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
  \[
  \{ \text{field}_1: \text{type}_1, \text{field}_2: \text{type}_2, \ldots, \text{field}_n: \text{type}_n \}
  \]

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

• A record is a structured data type in which each element is accessed by a unique name.
  – E.g. \{ name: string, age: int, 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.**
      ```ml
      type waitress = { name: string, wages: int, tips: int };
      ```
  - Named types can be used anywhere that an unnamed types can.
    - **E.g.**
      ```ml
      fun income (w: waitress) =
      #wages w + #tips w;
      ```

- **Named typed can be used in type declaration**
  - **E.g.**
    ```ml
    type waitresses = waitress list;
    ```
    - ```ml
    [{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**
    ```ml
    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);
    ```
    - ```ml
    [{name="sally", wages= 20, tips=10},
     {name="alice", wages= 15, tips=20},
     {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  
  (obj₁, obj₂, …)

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

- In general, the tuple (v₁,v₂,…) is a shorthand for {1=v₁,2=v₂,…}
  
  - That’s why #2(6,7,”abc”) works!
**ML: pattern matching on records**

- **Recall syntax**
  ```
  fun <func> <pattern_i> = <expression_i>
  | <func> <pattern_j> = <expression_j>
  ........
  | <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;
    ```

**ML: ...**

- **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 -> int
    - String list -> int
    - int list list -> 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 \rightarrow\space 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 \rightarrow\space 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\space int
    • real-length: real list \rightarrow\space int
    • string-length: string list \rightarrow\space 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{
    }
    ```

```java
Shape myShape;
myShape = new Rectangle();
myShape.draw();
myShape = myCircle;
myShape.draw();
```

ML: polymorphism types – cont’d

• 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
**ML: polymorphism – cont’d**

- 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)
  ```

- `fun swap(X,Y) = (Y,X);`
  - `val swap = fn : 'a * 'b -> 'b * 'a`
- `fun pair2list(X,Y) = [X,Y];`
  - `val pair2list = fn : 'a * 'a -> 'a list`
- `fun apply(Func,X) = Func X;`
  - `val apply = fn : ('a -> 'b) * 'a -> 'b`
- `fun applytwice(Func,X) = Func(Func X);`
  - `val applytwice = fn : ('a -> 'a) * 'a -> '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
  ```

  ```
  pair2list(1,2);
  val it = [1,2] : int list
  ```

  ```
  apply (hd, [1,2,3]);
  val it = 1 : int
  ```

  ```
  apply (length, [23,100]);
  val it = 2 : int
  ```

  ```
  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

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)
  ```
  nError = foo(...);
  if (nError == -1) { // error
      // error handling code
  } else { // no error, continue normally
      // normal code
  }
  ```
  - 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);
```

```ml
val fact = fn : int -> int
```

```ml
fact(5);
val it = 120 : int
```

```ml
fact(~5);
val it = uncaught exception NegArg raised at: ...
```

ML: exceptions

• How to handle an exception?

  – Syntax

```ml
<expression>
handle <exception> => <exception-handler1>
| <exception> => <exception-handler2>
| ...
| <exception> => <exception-handler_n>
```

  – If no exceptions are raised, then return the value of `<expression>`
  – If `<exception>` is raised then return the value of `<exception-handler>`

    • Only the first matching exception is considered.

• Example:

```ml
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));
```

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

```ml
mycomb (11,8);
val it = 165 : int
```

```ml
mycomb (~5,123);
val it = ~1 : int
```
ML: exceptions & scopes

- 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

f  
<expression>  
handle  
<exception>  =>  
<exception-handler>  
g  
<expression>  
handle  
<exception>  =>  
<exception-handler>  
h  
......  
raise <exception>  
......