~ Lecture VI ~

Keywords:

records; enumerated types; polymorphic `datatype`s: `option` type, disjoint-union type; abstract types; error handling; exceptions.

References:

- [MLWP]
  Section 2.9, § Records
  Chapter 4, § The `datatype` declaration
  Section 7.6, § The `abstype` declaration
  Chapter 4, § Exceptions

Record types

Records with *named fields* can be defined as a series of `name=value` pairs.

```
val r1 = { surname= "Jones"  , age=22 , gender= #"M" } ;
val r2 = { surname= "Clarke" , age=41 , gender= #"F" } ;
~
"p01"
```

```
> val r1 = {age = 22, gender = #"M", surname = "Jones"} :
    {age : int, gender : char, surname : string}
> val r2 = {age = 41, gender = #"F", surname = "Clarke"} :
    {age : int, gender : char, surname : string}
```


NB: Tuples are just records with numbered fields:

```
val t1 = { 1= "pi" , 2= 3.14159 } ;
val t2 = { 1= "e"  , 2= 2.71828 } ;
~
"p01"
```

```
> val t1 = ("pi", 3.14159) : string * real
> val t2 = ("e", 2.71828) : string * real
```

Fields can be accessed using `#field record`:

```
#age r1 ;
#1 t1 ;
```

```
> val it = 22 : int
> val it = "pi" : string
```



```
#badfield r2 ;
```

```
File "p01", line 15, characters 10-12:
! #badfield r2 ;
!           ^^
! Type clash: expression of type
!   {age : int, gender : char, surname : string}
! cannot have type
!   {age : int, badfield : 'a, ...}
! because record label  badfield  is missing
```

Records and Pattern Matching

Values in fields can be bound to names by pattern matching:

```
val { age = A1 , gender = _ , surname = S1 } = r1 ;

> val A1 = 22 : int
  val S1 = "Jones" : string
```

and one can just specify a subset of the fields:

```
- val { age = A2 , ... } = r2 ;

> val A2 = 41 : int
```


or use the field name as the value name:

```
val { age , surname , ... } = r1 ;
> val age = 22 : int
  val surname = "Jones" : string
```

Note however the following type mismatch:

```
val { age , surname } = r1 ;
! Toplevel input:
! val { age , surname } = r1 ;
!                         ^^
! Type clash: expression of type
!   {age : int, gender : char, surname : string}
! cannot have type
!   {age : int, surname : 'a}
! because record label  gender  is missing
```

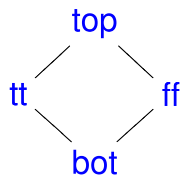

Record manipulation

Examples:

```
type person
  = { surname: string , age: int , gender: char } ;
fun adult( { age , ... }: person ) = age >= 16 ;
fun minor( p: person ) = (#age p) < 16 ;
~
"p03"
> type person
    = {age : int, gender : char, surname : string}
> val adult = fn :
    {age : int, gender : char, surname : string} -> bool
> val minor = fn :
    {age : int, gender : char, surname : string} -> bool
```


Enumerated types

Example:
The lattice

S = 

may be implemented as

```
datatype S = bot | tt | ff | top ;
fun lt (bot,_) = true
  | lt (_,top) = true
  | lt _ = false ;
~
"p04"
```

The `datatype` declaration defines the set of *constructors* of the datatype:

```
> New type names: =S
  datatype S
  = (S,{con bot : S, con ff : S,
        con top : S, con tt : S})
  con bot = bot : S
  con ff = ff : S
  con top = top : S
  con tt = tt : S
> val lt = fn : S * S -> bool
```


Polymorphic datatypes

1. The standard library declares the datatype `option`:

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

Note the polymorphism

```
> New type names: =option
  datatype 'a option =
  ('a option,{con 'a NONE : 'a option,
              con 'a SOME : 'a -> 'a option})
  con 'a NONE = NONE : 'a option
  con 'a SOME = fn : 'a -> 'a option
```

Type τ `option` contains a copy of type τ, augmented with the extra value `NONE`.

The `option` type can be used to supply optional data to a function, but its most obvious use is to indicate errors.

Example:

```
fun find P l
  = case dropwhile ( fn x => not(P x) ) l of
       [] => NONE
     | (h::_) => SOME h ;
> val 'a find
     = fn : ('a -> bool) -> 'a list -> 'a option
```


2. Disjoint-union type.

```
datatype ('a,'b)sum
  = In1 of 'a | In2 of 'b ;
~
"p05"
> New type names: =sum
  datatype ('a, 'b) sum =
  (('a, 'b) sum,
   {con ('a, 'b) In1 : 'a -> ('a, 'b) sum,
    con ('a, 'b) In2 : 'b -> ('a, 'b) sum})
  con ('a, 'b) In1 = fn : 'a -> ('a, 'b) sum
  con ('a, 'b) In2 = fn : 'b -> ('a, 'b) sum
```

Examples: `type intORreal = (int,real)sum ;`

`type 'a myoption = (unit,'a)sum ;`

Abstract types

Examples:

1. Complex numbers.

```
abstype complex = C of real * real
with
  fun ComplexRep( x , y ) = C(x,y) ;
  val origin = C( 0.0 , 0.0 ) ;
  fun X( C(x,y) ) = x ;
  fun Y( C(x,y) ) = y ;
  fun norm v = Math.sqrt( X(v)*X(v) + Y(v)*Y(v) ) ;
  fun scalevec( r, v ) = C( r*X(v) , r*Y(v) ) ;
  fun normal v = scalevec( 1.0/(norm v) , v ) ;
  fun CartRep v = ( X v , Y v ) ;
end ;
```



```
> New type names: complex
  type complex = complex
  val ComplexRep = fn : real * real -> complex
  val origin = <complex> : complex
  val X = fn : complex -> real
  val Y = fn : complex -> real
  val norm = fn : complex -> real
  val scalevec = fn : real * complex -> complex
  val normal = fn : complex -> complex
  val CartRep = fn : complex -> real * real
```



```
val v1 = normal( ComplexRep(2.0,2.0) ) ;
CartRep v1 ;
norm v1 ;
```

```
> val v1 = <complex> : complex
> val it
  = (0.707106781187, 0.707106781187) : real * real
> val it = 1.0 : real
```


2. Queues.

```
abstype 'a Queue = Q of 'a list * 'a list
  with
    val empty = Q([],[]) ;
    fun null( Q([],[]) ) = true
      | null _ = false ;
    fun enq x ( Q([],_) ) = Q( [x] , [] )
      | enq x ( Q(front,back) ) = Q( front , x::back ) ;
    fun deq( Q(_::[],back) ) = Q( rev back , [] )
      | deq( Q(_::rest,back) ) = Q( rest , back )
      | deq( _ ) = empty ;
    fun hd( Q(head::rest,_) ) = SOME head
      | hd( Q([],_) ) = NONE ;
  end ;
```

```
type 'a Queue
val empty = - : 'a Queue
val null = fn : 'a Queue -> bool
val enq = fn : 'a -> 'a Queue -> 'a Queue
val deq = fn : 'a Queue -> 'a Queue
val hd = fn : 'a Queue -> 'a option
```



```
enq 0 empty; enq 1 it; deq it ; hd it ;
```

```
val it = - : int Queue
val it = - : int Queue
val it = - : int Queue
val it = SOME 1 : int option
```

```
enq 0 empty; enq 1 it; deq it ; deq it ; hd it ;
```

```
val it = - : int Queue
val it = - : int Queue
val it = - : int Queue
val it = - : int Queue
val it = NONE : int option
```


Exceptions

Exceptions are raised on various runtime failures including failed pattern match, overflow, *etc.* One can also define custom exceptions and raise them explicitly.

Examples:

1.

```
exception Error of string ;
fun f b
  = ( if b
      then raise Error "It's true\n"
      else raise Error "It's false\n"
        ) handle Error m => print m ;
```

```
val f = fn : bool -> unit
```

```
- f true ;
```

```
It's true
```


2. Queues.

```
abstype 'a Queue = Q of 'a list * 'a list
  with
    exception E ;
    val empty = Q([],[]) ;
    fun null( Q([],[]) ) = true
      | null _ = false ;
    fun enq x ( Q([],_) ) = Q( [x] , [] )
      | enq x ( Q(front,back) ) = Q( front , x::back ) ;
    fun deq( Q(_::[],back) ) = Q( rev back , [] )
      | deq( Q(_::rest,back) ) = Q( rest , back )
      | deq( _ ) = empty ;
    fun hd( Q(head::rest,_) ) = head
      | hd( Q([],_) ) = raise E ;
  end ;
```

```
datatype 'a instruction
  = create of 'a Queue -> 'a Queue
  | observe of 'a Queue -> 'a ;

fun process obs []  q  = obs
  | process obs ( (create f)::ins ) q
      = process obs ins ( f q )
  | process obs ( (observe f)::ins ) q
      = process (  (f q)::obs handle E => []  ) ins q ;
val process = fn :
'a list -> 'a instruction list -> 'a Queue -> 'a list
```