 } [8,9];
  \]

\text{flip} is a function which takes a curried function and "flips" its two arguments. We define it on the next slide...

Type inference example

\[
\text{fun } \text{flip } f y x = f x y
\]

The type of \text{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 \(y\):
  \[
  (f \ x) : C \rightarrow D
  \]
  \[
  y : C \quad (f \ x \ y) : D
  \]
- Note that \(B\) must be the same as \(C \rightarrow D\). We say that \(B\) must \text{unify} with \(C \rightarrow D\).
- The return type of \text{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:

\[
\left( (x : \tau) \in E \right) \quad E \vdash x : \tau
\]

and the one for function calls:

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

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

\[
E \vdash e : \text{bool} \quad E \vdash e_1 : \tau \quad E \vdash e_2 : \tau \quad \frac{}{E \vdash \text{if } e \ \text{then } e_1 \ \text{else } e_2 : \tau}
\]

Passing functions

- \text{fun exists pred [ ] = false}
  \[
  \mid \text{exists pred (x::xs) = pred x orelse}
  \]
  \[
  \text{exists pred xs;}
  \]
- \text{val exists = fn : ('a -> bool) -> 'a list -> bool}

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

- \text{exists (fn i => i = 1) [2, 3, 4];}
- \text{val it = false : bool}
### Applying functionals

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

Now partially apply exists:

- \[
  \text{val hasOne = exists (fn i => i = 1);} \]
  
  \[
  \text{val hasOne = fn : int list \to bool}
  \]
  
  \[
  \text{val it = true : bool}
  \]
Another variant of mergesort

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;

qSort : (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 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
  (like name equivalence)
- introduces data constructors
  - data constructors can be used in patterns
  - they are also values themselves
**Datatype example**

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

Leaf and Node are *data constructors*:

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

**Pattern Matching**

We can define functions by pattern matching:

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

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

`flatten : tree → int list`

**Parameterized datatypes**

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

```ocaml
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 (...)`
  - 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:

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

Defining a simple lookup function:

```plaintext
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`:

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

Another lookup function

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

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

The type of this lookup:

```plaintext
(α → bool) → (α * β) list → β option
```
Useful library functions

- **map**: \((\alpha \rightarrow \beta) \rightarrow \alpha\text{ list} \rightarrow \beta\text{ list}\)

  \[
  \text{map } (\text{fn } i \Rightarrow i + 1) \ [7, 15, 3] \\
  \Rightarrow [8, 16, 4]
  \]

- **foldl**: \((\alpha \times \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\text{ list} \rightarrow \beta\)

  \[
  \text{foldl } (\text{fn } (a, b) \Rightarrow "(" \ ^\wedge\ a\ ^\wedge\ "+"\ ^\wedge\ b\ ^\wedge\ ")") \\
  "0" \ ["1", "2", "3"] \\
  \Rightarrow "(3+(2+(1+0)) )"
  \]

- **foldr**: \((\alpha \times \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\text{ list} \rightarrow \beta\)

  \[
  \text{foldr } (\text{fn } (a, b) \Rightarrow "(" \ ^\wedge\ a\ ^\wedge\ "+"\ ^\wedge\ b\ ^\wedge\ ")") \\
  "0" \ ["1", "2", "3"] \\
  \Rightarrow "(1+(2+(3+0)) )"
  \]

- **filter**: \((\alpha \rightarrow \text{bool}) \rightarrow \alpha\text{ list} \rightarrow \alpha\text{ list}\)

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

```ml
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.

```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;
```
ML structure

```ml
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;
```