



UNIVERSITETET  
I OSLO

# More on ML & Types

---

Gerardo Schneider  
gerardo@ifi.uio.no

Department of Informatics – University of Oslo

**Based on John C. Mitchell's slides (Stanford U.) ,**

# ML lectures

1. **08.09:** The Algol Family and ML (Mitchell's chap. 5 + more)
2. **15.09: More on ML & types (chap. 5, 6 + more)**
3. **06.10:** More on Types, Type Inference and Polymorphism (chap. 6)
4. **13.10:** Control in sequential languages, Exceptions and Continuations (chap. 8)

# Repetition

- ◆ Background for Algol-like languages
- ◆ Origin and background for ML, SML
- ◆ Types in ML
  - Basic types.
  - Compound types: unit, tuples, records, lists
- ◆ Value declarations and patterns
- ◆ Function declarations and pattern matching, multiple clause definitions
  - fun
- ◆ Case expressions
- ◆ Data-type declarations and type abbreviations
- ◆ Matching function declarations with datatype
- ◆ ML Imperative constructs
  - Assignment

# Outline

- ◆ ML imperative constructs
- ◆ More recursive examples
- ◆ Higher-order functions
- ◆ Something about equality
- ◆ Something on the ML module system
- ◆ Types in programming
- ◆ Type safety

# Imperative programming in ML

```
val i = ref 0;  
while !i < 5 do  
  (i := !i + 1 ;  
   print("i is :" ^ Int.toString(!i) ^ "\n")  
  );
```

## ◆ References

## ◆ In ML you evaluate a **series of expressions**

- By evaluating  $(e_1; e_2; \dots; e_n)$ , the expressions  $e_1$  to  $e_n$  are evaluated from left to right
- The result is the value of  $e_n$ . The other values are discarded

## ◆ While command : while e1 do e2

- **while e1 do e2**  $\equiv$  **if e1 then (e2; while e1 do e2) else ()** ;

## ◆ **print : string -> unit**

- print returns **it : ()** but has a side effect.

# Outline

- ◆ ML imperative constructs
- ◆ More examples on recursion
- ◆ Higher-order functions
- ◆ Something about equality
- ◆ Something on the ML module system
- ◆ Types in programming
- ◆ Type safety

# More on list functions

- ◆ Writing a recursive function is not difficult, but what about efficiency?
- ◆ Example: Reverse a list  
(remember  $[1,2] @ [3,4] = [1,2,3,4]$ )

```
fun rev [] = []  
  | rev (x::xs) = (rev xs) @ [x] ;
```

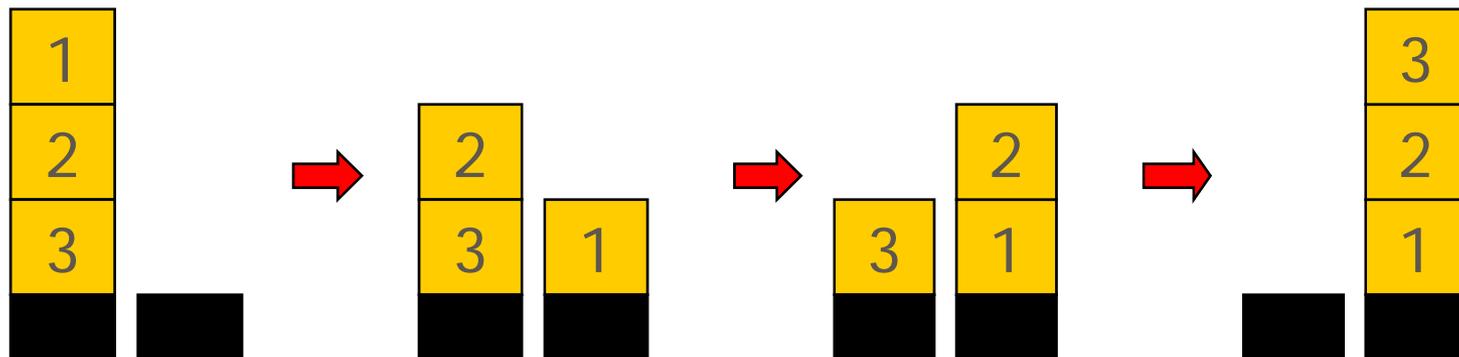
- ◆ Questions
  - How efficient is reverse?
  - Can you do this with only one pass through list?

# More efficient reverse function

```
fun revAppend ([],ys) = ys  
  | revAppend (x::xs,ys) = revAppend(xs,(x::ys)) ;
```

```
fun rev xs = revAppend(xs,[]);
```

Tail recursive function!



# Two factorial functions

## ◆ Standard recursion

```
- fun fact n =  
  if n = 0 then 1 else n * fact(n-1) ;
```

## ◆ Tail recursive (iterative)

```
- fun facti(n,p) =  
  if n = 0 then p else facti(n-1,n*p) ;  
- fun fact n = facti(n,1) ;
```

## ◆ More examples in Pucella sec. 2.7

# Outline

- ◆ ML imperative constructs
- ◆ More examples on recursion
- ◆ Higher-order functions
- ◆ Something about equality
- ◆ Something on the ML module system
- ◆ Types in programming
- ◆ Type safety

# Monomorphism vs. Polymorphism

- ◆ *Monomorphic* means "having only one form", as opposed to *Polymorphic*
- ◆ A type system is **monomorphic** if each constant, variable, etc. has unique type
- ◆ Variables, expressions, functions, etc. are **polymorphic** if they "allow" more than one type

Example. In ML, the *identity* function `fn x => x` is polymorphic: **it has infinitely many types!**

- `fn x => x`

> `val it = fn : 'a -> 'a`

**Warning!** The term "polymorphism" is used with different specific technical meanings (more on this in ML-lecture 3)

# Higher-order functions (functionals)

- ◆ IN ML functions are computational values ("first-class objects")
  - can be constructed during execution
  - stored in data structures
  - passed as arguments to other functions
  - returned as values

***A functional is a function that operates on other functions***

- ◆ Programs are more concise and clear when using functionals
- ◆ Functionals on lists have been very popular in Lisp
- ◆ The use of functionals is a powerful tool for *modularisation* which is what gives FPLs one of its conceptual advantages (Hughes 1984)

# Higher-order functions (functionals)

◆ Map: apply a function to every element in a list

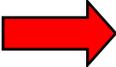
- fun map (f, nil) = nil

| map (f, x::xs) = f(x) :: map (f,xs);

> val map = fn : ('a -> 'b) \* 'a list -> 'b list

- fun incr x = x+1 ;

> val incr = fn : int -> int

- map (incr, [1,2,3]);            [2,3,4]

- map (fn x => x\*x, [1,2,3]);            [1,4,9]

# Higher-order functions (functionals)

◆ Map: apply a function to every element in a list

- fun map (f, nil) = nil

| map (f, x::xs) = f(x) :: map (f,xs);

> val map = fn : ('a -> 'b) \* 'a list -> 'b list

- fun bintoString(i) =

case x of 0 => "zero"

| 1 => "one"

| \_ => "illegal value";

> val bintoString = fn : int -> string

- map (bintoString , [1,0,2,0]);

> val it = ["one","zero","illegal value","zero"] : string list

# Higher-order functions (functionals)

◆ **filter**: apply a predicate to every element of list

- fun filter (p, nil) = nil

| filter (p, (x::xs)) = if p(x) then x :: (filter (p,xs))  
else filter (p,xs) ;

- val odd = fn : int -> bool

- val mylist = [1,2,3,4,5,6,7,8];

- filter (odd, mylist); > val it = [1,3,5,7] : int list

- map (fn x => x\*x, (filter(odd,mylist)));

> val it = [1,9,25,49] : int list

- val pairs = [(1,2),(4,3),(8,9),(0,9),(0,0),(5,1)] ;

- filter ((op <) , pairs);

> val it = [(1,2),(8,9),(0,9)] : (int \* int) list

# Curried functions

- ◆ A function can have only one argument
  - tuples are used for more than one argument
- ◆ Multiple arguments may be realized by giving a function as a result
  - *Currying* -> after the logician Haskell B. Curry
- ◆ A function over pairs has type  
 $'a * 'b \rightarrow 'c$   
while a curried function has type  
 $'a \rightarrow ('b \rightarrow 'c)$
- ◆ A curried function allows *partial application*: applied to its 1st argument (of type  $'a$ ), it results in a function of type  $'b \rightarrow 'c$

# Curried functions

## ◆ Example: function to add two numbers

```
- fun pluss (x,y) = x + y ;  
> val pluss = fn : int * int -> int  
- pluss (2,3) ;  
➤ val it = 5 : int
```

## ◆ Curried version of the same function

```
- fun cPluss x y = x + y ;  
> val cPluss = fn : int -> int -> int  
- cPluss 2 3 ;  
> val it = 5 : int  
- val addTwo = cPluss 2 ;  
> val addTwo = fn : int -> int  
- addTwo 5 ;  
> val it = 7 : int
```

# Curried functions

## ◆ Curry and uncurry

- fun curry f x y = f (x,y) ;

> val curry = fn : ('a \* 'b -> 'c) -> 'a -> 'b -> 'c

- fun uncurry f (x,y) = f x y ;

> val uncurry = fn : ('a -> 'b -> 'c) -> 'a \* 'b -> 'c

# Example: the map function

- ◆ Recall that map can be defined as

```
fun map (f, nil) = nil
```

```
| map (f, x::xs) = f(x) :: map (f,xs);
```

```
> val map = fn : ('a -> 'b) * 'a list -> 'b list
```

```
- map (fn x => x+1, [1,2,3]);
```

```
> val it = [2,3,4] : int list
```

- ◆ By currying it, we can define map as

```
fun map f nil = nil
```

```
| map f (x::xs) = (f x) :: map f xs;
```

```
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

```
- map (fn x => x+1) [1,2,3];
```

```
> val it = [2,3,4] : int list
```

# More on the map function

- ◆ We can have a function having as argument a function which has another function as an argument
- ◆ Thanks to currying, we can combine functionals to work on lists of lists

Example:

```
- map (map (fn x => x+1)) [[1], [1,2], [1,2,3]];
```

```
→ [ map (fn x => x+1) [1], map (fn x => x+1)[1,2], map (fn x => x+1)[1,2,3]]
```

```
→ [ [2], [2,3], [2,3,4]]
```

What does it give as a result?

```
> val it = [[2],[2,3], [2,3,4]] : int list list
```

# Outline

- ◆ ML imperative constructs
- ◆ More recursive examples
- ◆ Higher-order functions
- ◆ **Something about equality**
- ◆ Something on the ML module system
- ◆ Types in programming
- ◆ Type safety

# Equality

- ◆ Equality in (S)ML is defined for many types but not all – E.g., it is defined for:
  - Integers
  - Booleans
  - Strings
  - Characters
  
- ◆ What about floating points (reals), compound types (tuples, records, lists), functions, abstract data types, etc?

# Equality on "reals"

- ◆ In old versions of SML/NJ it was possible to compare floating points (reals) equality but **not anymore** (<http://www.standardml.org/Basis/real.html>)
- ◆ Example
  - $4.343 = 4.234234$ ;
  - > Error: operator and operand don't agree [equality type required]  
operator domain: "Z \* "Z  
operand:        real \* real  
in expression  $4.343 = 4.234234$

# Equality

## ◆ When are two expressions equal?

- The so-called *Leibniz's Principle of the Identity of Indiscernables*:

"e1 and e2 are **equal** iff they cannot be distinguished by any operation in the language"

"e1 and e2 are **distinct** iff there is some way to tell them apart"

## ◆ What is difficult about Leibniz's Principle?

# Problems with Equality

- ◆ Equality, as defined by Leibniz's principle, is **undecidable**

**In general, there is no program which determines whether two expressions are equal in Leibniz's sense**

Also:

- ◆ Problems with reference cells (aliasing)

- `val s = ref 1 ; val t = ref 1 ;`
- `s = t ;` > `false`
- `!s = !t` > `true`
- `val s = t ;`
- `s = t ;` > `true`

- ◆ Polymorphic equality complicates the compiler

# Equality Types

- ◆ An **equality type** is a type admitting equality test
- ◆ Types admitting equality in (S)ML
  - *int, bool, char, string*
  - *tuples* and *records*, if all their components admit equality
  - *datatypes*, if every constructor's parameter admits equality
  - *lists* admit equality if the underlying element type admits equality
    - Two lists are equal if they have the same length and the same elements in corresponding positions

# Equality Types (cont.)

## ◆ Do **not** admit equality in (S)ML

- *reals*
- *functions*
- *tuples, records and datatypes* not mentioned in the previous slide
- *abstract data types*

## ◆ Equality type variable: "a

- fun equals (x,y) = if x = y then true else false ;
- > stdIn:7.25 Warning: calling polyEqual
- val equals = fn : "a \* "a -> bool

# Equality: Examples

- ◆ Equality tests on functions is not computable since
$$f = g \text{ iff for all } x, f(x) = g(x)$$
- ◆ No "standard" notion of equality for an abstract type
  - What is supposed to be the equality on *trees*? Is it defined structurally? Is it over the list of their elements? By DFS or BFS?

- ◆ Ex:

```
fun find x nil = false
  | find x (y :: ys) = x = y orelse find x ys ;
```

```
> = stdIn:30.31 Warning: calling polyEqual
val find = fn : "a -> "a list -> bool
```

# Outline

- ◆ ML imperative constructs
- ◆ More recursive examples
- ◆ Higher-order functions
- ◆ Something about equality
- ◆ **Something on the ML module system**
- ◆ Types in programming
- ◆ Type safety

# Modularity: Basic Concepts

## ◆ Component

- Meaningful program unit
  - Function, data structure, module, ...

## ◆ Interface

- Types and operations defined within a component that are visible outside the component

## ◆ Specification

- Intended behavior of component, expressed as property observable through interface

## ◆ Implementation

- Data structures and functions inside component

# Example: Function Component

## ◆ Component

- Function to compute square root

## ◆ Interface

- function `sqrt (float x)` returns float

## ◆ Specification

- If  $x > 1$ , then  $\text{sqrt}(x) * \text{sqrt}(x) \approx x$ .

## ◆ Implementation

```
float sqrt (float x){  
    float y = x/2; float step=x/4; int i;  
    for (i=0; i<20; i++){if ((y*y)<x) y=y+step; else y=y-step; step = step/2;}  
    return y;  
}
```

# Something on ML Modules

- ◆ Signatures and structures are part of the standard *ML module system*
- ◆ An ML structure is a module, which is a collection of:
  - Types
  - Values
  - Structure declarations
- ◆ Signatures are module interfaces
  - Kind of "type" for a structure

# Example: Point

## ◆ Signature definition (Interface)

```
signature POINT =  
sig  
  type point  
  val mk_point : real * real -> point (*constructor*)  
  val x_coord : point -> real (*selector*)  
  val y_coord : point -> real (*selector*)  
  val move_p : point * real * real -> point  
end;
```

# Example: Point (cont.)

## ◆ Structure definition (Implementation)

```
structure pt : POINT =  
  struct  
    type point = real * real  
    fun mk_point(x,y) = (x,y)  
    fun x_coord(x,y) = x  
    fun y_coord(x,y) = y  
    fun move_p((x,y):point,dx,dy) = (x+dx, y+dy)  
  end;
```

## ◆ To be able to use the implementation:

- open pt;

# Example: Point (cont.)

Open the structure by writing `open <structname>`

```
- open pt;
```

```
...
```

After that you may use the struct operations

```
- val p1 = mk_point(4.3, 6.56);
```

```
> val p1 = (4.3,6.56) : point
```

```
- y_coord (p1);
```

```
> val it = 6.56 : real
```

```
- move_p (p1, 3.0, ~1.0);
```

```
> val it = (7.3,5.56) : point
```

You may use the struct without opening it by prefixing a function with the struct name.

```
- pt.mk_point(1.0,1.0);
```

```
> val it = (1.0,1.0) : point
```

E.g. we would like to use the `min` function to get the smallest of two ints.

```
- min(1,2);
```

```
> stdIn:1.1-1.4 Error: unbound variable or constructor: min
```

The function is defined in the `Int` struct so we must use `Int` as a prefix

```
- Int.min(1,2);
```

```
> val it = 1 : int
```

See: <http://www.smlnj.org/doc/basis/pages/sml-std-basis.html> for an overview of the structures and signatures in The Standard ML Basis Library. Follow the link: [Top-level Environment](#) to see which functions are available in the top level environment, i.e. which you can use without prefixes.

# Outline

- ◆ ML imperative constructs
- ◆ More recursive examples
- ◆ Higher-order functions
- ◆ Something about equality
- ◆ Something on the ML module system
- ◆ **Types in programming**
- ◆ Type safety

# Type

A **type** is a collection of computational entities sharing some common property

## ◆ Examples

- Integers
- [1 .. 100]
- Strings
- $\text{int} \rightarrow \text{bool}$
- $(\text{int} \rightarrow \text{int}) \rightarrow \text{bool}$

## ◆ “Non-examples”

- {3, true, 5.0}
- Even integers
- $\{f:\text{int} \rightarrow \text{int} \mid \text{if } x > 3 \text{ then } f(x) > x^*(x+1)\}$

Distinction between types and non-types is language dependent

# Uses for types

## ◆ Program organization and documentation

- Separate types for separate concepts
  - E.g., customer and accounts (banking program)
- Types can be checked, unlike program comments

## ◆ Identify and prevent errors

- Compile-time or run-time checking can prevent meaningless computations such as `3 + true` - “Bill”

## ◆ Support optimization

- Short integers require fewer bits
- Access record component by known offset

# Type errors

## ◆ Hardware error

- Function call `x()` (where `x` is not a function) may cause jump to instruction that does not contain a legal op code
  - If `x = 512`, executing `x()` will jump to location 512 and begin execute “instructions” there

## ◆ Unintended semantics

- `int_add(3, 4.5)`: Not a hardware error, since bit pattern of float 4.5 can be interpreted as an integer

# General definition of type error

- ◆ A *type error* occurs when execution of program is not faithful to the intended semantics
- ◆ Type errors depend on the concepts defined in the language; **not** on *how* the program is executed on the underlying software
- ◆ All values are stored as sequences of bits
  - Store 4.5 in memory as a floating-point number
    - Location contains a particular bit pattern
  - To interpret bit pattern, we need to know the type
  - If we pass bit pattern to integer addition function, the pattern will be interpreted as an integer pattern
    - Type error if the pattern was intended to represent 4.5

# Subtyping

- ◆ **Subtyping** is a relation on types allowing values of one type to be used in place of values of another
  - **Substitutivity:** If  $A$  is a subtype of  $B$  ( $A <: B$ ), then any expression of type  $A$  may be used without type error in any context where  $B$  may be used
- ◆ In general, if  $f: A \rightarrow B$ , then  $f$  may be applied to  $x$  if  $x: A$ 
  - Type checker: If  $f: A \rightarrow B$  and  $x: C$ , then  $C = A$
- ◆ In languages with subtyping
  - Type checker: If  $f: A \rightarrow B$  and  $x: C$ , then  $C <: A$

Remark: **No subtypes in ML!**

# Outline

- ◆ ML imperative constructs
- ◆ More recursive examples
- ◆ Higher-order functions
- ◆ Something about equality
- ◆ Something on the ML module system
- ◆ Types in programming
- ◆ **Type safety**

# Type safety

- ◆ A Prog. Lang. is *type safe* if no program can violate its type distinction
  - E.g. use an integer as a function
  - Access memory not allocated to the program.
- ◆ Examples of not type safe language features:
  - Type casts (a value of one type used as another type)
    - Use integers as functions (jump to a non-instruction or access memory not allocated to the program) (C)
  - Pointer arithmetic
    - $*(p)$  has type A if p has type A\*
    - $x = *(p+i)$  what is the type of x?
  - Explicit deallocation and dangling pointers
    - Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p

# Relative type-safety of languages

- ◆ **Not safe**: BCPL family, including C and C++
  - Casts; pointer arithmetic
- ◆ **Almost safe**: Algol family, Pascal, Ada.
  - Explicit deallocation; dangling pointers
    - No language with explicit deallocation of memory is fully type-safe
- ◆ **Safe**: Lisp, ML, Smalltalk, Java, Haskell
  - Lisp, Smalltalk: dynamically typed
  - ML, Haskell, Java: statically typed

# Compile-time vs. run-time checking

- ◆ Lisp uses run-time type checking
  - (car x) check first to make sure x is list
- ◆ ML uses compile-time type checking
  - f(x) must have  $f : A \rightarrow B$  and  $x : A$
- ◆ Basic tradeoff
  - Both prevent type errors
  - Run-time checking slows down execution (compiled ML code, up to 4 times faster than Lisp code)
  - Compile-time checking restricts program flexibility
    - Lisp list: elements can have different types
    - ML list: all elements must have same type
- ◆ Combination of Compile/Run-time eg. Java
  - Static type checking to distinguish arrays and integers
  - Run-time checking to detect array bounds errors

# Compile-time type checking

- ◆ *Sound* type checker: no program with error is considered correct
- ◆ *Conservative* type checker: some programs without errors are considered to have errors
- ◆ Static typing is always conservative

if (possible-infinite-run-expression)  
  then (expression-with-type-error)  
  else (expression-with-type-error)

Cannot decide at compile time if run-time error will occur  
(from the undecidability of the Turing machine's halting problem)

# Remarks – Further reading

- ◆ Mitchell doesn't cover the material presented on Equality – See section 2.9 of Pucella's notes
- ◆ *signatures* and *structures* are part of ML Module system. See section 9.3.2 of Mitchell's book
- ◆ Types: Mitchell's section 6.1, 6.2
- ◆ Imperative programming in ML: See chapter 8 of Paulson's book

# ML lectures

1. 08.09: The Algol Family and ML (Mitchell's chap. 5 + more)
2. 15.09: More on ML & types (chap. 5,6 + more)
3. **06.10: More on Types, Type Inference and Polymorphism (chap. 6)**
4. 13.10: Control in sequential languages, Exceptions and Continuations (chap. 8)