ML

Features

- Functional Language
- Garbage collected
- Applicative order evaluation
- Polymorphism
- Pattern Matching
- Abstract datatypes
- Strongly Typed
- Type inference
- Advanced module system

Tuples & Lists

- Tuple:
  - List of 2 or more expressions of any type (heterogeneous), comma separated and surrounding them with round parentheses.
  - Example: val t=(4, 5.0,”hello world”)
  - Function arguments when there are more than 1 parameter, tuples are used.
  - ML response to tuples is formation of product type using (*). Do not confuse with * with multiplication. A product type is formed from two or more types T1, T2, … Tn by putting * between them like T1 * T2 * ..*Tn. Values of this type are tuples with k components, the first if which is of type T1, second is of type T2 and so on.
  - Example: (1, 3.0, “Hi”) is of type int * real * string

Tuples & Lists (Cont)

- Lists:
  - Elements are of the same type (homogenous). Comma separated, surrounded by square brackets.
  - Example:
    - [1,2,3]
      - ML response: *val it=[1,2,3] : int list*
    - [];
      - ML response: *val it = [] : 'a list*
  - Head and tail of the list: Except for the empty list, every list has a head (First element of the list) and tail (List of all elements but the first). Similar to car and cdr in Scheme.

- Concatenation (@): Two lists can be concatenated using @ operator.
- Cons: The cons operator in ML is represented as (::), takes an element (head) and a list of elements of the same type as the head and produces a single list.
  - Example: 2::[3,4]
- Empty list: The type of an empty list is *'a list* i.e. a list if elements of any one type. Identifiers beginning with ' (quote mark) denote types. *'a list* is ML's ways of saying "any-type list"

Type inference

- ML has a strong type system, but types don’t need to be declared by the programmer
- Interpreter uses a type inference system to deduce the types of functions and variables
- Things to remember:
- The operator + can take either integer or real arguments but not both. Thus, both operands must be of same type.
- Division / applies only to reals.
- If-then-else:
  - The expression following if must have a Boolean type.
  - The expression following then and else can be of any one type but they must be of the same type.

How ML deduces types:

1. Types of operands and result of arithmetic operations must all agree.
2. When arithmetic comparison is made, you can be sure that the operands are of the same type and the result is Boolean.
3. In a conditional expression, the subexpressions following the then and else must be of the same type.
4. If a variable or expression used as an argument of a function is of a known type, then the corresponding parameter of the function must be of that type.

How ML deduces types:

5. Return type of same function must be consistent (e.g. in pattern matching, or if the same function is called multiple times.)
6. If there is no way to determine the type of a particular use of an overloaded operator exists, then the type of that operator is defined to be the default for that operator, normally integer.

An operator is overloaded if it can apply to 2 or more different types. For example +.

7. Types that cannot be specified are given type variables.

Polymorphism

- While ML is strictly typed, it is possible to write functions such that one or more parameters are not restricted to a single type. The types of these parameters are assigned type variables by the type-checker.
- Ex. - fun nonNegative f a = if f a > 0 then f a else 0;
  val nonNegative = fn : ('a -> int) -> 'a -> int
- This is an example of polymorphism, the use of a single operation with multiple types. Datatypes can also be polymorphic.
- The above function will work correctly with different types of inputs:
- - nonNegative length [0, 1, 2];
  val it = 3 : int
  - nonNegative abs ~4;
  val it = 4 : int

Pattern Matching

- ML makes heavy use of pattern matching to deal with different cases of inputs.
- Pattern matching is the standard way to write recursive functions
- Example:

```
- fun  fib 0 = 1
|       fib 1 = 1
|       fib n = fib (n - 1) + fib (n - 2);
val fib = fn : int -> int
```

- This is especially common when dealing with lists, records, datatypes
- Constants will only match their value, while variables will match any value and bind a name to it.

Pattern Matching

- This is especially common when dealing with lists, records, datatypes.

```
- fun summap _ [] = 0
- |   summap f (x::xs) = f x + summap f xs;
val summap = fn : ('a -> int) -> 'a list -> int
```

- The '_' pattern is a wildcard. It matches any value but doesn't bind a name to it.
- When run, a value will match the first pattern it can match with, and ignore the rest.
- ML will warn you if your patterns are non-exhaustive.

Datatypes

- Used for defining new types.
- ML has the ability to define recursive datatypes.
- Type is defined as the set of values and operators that operate on those values.
- Datatype definition involves two kinds of identifiers:
  - A type constructor that is the name of the datatype.
  - One or more data constructors which are identifiers used as operators to build values belonging to new datatype.
- Examples:
  - datatype fruit=Apple|Pear|Grape;
- Datatypes using data constructors:

```
datatype (<list of parameters>) <identifier>=
        <first constructor expression> |
        <second constructor expression>|..
        |<last constructor expression>
```

Datatypes

- Constructor expression consists of constructor name, the keyword ‘of’ and type expression.
- Example:
  - Leaf of int
  - datatype (‘a,’b) element = pair of ‘a * ‘b | Null
- Recursive datatypes:
  - Type whose values may incorporate elements of that type.
  - datatype ‘a tree=leaf of ‘a | node of ‘a tree * ‘a tree;
  - val myTree = node (node (leaf 5, leaf 2), leaf 3);

Infix Operators

- You can write your own infix operators like +, *, ::
- Define a function that takes a 2-tuple:
  - fun c (a, b) = 2*a + b
  - infix c
- Once you call 'infix' on your function, it can be used as an infix operator
  - - 4 c 5
  - Val it = 13 : int

Infix Operators

- You can define functions with parameters that are infix operators
- The keyword ‘op’ is used to specify that a name is an infix operator.
- You can then use that as an infix operator in your function.
- - fun mult (op *) x y = x * y
- val mult = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
- (Note: the ‘*’ in the function above is a variable, not the built in multiplication operator. That’s why the function is generic. However, the variable that you use must be a built-in or previously defined infix operator. This can be confusing.)

Infix Operators

- - mult (op *) 3 4;
- val it = 12 : int
- - mult (op >) 3 4;
- val it = false : bool
- (Here the operators have their usual meanings.)