CS312 Lecture 4
Polymorphism and Parameterized Types

Polymorphism

Type systems are nice but they can get in your way. In a lot of programming languages (e.g., Java) we find that we end up rewriting the same code over and over again so that it works for different types. SML doesn't have this problem, but we need to introduce new features to show how to avoid it. Suppose we want to write a function that swaps the position of values in an ordered pair:

```
fun swapInt(x: int, y: int): int*int = (y,x)
fun swapReal(x: real, y: real): real*real = (y,x)
fun swapString(x: string, y: string): string*string = (y,x)
```

This is tedious, because we're writing exactly the same algorithm each time. It gets worse! What if the two pair elements have different types?

```
fun swapIntReal(x: int, y: real): real*int = (y,x)
fun swapRealInt(x: real, y: int): int*real = (y,x)
```

And so on. There has to be a better way... and here it is:

```
- fun swap(x: 'a, y: 'b): 'b * 'a = (y,x)
val swap = fn : 'a * 'b -> 'b * 'a
```

Instead of writing explicit types for `x` and `y`, we write **type variables** `'a` and `'b`. The type of `swap` is `'a*'b -> 'b*'a`. What this means is that we can use swap as if it had any type that we could get by *consistently* replacing `'a` and `'b` in its type with a type for `'a` and a type for `'b`. We can use the new swap in place of all the old definitions:

```
swap(1,2);          (* swap : (int * int) -> (int * int) *)
swap(3.14,2.17);    (* swap : (real * real) -> (real * real) *)
swap("foo","bar");  (* swap : (string * string) -> (string * string) *)
swap("foo",3.14);   (* swap : (string * real) -> (real * string) *)
```

This ability to use swap as though it had many different types is known as **polymorphism**, from the Greek for "many shapes". If we think of swap as having a "shape" that its type defines, then swap can have many shapes: it is **polymorphic**. Notice that the requirement that type variables be substituted consistently means that some types are ruled out; for example, it is impossible to use swap at the type `(int*real) -> (string*int)`, because that type would consistently substitute for the type variable `'a` but not for `'b`.

ML programmers typically read the types `'a` and `'b` as "alpha" and "beta". This is easier than saying "single quotation mark a", and also they wish they could write Greek letters instead. In fact a type variable may be any identifier preceded by a single quotation mark; for example, `'key` and `'value` are also legal type variables. The ML compiler needs to have these identifiers preceded by a single quotation mark so that it knows it is seeing a type variable.

It's important to note that `swap` doesn't use its arguments `x` or `y` in any interesting way. It treats them as if they were black boxes. When the SML type checker is checking the definition of swap, all it knows is that `x` is of some arbitrary type `'a`. It doesn't allow any operation to be performed on `x` that couldn't be performed on an arbitrary type. This means that the code is guaranteed to work for any `x` and `y`. If we want some operations to be performed on values whose types are type variables, we have to provide them as function values. For example,

```
- fun appendString(x: 'a, s: string, toString: 'a->string): string =
    (toString x) ^ " " ^ s
val appendString = fn : 'a * string * ('a -> string) -> string
- appendString(312, "class", Int.toString)
val it = "312 class" : string
- appendString("three", "twelve", fn(s:string) => s)
val it = "three twelve" : string
```

Parameterized Types

The ability to write polymorphic code is pretty useless unless it comes with the ability to define data structures whose types depend on type variables. For example, last time we defined lists of integers as

```
datatype IntList = Nil | Cons of  int * IntList
```

But we'd like to be able to make lists of any kind of value, not just integers. (The built-in lists have this capability, of course). Further, using this definition of **`IntList`**, we can write lots of functions for manipulating lists, yet many of these functions don't depend on what kind of values are stored in the list. The **`length`** function is a good example:

```
fun length(lst: IntList): int =
    case lst of
      Nil => 0
    | Cons(_, rest) => 1 + (length rest)
```

We can avoid defining lots of list datatypes and associated operations by declaring a **parameterized datatype** instead:

```
datatype 'a List = Nil | Cons of 'a * 'a List
```

A parameterized datatype is a recipe for creating a family of related datatypes. The type variable **`'a`** is a **type parameter** for which any other type may be supplied. For example, **`int List`** is a list of integers, **`real List`** is a list of reals, and so on. However, **`List`** itself is not a type. Notice also that we cannot use **`List`** to create a list each of whose elements can be any type. All of the elements of a **`T List`** must be **`T`**'s.

```
val il : int List = Cons(1,Cons(2,Cons(3,Nil)))    (* [1,2,3] *)
val rl : real List = Cons(3.14,Cons(2.17,Nil))     (* [3.14,2.17] *)
val sl : string List = Cons("foo",Cons("bar",Nil)) (* ["foo","bar"] *)
val srp : (string*int) List =
    Cons(("foo",1),Cons(("bar",2),Nil))  (* [("foo",1), ("bar",2)] *)
val recp : {name:string, age:int} List =
    Cons({name = "Greg", age = 150},
         Cons({name = "Amy", age = 3},
              Cons({name = "John", age = 1}, Nil)))
```

Notice **`List`** itself is *not* a type. We can think of **`List`** as a function that, when applied to a type like **`int`**, produces another type (**`int List`**). A parameterized datatype is an example of a **parameterized type constructor**: a function that takes in parameters and gives back a type. Other languages have parameterized type constructors. For example, in Java you can declare a **parameterized class**:

```
class List<T> {
    T head;
    List <T> tail;
    ...
}
```

In SML, the only type constructors that can be parameterized are datatypes. This is a language design choice that works well for SML.

We can also define polymorphic functions that know how to manipulate any kind of lists:

```
(* is the list empty? *)
fun isEmpty(lst: 'a List): bool =
    case lst of
        Nil => true
      | _ => false;
```

```
(* return the length of the list *)
fun length(lst: 'a List): int =
    case lst of
      Nil => 0
    | Cons(_, rest) => 1 + (length rest)

(* append two lists:  append([a,b,c],[d,e,f]) = [a,b,c,d,e,f] *)
fun append(x: 'a List, y: 'a List): 'a List =
    case x of
       Nil => y
     | Cons(h,t) => Cons(h, append(t, y))

val il2 = append(il,il)
val il3 = append(il2,il2)
val il4 = append(il3,il3)
```

```
val sl2 = append(sl,sl)
val sl3 = append(sl2,sl2)

(* reverse the list:  reverse([a,b,c,d]) = [d,c,b,a] *)
fun reverse(x: 'a List): 'a List =
    case x of
        Nil => Nil
      | Cons(h,t) => append(reverse t, Cons(h,Nil));

val il5 = reverse(il4);
val sl4 = reverse(sl3);

(* apply the function f to each element of x:
 *    map f [a,b,c] = [f(a),f(b),f(c)] *)
fun map (f: 'a->'b) (x: 'a List): 'b List =
    case x of
        Nil => Nil
      | Cons(h,t) => Cons(f h, map f t)

val sl5 = map Int.toString il5

(* insert sep between each element of x:
 *    separate(s,[a,b,c,d]) = [a,s,b,s,c,s,d] *)
fun separate(sep: 'a, x: 'a List) =
    case x of
        Nil => Nil
      | Cons(h,Nil) => x
      | Cons(h,t) => Cons(h, Cons(sep, separate(sep,t)))

(* prints out a list of elements as long as we can convert the
 * elements to a string using to_string. *)
fun printList (toString: 'a -> string) (x: 'a List): unit =
    let val strings = separate(",", map toString x)
    in
        print("[");
        map print strings;
        print("]\n")
    end

fun printInts(x: int List): unit =
    printList Int.toString x

fun printReals(x: real List): unit =
    printList Real.toString x

fun printStrings(x: string List): unit =
    printList (fn s => "\"" ^ s ^ "\"") x
```

Lists are useful, but they are hardly the only use for type parameterization. For example, we can define a datatype for binary trees:

```
datatype 'a Tree = Leaf | Node of ('a Tree) * 'a * ('a Tree)
```

Abstract Syntax and datatypes

Earlier we noticed that there is a similarity between BNF declarations and datatype declarations. In fact, we can define datatype declarations that act like the corresponding BNF declarations. The values of these datatypes then represent legal expressions that can occur in the language. For example, consider a BNF definition of legal SML type expressions:

(base types) b ::= **int** | **real** | **string** | **bool** | **char**

(types) t ::= b | t **->** t | t_1 * t_2 *...* t_n | **{** x_1 **:** t_1**,** ..., x_n**:** t_n **}** | X

This grammar has exactly the same structure as the following datatype declarations:

```
type id = string
datatype baseType = Int | Real | String | Bool | Char
datatype mlType = Base of baseType | Arrow of mlType*mlType | Product of mlType List
                | Record of (id*mlType) List | DatatypeName of id
```

Any legal SML type expression can be represented by a value of type **Type** that contains all the information of the corresponding type expression. This value is known as the *abstract syntax* for that expression. It is abstract, because it doesn't contain any information about the actual symbols used to represent the expression in the program. For example, the abstract syntax for the expression **int*bool->{name: string}** would be:

```
Arrow( Product(Cons(Base Int, Cons(Base Bool, Nil))),
       Record(Cons(("name", Base String), Nil)))
```

The abstract syntax would be exactly the same even for a more verbose version of the same type expression: `((int*bool)->{name: string})`. Compilers typically use abstract syntax internally to represent the program that they are compiling. We will see a lot more abstract syntax later in the course when we see how ML works.