CIS 425 - ML

1 ML Background

ML (which stands for Meta Language) is a functional language with imperative features, it also has a concurrent extension and a object-oriented extension. OCaml is essentially the same thing as ML but has different syntax (used by Jane Street `http://www.janestreet.com`).

ML is an expression oriented language, instead of being statement oriented as the languages you have seen so far; it always returns something, even if your program is only a simple assignment.

Example 1.1. Everything Has a Return Value

Consider the following ML expression:

```
2 + 3;
```

Since ML is an expression oriented language this must have a return value, once this program is compiled we see that the output is:

```
val it = 5 : int
```

The compiler provides the name *it* since we didn't provide a name ourselves. Also notice how the compiler determined that the type was an **int**; this will be discussed further in the next section.

A distinguishing feature of ML is its type system, which unlike the C type system, does not allow any loopholes. The type system is sound in a precise mathematical sense. If you have an expression which returns a pointer to a string:

```
  e :   pointer string
```

then if `e` terminates "normally" then `e` cannot return a dangling pointer, that is, a pointer to some memory which has been deallocated, or a pointer to an `int`. What if `e` raises an error? We will see that there are two schools:

```
ML          1 + raise ERROR   : int
Haskell     1 + raise ERROR   : int with the possibility of errors
                                The side effects appears in the type
```

The type system is not overwhelming since most of the times the types can be automatically inferred.

ML was designed by Milner (Turing award 1991) in the context of the LCF project, "Logic for Computable Functions". His goal was to build a system which would allow one to prove properties of functional programs in an automated or semi-automated manner. It was used to write tactics (that is why they call it ML), which given a formula tries to find a proof of it:

```
    tactic :  formula -> proof
```

Another emphasis of ML is the use of higher-order functions. Examples are: map and reduce. Map: applies argument function to each element in a collection; Reduce: takes a collection, an initial value, and a function, and combines the elements in the collection according to the function. Google uses Map/Reduce to parallelize and distribute massive data processing tasks.

We will use SML (i.e. Standard ML) in the read-eval-print loop. You enter one expression or declaration at the time. The expression is then compiled and executed.

2 ML Types

Unlike lambda calculus or Javascript, ML has types.

ML uses something called *type inference*.

- *Type inference* refers to the automatic detection of the type of an expression. In other words, ML doesn't ask a user for a type unless it can't be derived by the compiler.
- As mentioned before, it is guaranteed that if a program has a type then that program cannot cause a segmentation fault or core dump.

ML has no *implicit coercion*.

- *Implicit coercion* refers to hidden type conversions with non-obvious side-effects that implicitly occur.

Basic types can can be described by the following grammar:

$$t ::= int \mid real \mid bool \mid string \mid \alpha$$

in other words, a type can be:

- int , e.g. 5
- real, e.g. 5.1
- bool, e.g. true, false
- string, e.g. "ciao", "bye"
- α (alpha)
  - good for when you don't know the type
  - also written as 'a
  - a type containing a type variable is called a polymorphic type
- unit
  - ML's version of the "void" type. However, there is a difference between unit and void: void represents the empty set $\{\}$ whereas unit represents a set with one element $\{()\}$.

You can think of types in terms of sets; we now introduce ways to form new sets from the predefined ones.

Structured types can be given by adding the following productions to our grammar of types:

$$t ::= t \star t \mid t \star t \star t \mid \cdots \mid t \rightarrow t$$

- product type, written as $t \star t$, corresponds to a pair. We can also have $t \star t \star t$. In general, we refer to the values of a product type as tuples. The product type corresponds to the cartesian product of sets.
- function type, written as $t_1 \rightarrow t_2$, corresponds to a mapping between the domain set t_1 and codomain t_2.

Example 2.1. Type Inference

Consider the following ML expression:

```
val x = 5;
```

Once this program is compiled we see that the output is:

```
val x = 5 : int
```

Here, notice how the input you give to the compiler doesn't have to specify a type. Instead, it will give one for you.

Example 2.2. Implicit Coercion

Consider the following ML code:

```
val x = 5;
val z = 5.0;
x +  z;
```

The final line would result in the error

```
operator and operand do not agree
operator domain: int * int
  operand:          int * real
  in expression:
    x + z
```

which says that `+` is expecting two integers, and instead you provide an integer and a real. If you want to add them, you will have to either convert the integer to a real, as so:

```
(Real.fromInt x)+z;
```

or convert the real to an integer, as so:

```
x + (round z);
```

Example 2.3. Unit

The only value of type unit is:

```
();
```

Once this program is compiled and ran, the output will be:

```
val it = () : unit
```

Example 2.4. Product Types

Consider the following ML expression:

```
(2, true, "ciao");
```

Once this program is compiled and ran, the output will be:

```
val it = (2, true, "ciao") :  int * bool * string
```

How to type If-Expressions

In ML there are two crucial rules to follow when defining if-expressions. Consider

If P then *then-clause* else *else-clause*

1. The condition P must be a boolean.
2. The type of the return value(s) in the *then-clause* and *else-clause* must be the same. That is, if t_1 is the type of the *then-clause* and t_2 is the type of *else-clause*, it must be the case that $t_1 = t_2$.

Example 2.5. Valid If-Expression

Consider the ML program below:

```
if true then "ciao" else "bye";
```

it returns:

```
val it = "ciao" : string
```

Here we see that P is true (bool), the *then-clause* is "ciao" (string), and the *else-clause* is "bye" (string). This follows both rule 1 stating that P must be a bool and rule 2 stating that *then-clase* and *else-clause* must be the same type (in this case string).

Example 2.6. Invalid If-Expression

Consider the ML program below:

```
if true then 5 else "bye";
```

During compilation this will produce a typing error. Rule 1 is satisfied since our P value is of type bool. However, rule 2 is not satisfied since the types of the *then-clause* is int which is not the same type as the type of the *else-clause* which is a string.

Example 2.7. Invalid If-Expression with Product Types

Consider the ML program below:

```
if true then (2, true, "ciao") else (3, 3, "bye");
```

Similar to example 3.2, during compilation this will produce a typing error. Why?

3 Functions

ML provides *Higher-Order functions*, that is, one can define functions which accepts other functions as arguments or returns a function as a result. Functions are also said to be *first-class citizens*. Functions in ML take only **one** parameter, and they are defined as follows:

$$\text{fun } <\textit{function name}> <\textit{function parameter}> = <\textit{function body}>;$$

Notice that no return keyword is necessary since everything must return something anyways in ML.

As in lambda-calculus, we are not forced to give a name to a function. An anonymous function is defined as

$$\text{fn } <\textit{function parameter}> => <\textit{function body}>$$

The type of a function can be denoted as follows:

$$\textit{input type} \rightarrow \textit{output type}$$

Example 3.1. Consider the ML program below:

```
fun succ x = x + 1;
```

Once compiled, this returns:

```
val succ = fn : int -> int
```

Note that *fn* denotes that the variable *succ* is a function. Additionally, the input type is recognized as an int since the only type that can add 1 to itself is an int. Similarly, the only type that can be produced from adding 1 is an int, therefore the output type must also be an int.

Example 3.2. Consider the ML program below:

```
fun foo x = if x then 1 else 2;
```

Once compiled, this returns:

```
val foo = fn : bool  ->  int
```

Here the compiler recognizes rule 1 from section 3 stating that the condition variable in an if expression must be of type bool, thus the input type is bool. The output type is recognized from the return values in the then/else section and is therefore an int.

Example 3.3. Consider the ML program below:

```
fun foo x = if true then x else 2;
```

Once compiled, this returns:

```
val foo = fn : int  ->  int
```

Here the compiler recognizes rule 2 from section 3 stating that the return values in the then/else section must be of the same type and therefore our input type is an int. Similarly, our output type must also be an int.

3.1 Polymorphism

Example 3.4. Consider the following function:

```
fun id x = x;
```

As you might be able to see off first glance, the compiler has no way of deriving an input and output type. For instance, this function could be of type *int* →*int*, *bool* →*bool*, *string* →*string*, etc. However, note that it CANNOT be of type *int* →*bool* or any other type where the input type and output type are not the same.

This is a situation where we do not know the type which means it's a good time to use the α type. Using α, we define the return type of the identity function to be:

$$\alpha \to \alpha$$

This allows our input and output type to be any type we want (as long as the input and output types match).

Example 3.5. Consider the recursive function below:

```
fun loop x = loop x + 1;
```

Once compiled, this returns:

```
val loop = fn : 'a ->  int
```

The compiler views the body of this function as *(loop x) + 1* meaning that x, the input variable can have any type since there's nothing explicit telling us it can't. However, the output type *must* be an int since we add 1 after *loop x* is evaluated.

Example 3.6. Consider the recursive function below:

```
fun loop x = loop (x + 1);
```

Once compiled, this returns:

```
val loop = fn : int -> 'a
```

The compiler will see that before executing *loop* again, x must be added to 1 so it has to be an int, therefore the input type is int. The output type cannot be identified so it must be α.

Example 3.7. Consider the recursive function below:

```
fun loop x = loop x;
```

Once compiled, this returns:

```
val loop = fn : 'a -> 'b
```

Both the input and output cannot be derived. Because of this, we start with defining the input to be α but now we ask: does the output have to be the same as the input? The answer in this situation is no. Since the output type can be different from the input type, and we already used α, it's time to go to the next Greek letter: β.

Example 3.8. Consider the following ML function definition:

```
fun apply (f, x) = f x;
```

What is the type of *apply*? We see that our function takes a tuple of two arguments, f and x. We see that we are applying the value f to x, so f must be a function. Since we do not know what the type of x is, nor what type f returns, we define the type of f as:

```
f : 'a -> 'b
```

Logically, x must be of type 'a, since that is what f takes in as arguments. Furthermore, we now have defined that f returns values of type 'b, so f applied to x must be a value of type 'b. This helps us to write the full type of *apply* as follows:

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

So, we see that the function *apply* is considered a higher-order function, because it is able to accept another function as an argument.

Example 3.9. Consider the same ML function, adding a call to apply:

```
fun apply (f, x) = f x;
apply (fn x => x+1, 5);
```

Note first that here, the function provided as the first argument to apply is an example of an *anonymous* function. *anonymous function* is exactly what it sounds like: a function without a name. Sometimes, all we need a function for is to pass it as an argument to another function, and so we can skip the need to define and name that function by using the **fn** expression.

In this example, the type of the anonymous function would in fact be

```
fn (x => x+1): int -> int
```

So, while in Example 3.8 we saw that we had:

```
f : 'a -> 'b,
```

It is possible that 'a and 'b are the same, as they are in this example, where 'a : int and 'b : int, but it is not **required**.

Example 3.10. What if we changed our call to $a$pply to be the following?

```
apply (fn x => x + 1, true);
```

This would lead to an error! Why?

Because in our call to $a$pply where we give the anonymous function:

```
fn x => x + 1
```

we are mandating that the type of f be

```
f : int -> int
```

but then instead providing f with an argument of type bool.

Example 3.11. Consider several successive calls to *apply* as follows:

```
fun apply (f, x) = f x;
apply (fn x => x+1, 5);
apply (fn x => true, false);
apply (fn x => x + 1.0, 5.2);
```

Will this result in an error? I.e, does the first call to apply permanently set the required type of f? **No! And that is the beauty of polymorphic instantiation.** All of the above calls are permitted, even one after the other.

Example 3.12. Let us consider a new function, called apply' (read: *apply prime*), which we will then call with specific arguments.

```
fun apply' f = fn x => f x;
```

What is the type of apply'? Well, we first see that apply' takes in an argument **f**. We also see that f is being applied to x, so f itself must be a function. We also see that the return of apply' is actually an anonymous function, and that this anonymous function accepts an argument x, and then applies f to x. This give us the type of apply' as follows:

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

Say we called apply' as follows:

```
val a =   apply' (fn x => x + 1);
```

What would this return? it will return a function:

```
val  a = fn : int -> int
```

which can then be invoked as so:

```
a 5
```

which returns **6.**

4 Currying

Currying is the technique of representing a function that takes multiple arguments into a sequence of functions that each takes a single argument.

Currying is what allows us to perform something called *partial application*, wherein not all the parameters to a curried function need to be provided all at once.

Example 4.1. For instance, consider the following function definition:

```
fun f x1 x2 x3 = e;
```

This is an example of a curried function, where the arguments `x1`, `x2` and `x3` do NOT need to be supplied all at the same time. The above function is said to be *syntactic sugar* for the following function:

```
fun f x1 = fn x2 => fn x3 => e;
```

In other words, the first definition entered by the user is translated to the second one.

As a reminder, application associates to the left:

```
e1 e2 e3
```

is to be interpreted as:

```
((e1 e2) e3)
```

Example 4.2. Consider another series of function definitions:

```
    fun sum (x, y, z) = x + y + z;
    fun add x y z = x + y + z;
    fun add' x => fn y => fn z => x + y + z;
```

As per Example 6, *add'* is simply syntactic sugar for *add*. But are sum and add equivalent? No, and to see why, we will look at their types and how we are able to call them. The type of *sum* is:

```
    val sum = fn : int * int * int -> int
```

What this means is that *sum* accepts three ints in a tuple as its argument, and returns a single int. On the other hand, the type of *add* is:

```
    val add = fn: int -> int -> int -> int
```

What does this mean? This means that, unlike *sum*, *add* does NOT need all of the three integers it accepts to compute its addition to be given all at once. In fact, trying to do something like:

```
    add (1, 2, 3);
```

Will result in an error.
Why? because you are supplying the function with a tuple of type (int * int * int), when we only asked for an int!

So how can we call add? In any of the following ways:

```
    fun add x = fn y => fn z => x + y + z;

    add 1;
    add 1 2;
    add 1 2 3;
```

Remember again that application is left to right: add 1 2 3 is to be interpreted as (((add 1) 2) 3)

But what are the values returned by these three function calls? The type of add is:

```
    val add = fn : (int -> (int -> (int -> int)))
```

The values of the calls to add are as follows:

```
    add 1     = fn y => fn z => 1 + y + z
    add 1 2   = fn z => 1 + 2 + z
    add 1 2 3 = 1 + 2 + 3
```

In summary, the add function shows us an example of what is known as partial application of a curried function, where not all the arguments need to be supplied at once. Functions in ML only take one parameter. While it may be a tuple, such as with the *sum* function, they are still only one argument.

Overloading

Consider the function below:

```
fun add(x, y) = x + y;
```

Once compiled, this returns:

```
val add = fn : int * int  -> int
```

The compiler knows that addition can only happen between ints and reals so it must choose one of these two types to associate with x and y. This defaults to ints and therefore the input type becomes int * int and the output type must be an int as well.

Note: to change this and return *real * real -¿ real* instead, simply add in *+ 0.0* to the function body, as so:

```
fun add(x, y) = x + y + 0.0;
```

which gives the type:

```
val add = fn : real * real -> real
```

We say that + is an overloaded symbol. You need to understand the difference between overloading and polymorphism.

5 Defining New Types

How do we define a new type? There are two steps to address:

```
1. Introduce the name of our new type
2. Define all possible ways to build new values of that type
```

Example 5.1. Say, for instance, we wanted to define the boolean datatype ourselves. We could do so as follows:

```
datatype bool = True | False;
```

By doing this, we define the name of our new type, *bool*, and we define all the possible ways to build new values of that type, by declaring that there are two potential values of type bool, either **True** or **False**.

After defining this datatype, we could then do the following:

```
True;
False;
```

Giving us two values:

```
val it = True : bool
val it = False : bool
```

But how do we define functions over these values? Through *pattern matching*.

Example 5.2. Say we wanted to define the boolean *not* function. This could be done as the following:

```
fun not True  = False
|   not False = True
```

With a type of

```
val not = fn : bool -> bool
```

This definition says that if we are given something of type bool with the value True, then *not* will return False. If instead we are given something of type bool with the value False, then *not* will return True.

Example 5.3. Defining the natural numbers. Say instead we wanted to consider defining a different dataype, one which gives us a definition of the natural numbers. How would we go about this?
First, we can consider the definition of natural numbers from discrete math to help us in our datatype definition.
Natural Numbers:

```
The set of natural numbers is defined inductively as follows:

Base case: 0 is a Nat
Otherwise: If n is a Nat, n+1 is a Nat.
```

This leads us to our definition of Nat as follows:

```
datatype Nat = Zero | S of Nat;
```

What does this mean? Well, we see that we have a new dataype called *Nat*, and that it has two possible ways of building a new Nat. Either we have something with the value *Zero*, or we have something which requires another constructor called *S*, which accepts something of type Nat.

The type of the constructor *S* is of course:

```
S : Nat -> Nat
```

and this allows us to build our dataype to effectively represent the natural numbers. We can use successive calls to our new constructor to build values which act like natural numbers:

```
val zero = Zero;
val one = S(Zero);
val two = S(Succ(Zero));
val three = S(two);
val four = S(three);
```

And so on and so forth.

Example 5.4. Consider one last dataype:

```
datatype foo = A of foo;
```

Would this be an effective definition of the foo datatype? No! Why?
Because there is no base case. The set of values of type foo is empty

Note that all values of a datatype must be finite. If you want to define infinite objects you need to get into coinductive datatype definitions, topic which is beyond this course.