Principles of Programming Languages VI

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.dgp.toronto.edu/~wael/324.html
Today

- ML Types
- ML Polymorphism
- ML Exceptions
ML: record type

- We have seen list and tuple...

- Record syntax
  \[
  \{ \langle \text{field}_1 \rangle : \langle \text{type}_1 \rangle, \langle \text{field}_2 \rangle : \langle \text{type}_2 \rangle, \ldots, \langle \text{field}_n \rangle : \langle \text{type}_n \rangle \}\]

- A record instance is defined as
  \[
  \{ \langle \text{label}_1 \rangle = \langle \text{value}_1 \rangle, \langle \text{label}_2 \rangle = \langle \text{value}_2 \rangle, \ldots, \langle \text{label}_n \rangle = \langle \text{value}_n \rangle \}\]

- A record is a structured data type in which each element is accessed by a unique name.
  - E.g.
    \[
    \{ \text{name: string}, \\
    \text{age: int}, \\
    \text{salary: int} \}
    \]
    - {name = “Dave”, age = 77, salary=99000}
    > val it = {name = “Dave”, age = 77, salary=99000} : {name:string, age:int,salary:int}
ML: record type – cont’d

• Operations
  – # operator to extract a field from a record instance
  – E.g.

    - #salary {name =“john”, age=35, salary=90};
    > val it = 90 : int

    - #options {startcity="toronto",endcity="boston",
                options=("12",10,"K")};
    > val it = ("12",10,"K") : string * int * string
ML: record type – cont’d

• **Named Types**
  – It is inconvenient to repeatedly write out a complex type definition. So, ML provides a way to give a name to a type using `type`. Such is called **Named types**
    - E.g. - `type waitress = { name: string, wages: int, tips: int };`
  
  – Named types can be used anywhere that an unnamed types can.
    - E.g. - `fun income (w: waitress) =
          #wages w + #tips w;
    > val income = fn : waitress -> int
    - `fun income (w: { name: string, wages: int, tips: int }) =
          #wages w + #tips w;
    > val income = fn : {name:string, tips:int, wages:int} -> int`

  – Named typed can be used in type declaration
    - E.g. - `type waitresses = waitress list;
      - [{name=“sally”, wages= 20, tips=10},
            {name=“alice”, wages= 15, tips=15},
            {name=“sue”, wages= 25, tips=10}]`
ML: record type – cont’d

- Named Types – cont’d:
  - E.g.: finding the total income of all waitresses
    > type waitress = { name: string, wages: int, tips: int };
    > fun income (w: waitress) =
      #wages w + #tips w;
    > type waitresses = waitress list;
    > fun total (WL: waitresses) =
      if WL = [] then 0
      else income(hd WL) + total(tl WL);

- [{name="sally", wages= 20, tips=10},
  {name="alice", wages= 15, tips=20},
  {name="sue", wages= 25, tips=20}]

  30 +

- [{name="alice", wages= 15, tips=20}
  {name="sue", wages= 25, tips=20}]

  30 + 35 +

- [{name="sue", wages= 25, tips=20}]

  30 + 35 + 45
ML: type synonyms

• A type synonym is different from a type, why?

• Syntax  
  
  `type <type-name> = <type-specification>;`

• Example:
  
  - `type DATE = int * string * int;`  
    (* tuple of int, string, int *)
  
  - `fun age(birth: DATE) = ....;`
**ML: tuples & records**

- **Recall tuple syntax** \((\text{obj}_1, \text{obj}_2, \ldots)\)

- **Tuples are actually a special case/simplification of record types in which field have numbers.**
  - E.g.
    - The tuple \((7, \text{“ab”}, \text{“cd”})\) is really a short hand notation for the record
      \[
      \{ 1 = 7, 2 = \text{“ab”}, 3 = \text{“cd”}\}
      \]

- **In general, the tuple \((v_1, v_2, \ldots)\) is a shorthand for \(\{1=v_1, 2=v_2, \ldots\}\)**
  - That’s why \#2(6,7,”abc”) works!
ML: pattern matching on records

- Recall syntax
  ```
  fun <func> <pattern_1> = <expression_1>
  | <func> <pattern_2> = <expression_2>
  ........
  | <func> <pattern_n> = <expression_n>
  ```

- You can use patterns to match on records
  - E.g. finding the total income of all waitresses
    ```
    type waitress = { name: string, wages: int, tips: int };
    fun income (w: waitress) =
        #wages w + #tips w;
    ```
    ```
    type waitresses = waitress list;
    fun total ([]: waitresses) = 0
    | total(W::WL) = (income W) + (total WL);
    ```

- You can also use wild cards ...
  - E.g.
    ```
    fun costly({price:int, …}: footype) = price > 100.0;
    ```
What does ML infer about this function?

- **fun length L =**
  - if (null L) then 0
  - else 1 + length(tl L);

- length[1,2,3,4]
  > val it = 4 : int
- length["ab","cd","xy"];
  > val it = 3 : int
- length[[1,2],[3,4],[123,123,222],[1]]
  > val it = 4 : int

- Seems length has/accept these types
  - int list \(\rightarrow\) int
  - String list \(\rightarrow\) int
  - int list list \(\rightarrow\) int

- Obviously, we would like length to apply to any kind of list.
• What does ML infer about this function?
  - `fun length L =`
    `if (null L) then 0`
    `else 1 + length(tl L);`
  - `length[1,2,3,4]`
    `> val it = 4 : int`
  - `length["ab","cd","xy"];`
    `> val it = 3 : int`
  - `length[[1,2],[3,4],[123,123,222],[1]];`
    `> val it = 4 : int`

  – In ML, `length` has all of these types. This is written as `length : 'a list -> int`
    • `'a` is a type variable. It stands for any type
    • This means that the input to `length` is a list of items all of type `'a` where `'a` can be `int`, `string`, `int list`, or any other type.

  – In fact, that’s what ML infers for this function
    - `fun length L = if (null L) then 0 else 1 + length(tl L);`
    `> val length = fn : 'a list -> int`
ML: polymorphism

- **Greek:** *poly* = many, *morph* = form

- **Definitions:**
  - **Polymorphism:**
    - dictionary.com: the capability of assuming different forms; the capability of widely varying in form. The occurrence of different forms, stages, or types
    - Software: a value/variable can belong to multiple types
  - **Monomorphism:**
    - Dictionary.com: having only one form, same genotype…
    - Software: every value/variable belongs to exactly one type

- **Why is useful?**
  - We would have to define to define many different kinds of *length* functions:
    - int-length : int list \(\rightarrow\) int
    - real-length: real list \(\rightarrow\) int
    - string-length: string list \(\rightarrow\) int ………..  
    - And the code for each of these functions would be virtually identical!
  - Polymorphism adds flexibility & great convenience…. but…
ML: polymorphism types

• **Ad-hoc polymorphism:**
  – Different operations on different types known by the same name (*also called overloading*)
  – E.g. `3.0 + 4` compiler/interpreter must change 4 to 4.0 first

• **Inheritance polymorphism:**
  – Use sub-classing to define new versions of existing functions (*OO*)
  – E.g.:
    ```java
    public class Shape{
        public void draw( int x, int y){
            // do nothing
        }
    }
    
    public class Rectangle extends Shape{
        public void draw( int x, int y){
            // draws a rectangle
        }
    }
    
    public class Circle extends Shape{
        public void draw( int x, int y){
            // do nothing
        }
    }
    
    public class Shape{
        public void draw( int x, int y){
            // do nothing
        }
    }
    ```

    Shape myShape;
    myShape = new Rectangle( );
    myShape.draw( );
    myShape = myCircle;
    myShape.draw( );
    ```
Parametric Polymorphism (ML):
- Allows types to be parameters to functions and other types.
- Basic idea is to have a type variable…
- Type of function depend on type of parameter
- Implementation (ML):
  - One copy of code is generated
  - Polymorphic parameters must internally be implemented as pointers
Polymorphic functions are very common in ML:

- **fun** id X = X;  
  > val id = fn : 'a -> 'a

- **fun** listify X = [X];  
  > val listify = fn : 'a -> 'a list

- **fun** double X = (X,X);  
  > val double = fn : 'a -> 'a * 'a

- **fun** inc(N,X) = (N+1,X);  
  > val inc = fn : int * 'a -> int * 'a

- id 7;  
  > val it = 7 : int

- id "abc";  
  > val it = "abc" : string

- listify 3;  
  > val it = [3] : int list

- listify 7.3;  
  > val it = [7.3] : real list

- double "xy";  
  > val it = ("xy","xy") : string * string

- double [1,2,3];  
  > val it = ([1,2,3],[1,2,3]) : int list * int list

- inc (2,5);  
  > val it = (3,5) : int * int

- inc (4,(34,5));  
  val it = (5,(34,5)) : int * (int * int)
Polymorphic functions are very common in ML:

- `fun swap(X,Y) = (Y,X);`
  - `val swap = fn : 'a * 'b -> 'b * 'a`
  - `swap ("abc",7);`  
    - `val it = (7,"abc") : int * string`
  - `swap (13.4,[12,3,3]);`
    - `val it = ([12,3,3],13.4) : int list * real`

- `fun pair2list(X,Y) = [X,Y];`
  - `val pair2list = fn : 'a * 'a -> 'a list`
  - `pair2list(1,2);`
    - `val it = [1,2] : int list`
  - `pair2list(1,"cd");`
    - `?`

- `fun apply(Func,X) = Func X;`
  - `val apply = fn : ('a -> 'b) * 'a -> 'b`
  - `apply (hd, [1,2,3]);`
    - `val it = 1 : int`
  - `apply (length, [23,100]);`
    - `val it = 2 : int`

- `fun applytwice(Func,X) = Func(Func X);`
  - `val applytwice = fn : ('a -> 'a) * 'a -> 'a`
  - `applytwice (square,3);`
    - `val it = 81 : int`
  - `applytwice (tl, [1,2,3,4]);`
    - `?`
  - `applytwice (hd, [1,2,3,4]);`
    - `?`
ML: polymorphism – cont’d

- **Operators that restrict polymorphism**
  - Arithmetic operators: +, -, *, and –
  - Division-related operations such as /, div and mod
  - Inequality comparison operators: <, <=, >=, and >
  - Boolean connectives: andalso, orelse and not
  - String concatenation operator: ^
  - Type conversion operators
    - E.g. ord, chr, real, str, floor, ceiling, round, truncate,…

- **Operators that allow polymorphism**
  - Tuple operators
  - List operators
  - Equality operators = and <>
Exceptions: introduction

- An exception is any unusual event, erroneous or not, that is detectable either by hardware or software and that may require special processing.

- The special processing that may be required by the detection of an exception is called exception handling. This processing is done by a code unit called the exception handler.

- Why do we need exceptions if the language is strongly typed?
  - A strongly typed compiler/interpreter only checks the types of parameters and not their values.
  - In a language without exception handling: when an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated.
  - In a with exception handling: programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing.
Exceptions: execution flow

Executing code

```
... begin ...
end; ...
```

Exception is raised

some statement;

Exception to handler binding?

Exception handlers

```
when ...
begin ...
end;
when ...
begin ...
end;
when ...
begin ...
end;
```

Termination

Continuation
Exceptions: why?

- **How was error handling done in early programming languages?**
  - Send an auxiliary parameter or use the return value to indicate the return status of a subprogram (e.g. C standard library functions)
    ```c
    nError = foo(....);
    if( nError == -1){ // error
    }
    else{ // no error, continue normally
    }
    ```
  - Pass a label parameter to the subprogram. If an error occurs, use the label to jump to another location in the program (e.g. FORTRAN)
  - Pass an error-handler subprogram to the called subprogram.

- **Advantages of built-in exceptions:**
  - Error detection code is tedious to write and it clutters the program
  - Exception propagation allows a high level of reuse of exception handling code
ML: exceptions

- Syntax
  ```ml
  exception <exception-name> of <type-expression>
  ```

- Example:
  ```ml
  exception NegArg of int;
  fun fact N = 
    if N = 0 then 1
    else if N > 0 then N * fact(N-1)
    else raise NegArg(N);
  val fact = fn : int -> int
  fact(5);
  val it = 120 : int
  fact(~5);
  uncaught exception NegArg raised at: …
  ```
ML: exceptions

• **How to handle an exception?**
  
  – **Syntax**
    
    `<expression>`
    handle `<exception_1>` => `<exception-handler_1>`
    | `<exception_2>` => `<exception-handler_2>`
    | ....
    | `<exception_n>` => `<exception-handler_n>`
  
  – If no exceptions are raised, then return the value of `<expression>`
  – If `<exception_i>` is raised then return the value of `<exception-handler_i>`
    
    • Only the first matching exception is considered.

• **Example:**

  - exception Negative of int;
  - exception TooBig of int;
  - fun comb (N,M) =
    
    if N < 0 then raise Negative(N)
    else if M < 0 then raise Negative(M)
    else if M > N then raise TooBig(M)
    else
      fact(N) div (fact(M) * fact(N-M));
  
  > val comb = fn : int * int -> int

  - fun mycomb (N,M) =
    
    comb(N,M)
    handle Negative(X) => ~1
    | TooBig(M) => 0;

  > val mycomb = fn : int * int -> int

  > val mycomb = fn : int * int -> int

  - mycomb(11,8);

  > val it = 165 : int

  - mycombt(~5,123);

  > val it = ~1 : int
Suppose f calls g calls h, and h raises an exception: g handler is used

Example:

- exception e1;
- exception e2;
- exception e3;
- fun h(1) = raise e1
  |   h(2) = raise e2
  |   h(3) = raise e3
  |   h(_) = "ok";
- fun g(N) = h(N)
  handle e2 => "error g2"
  |   e3 => "error g3";
- fun f(N) = g(N)
  handle e1 => "error f1"
  |   e2 => "error f2";

- f(4);
  > val it = "ok" : string
- f(3);
  > val it = "error g3" : string
- f(2);
  > val it = "error g2" : string
- f(1);
  > val it = "error f1" : string
- f(0);
  > val it = "ok" : string