CSE 341: Polymorphic types and higher-order functions

Polymorphic types and type variables

Recall ML's response when we typed in `nil` at the prompt:

```
- nil;
val it = [] : 'a list
```

The `'a list` type annotation is a **polymorphic type**, because it contains the **type variable** `'a`. Type variables in ML are **universally quantified**, which means they can stand for "any type" --- think of `'a list` as "for all a, list of a".

The presence of a type variable in a type indicates that the type is a "factory for types". A polymorphic type can be **instantiated** with any type substituted for the type variables.

ML automatically performs instantiation whenever an expression is used in a fashion requiring instantiation --- so, for example, in the expression:

```
- 1::nil;
val it = [1] : int list
```

In order to make the cons expression typecheck, `int` was substituted for the type variable `'a`, thereby making the cons legal.

Here's another value with polymorphic type:

```
- fun identity x = x;
val identity = fn : 'a -> 'a
```

This function returns whatever is passed to it (it's called the identity function). Clearly, this function can be applied to any value --- it imposes no constraints on its argument --- and so ML infers a polymorphic type `'a -> 'a`, meaning that this function returns the same type that is passed as its argument. Indeed, when the function is applied, its result type is appropriately instantiated to be whatever its argument type was:

```
- identity 3;
val it = 3 : int
- identity "hi";
val it = "hi" : string
```

Note that type variable substitution (like all type operations in ML) is performed statically (at compile/typecheck time), not dynamically (at run time); to prove this, apply identity inside the body of an unevaluated function:

```
- fn (s:string) => identity s;
val it = fn : string -> string
```

We have not evaluated the body of the `fn` form, but ML still deduces that the result type is `string`. In order to do this, it must have statically instantiated the type variables in `identity`'s type.

Multiple type variables

Types are not restricted to having one type variable in their type. Consider the following function, which swaps the elements of a pair:

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

The operation of swapping the elements of a pair is insensitive to the types of the elements. Furthermore, the types of the elements need not be the same. Therefore, ML assigns two different type variables to the element types. This function can be applied to pairs of any type:

```
- swap("hi", 123);
val it = (123,"hi") : int * string
- swap(#"a", (1, 2));
val it = ((1,2),#"a") : (int * int) * char
- swap("foo", "bar");
val it = ("bar","foo") : string * string
```

The last example above demonstrates that different type variables may be instantiated to the same type. The use of different variable names simply indicates that they may be instantiated to different types, not that this must be so.

Built-in polymorphic functions

Some of the functions you've seen already have polymorphic type. To wit:

```
- hd;
val it = fn : 'a list -> 'a
- tl;
val it = fn : 'a list -> 'a list
- op ::;
val it = fn : 'a * 'a list -> 'a list
- op @;
val it = fn : 'a list * 'a list -> 'a list
```

(Note that prefixing an infix operator with the `op` keyword allows you to refer to it as a regular name.)

Parametric vs. subtype polymorphism: `'a` versus `Object`

ML type variables stand for "any type". But this is different from, e.g., `Object` in Java, which also stands for any (object) type --- but in a different way. Consider a method:

```
static Object foo(Object o) { ... }
```

If we use ML-like arrow syntax to describe the type of this method, we might write `Object -> Object`. But this does not mean that `foo`returns a value having the same type as its argument, like the ML identity function. `foo` might return a different type altogether. In fact, there is no known relationship between the argument and result type of `foo`. Here is a possible legal implementation of `foo`:

```
static Object foo(Object o) { return "hello"; }
```

Type variables in ML polymorphic types must be substituted consistently throughout a type. Java has a different form of polymorphism, called **subtype polymorphism** (where any type may be substituted for any of its supertypes in any position). We'll return to this later in the quarter, when we discuss static typing in object-oriented languages.

The strength of ML-style polymorphism is that it "preserves more information". Imagine if `foo` really were an identity function:

```
static Object foo(Object o) { return o; }
```

Now, imagine some caller who wished to use this:

```
String s = ...;
String t = (String)foo(s);
```

The caller must re-cast the result of `foo` back to `String`, because the static type of `foo` loses the information that `foo` returns the same static type as its argument.

Higher-order functions

Review: Functions as arguments and return values

Recall that functions are first class. They can be stored in data structures, or serve as function arguments or return values. Here's a trivial function that takes `unit`, and returns a function that prepends `"Hi, "` to its argument:

```
- fun prependHi () = fn s => "Hi, " ^ s;
val prependHi = fn : unit -> string -> string
```

A function that operates over other functions is called **higher-order**(to contrast with **first-order** functions, which do not operate over functions).

Higher-order functions are invaluable because they enable novel (de)compositions of behavior --- a function can delegate some of the responsibility for defining behavior to its caller, and library functions can be written that factor out patterns of behavior commonly repeated in different clients.

Polymorphic types and higher-order types go naturally together. Here's a function that applies its first argument to its second:

```
fun applyF (f, v) = f v;
- val applyF = fn : ('a -> 'b) * 'a -> 'b
```

Note the polymorphic type. The argument of `f` must have the same type as `v`, to which it will eventually be applied --- hence the matching `'a` type variables --- and the result of the whole function must be the same as the result of `f` --- hence the matching `'b` type variables.

All non-trivial languages (even C) have some form of higher-order function. ML, and functional languages generally, provide higher-order functions in a particularly convenient form, which encourages their use.

Here's a more interesting higher-order function that returns the maximum of two elements in a tuple, based on a comparison function that is passed as an argument:

```
- fun max (greaterThan, (a, b)) =
    if greaterThan(a, b) then a else b;
val max = fn : ('a * 'a -> bool) * ('a * 'a) -> 'a
```

This could be used to compare records by two different fields --- one would simply need to pass different functions. This form of parameterization is common in, e.g., sorting algorithms.

Currying

Any function taking a tuple of arguments can be written as a higher-order function taking one argument. Consider addition:

```
- val add = fn (x, y) => x + y;
val add = fn : int * int -> int
- val addX = fn x => fn y => x + y;
val addX = fn : int -> int -> int
- val add5 = addX 5;
val add5 = fn : int -> int
- add5 4;
val it = 9 : int
- add5 6;
val it = 11 : int
```

`add` and `addX` both add two integers. But whereas `add` does this directly, `addX` does it in two steps: when given an argument `x`, returns a function that adds `x` to its argument. This function can then be applied to yield the sum.

Currying allows you to partially apply a function to some of its arguments, leaving a residual function which can be further evaluated later.

This transformation --- whereby a function with a tuple of arguments is transformed into a higher-order function --- is called currying, after the mathematician Haskell Curry, who employed a similar transformation in some of his work.

Currying is so common in ML that there is a special syntax for it --- simply put a space between curried arguments to a function:

```
- fun addX x y = x + y;
val addX = fn : int -> int -> int
```

Note the curried function type. Because function application is left-associating, curried function application can be written by following the function expression by its space-separated arguments:

```
- addX 3 4;
val it = 7 : int
```

Many of the functions in the ML standard library are curried.

Standard higher-order list functions

We have mentioned that higher-order function can factor out common "patterns" of behavior. We'll examine some functions from the ML standard library that do just this.

`map`

Functional programmers often write functions that take a list and return a different list, whose elements are computed by doing "something" to each of the input lists' elements.

Consider the function that converts a list of integers to a list of strings:

```
fun allToString nil = nil
  | allToString (x::xs) = (Int.toString x) :: (allToString xs);
val allToString = fn : int list -> string list
```

Now consider the function that computes the factorial of each element of an integer list:

```
fun factorial 0 = 0
  | factorial n = n * factorial (n-1);

fun allFactorial nil = nil
  | allFactorial (x::xs) = (factorial x) :: (allFactorial xs);
```

Now consider the function that converts a list of Celsius temperatures to Fahrenheit temperatures:

```
fun c2f temp = (temp * 9.0/5.0) + 32.0;

fun allC2F nil = nil
  | allC2F (x::xs) = (c2f x) :: (allC2F xs);
```

You should notice a pattern. Each of these recursive functions has:

- The same base case: for the empty list, return the empty list
- A function that **maps** elements of the input list to elements of the output list
- An inductive case that applies the function to the head, then conses this onto a recursive invocation of the tail.

This is a pattern that can be factored out into a higher-order function; we'll call it `myMap` (to avoid conflicting with the built-in library function `map`):

```
fun myMap (f, nil) = nil
  | myMap (f, x::xs) = (f x) :: myMap (f, xs);
val myMap = fn : ('a -> 'b) * 'a list -> 'b list
```

In `myMap`, the function that maps input list elements to output list elements is factored out as a parameter. The `myMap` function itself handles only the list traversal and construction behavior. We can write each of the "mapping pattern" functions above using an application of `myMap`:

```
fun allToString' aList = myMap (Int.toString, aList);
fun allFactorial' aList = myMap (factorial, aList);
fun allC2F' aList = myMap (c2f, aList);
```

Applications of map (and other higher-order functions) are so common that functional programmers often don't bother to bind the function argument to a name; they use anonymous functions directly:

```
fun allC2F'' aList =
    myMap (fn temp => (temp * 9.0/5.0) + 32.0, aList);
```

Final note: the standard library `map` uses the curried form instead of the tuple form:

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```

`List.filter`

Another common pattern is to filter a list into only those that satisfy some common predicate. For example, suppose you wanted to write a function that returned only the positive elements of a real list. You could write it as follows:

```
fun positives nil = nil
  | positives (x::xs) =
    if x > 0.0 then
        x :: (positives xs)
    else
        positives xs;
```

But we can generalize this by making the filter expression --- here `x > 0.0` --- into an application of a function parameter. Call this function the **predicate**; then, we can define a generic filter function as follows (note that we use the curried syntax):

```
fun myFilter pred nil = nil
  | myFilter pred (x::xs) =
    if pred x then
        x :: (myFilter pred xs)
    else
        myFilter pred xs;
val myFilter = fn : ('a -> bool) -> 'a list -> 'a list
```

Now, we can obtain `positives` as a special case of this function:

```
fun positives' aList = myFilter (fn x => x > 0.0) aList;
```

Thought exercise: Here's a version of the above function that exploits currying; why does it work?

```
val positives'' = myFilter (fn x => x > 0.0);
```

The standard library version of filter is named `List.filter`.

`List.find`

Another common pattern is to pick only the first element of a list that satisfies a given predicate. We'll skip directly to the generic function definition this time:

```
fun myFind pred nil = raise Fail "No such element"
  | myFind pred (x::xs) =
    if pred x then x else myFind pred xs;
val myFind = fn : ('a -> bool) -> 'a list -> 'a
```

(Ignore the `raise` expression for now; this is the ML way of signaling an error. We'll cover exceptions soon.) Here's an application of `myFind`:

```
myFind (fn x => x > 0.0) [~1.2, ~3.4, 5.6, 7.8];
val it = 5.6 : real
```

The standard library version of find is named `List.find`.

`foldl`

We can generalize the notion of recursion over lists still further. All recursion has a base case, an iterative case, and a way of combining results. Suppose we made all three of these parameters; then we'd get `foldl` ("fold left"), which functional programmers sometimes call `reduce` (we'll just call our function by that name):

```
fun reduce f b nil = b
  | reduce f b (x::xs) =
    let
        val rest = reduce f b xs;
        val combined = f (x, rest);
    in
        combined
    end;
val reduce = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

`reduce` operates as follows:

- For `nil`, it returns the base case.
- For the inductive case, it processes the rest of the list, and then combines the result of processing the rest of the list with the result of processing the head.

To understand the uses of reduce, consider a function that sums the elements of a list, which we'll write in the slightly more verbose form to make the parallel clear:

```
fun sumList nil = 0
  | sumList (x::xs) =
    let
        val rest = sumList xs;
        val combined = x + rest;
    in
        combined
    end;
```

Clearly this function can be written by plugging in zero (the base case) and addition (the combining function) where `b` and `f` would go in `reduce`. Passing them as parameters allows us to do exactly that:

```
fun sumList' aList = reduce (op +) 0 aList;
val sumList' = fn : int list -> int
sumList' [1, 2, 3];
val it = 6 : int
```

The standard library version of reduce is named `foldl` (there is also a `foldr`, which "folds" the list up in the reverse order; you can experiment with `foldr` to see what it does).

Summary: higher-order functions as control structures

`map` and `reduce` serve purposes similar to `for` loops in Java (or `foreach` loops in many other languages, e.g. Perl): programmers use them to iterate over the elements of a list. But these need not be hard-coded into the language, and iteration functions can easily be defined for more complex data structures such as trees.

Languages which make higher-order functions available in a convenient form allow programmers to effectively program their own control structures.

We'll see an even more aggressive use of anonymous functions and higher-order functions for control structures when we learn Smalltalk.

Supplemental exercises

1. For each of `map`, `filter`, and `find`, determine whether it is tail-recursive. Explain why or why not; and if not, write a tail-recursive version. Write a few test expressons to show that your function works properly. (Note that you may have to reverse the elements when you're done processing the list.)
2. Rewrite `myMap` as a hand-curried function (i.e., using explicit `fn`forms in the body) instead of a function taking a tuple. Rewrite `sumList'` in a form that exploits currying.
3. Write the equivalent of `map` in C++ or Java, using an object with a mapping function as the function parameter.

Solutions are available.