UNIVERSITETET
I OSLO

More on ML & Types

Arild B. Torjusen
aribraat@ifi.uio.no

Department of Informatics – University of Oslo

Based on John C. Mitchell's slides (Stanford U.) , adapted by Gerardo Schneider, UiO.

INF 3110/4110 - 2006

ML lectures

1. 04.09: The Algol Family and ML (Mitchell's chap. 5 + more)
2. **11.09: More on ML & types (chap. 5 and 6)**
3. 18.09: More on Types, Type Inference and Polymorphism (chap. 6)
4. 02.10: Control in sequential languages, Exceptions and Continuations (chap. 8)

Outline

- More recursive examples
- More on 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  reverse [] = []
   |  reverse (x::xs)  = (reverse xs) @ [x] ;
```

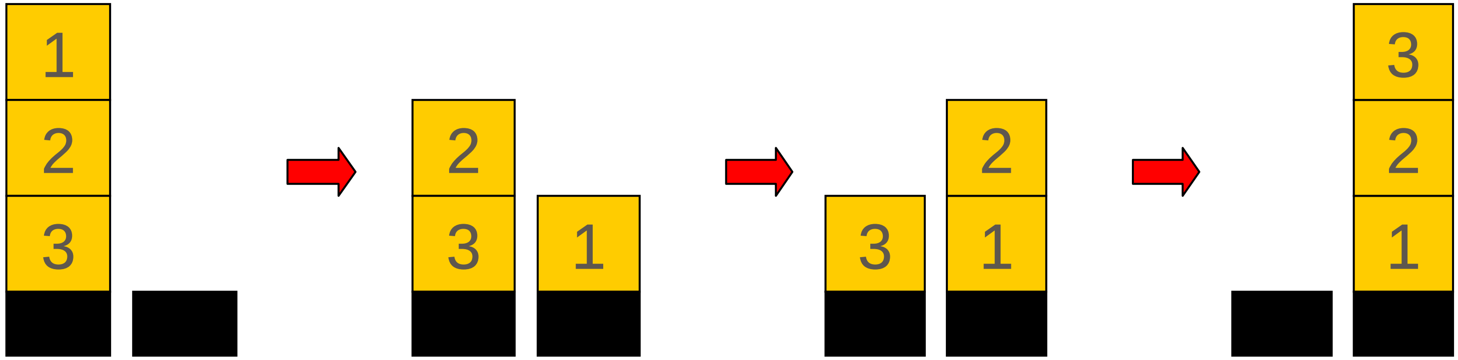
- Questions
  - How efficient is reverse?
  - Can you do this with only one pass through list?

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 (iteritative)

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

Outline

- More recursive examples
- More on higher-order functions
- Something about equality
- Something on the ML module system
- Types in programming
- Type safety

Higher-order functions (functionals)

- Functions are computational values, hence they can be passed as an argument to another function

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

- Program are more concise and clear when using functionals
- Functionals on lists have been very popular in Lisp

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 -> ’c

  while a curried function has type

  ’a -> (’b -> ’c)
- A curried function allows *partial application*: applied to its 1st argument (of type ’a), it results in a function of type ’b -> ’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]];
```

  What does it give as a result?

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

Outline

- More recursive examples
- More on 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), compund 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

- 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
```

INF 3110/4110 - 2006

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)
- Polymorphic equality complicates the compiler

Equality Types

- An equality type is a type admiting equality test
- Types admiting 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

    Ex: *lists* admit equality if the underlying element type admits equality. Moreover, 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

INF 3110/4110 - 2006

Equality: Examples

- Equality tests on functions is not computable since

  $$f = g \quad \text{iff} \quad \text{for all } x,\ f(x) = g(x)$$

- There is 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?
- Mitchell doesn’t cover the material presented on Equality – Check, for instance, Section 2.9 of Pucella’s notes

Outline

- More recursive examples
- More on 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 sqrt(x)*sqrt(x) ≈ x.
- **Implementation**

```
float sqroot (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.)

- Tests:

```
- 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
```

Remarks – Further reading

- *signatures* and *structures* are part of ML Module system. Modules, in general, will be developed later on this course. For the present lecture you might want to read Section 9.3.2 of Mitchell's book

Outline

- More recursive examples
- More on 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
  - int $\rightarrow$ bool
  - (int $\rightarrow$ int) $\rightarrow$bool
- "Non-examples"
  - {3, true, 5.0}
  - Even integers
  - {f:int $\rightarrow$ int | if x>3 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 -> B, then f may be applied to x if x: A
  - Type checker: If f: A -> B and x: C, then C = A
- In languages with subtyping
  - Type checker: If f: A -> B and x: C, then C <: A

Remark: No subtypes in ML!

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 that later)

Outline

- More recursive examples
- More on 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. functions and integer)
- 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)
  - 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
  - Lisp, Smalltalk: dynamically typed
  - ML, 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

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 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)