CSCI3180: Principles of Programming Languages

Declarative Programming (Part 2)

Wallace Mak

2022-23 Term 2

Topics

- Introduction to Functional Programming
- Functional Programming in ML
- Summary

Introduction to Functional Programming

- Functional programming supports another form of declarative programming
  - The building blocks are “true” functions, i.e., mathematical functions
- Mathematical functions describe the transformation of input values to output values with the environment in which the function is used
- From the programming point of view, this is attractive in that functional programs look like the kind of hierarchical specifications so often used in software-engineering

Functional Programming in ML

- The conceptual model of functional programming is that of a pocket calculator
  - You enter an expression, the calculator returns its value

```
- 2 + 2;
val it = 4 : int
- it;
val it = 4 : int
```

```
- exp(0.5);
val it = 1.64872127070013 : real
- sin(1.0472)*sin(1.0472)+cos(1.0472)*cos(1.0472);
val it = 1.0 : real
```

What is `it` ?

Functional Programming in ML

- Standard ML (SML), OCaml, and F# are common ML (Meta Language) dialects
  - We will be using SML for learning ML and functional programming in this course
  - Note that SML is considered as an impure functional programming language
    - Some of its features can produce side-effects
    - We will focus on the pure functional programming part
  - The concepts and principles learned can be applied to other dialects and functional programming in general

No Variables Allowed

- In functional programming, there is no variable like those we have in imperative programming, which support destructive assignments
- However, we can introduce names and assign values to them
- While you can reuse the same name at a later time if you must, you are not allowed to mutate the value
  - You are essentially creating a new named value which has the same name

```
- val b = exp(0.5);
val b = 1.64872127070013 : real
- val bb = ln(b);
val bb = 0.5 : real
- val b = "haha";
val b = "haha" : string
```

```
- val s = sin(1.0472);
val s = 0.866026628183543 : real
- val c = cos(1.0472);
val c = 0.499997879272546 : real
- s*s+c*c;
val it = 1.0 : real
```

Functional Value

- To create a functional value:

```
- fn((x:real),(y:real)) => x*x+y*y;
val it = fn : real * real -> real
```

- The value of the last expression is, although without a name, a function
  - This value should be distinguished from any of the results obtained by applying the function to an argument
- We can apply the last functional expression to an argument:

```
- fn((x:real),(y:real)) => x*x+y*y (s,c);
val it = 1.0 : real
- fn((x:real),(y:real)) => x*x+y*y (5.0,2.0);
val it = 29.0 : real
```

Functional Value

- To avoid repeatedly typing a lengthy expression, we name it:

```
- val sumsq = fn((x:real),(y:real)) => x*x+y*y;
val sumsq = fn : real * real -> real
- sumsq(s,c);
val it = 1.0 : real
- sumsq(5.0,2.0);
val it = 29.0 : real
```

- The form “`val name = fn …`” is used so often that a special purpose notation exists in ML:

```
- fun sumsq ((x:real),(y:real)) = x*x+y*y;
val sumsq = fn : real * real -> real
```

Typing in ML

- Our use of expressions, such as `5.0`, may have given you the impression that ML is rather picky about types
  - E.g., the following is unacceptable because multiplication is either between integers or between reals, but not between an integer and a real

```
- 1*1.0;
ERROR
```

What would you actually see if you execute `1*1.0;` in ML?

Typing in ML

- There is a way in which we can transform the integer 1 to its real counterpart

```
- (real 1)*1.0;
val it = 1.0 : real
```

- Here, we have applied a built-in function of type `int -> real`, as can be verified with the following

```
- real;
val it = fn : int -> real
```

- ML is a highly strongly-typed language in that it requires types of operators and operands to be consistent
  - No coercion is allowed at all!

Do you remember what is a coercion?

Typing in ML

- Basic types
  - int
    - Examples: `0`, `1337`, `~3`
      - Negative values are denoted using the `~` unary operator
    - Operations: `+`, `-`, `*`, `div`, `mod`
  - real
    - Examples: `3.14`, `2E4`, `~123.4`
      - 2E4 is 20000.0
    - Operations: `+`, `-`, `*`, `/`

What is the difference between `/` and `div` ?

Typing in ML

- bool
  - Examples: `true`, `false`
  - Operations: `not`, `and`, `or`, `andalso`, `orelse`
    - `andalso` and `orelse` use short-circuit evaluation whereas `and` and `or` do not
- string
  - Examples: `"doge"`, `"is"`, `"awesome"`
  - Operation: `^`

```
- "doge" ^ "is" ^ "awesome";
val it = "dogeisawesome" : string
```

Type Inference

- In many cases, ML can infer the types of an expression without the user declaring any type
- In some cases, however, we must give ML some clues as to what types are intended for the arguments for functions
  - Because some operators, like the arithmetic or comparison operators, are overloaded - they apply to values of several types
  - The type will be defaulted to a default type if it cannot be inferred
    - E.g. in SML, the default type for `*`, `+`, `-` is `int`

```
- fun sq x = x*x;
val sq = fn : int -> int
- sq 3;
val it = 9 : int
- sq 3.3;
ERROR
```

What is that error exactly?

Type Inference

- Although ML is picky about typing, it is also rather generous in not insisting on redundant type constraints
  - E.g., all of the following define the same function due to type inference

```
- fun sq (x:int) = x*x;
val sq = fn : int -> int
```

```
- fun sq x = x*(x:int);
val sq = fn : int -> int
```

```
- fun sq x = (x:int)*x;
val sq = fn : int -> int
```

```
- fun sq x = (x*x):int;
val sq = fn : int -> int
```

Conditional Expression

- Here is an example of a free-standing conditional expression

```
- if (floor (323.43*sin(1.0)) mod 2) = 0
=      then "even" else "odd";
val it = "even" : string
```

- The example is not typical but is effective in illustrating that a conditional expression is, after all, an expression
- However, conditional expressions are usually in functions

```
- fun abs x = if x >= 0 then x else ~x;
val abs = fn : int -> int
- abs(4-7);
val it = 3 : int
- fun negative x = x < 0;
val negative = fn : int -> bool
- negative(~3);
val it = true : bool
- fun div6 n = n mod 2 = 0 andalso n mod 3 = 0;
val div6 = fn : int -> bool
- div6(12);
val it = true : bool
- fun anniversary age = age mod 10 = 0 orelse age mod 25 = 0;
val anniversary = fn : int -> bool
- anniversary(60);
val it = true : bool
- fun nonneg x = not(negative x);
val nonneg = fn : int -> bool
- nonneg(0);
val it = true : bool
```

How do you do equality comparison in ML?

Recursive Functions, Tuples

- Conditional expressions are often used in defining recursive functions

```
- val rec fact = fn n =>
=     if n <= 0 then 1
=     else n*fact(n-1);
val fact = fn : int -> int
```

```
- fun fact n = if n <= 0 then 1
=     else n*fact(n-1);
val fact = fn : int -> int
- fact(4);
val it = 24 : int
```

- Values can be combined into tuples
  - For example, we can represent a point in the 3D-space as a tuple of coordinates

```
- val origin = (0.0,0.0,0.0);
val origin = (0.0,0.0,0.0) : real * real * real
- fun length (x,y,z) = Math.sqrt(x*x+y*y+z*z);
val length = fn : real * real * real -> real
```

sqrt is a Math library function

Tuples

- Note how the type of `x`, `y`, `z` and `length` is determined by the type of the built-in Math library function `sqrt`
- Note also that every function in ML is in fact a one-argument function
  - While a function looks like a multi-argument function, the only argument it has is just a tuple

```
- length(1.0,1.0,1.0);
val it = 1.73205080757 : real
- length origin;
val it = 0.0 : real
```

Selector Functions

- We could also have used selector functions that explicitly select the components of a tuple, as defined in the following
- The underscore _ is the wildcard symbol used in pattern matching

```
- fun first (x,_,_) = x;
val first = fn : 'a * 'b * 'c -> 'a
- fun second (_,y,_) = y;
val second = fn : 'a * 'b * 'c -> 'b
- fun third (_,_,z) = z;
val third = fn : 'a * 'b * 'c -> 'c
- fun sqr (x:real) = x*x;
val sqr = fn : real -> real
- fun len p = sqrt(sqr(first p)+sqr(second p)+sqr(third p));
val len = fn : real * real * real -> real
- len(1.0,1.0,1.0);
val it = 1.73205080757 : real
```

Local Declaration

- The formula of Heron

$$Area = \sqrt{p(p-a)(p-b)(p-c)} \quad \text{where} \quad p = \frac{a+b+c}{2}$$

  giving the area of a triangle in terms of its sides a, b, and c is known from antiquity

- It is interesting to see how the auxiliary quantity p can simplify a formula and therefore a program
- The quantity p is only relevant for the computation of $Area$ and should therefore be local to the function

Local Declaration

- We use a new construct: local declaration with `let`

```
- fun area (a,b,c) = let val p = (a+b+c)/2.0 in
=     sqrt(p*(p-a)*(p-b)*(p-c))
= end;
val area = fn : real * real * real -> real
- val sides1 = (1.0,2.0,3.0);
val sides1 = (1.0,2.0,3.0) : real * real * real
- val sides2 = (3.0,4.0,5.0);
val sides2 = (3.0,4.0,5.0) : real * real * real
- area sides1;
val it = 0.0 : real
- area sides2;
val it = 6.0 : real
```

Local Declaration

- Declarations can be local to any expression, not only to function declarations

```
- val (a,b,c) = (3.0,4.0,5.0);
val a = 3.0 : real
val b = 4.0 : real
val c = 5.0 : real
- let val p = (a+b+c)/2.0 in
= sqrt(p*(p-a)*(p-b)*(p-c))
= end;
val it = 6.0 : real
- fun anniversary age =
=     let fun divides (x,y) = y mod x = 0 in
=     divides(10,age) orelse divides(25,age)
= end;
val anniversary = fn : int -> bool
- anniversary 11;
val it = false : bool
```

Local Declaration

- Here, a global variable anniversary was introduced
- Note that it is not necessary and not desirable if its value is not needed later on
- So always be aware of the alternative

```
- (let fun divides(x,y) = y mod x = 0 in
=       fn age => divides(10,age) orelse divides(25,age)
= end) 10;
val it = true : bool
```

- The last expression contains only local declarations
  - After its evaluation, it leaves not a single name defined

User-Defined Types: Enumeration

- So far, we have only encountered functions using built-in types
- ML supports user-defined types
- The following simple example is basically an enumeration type

```
- datatype DIRECTION = North | East | South | West;
datatype DIRECTION = East | North | South | West
- val dir = East;
val dir = East : DIRECTION
```

Do you remember what is an enumeration type?

Case Expression

- The `case` expression is often used naturally in conjunction with enumeration types

```
- case dir of North => 0
=    |          East  => 90
=    |          South => 180
=    |          West  => 270;
val it = 90 : int
```

- A common use of patterns is in a function

```
- (fn dir =>
=   case dir of North => 0
=     |          East  => 90
=     |          South => 180
=     |          West  => 270
= ) East;
val it = 90 : int
```

What if you miss some of the cases?

Case Expression

- ML allows a shorthand for such use

```
- (fn North => 0
=    | East  => 90
=    | South => 180
=    | West  => 270
= ) East;
val it = 90 : int
```

Case Expression

```
- datatype SUIT = Spades | Hearts | Diamonds | Clubs;
datatype SUIT = Clubs | Diamonds | Hearts | Spades
- fun gt_suit(_,Spades) = false
=   | gt_suit(Spades,_) = true
=   | gt_suit(Clubs,_)  = false
=   | gt_suit(_,Clubs)  = true
=   | gt_suit(s1,s2)    = s1 = Hearts andalso s2 = Diamonds;
val gt_suit = fn : SUIT * SUIT -> bool
- fun gt_card((s1,v1),(s2,v2)) =
=    (v1:int) > v2 orelse
=    (v1 = v2 andalso gt_suit(s1,s2));
val gt_card = fn : (SUIT * int) * (SUIT * int) -> bool
- gt_card((Clubs,12),(Spades,12));
val it = false : bool
```

Do you think the order of the cases is important?

User-Defined Types: Union

- We can also define a more complicated type which is basically a union type

```
- datatype num = i of int | r of real;
datatype num = i of int | r of real
- val a = i(3);
val a = i 3 : num
- val b = r(4.0);
val b = r 4.0 : num
```

Do you remember what is a union type?

User-Defined Types: Polymorphic Types

- We can also define polymorphic data types

```
- datatype 'a bTree = empty | node of 'a bTree * 'a * 'a bTree;
datatype 'a bTree = empty | node of 'a bTree * 'a * 'a bTree
- fun btMem(e,empty) = false
=   | btMem(e,node(left,r,right)) = e = r orelse
=          btMem(e,left) orelse btMem(e,right);
val btMem = fn : ''a * ''a bTree -> bool
- val t = node(node(empty,2,node(empty,4,empty)),
=              1,
=              node(node(empty,5,empty),3,empty));
val t = node (node (empty,2,node #),1,node (node #,3,empty)) : int bTree
- btMem(3,t);
val it = true : bool
- btMem(0,t);
val it = false : bool
```

List

- List is an important data structure in ML
- The empty list `[]` is a list
- Every non-empty list L consists of:
  - A first element H (the head of L), and
  - A list T (the tail of L) consisting of zero or more remaining elements
- The head and tail are combined by the `cons` operator `::`, written as $H :: T$
  - E.g.
    - `1 :: []`
    - `2 :: (1 :: [])`

List

- The elements of a non-empty list are ordered by the rule according to which the head *H* precedes all elements of the tail *T* (if any)
- There is an alternative notation for lists consisting of the elements in order between square brackets, separated by commas
  - E.g.:
    - `2 :: (1 :: [])` is the same as `[2,1]`
    - `3 :: (4 :: (5 :: (6 :: [])))` is the same as `[3,4,5,6]`

```
- 2 :: (1 :: []) ;
val it = [2,1] : int list
- 3 :: (4 :: (5 :: (6 :: []))) ;
val it = [3,4,5,6] : int list
```

List

- Lists in ML are homogeneous: elements of a list must be of the same type

```
- [1,2,3];
val it = [1,2,3] : int list
- ["husky","shiba"];
val it = ["husky","shiba"] : string list
- [[1,2,3],[4,5,6]];
val it = [[1,2,3],[4,5,6]] : int list list
- [1,true];
ERROR
```

Pattern Matching for Lists

- An alternative to “if-then-else” with selectors is the use of pattern matching

```
- fun sumList [] = 0
=    | sumList(a::list) = a + sumList list;
val sumList = fn : int list -> int
- sumList [1,2,3,4];
val it = 10 : int
```

Operating on Lists

- The built-in functions `hd` (for "head") and `tl` (for "tail") make it possible to decompose a list into its two components

```
- hd([3,4,5,6,7,8,9]);
val it = 3 : int
- tl([3,4,5,6,7,8,9]);
val it = [4,5,6,7,8,9] : int list
- hd(tl(tl(tl([3,4,5,6,7,8,9]))));
val it = 6 : int
```

- The following is a function to compute the sum of elements in a list of integers

```
- fun sumList list = if null list then 0
=          else (hd list) + sumList(tl list);
val sumList = fn : int list -> int
```

Operating on Lists

- The @ operator can be used to append a list to another

```
- [1,2,3] @ [4,5,6,7];
val it = [1,2,3,4,5,6,7] : int list
```

```
- it @ [42] @ [~9];
val it = [1,2,3,4,5,6,7,42,~9] : int list
```

```
- [1,2,3] @ [4.0];
ERROR
```

Why won't the last expression work?

Higher-Order Functions

- The most dominant feature of functional programming is the notion of higher-order functions, which are functions that take functions as arguments and/or produce functions as values
- Functions are first-class citizens in ML
  - They have the same status as other values

Higher-Order Functions

- Suppose we want to have a function that takes a function f as input and return another function that always returns the twice the result of f

```
- fun double f = fn x => 2 * f(x);
val double = fn : ('a -> int) -> 'a -> int
- fun inc x = x + 1;
val inc = fn : int -> int
- fun inc2 x = x + 2;
val inc2 = fn : int -> int
- double inc 3;
val it = 8 : int
- double inc2 3;
val it = 10 : int
- double inc;
val it = fn : int -> int
```

Higher-Order Functions

- A very interesting function is the `map` function that takes as input a function `f` and a list $[a_1, \ldots, a_n]$, and produces the list $[f(a_1), \ldots, f(a_n)]$ as output

```
- fun map(f,[]) = []
= | map(f,x::xs) = f(x)::map(f,xs);
val map = fn : ('a -> 'b) * 'a list -> 'b list
- fun inc x = x + 1;
val inc = fn : int -> int
- map(inc,[1,2,3,4]);
val it = [2,3,4,5] : int list
```

Higher-Order Functions

- `map` is actually a built-in function, but the built-in one needs to be without the round brackets around the arguments
  - The reason is that it uses currying in its definition
    - Currying is out of the scope for this course
  - You may try the following before you define the map function yourself
    - It works the same as the previous example with a slightly different syntax

```
- fun inc x = x + 1;
val inc = fn : int -> int
- map inc [1,2,3,4];
val it = [2,3,4,5] : int list
```

Higher-Order Functions

- Another interesting function is the `foldl` function
  - It takes as input a function `f`, an initial value x, and a list $[a_1, \ldots, a_n]$
  - It produces as output the value $f(a_n,...,f(a_2, f(a_1, x))...)$
  - Or simply x if the list is empty

```
-  fun add(x,y) = x + y;
val add = fn : int * int -> int
- foldl add 0 [2,3,4,5];
val it = 14 : int
```

- There are other common higher-order functions which are often supported readily in many functional programming languages, such as `foldr` and `filter`

Summary

- With declarative programming, we focus on declaring “what” the problem is instead of going into exact details on “how” to solve it
  - It is very different from imperative programming in which most of the effort is put on giving exact step-by-step machine instructions on how to solve the problem from the machine point-of-view
- Logic programming and functional programming have been discussed via practicing them in Prolog and ML
  - However, the principles and concepts apply in general to those paradigms