Chapter 8: Type Inference and Polymorphism

1. Type Inference
    a. Pleasant feature of ML is that it allows to omit the type information when declaring a variable whenever it can be determined from the context → type inference (compiler inferring the missing type information based on context)
    b. fn s:string => s ^ "\n" → no other type for s other than string
    Therefore, it can be written as
    fn s => s ^ "\n"
    c. Principal typing property of ML: whenever type information is omitted, there is always a mos general way to recover the omitted type information
    d. If there is no way to recover the omitted type information, then the expression is ill-typed
    e. The best way is to fill in the blanks → always be found by compiler
    f. fn x => x
    The body of the function places no constraints on the type of x since there is no performing computation → polymorphic
    It has infinitely many types, one for each choice of the type of the parameter x
    g. Type scheme is type expression involving one or more type variables standing for an unknown but arbitrary type expression → written as ‘a
    h. Instance of type scheme is obtained by replacing each of the type variables occurring in it with a type scheme, with the same type scheme replacing each occurrence of a given type variable
    i. Can be int -> int (same type) but not int -> string (different type)
    j. Type scheme ‘a -> ‘b can be any instance
    k. fn (x,y) = x constraints neither the type of x nor the type of y → choose their types freely and independent
    l. We can write it as fn(x: ‘a, y: ‘b) => x with type scheme ‘a * ‘b -> ‘a (‘a * ‘a -> ‘a is overly restrictive and ‘a * ‘b -> ‘c does not have same type of x)
    m. Every expression in ML has best or more general way to infer types for expression that maximizes generality
    n. Type inference is process of constraint satisfaction
    o. First, expression determines a set of equations governing the relationships among the types of its subexpressions
    p. Second, constraints are solved using process similar to Gaussian elimination called unification
    q. Equations can be classified by their solution sets:
        i. Overconstrained: no solution, error
        ii. Underconstrained: many solutions: ambiguous or polymorphic
        iii. Uniquely determined: one solution

2. Polymorphic Definitions
    a. Binding identity function to variable I
       val I : ‘a -> ‘a = fn x=> x
       OR
       fun I(x: ‘a): ‘a = x
    b. Each use of I determines a distinct instance of the ascribed type scheme ‘a -> ‘a
    c. ML also provides form of type inference on value bindings that eliminates the need to ascribe a type scheme to the variable when it is bound
    d. If no type is ascribed to a variable introduced by val binding, it is implicitly ascribed the principal type scheme of the right-hand side
    e. EX) val I = fn x => x OR fun I(x) = x as a binding for the variable
    f. Variables introduced by val binding are allowed to be polymorphic only if the right-hand side is a value → value restriction on polymorphic declarations
    g. EX) val J = I I -> not polymorphic since the right-hand side is not value but requires computation to determine the value
    h. EX) val J x = I I x => binding of J is lambda, which is value
    i. In some circumstances, this is not possible
    j. EX) val l = nil @ nil (right side is a list, cannot apply defining l to function)
       OR val l = nil
3. Overloading
    a. Few corner cases that create problems for type inference
    b. fn x => x + x (cannot omit type information since problems arise) → Does + stand for integer or floating point additions
    c. When such is presented, compiler has two choices
        i. Declare expression ambiguous, and force programmer to provide enough explicit type information to resolve it
        ii. Arbitrarily choose “default” interpretation that forces one interpretation
    d. Many compilers choose second approach → but issue a warning indicating that it has done so
    e. Considered ambiguous
    f. EX)
       let
            val double = fn x => x + x
       in
            (double 3, double 4)
       end
    g. It will be flagged as ambiguous even though its only uses are with integer arguments because value bindings are considered to be “units” of type inference for which all ambiguity must be resolved before type checking continues

h. Compiler giving int as default will give warning but will run, but the following would be rejected
i. let

```
    val double = fn x => x + x
in
    (double 3.0, double 4.0)
end
```

j. Ambiguity must be resolved at val binding, which means compiler must commit at that point to treating the addition operation as integer or floating point → no single choice can be correct since we subsequently use double at both types

Chapter 12: Exceptions

a. Effect → any action resulting from evaluation of an expression other than returning a value
b. Main examples
    i. Exceptions: evaluation may be aborted by signalinig an exceptional condition
    ii. Mutation: storage may be allocated and modified during evaluation
    iii. Input/output: it is possible to read from an input source and write to an output sink during evaluation
    iv. Communication: data may be sent to and received from communication channels

1. Exceptions as Errors
    a. Dynamic violations are signalling by raising exceptions
    b. Primitive Exceptions
        i. 3 + "3" is ill-typed and can't be evaluated while 3 div 0 is well typed that gives a run-time error that signals by raising exception Div
        ii. We can write the above as 3 div 0 ⇓ raise Div
        iii. Exception such as this reports an error message of the form "Uncaught exception Div" with line number of the point in the program
        iv. Exception has names
        v. ML programs are implicitly "bullet-proofed" against failures of pattern matching but no inexhaustive match warnings arise during type checking
        vi. Related situation is use of pattern in val binding to destructure a value
        vii. If pattern can fail to match a value of this type, then Bind exception is raised
        viii. EX) val h :: _ = nil → exception Bind since the pattern does not match the value nil

c. User-Defined Exceptions
    i. Introduce new exception using an exception declaration, and signal it using a raise expression when runtime violation occurs (easing the process of tracking down the source of error)

```
exception Factorial
fun checked_factorial n =
    if n < 0 then
       raise Factorial
    else if n=0 then
       1
    else n * checked_factorial (n-1)
```

    ii.
    iii. Declaration exception Factorial introduces exception Factorial, which we raise in case checked_factorial is applied to negative number
    iv. Problems:
        1. It does not make effective use of pattern matching but relies on explicit comparison operations
        2. Repeatedly checks the validity of its argument on each recursive-call even though we can prove that if initial argument is non-negative, the argument on each recursive call is non-negative
    v. Improve by

```
exception Factorial
local
   fun fact 0 = 1
     | fact n = n * fact (n-1)
in
   fun checked_factorial n =
       if n >= 0 then

          fact n
       else
          raise Factorial
end
```

2. Exception Handlers
    a. Use of exceptions to signal error conditions suggests raising an exception is fatal since execution of program terminates with the raised exception
    b. Singaling an error is only one use of exception mechanism
    c. Exceptions can be used to effect non-local transfers of control by using exception handler (catch a raised exception and continue evaluation along some other path)

```
fun factorial_driver () =
    let
        val input = read_integer ()
        val result =
            toString (checked_factorial input)
    in
        print result
    end
    handle Factorial => print "Out of range."
```

    d.

e. Expression of the form exp handle match is called an exception handler → evaluated by attempting to evaluate exp (if it returns a value, then that is the value of the entire expression, but if it raises an exception exc, then exception value is matched against the clauses of the match to determine how to proceed)
f. If pattern of a clause matches the exception exc, then evaluation resumes with the expression part of that clause
g. If no pattern matches, the exception exc is re-raised so that outer exception handlers may dispatch on it
h. In factorial_driver, evaluation proceeds by attempting to compute the factorial of a given number, printing the result if the given number is in range, and otherwise reporting that the number is out of range
i. Generalizing it in a number of ways

```
handle EndOfFile => print "Done."
     | SyntaxError =>
       let
           val _ = print "Syntax error."
       in
           factorial_driver ()
       end
     | Factorial =>
       let
           val _ = print "Out of range."
       in
           factorial_driver ()
       end
```

j.
k. Repeatedly read integers until user terminates input stream
   AND
   Expect function read_integer raises exception SyntaxError in the ase input is not properly formulated (gives string)
l. Primary benefits of exception mechanism
   i. Force you to consider exceptional case and
   ii. Allow to segregate special case from the normal case in the code
m. Implement backtracking
n. Given 5 cents and 2 cents, make 16
o. Only way done is 2 5s and 3 2s

```
exception Change
fun change _ 0 = nil
  | change nil _ = raise Change
  | change (coin::coins) amt =
    if coin > amt then
       change coins amt
    else
       (coin :: change (coin::coins) (amt-coin))
       handle Change => change coins amt
```

p.

3. Value-Carrying Exceptions
    a. Useful to attach additional information that is passed to exception handler
    b. Achieved by attaching values to exceptions
    c. Ex) raise SyntaxError “Integer expected” by declaring
       exception SyntaxError of string
    d. Exception constructors are in many ways similar to value constructors
    e. Exception constructors can be used to create exception, match an exception

Chapter 13: Mutable Storage

1. Reference Cells
    a. A mutable cell may be thought of as a container in which data value of a specified type is stored
    b. During execution of a program the contents of a cell may be retrieved or replaced by any other value of the appropriate type
    c. Programming with cells is called imperative programming
    d. Reference cells, like all values, are first class – they may be bound to variables, passed as arguments to functions, returned as results of functions, appear within data structures, and even be stored within other reference cells
    e. Reference cell is created or allocated by function ref of type typ -> typ ref
    f. Ref allocates a “new” cell, initializes its content to val, and returns a reference to the cell
    g. Contents of a cell of type typ is retrieved using function ! of type typ ref -> typ
    h. Applying ! to a reference cell yields the current contents of that cell
    i. Contents of a cell is changed by applying the assignment operator op :=, which has type typ ref * typ -> unit
    j. Assignment is usually written using infix syntax
    k. When applied to cell and value, it replaces the content of that cell with that value and yields the null-tuple as result
    l. EX)

```
val r = ref 0
val s = ref 0
val _ = r := 3
val x = !s + !r
val t = r
val _ = t := 5
val y = !s + !r
val z = !t + !r
```

    m. x is bound to 3, y is bound 5, and z is bound to 10
    n. Usage of val binding of the form val _ = exp when exp is to be evaluated purely for its effect (value of exp is discarded by binding, since left hand is _)

o. In ML, exp1; exp2 is same as
```
let
        val _ = exp1
in
        exp2
end
```

2. Reference Patterns
    a. Use ref patterns → pattern of the form ref pat matches a reference cell whose content matches the pattern pat
    b. This means that the cell's contents are implicitly retrieved during pattern matching and may be subsequently used without explicit de-referencing
    c. fun !(ref a) = a
    d. When called with a reference cell, it is de-referenced and its contents is bound to a, which is returned as result
3. Identity
    a. We can tell two expressions are equal iff they cannot be distinguished by any operation in the language
    b. We can tell two expressions apart iff there is a complete program containing one of the expressions whose observable behavior changes when we replace that expression by the other
    c. Complete program behavior includes
        i. Its values, either by non-termination or by raising an uncaught exception
        ii. Visible side effects, include visible modifications to mutable storage or any input/ouput in may perform
    d. Not count as observations
        i. Execution time or space usage
        ii. "Private" use of storage
        iii. The name of uncaught exceptions
    e. Therefore, taking these in mind,
    val r = ref 0
    val s = ref 0
    r and s are not equivalent (distinguish r from s and therefore are distinct)
    But val s = r (now s and r are equivalent → bound to same reference cell)
    f. val r = ref ()
    val s = ref ()
    r and s are distinct
4. Aliasing
    a. Problem of aliasing → any two variables of the same reference type might be bound to the same reference cell, or to two different reference cells

b. val r = ref 0
   val s = ref 0
   Two variables are not aliases but after declaration
   val r = ref 0
   val s = r
   They are not aliases for the same reference cell

5. Programming Well with References
   a. Using reference to define factorials

```
fun imperative_fact (n:int) =
    let
        val result = ref 1
        val i = ref 0
        fun loop () =
            if !i = n then
               ()
            else
               (i := !i + 1;
                result := !result * !i;
                loop ())
    in
        loop (); !result
    end
```

→ bad style to program in this fashion

   b. Private Storage
      i. Use of higher-order functions to manage shared private state
      ii. This programming style is closely related to the use of objects to manage state in object-oriented programming languages

```
local
   val counter = ref 0
in
   fun tick () = (counter := !counter + 1; !counter)
   fun reset () = (counter := 0)
end
```

      iii.
      iv. tick of type unit -> int and reset of type unit -> unit
      v. Their definitions share a private variable counter that is bound to a mutable cell containing the current value of a shared counter
      vi. tick increments counter and returns new value and reset operation resets its value to zero
      vii. Counter generator(or constructor)

```
fun new_counter () =
    let
        val counter = ref 0
        fun tick () = (counter := !counter + 1; !counter)
        fun reset () = (counter := 0)
    in
        { tick = tick, reset = reset }
    end
```

      viii.
      ix. The type of new_counter is
      unit -> {: tick: unit -> int, reset: unit -> unit}

x. Function new_counter can be thought of as a constructor for class of counter objects where each object has ap private instance variable counter that is shared between methods tick and reset

xi. Usage of counters

```
val c1 = new_counter ()
val c2 = new_counter ()
#tick c1 ();
(* 1 *)
#tick c1 ();
(* 2 *)
#tick c2 ();
(* 1 *)
#reset c1 ();
#tick c1 ();
(* 1 *)
#tick c2 ();
(* 2 *)
```

c. Mutable Data Structures

i. Second important use of references is to build mutable data structures

ii. Simple example is the type of possibly circular lists (pcl)

iii. pcl is finite graph in which every node has at most one neighbor called predecessor, in the graph

iv. Employ backpatching: predecessor is initialized to Nil so that node and its ancestors can be constructed

v. This can be achieved in ML using datatype declaration
datatype ‘a pcl = Pcl of ‘a pcell ref and ‘a pcell Nil | Cons of ‘a * ‘a pcl;

vi. Value of type typ pcl is reference to a value of type typ pcell

vii. Value of type pcell is either Nil, the cell at the end of a noncircular possibly-circular list, or Cons(h,t) where h is value of type typ and t is another such possibly-circular list

viii. Convenient functions for creating and taking apart possibly circular lists:

```
fun cons (h, t) = Pcl (ref (Cons (h, t)));
fun nill () = Pcl (ref Nil);
fun phd (Pcl (ref (Cons (h, _)))) = h;
fun ptl (Pcl (ref (Cons (_, t)))) = t;
```

ix. To implement backpatching, need a way to “zap” the tail of a possibly circular list

```
fun stl (Pcl (r as ref (Cons (h, _))), u) =
    (r := Cons (h, u));
```

x. It would make sense to require that the tail of the Cons cell be the empty pcl, so that you're only allowed to backpatch at the end of a finite pcl

xi. Finite and infinite pcl:

```
val finite = cons (4, cons (3, cons (2, cons (1, nill ()))))
val tail = cons (1, nill());
val infinite = cons (4, cons (3, cons (2, tail)));
val _ = stl (tail, infinite)
```

xii. Last step backpatches the tail of the last cell of infinite to be infinite itself, creating circular list

xiii. Size of pcl

```
local
  fun race (Nil, Nil) = 0
    | race (Cons (_, Pcl (ref c)), Nil) =
      1 + race (c, Nil)
    | race (Cons (_, Pcl (ref c)), Cons (_, Pcl (ref Nil))) =
      1 + race (c, Nil)
    | race (Cons (_, l), Cons (_, Pcl (ref (Cons (_, m))))) =
      1 + race' (l, m)
  and race' (Pcl (r as ref c), Pcl (s as ref d)) =
      if r=s then 0 else race (c, d)
in
  fun size (Pcl (ref c)) = race (c, c)
end
```

6. Mutable Arrays
   a. ML also provides mutable arrays as primitive data structure
   b. The type typ array is the type of arrays carrying values of type typ
   c. Basic operations on arrays are

```
val array : int * 'a -> 'a array
val length : 'a array -> int
val sub : 'a array * int -> 'a
val update : 'a array * int * 'a -> unit
```

   d. array creates a new array of a given length with the given value as initial value of every element of array
      length returns length of array
      sub performs a subscript operation, returning ith element of an array
   e. One simple use of arrays is for memoization
   f. Computing nth Catalan number by making use of an auxiliary summation function that can easily be defined (applying sum to f and n computers the sum of f1 + … + fn)

```
fun C 1 = 1
  | C n = sum (fn k => (C k) * (C (n-k))) (n-1)
```

g. Can do better by caching previously-computed results in an array, leading to an enormous improvement in execution speed

```
local
        val limit : int = 100
        val memopad : int option array =
            Array.array (limit, NONE)
in
        fun C' 1 = 1
          | C' n = sum (fn k => (C k)*(C (n-k))) (n-1)
        and C n =
            if n < limit then
               case Array.sub (memopad, n)


                of SOME r => r
                 | NONE =>
                   let
                       val r = C' n
                   in
                       Array.update (memopad, n, SOME r);
                       r
                   end
           else
              C' n
end
```

h. Function C is memoized version of Catalan number function

i. When called it consults the memopad to determine whether or not the required result has already been computed

j. If so, answer is simply retrieved from memopad, otherwise the result is computed, stored in the cache, and returned