CS6202: Advanced Topics in
Programming Languages and Systems

Lecture 2/3: **Standard ML**

A type-safe language that embodies many innovative ideas in language design.
**Standard ML**

- Great programming language – reusability, abstraction, quite efficient.
- Expression-Oriented.
- Values, Types and Effects
- Polymorphic Types and Inference
- Products, Records and Algebraic Types
- Higher-Order Functions
- Exceptions and Reference Types
- Rich Module Language

Reference --- Programming in Standard ML:  
http://www.cs.cmu.edu/~rwh/introsml/
Example ML Program

• Problem – matching string against a regular expression.

```
signature REGEXP = sig
  datatype regexp =
    Zero | One | Char of char |
    Plus of regexp * regexp |
    Times of regexp * regexp |
    Star of regexp
  exception SyntaxError of string
  val parse : string -> regexp
  val format : regexp -> string
end

signature MATCHER = sig
  structure RegExp : REGEXP
  val match : RegExp.regexp -> string -> bool
end
```

• Structure is implementation, while signature denotes interface.
**Signature**

- Signature – describe interface of modules.

- Signature Expression:
  
  \[ \text{sigexp ::= sig specs end} \]

- Contains basic *specifications* for type, datatype, exception, values.

- Signature binding:
  
  \[ \text{signature sigid = sigexp} \]
Implementation

- Implementation of signature is called structure.

```
structure RegExp ==> REGEXP = ...  
structure Matcher ==> MATCHER = ...
```

- Components referred by long identifiers.

```
val regexp =  
  Matcher.RegExp.parse "(a+b)*)"  
val matches =  
  Matcher.match regexp

val ex1 = matches "aabba"   (* yields true *)  
val ex2 = matches "abac"    (* yields false *)
```
**Structure**

- A unit of program with declarations for types, exceptions and values.

- Structure Expression:
  
  \[ \text{strexp ::= struct } \text{decs } \text{end} \]

- Contains *definitions* for type, datatype, exception, values.

- Structure binding:

  \[
  \text{structure strid = strexp}
  \]
**Computation Model**

- Emphasis is on evaluation of *expressions* rather than command.

- Each expression has three characteristics: (i) *type*, (ii) *value* and (iii) possible *effect*.

- Type is a description of the value it is supposed to yield.

- Evaluation may cause an effect, such as *input/output*, *exception* or *mutation*.

- Pure expression (e.g. mathematical functions) does not have side-effects.
Values

- Expression has a type, denoted by \( \text{exp} : \text{typ} \)

\[
\begin{align*}
3 & : \text{int} \\
3 + 4 & : \text{int} \\
4 \text{ div } 3 & : \text{int} \\
4 \text{ mod } 3 & : \text{int}
\end{align*}
\]

- Can be evaluated to a value, denoted by \( \text{exp} \downarrow \text{val} \)

\[
\begin{align*}
5 & \downarrow 5 \\
2+3 & \downarrow 5 \\
(2+3) \text{ div } (1+4) & \downarrow 1
\end{align*}
\]
Types

• Some examples of base types:
  
  • **Type name:** real
    
    - **Values:** 3.14, 2.17, 0.1E6, ...
    - **Operations:** +, −, *, /, =, <, ...
  
  • **Type name:** char
    
    - **Values:** "a", "b", ...
    - **Operations:** ord, chr, =, <, ...
  
  • **Type name:** string
    
    - **Values:** "abc", "1234", ...
    - **Operations:** ^, size, =, <, ...
  
  • **Type name:** bool
    
    - **Values:** true, false
    - **Operations:** if exp then exp₁ else exp₂
Declarations

• Any type may be given a name through type binding

    type float = real
    type count = int and average = real

• A value may be given a name through a value binding. Such bindings are type-checked, and rejected if ill-typed.

    val m : int = 3+2
    val pi : real = 3.14 and e : real = 2.17
**Limiting Scope**

- Scope of a type variable or type constructor may be delimited, as follows:

  
  \[
  \text{let } \text{dec in } \text{exp end}
  \]

  
  \[
  \text{local } \text{dec in } \text{dec' end}
  \]

- An Example.

  
  ```
  val m : int = 2
  val r : int =
    let
      val m : int = 3
      val n : int = m*m
    in
      m*n
    end
  end * m
  ```
Functions

• Two main aspects:
  • *algorithmic* – how it is computed
  • *extensional* – what is being computed

• Each function has a type:
  \[ \text{typ} \rightarrow \text{typ}' \]

• Anonymous function written using syntax:

```plaintext
fn var : typ => exp
```

Example:

```plaintext
fn x : real => Math.sqrt (Math.sqrt x)
```
**Functions**

- Function is also a value:

\[
\text{val } \text{var} : \text{typ} = \text{exp}
\]

- An example of function value:

\[
\text{val fourthroot : real } \to \text{ real } = \\
\quad \text{fn } x : \text{real } \to \text{Math.sqrt (Math.sqrt x)}
\]
**Tuple and Product Type**

- Aggregate data structures, such as tuples, lists, can be easily created and manipulated.

- An n-tuple is a finite ordered sequence:

\[(val_1, \ldots, val_n) : \text{typ}_1 \times \ldots \times \text{typ}_n\]

**Example**

```scala
val pair : int * int = (2, 3)
val triple : int * real * string = (2, 2.0, "2")
val quadruple : int * int * real * real
                         = (2, 3, 2.0, 3.0)
val pair_of_pairs : (int * int) * (real * real)
                      = ((2, 3), (2.0, 3.0))
```
**Tuple Pattern**

- Allows easy access of components. General form:

  \[
  \text{val } \text{pat} = \text{exp}
  \]

- Permitted form of tuple pattern:

  1. A variable pattern of the form \( \text{var:typ} \).

  2. A tuple pattern of the form \((\text{pat}_1, \ldots, \text{pat}_n)\), where each \(\text{pat}_i\) is a pattern. This includes as a special case the null-tuple pattern, \((\ )\).

  3. A wildcard pattern of the form \(\_\).

- Example:

  \[
  \text{val } ((\_), (r:real, \_)) = \text{val}
  \]
**Record Types**

- Record type allows a label to be associated with each component.

- A record value and its type:

  \[
  \{lab_1=val_1, \ldots , lab_n=val_n\} : \{lab_1:typ_1, \ldots , lab_n:typ_n\}
  \]

  - record value
  - record type

- Record binding:

  \[
  val \quad \{lab_1=pat_1, \ldots , lab_n=pat_n\} = \{lab_1=val_1, \ldots , lab_n=val_n\}
  \]
Record Example

record type

type hyperlink =
  { protocol : string,
    address : string,
    display : string }
**Selectors**

- A list of predefined selection function for the i-th component of a tuple.

\[
\text{fun } \#i \ (\_, \ldots, \_, x, \_, \ldots, \_ ) = x
\]

- Predefined selector for record fields:

\[
\text{fun } \#\text{lab} \ \{\text{lab}=x, \ldots\} = x
\]

- Use sparingly as patterns are typically clearer.
**Case Analysis**

- Clausal function expression useful for cases.

\[
\text{fn } pat_1 \Rightarrow exp_1 \\
| \ldots \\
| pat_n \Rightarrow exp_n
\]

- An example:

\[
\text{val recip : int } \rightarrow \text{ int } = \\
\text{fn 0 } \Rightarrow \text{ 0 } | \text{ n: int } \Rightarrow \text{ 1 div n}
\]

- Alternative form:

\[
\text{case } exp \\
\text{of } pat_1 \Rightarrow exp_1 \\
| \ldots \\
| pat_n \Rightarrow exp_n
\]  
\[
\equiv \\
(\text{fn } pat_1 \Rightarrow exp_1 \\
| \ldots \\
| pat_n \Rightarrow exp_n ) \\
\text{exp}.
\]
**Recursive Function**

- Use `rec` to indicate recursive value binding.

```ocaml
val rec factorial : int -> int = 
   fn 0 => 1 | n:int => n * factorial (n-1)
```

- Or use fun notation directly:

```ocaml
fun factorial 0 = 1
   | factorial (n:int) = n * factorial (n-1)
```
**General Recursion**

- Requires linear stack space.

```plaintext
fun factorial 0 = 1
  | factorial (n:int) = n * factorial (n-1)
```

- Example:

```
factorial 3
3 * factorial 2
3 * 2 * factorial 1
3 * 2 * 1 * factorial 0
3 * 2 * 1 * 1
3 * 2 * 1
3 * 2
6
```
**Iteration via Tail-Recursion**

- Loop is equivalent to tail-recursive code

```plaintext
fun helper (0, r: int) = r
  | helper (n: int, r: int) = helper (n-1, n*r)
fun factorial (n: int) = helper (n, 1)
```

- Example:

```
helper (3, 1)
helper (2, 3)
helper (1, 6)
helper (0, 6)
6
```

- What is a tail call, and why is it more efficient?
Polymorphism / Overloading

- Some functions have generic type. For example, the identity function has a principal type `\texttt{a -> a}

\begin{verbatim}
val I : 'a->'a = fn x=>x
fun I(x:'a):'a = x
\end{verbatim}

- Overloading uses the same name for a class of operator.

\begin{verbatim}
fn x:int => x+x
fn x:real => x+x
\end{verbatim}

- Hard problem:

\begin{verbatim}
let
  val double = fn x => x+x
in
  (double 3, double 3.0)
end
\end{verbatim}
**Algebraic Data Types**

- Data type declaration via `datatype` contains:
  - Type constructor
  - Value constructor(s)

- Examples of non-recursive data types.

```haskell
datatype suit = Spades | Hearts | Diamonds | Clubs

fun outranks (Spades, Spades) = false
  | outranks (Spades, _) = true
  | outranks (Hearts, Spades) = false
  | outranks (Hearts, Hearts) = false
  | outranks (Hearts, _) = true
  | outranks (Diamonds, Clubs) = true
  | outranks (Diamonds, _) = false
  | outranks (Clubs, _) = false
```
**Algebraic Data Types**

- Some may have type parameters, e.g.

\[
\text{datatype } 'a \text{ option } = \text{NONE } | \text{SOME of } 'a
\]

- An example of its use:

\[
\text{fun reciprocal 0 } = \text{NONE} \\
| \text{reciprocal } n = \text{SOME (1 div } n)\]

- Recursive type is also possible:

\[
\text{datatype } 'a \text{ tree } = \\
\text{Empty } | \\
\text{Node of } 'a \text{ tree } * 'a * 'a \text{ tree}
\]
**Algebraic Data Types**

- Recursive functions:

  ```
  fun height Empty = 0
  | height (Node (lft, _, rht)) = 1 + max (height lft, height rht)
  ```

  ```
  fun size Empty = 0
  | size (Node (lft, _, rht)) = 1 + size lft + size rht
  ```

- Mutual recursive data types (a bit contrived):

  ```
  datatype 'a tree =
      Empty |
      Node of 'a * 'a forest
  and 'a forest =
      None |
      Tree of 'a tree * 'a forest
  ```

- Disjoint union types:

  ```
  datatype int_or_string =
      Int of int |
      String of string
  ```
Abstract Syntax Tree

• Easy to model symbolic data structures:

```
datatype expr =
    Numeral of int |
    Plus of expr * expr |
    Times of expr * expr
```

• An interpreter:

```
fun eval (Numeral n) = Numeral n |
    eval (Plus (e1, e2)) = let
    val Numeral n1 = eval e1
    val Numeral n2 = eval e2
    in Numeral (n1+n2) end |
    eval (Times (e1, e2)) = let
    val Numeral n1 = eval e1
    val Numeral n2 = eval e2
    in Numeral (n1*n2) end
```
Lists

- A built-in data type with 2 value constructors.

```plaintext
val nil : typ list
val (op ::) : typ * typ list -> typ list

val₁::val₂::...::valₙ::nil abbreviated [ val₁, val₂, ..., valₙ ]
```

- Some functions on list:

```plaintext
fun length nil = 0
  | length (_::t) = 1 + length t

fun append (nil, l) = l
  | append (h::t, l) = h :: append (t, l)

fun rev nil = nil
  | rev (h::t) = rev t @ [h]
```

infix version of append
Higher-Order Functions

- Functions are first-class: pass as arguments, return as result, contain inside data structures, has a type.

- Key main uses:
  - abstracting control
  - staging computation

- Example – applies a function to every element of list

\[
\text{fun map'} (f, \text{nil}) = \text{nil} \\
| \text{map'} (f, h::t) = (f \ h) :: \text{map'} (f, t)
\]

\[
\text{map'} (\text{fn } x \Rightarrow x+1, [1,2,3,4]) \Rightarrow [2,3,4,5]
\]
Higher-Order Functions

- Returning function as result:

  ```plaintext
  val constantly = fn k => (fn a => k)
  fun constantly k a = k
  ```

- Curry function to untupled argument:

  ```plaintext
  fun curry f x y = f (x, y)
  ```

  ```plaintext
  ('a*'b->'c) -> ('a -> ('b -> 'c))
  ```
Abstracting Control

- Abstracting similar patterns of control

```ml
fun add_up nil = 0
  | add_up (h::t) = h + add_up t
fun mul_up nil = 1
  | mul_up (h::t) = h * mul_up t

fun reduce (unit, opn, nil) = unit
  | reduce (unit, opn, h::t) = opn (h, reduce (unit, opn, t))
```

- What is the principal type of this reduction?

```ml
val reduce : 'b * ('a*'b->'b) * 'a list -> 'b
```
**Staging**

- Distinguish early from late arguments:

  ```ml
  fun staged_reduce (unit, opn) = 
    let
      fun red nil = unit
      | red (h::t) = opn (h, red t)
    in
      red
    end
  ```

  *early argument

  *late argument

- Improve by early evaluation and then sharing.

  ```ml
  fun staged_append nil = fn l => l
   | staged_append (h::t) = 
     let
       val tail_appender = staged_append t
     in
       fn l => h :: tail_appender l
     end
  ```

  **staged_append** [v₁, ..., vₙ]

  ```ml
  fn l => v₁ :: v₂ :: ... :: vₙ :: l.
  ```
Exceptions

- Are useful to catch runtime errors.

\[
\begin{align*}
\text{3 div 0} & \downarrow \text{raise Div} \\
\text{hd nil} & \downarrow \text{raise Match}
\end{align*}
\]

- An example of user-defined exception:

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

- Exception handler can be used to catch a raised exception. This can make software more robust.

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

- Handler has the syntax:

```plaintext
exp handle match
match ::= pat => exp
```
Exceptions

• Exception can implement back-tracking.

```ml
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
```

• Exception may carry values.

```ml
declare exception SyntaxError of string

raise SyntaxError "Identifier expected"

catch ...
  handle SyntaxError msg => print "Syntax error: " ^ msg
```
Mutable Store

- Mutable cell contains a value that may change:

- Create a mutable cell with an initial value:
  
  \[
  \text{ref : } \ 'a -> \ 'a \text{ ref}
  \]

- Contents can be retrieved using:
  
  \[
  \text{! : } \ 'a \text{ ref } -> \ 'a
  \]

  Can use a ref pattern:
  
  \[
  \text{fun } !\text{(ref a)} = a
  \]

- How is equality implemented for reference?

  \[
  \text{val r = ref ()}
  \text{val s = ref ()}
  \]

  \[
  \text{if s=r then "it’s r" else "it’s not"}
  \]
Bad Imperative Programming

- A factorial function: can you follow?

```ml
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
```
**OO Programming Style**

- An single counter:

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

- A class of counters:

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

Type of `new_counter`:

```
unit -> { tick : unit->int, reset : unit->unit }
```
**Mutable Array**

-Mutable array as a primitive data structure:
  ```ml
  val array : int * 'a -> 'a array
  val length : 'a array -> int
  val sub : 'a array * int -> 'a
  val update : 'a array * int * 'a -> unit
  ```

-Can be used for memoization where many redundant calls, e.g. n-th Catalan number:
  ```ml
  fun C 1 = 1
  | C n = sum (fn k => (C k) * (C (n-k))) (n-1)
  ```

\[ \text{sum } f \ n = (f \ 0) + ... + (f \ n) \]
Memoization

- Repeated calls are retrieved rather than recomputed.

```latex
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
```
**Memoization**

- Apply the same idea to computing fibonacci efficiently.

```ml
local
  val limit : int = 1000
  val memo : int option array = Array.array(limit,NONE)
in
  fun fib' 0 = 1
    | fib' 1 = 1
    | fib' n = fib(n-1) + fib(n-2)
  and fib n =
    if n<limit then
      case Array.sub (memo,n) of
        SOMR r => r
      | None => let r=fib' n in
        Array.update(memo,n,SOME r)
      end
    else fib' n
  end
```
**Tupling**

- Is there no hope for purity?
  Use tupled function

\[
\text{fibtup } n = (\text{fib}(n+1), \text{fib}(n))
\]

Optimised code with reuse:

\[
\begin{align*}
\text{fun fibtup } 0 &= (1,1) \\
\mid \text{fibtup } n &= \text{case fibtup}(n-1) \text{ of} \\
&\quad (u,v) \Rightarrow (u+v,u) \\
\text{and fib } n &= \text{snd(fibtup}(n))
\end{align*}
\]

More optimization – tail recursion? logarithmic time?
**Input/Output**

- Standard input/output organized as streams.
- Read a line from an input stream.
  \[
  \text{inputLine} : \text{instream} \rightarrow \text{string}
  \]
- Write a line to stdout stream.
  \[
  \text{print} : \text{string} \rightarrow \text{unit}
  \]
- Write a line to a specific stream.
  \[
  \text{output} : \text{outstream} \times \text{string} \rightarrow \text{unit}
  \]
  \[
  \text{flushout} : \text{outstream} \rightarrow \text{unit}
  \]
- A blocking input that reads current available string
  \[
  \text{input} : \text{instream} \rightarrow \text{string}
  \]
- Non-blocking input that reads upto n-char string
  \[
  \text{caninput} : \text{instream} \times \text{int} \rightarrow \text{string}
  \]
Lazy Data Structures

- ML philosophy – laziness is a special case of eagerness. Can treat an unevaluated expression as a value.

- Applications
  - (i) infinite structures (e.g. streams)
  - (ii) interactive system
  - (iii) better termination property

activate SML/NJ option

```
Compiler.Control.lazysml := true;
open Lazy;
```

- Infinite stream and accesses:

```plaintext
datatype lazy 'a stream = Cons of 'a * 'a stream
val rec lazy ones = Cons (1, ones)
val Cons (h, t) = ones
val Cons (h, (Cons (h', t'))) = ones
```
Lazy Function Definitions

- Function over lazy stream is already lazy.
  
  ```
  fun shd (Cons (h, _)) = h
  fun stl (Cons (_, s)) = s
  ```

- So how can a function be made lazier?
  
  ```
  fun lazy lstl (Cons (_, s)) = s
  ```

- An example of difference in laziness
  
  ```
  val rec lazy s = (print "."; Cons (1, s))
  val _ = stl s (* prints "." *)
  val _ = stl s (* silent *)
  val rec lazy s = (print "."; Cons (1, s))
  val _ = lstl s (* silent *)
  val _ = stl s (* prints "." *)
  ```
Programming with Streams

- Lazily set up stream computation, not perform them.

```plaintext
fun smap f =
    let
        fun lazy_loop (Cons (x, s)) =
            Cons (f x, loop s)
    in
        loop
    end
```

- Lazy feature suspends a function call, an example:

```plaintext
val one_plus = smap (fn n => n + 1)
val rec lazy nats = Cons (0, one_plus nats)

Result: [0, 1, 2, 3, 4, 5, 6, .... ]
```
**Infinite Primes**

- Using Sieve of Erastotene method that sieves away non-prime.

```ml
fun sfilter pred = 
  let
    fun lazy loop (Cons (x, s)) = 
      if pred x then
        Cons (x, loop s)
      else
        loop s
  in
    loop
  end
fun m mod n = m - n * (m div n)
fun divides m n = n mod m = 0
fun lazy sieve (Cons (x, s)) = 
  Cons (x, sieve (sfilter (not o (divides x)) s))
val nats2 = stl (stl nats)
val primes = sieve nats2
```
**Modules in ML**

- *Signatures* and *Structures* are fundamental constructs of ML module system.

- Four basic forms of specifications are:

  1. A *type specification* of the form

     ```
     type (tyvar₁,...,tyvarₙ) tycon [ = typ ] ,
     ```

     where the definition *typ* of *tycon* may or may not be present.

  2. A *datatype specification*, which has precisely the same form as a datatype declaration.

  3. An *exception specification* of the form

     ```
     exception excon of typ .
     ```

  4. A *value specification* of the form

     ```
     val id : typ .
     ```
Signatures

• An example of signature definition.

```plaintext
signature QUEUE =
  sig
  type 'a queue
  exception Empty
  val empty : 'a queue
  val insert : 'a * 'a queue -> 'a queue
  val remove : 'a queue -> 'a * 'a queue
end
```

• Above signature requires its structure to provide a unary type constructor, an exception and three polymorphic value/functions.
**Signature Inheritance**

- Signatures can use two kinds of inheritance mechanism - *inclusion* or *specialization*.

- An example with inclusion:

  ```ml
  signature QUEUE_WITH_EMPTY =
  sig
    include QUEUE
    val is_empty : 'a queue -> bool
  end
  ```

- Same as expanded version:

  ```ml
  signature QUEUE_WITH_EMPTY =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
    val is_empty : 'a queue -> bool
  end
  ```
**Signature Specialization**

- Can augment an existing signature with extra type definitions.

```plaintext
signature QUEUE_AS_LISTS =
    QUEUE where type 'a queue = 'a list * 'a list
```

- But must **not** re-define a type that is already defined.

```plaintext
signature QUEUE_AS_LISTS_AS_LIST =
    QUEUE_AS_LISTS where type 'a queue = 'a list

signature QUEUE_AS_LIST =
    QUEUE where type 'a queue = 'a list
```
Structures

- Structures are implementation of signatures, while signatures are the *types* of structures.
- Four basic forms of structures.

1. A *type declaration* defining a type constructor.
3. An *exception declaration* defining a new exception constructor with a specified argument type.
4. A *value declaration* defining a new value variable with a specified type.
Structure Binding

• An example:

```ml
structure Queue =
  struct
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (b,f)) = (x::b, f)
    fun remove (nil, nil) = raise Empty
    | remove (bs, nil) = remove (nil, rev bs)
    | remove (bs, f::fs) = (f, (bs, fs))
  end
```

• Long identifiers of the form: `strid.id`

```ml
Queue.empty : 'a Queue.queue
Queue.insert : 'a * 'a Queue.queue -> 'a Queue.queue
'a Queue.queue = 'a list * 'a list
```

exposed details


**Structure Abbreviation**

- Use shorter names:

  ```
  structure Q = Queue
  ```

- An open declaration can inline the bindings directly:

  ```
  open strid_1 ... strid_n
  ```

  ```
  open Queue Stack
  ```

- Caveat: if an identifier is re-declared, it shadows/overrides the previous version.
Structure Matching

• When does a structure implement a signature? All components must satisfy all type definitions in signature.

• Rules of thumb:

  • To minimize bureaucracy, a structure may provide more components than are strictly required by the signature. If a signature requires components x, y, and z, it is sufficient for the structure to provide x, y, z, and w.

  • To enhance reuse, a structure may provide values with more general types than are required by the signature. If a signature demands a function of type int→int, it is enough to provide a function of type 'a→'a.

  • To avoid over-specification, a datatype may be provided where a type is required, and a value constructor may be provided where a value is required.

  • To increase flexibility, a structure may consist of declarations presented in any sensible order, not just the order specified in the signature, provided that the requirements of the specification are met.
Principal Signature

- Captures the most specific description of the components of structure.

- Briefly it contains all type definitions, datatype definitions, exception bindings plus principal types of value bindings.

- A candidate signature matches another one if it has all components and all type equations of the latter.

- Target is considered a weakening of the candidate.
Signature Matching

```ml
signature QUEUE =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
  end
signature QUEUE_WITH_EMPTY =
  sig
    include QUEUE
    val is_empty : 'a queue -> bool
  end
signature QUEUE_AS_LISTS =
  QUEUE where type 'a queue = 'a list * 'a list
```

Queue_with_Empty match Queue
Queue_with_Lists match Queue

but not vice-versa!
Polymorphic Instantiation

- Signature matching may involve an instantiation of polymorphic types.

```ml
signature MERGEABLE_QUEUE =
  sig
    include QUEUE
    val merge : 'a queue * 'a queue -> 'a queue
  end
matches the signature

signature MERGEABLE_INT_QUEUE =
  sig
    include QUEUE
    val merge : int queue * int queue -> int queue
  end
```

- ``a queue` has been instantiated to `int queue`
**Datatype Refinement**

- A datatype spec matches a type with same name but no definition.

```ml
signature RBT_DT =
  sig
  datatype 'a rbt =
    Empty |
    Red of 'a rbt * 'a * 'a rbt |
    Black of 'a rbt * 'a * 'a rbt
  end

matches the signature

signature RBT =
  sig
    type 'a rbt
    val Empty : 'a rbt
    val Red : 'a rbt * 'a * 'a rbt -> 'a rbt
  end
```

- A structure implements a signature safely if its principal signature matches with the latter signature.
**Signature Ascription**

- Signature *ascription* imposes the requirement that a structure implements a signature, hence weakening its signature for all subsequent uses.

- Two forms of ascriptions

  - transparent (descriptive)
    
    \[
    \text{structure } strid : \text{ sigexp } = \text{ strexp}
    \]

  - opaque (restrictive)
    
    \[
    \text{structure } strid : \Rightarrow \text{ sigexp } = \text{ strexp}
    \]
Opaque Ascription

• Primary use is to enforce data abstraction.

structure Queue :> QUEUE =
  struct
  type 'a queue = 'a list * 'a list
  val empty = (nil, nil)
  fun insert (x, (bs, fs)) = (x::bs, fs)
  exception Empty
  fun remove (nil, nil) = raise Empty
    | remove (bs, f::fs) = (f, (bs, fs))
    | remove (bs, nil) = remove (nil, rev bs)
end

• The type `a Queue.queue is abstract. Cannot rely on the fact that it is implemented as (`a list * `a list).
Exposing Opaque Ascription

- Occasionally some type need to be exposed.

```
signature PQ =
  sig
    type elt
    val lt : elt * elt -> bool
    type queue
    exception Empty
    val empty : queue
    val insert : elt * queue -> queue
    val remove : queue -> elt * queue
  end
```

- Cannot compare unless we know what elt type is.

```
signature STRING_PQ = PQ where type elt = string
structure PrioQueue :> STRING_PQ = ...
```
**Transparent Ascription**

- Cuts down need for explicit exposure of type definitions.

```ml
signature ORDERED =
  sig
    type t
    val lt : t * t -> bool
  end

structure String : ORDERED =
  struct
    type t = string
    val clt = Char.<
    fun lt (s, t) = ... clt ...
  end
```

- Not useful unless we know the definition for \( t \).
**Transparent Ascription**

- Can help document an interpretation without rendering it abstract.
- Two ways of ordering integers.

```ml
structure IntLt : ORDERED =
  struct
    type t = int
    val lt = (op <)
  end

structure IntDiv : ORDERED =
  struct
    type t = int
    fun lt (m, n) = (n mod m = 0)
  end
```
Module Hierarchies

- During structure implementation, some type may be specialised to different possibilities.

```ml
signature MY_STRING_DICT =
  sig
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * string * 'a -> 'a dict
    val lookup : 'a dict * string -> 'a option
  end

structure MyStringDict => MY_STRING_DICT =
  struct
    datatype 'a dict =
      Empty |
      Node of 'a dict * string * 'a * 'a dict
    val empty = Empty
    fun insert (d, k, v) = ...
    fun lookup (d, k) = ...
  end
```
Substructures

- Can organise as a structure within a structure.

```ml
signature ORDERED =
  sig
    type t
    val lt : t * t -> bool
    val eq : t * t -> bool
  end

signature DICT =
  sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * Key.t * 'a -> 'a dict
    val lookup : 'a dict * Key.t -> 'a option
  end
```

```ml
(* Lexicographically ordered strings. *)
structure LexString : ORDERED =
  struct
    type t = string
    val eq = (op =)
    val lt = (op <)
  end

(* Integers ordered conventionally. *)
structure LessInt : ORDERED =
  struct
    type t = int
    val eq = (op =)
    val lt = (op <)
  end

(* Integers ordered by divisibility. *)
structure DivInt : ORDERED =
  struct
    type t = int
    fun lt (m, n) = (n mod m = 0)
    fun eq (m, n) = lt (m, n) andalso lt (n, m)
  end
```
Substructures

- Different possible implementations:

```haskell
structure LessIntDict :> INT_DICT =
  struct
    structure Key : ORDERED = LessInt
  datatype 'a dict =
    Empty |
    Node of 'a dict * Key.t * 'a * 'a dict
  val empty = Empty
  fun insert (None, k, v) = Node (Empty, k, v, Empty)
  fun lookup (Empty, _) = NONE
  | lookup (Node (dl, l, v, dr), k) =
      if Key.lt(k, l) then
        lookup (dl, k)
      else if Key.lt(l, k) then
        lookup (dr, k)
      else
        v
  end
```

- Can generalize to parameterised signatures.
Sharing Specifications

- Substructures express dependency between one abstraction and another.
Sharing Specifications

- Opaque ascriptions make type abstract and different.

```
signature SPHERE =
  sig
    structure Vector : VECTOR
    structure Point : POINT
    type sphere
    val sphere : Point.point * Vector.vector -> sphere
  end
```

Point.Vector and Vector are treated as different types!

```
signature SPHERE =
  sig
    structure Vector : VECTOR
    structure Point : POINT
    sharing Point.Vector = Vector
    type sphere
    val sphere : Point.point * Vector.vector -> sphere
  end

signature GEOMETRY =
  sig
    structure Point : POINT
    structure Sphere : SPHERE
    sharing Point = Sphere.Point
      and Point.Vector = Sphere.Vector
  end
```
Sharing Specifications

- Can re-organise to cut down redundant substructures.

```plaintext
signature SPHERE =
   sig
   structure Point : POINT
   type sphere
   val sphere :
       Point.point * Point.Vector.vector -> sphere
   end
```

- One fewer sharing constraint.

```plaintext
signature GEOMETRY =
   sig
   structure Point : POINT
   structure Sphere : SPHERE
   sharing Point = Sphere.Point
   end
```

use Vector from Point
Parameterized Modules

- Can support code/spec reuse.
- Functor – module level function that takes a structure as argument to return a structure as result.

\[
\text{functor } \text{funid}(\text{decs}): \text{sigexp} = \text{strexp}
\]

result signature is opaquely described
**An Example Functor**

- A parametric implementation of dictionary.

```ml
functor DictFun
  (structure K : ORDERED) =>
  DICT where type Key.t = K.t =
struct
  structure Key : ORDERED = K
datatype 'a dict =
    Empty |
    Node of 'a dict * Key.t * 'a * 'a dict
  val empty = Empty
fun insert (None, k, v) =
  Node (Empty, k, v, Empty)
fun lookup (Empty, _) = NONE
  | lookup (Node (dl, l, v, dr), k) =
    if Key.lt(k, l) then
      lookup (dl, k)
    else if Key.lt (l, k) then
      lookup (dr, k)
    else
      v
end
```
**Functor Application**

- Format `funid(binds)` where `binds` is a sequence of bindings of arguments of the `functor`

```plaintext
structure LtIntDict = DictFun (structure K = LessInt)
structure LexStringDict = DictFun (structure K = LexString)
structure DivIntDict = DictFun (structure K = DivInt)
```

- Corresponding opaque signatures:

```plaintext
DICT where type Key.t = int
so that IntLtDict.Key.t is equivalent to int, and we deduce that the signature of LexStringDict is
DICT where type Key.t = string
and that the signature of DivIntDict is
DICT where type Key.t = int.
```
Functor and Sharing

- Functor can facilitate sharing of specification

Without functor:

```
signature GEOMETRY =
  sig
    structure Point : POINT
    structure Sphere : SPHERE
    sharing Point = Sphere.Point
    and Point.Vector = Sphere.Vector
    and Sphere.Vector = Sphere.Point.Vector
  end
```

With functor:

```
functor PointFun
  (structure V : VECTOR) : POINT = ...
functor SphereFun
  (structure V : VECTOR
      structure P : POINT) : SPHERE =

functor GeomFun
  (structure P : POINT
      structure S : SPHERE) : GEOMETRY =
  struct
    structure Point = P
    structure Sphere = S
  end
```

May be Wrongly typed!
Functor and Sharing

- Add sharing constraints to parameter list of functors.

```lisp
functor SphereFun
  (structure V : VECTOR
   structure P : POINT
   sharing P.Vector = V) : SPHERE =
```

```lisp
functor GeomFun
  (structure P : POINT
   structure S : SPHERE
   sharing P.Vector = S.Vector and P = S.Point) : GEOMETRY =
  struct
  structure Point = P
  structure Sphere = S
end
```

- Is sharing constraint avoidable? Parameterize on `Point` but there is a loss of generality.
Summary

• Values, Types and Effects
• Polymorphic Types and Inference
• Products, Records and Algebraic Types
• Higher-Order Functions
• Exceptions, Mutable State, Memoization
• Lazy Evaluation
• Module – Signature, Structure, Functors

Reference --- Programming in Standard ML:
http://www.cs.cmu.edu/~rwh/introsml/