CIS 425 - ML (Continued)

1 Datatypes

To define a datatype in ML, we need to give the name and the constructor of the datatype.

```
datatype 'a myList = empty | B of 'a * 'a myList;
```

As we can see in this example, we give the name of the datatype `myList` and the constructors are `empty` and `B` :

```
empty;
val it = empty : 'a myList
B;
val it = fn : 'a * 'a myList -> 'a myList
```

As we can see here, we can give our datatype a polymorphic type: this list datatype that we just constructed could be a list of ints, reals, bools, or even any other datatype that we create.

We can make a list of ints

```
val a = B(5,empty);
```

or a list of bools.

```
val a = B(true, empty);
```

However, we cannot have a list of multiple datatypes. Once we have set α to a certain datatype, it must stay consistent.

```
val a = B(5, B(true, empty)); (* This will give an error! *)
```

However, we might want to be able to make a list of multiple datatypes. How would we be able to make a list of ints and bools?

To do that, we will need a new datatype! Just as how we invented the category "a Piece of Chex Mix" to include cereal, pretzels, bagel chips, and whatever else they put in chex mix, we can invent our own Bool and Int Mix.

```
datatype mix = I of int | C of bool;
```

(We are, slightly awkwardly, using `C` instead of `B`, since we already used `B` in our definition of the `myList` datatype.)

Now that we have our mix defined, we can finally represent our list of bools and ints.

```
val a = B(I 5, B(C true, empty)); (* No errors this time! *)
```

2 Map

We want to define the map function: that is a function that takes an inner function and a list, and returns a list where each element has that inner function applied to it.

In ML, we define functions using pattern matching. Recall our myList datatype. There were two patterns that defined that datatype: the empty list, and the list with an initial element paired with the rest of the list. For map, there are two patterns that we have to match: the empty list, and the list with a first element joined to the remainder of the list.

```
fun map f [] = (* something *)
  | map f (x::xs) = (* something else *)
```

With map, we want to apply `f` to every element of the list. If we apply `f` to the empty list `[]`, we should get back the empty list, since there are no elements. On the other hand, we can apply `f` to a non-empty list recursively, by applying it to the first element, and then calling map on the remainder of the list. Our implementation looks like this.

```
fun map f [] = []
  | map f (x::xs) = (f x) :: map f xs;
```

We could just as easily have passed the arguments as a tuple, in which case our function would look like this.

```
fun map (f, []) = []
  | map (f, (x::xs)) = (f x) :: map(f, xs);
```

The first function is curried, the second is uncurried. Note that both functions take just one argument. The curried version of `map` returns a function that acts on the list, as shown below:

```
fun map f   =  fn []        => []
              |  x :: xs    => (f x) :: map f xs;
```

while the uncurried version takes arguments as a single tuple.

2.1 Digression 1: Syntactic Sugar

When we talk about the list `[1,2,3]` in ML, this is really syntactic sugar for `1::2::3::nil`. Syntactic sugar is notation that makes programs easier to read and write, but doesn't actually add any functionality to the programming language.

2.2 Digression 2: Advantages of Partial Computations

Suppose we have the following function and function call.

```
fun f x y = x + 1;
val a = f 3;
```

What is the value of `a`? It is bound to the function `fn y => 3+1`. However, at this point the compiler could apply some optimizations and execute the addition instead of generating code for it. In this way, `a` will be bound to the function `fn y => 4`. That is, the computation inside of function f that did not depend on y has been computed, before we get the value of y. This way, we can call `a` as many times as we want, and it will not require performing the calculation many times.

3 Reduce

Like the map function, the reduce function will apply another function to every element of a list. However, this time, for each element of the list we want to update the value of another variable with the result of the function applied to the element, and that variable. Like the map function, we will need to consider the base case of the empty list and the inductive case of the non-empty list. Here is a proposed version of the reduce function:

```
fun reduce f [] a = a
  | reduce f (x::xs) a = f(x,a)::reduce f xs a; (* Will give us an
    error! *)
```

This function is wrong! How can we tell? One easy way is by considering the types. The base case will give us an int, the type of `a`, whereas the inductive case will give us a list (which isn't what we want reduce to give us anyway). In ML, the types of the different cases must match. The correct function is shown below.

```
fun reduce f [] a = a
  | reduce f (x::xs) a = reduce f xs (f(x,a));
```

(There are two ways of defining reduce: reducing from the left end of the list, and reducing from the right end of the list. This is one of them, you could try to find the other if you want a "`fun`" exercise.

4 What's the difference between a datatype and a C union type?

...I hear you ask.

Well here's the answer. Let's consider a C union type, which we will ominously call `unsafe`.

```
union {
    float f;
    int i;
} unsafe;

unsafe.f = 1.0;
printf("%d\n", unsafe.i + 2);
```

What does this print? Did you guess 1065353218? What we just did was read a float as an int, and the compiler raised no errors.

This kind of behavior is not allowed in the high-level, lawful palace known as ML. Consider our Chex Mix datatype from above, where we had both ints and bools. The datatype is not quite the union of the int and bool domain, but is what the set theorists call a ”disjoint union,” notated as int$\uplus$bool instead of int$\cup$bool. This means that if we take an element from the datatype, we will be able to know if it's an int or a bool: The constructors in an ML datatype are represented at a lower level as a bit or bits called ”tags,” set aside to tell us which type it is. The same program written in ML will raise an exception:

```
datatype safe = I of int | R of real; (* The SML response to the c
    union above *)
val safe = R 1.0;
print (Int.toString 5);
let val (I x) = safe
     in print (Int.toString (x+2))
end;

uncaught exception Bind
```

5 Programs That Take Programs as Input

Consider the grammar $E ::= n|E + E$. To represent this grammar in ML, we can use the following datatype.

```
datatype E = I of int | Plus of E * E;
```

So, the expression $(1 + 2) + 3$ in E can be represented as

```
val test = Plus(Plus(I 1, I 2), I 3)
```

We can now write an interpreter for our simple language. The interpreter will have the type `E -> int`. As always, we will need a line of our function to consider each of the two constructors in our datatype.

```
fun interp (I x) = x (* Base case *)
  | interp (Plus(x,y)) = (interp x) + (interp y)
```

Interestingly, if we compile with just the base case, we won't get an error, just a match non-exhaustive warning. But when we try to apply our interpreter function, we will get a runtime exception.

6 Eager vs. Lazy Languages

Lets say we have this code. What happens when we run it?

```
fun loop x = loop x;
fun zero x = 0;
zero(loop(9));
```

This code loops forever in ML, and would loop forever in almost all languages. This is because most programming languages are "eager": they compute things, like the invocation of the loop function even if they might not need them. Other languages, such as Haskell, are "lazy," waiting until computations are needed before triggering them. In a lazy language, this will not loop forever.

Because of this, lazy languages can work with infinite structures. Consider the structure below, which attempts to represent the (famously never-ending) natural numbers.

```
fun nats n = n :: nats (n+1)
nats 0;
```

Once again, this will run forever! So, how can we get ML (or any other eager language) to behave like a lazy language, and wait to generate the next natural number until we directly request it?

The answer, broadly, is through functions. Consider the delay function below:

```
val delay = fn() => 7 + 1;
```

At this point in time, we have not calculated $7+1$ yet. It does not get executed until we actually call the function.

```
delay(); (* Now we calculate 7 + 1 *)
```

We can implement this kind of scheme when defining our sequence of natural numbers, like this.

```
datatype seq = S of int * (unit -> seq);
fun nats_l n = S(n, fn() => nats_l (n+1));
fun get_n 1 (S(x, promise)) = x
  | get_n n (S(x, promise)) = get_n (n-1) (promise());
```

Using this method, we can represent infinite sequences in any eager language, not just ML.

7 Polymorphism vs. Overloading

In ML, the `+` operator is overloaded, and the `map` function is polymorphic. Both operators can work on different datatypes. What's the difference?

As we know, `+` can work on both ints and reals. However, it is not the same function at runtime. When the compiler sees a `+` sign, it determines whether it is acting on ints or on reals, and decides, before runtime, which addition function to call.

This is not the case for polymorphic functions. In this case, the compiler generates one piece of code, that will be used for any data type passed through it. However, different data types have different sizes: a real is often more bytes than an int, and custom datatypes could be even larger. How does the compiler generate a single piece of machine code, if it doesn't know how many bytes to allocate for the function argument?

The solution to that is to exclusively pass by reference in a polymorphic function. All pointers have the same size, so the compiler only needs to allocate enough space for a pointer for the argument. This solution comes at a cost. Passing by reference is often less efficient than passing by value, which is a reason why many languages do not support polymorphic functions.

To further illustrate the difference between a function such as + and map, let us consider the code for the `swap` function in both ML[1]. and C++:

```
fun swap a b = let val temp = !a
                in (a:= !b; b := temp)

                end;

template <typename T>
void swap(T& x, T& y){
T tmp = x; .....   ;
}
```

The C++ version doesn't generate a single piece of code for each type. It generates unique code for each type. C++ is overloaded because different code is executed at run time depending on the type. While ML is polymorphic because the same code is generated independent of the types. This require a uniform representation, which in case of ML consists of a pointer. For example, the variable temp will correspond to a pointer to an int, in case we are swapping integers. Instead, the variable tmp in the C++ code will be allocated on the stack and it will occupy the space for an int.

[1]a and b are pointers, and !a dereferences the pointer