Lecture 21: Even More OCaml
10:00 AM, Oct 24, 2018

Contents

1 Parameterized types and polymorphic procedures and type variables 1
  1.1 searching a list ................................................................. 2
  1.2 map .................................................................................. 4

1 Parameterized types and polymorphic procedures and type variables

Note that list is not a type; it is analogous to a mathematical function that takes a type argument and returns a type. Some types are:

- int list, which is the type consisting of lists of ints. (int*int) list, which is the type consisting of lists of pairs of ints, and int list list, which is the type consisting of lists of lists of ints.

Recall the length procedure:

let rec length = function
| [] -> 0
| _::rest -> 1 + length rest

This single procedure can be applied, for example, to values of type int list or int * int list or int list list:

# length [17; 18];;
- : int = 2
# length [15,16 ; 17, 18];;
- : int = 2
# length [ [15, 16, 22, 33], [17, 18, 22, 32]];;
- : int = 1
# length [ [15; 16; 22; 33] ; [17; 18; 22; 32]];;
- : int = 2

From Racket, you’re used to a procedure like length being applied to lists consisting of different types of elements, but how does this work in , where each list has a specific type?

The procedure length is a polymorphic procedure, which means that it can accept arguments of different types. Note in this case that they all have to be lists.

There are several forms of polymorphism, especially in object-oriented programming.
• This form of polymorphism is called \textit{parametric} polymorphism.

• Another is called \textit{ad hoc polymorphism}. An example is the addition procedure in Racket: it accepted integers and floating-point numbers but used a different algorithm for each of them. This is often called \textit{overloading}.

Another example is equality testing in OCaml. The = can compare a pair of objects from pretty much any type, but the method of comparison depends on the that type.

• In object-oriented programming, you will see \textit{subtype} polymorphism.

Let's write a simpler example of a polymorphic procedure.

\begin{verbatim}
let one = function x -> 1
\end{verbatim}

This takes an argument of any type, and returns 1.

What is the type of such a procedure? OCaml tells us its type is ‘a \rightarrow \texttt{int}. This seems to mean that it takes an argument of type ‘a and returns a value of type \texttt{int} but ‘a is not a type; it is a \textit{type variable}. It can be assigned any type.

With that understanding, let's look at the type of \texttt{length}. It is ‘a \texttt{list} \rightarrow \texttt{int}. That means that the procedure accepts an argument that is a list of any type.

Type variables can tell us more than that, however. What is the type of the \texttt{car} procedure?

\begin{verbatim}
let car = function x::_ -> x
\end{verbatim}

The type is ‘a \texttt{list} \rightarrow ‘a. This means that whatever the element type of the input list, the output type is that element type. If the procedure is given an \texttt{int list}, the output is an \texttt{int}, and so on.

There can be more than one type variable in a type. Consider the \texttt{swap} procedure:

\begin{verbatim}
let swap = function (x, y) -> (y, x);
\end{verbatim}

Its type is ‘a \times ‘b \rightarrow ‘b \times ‘a. For example, if given a pair consisting of an \texttt{int} and a \texttt{bool}, it returns a pair consisting of a \texttt{bool} and an \texttt{int}. If given a pair of \texttt{bool}s, it returns a pair of \texttt{bool}s.

You would think that every value would have a concrete type, with no variables. However, what about the empty list? It sort of has the potential to become part of any kind of list at all, by consing, so it itself has type ‘a \texttt{list}.

\textbf{Quiz:} Guess the type of \texttt{[]}, \texttt{[]}, \texttt{[true]}.  

\textbf{Answer:} ‘a \texttt{list} \times ‘b \texttt{list} \times \texttt{bool list}

\subsection{1.1 searching a list}

Let's write a procedure that takes a data object and a list, and searches for that data object in that list.
containsP

input: tuple (x, L)
output: true if x is an element of L, false otherwise

Quiz: What is the type of the procedure?

Using =, we can write this in a way that does not depend on the type of x. This suggests that x should have the type 'a. What about the list L? Well, for us to compare x to an element of L, the element must be of the same type as x, so L must be of type 'a list. Thus the type of the procedure should be ('a * 'a list) -> bool

Quiz: write the procedure.

```ocaml
let rec containsP : 'a * 'a list -> bool =
    function (x, alod) ->
    match alod with
        ... | first::rest -> if x=first then true else containsP (x, rest)
```

The base case is that of the empty list, for which the answer is always false.

```ocaml
let rec containsP : 'a * 'a list -> bool =
    function (x, alod) ->
    match alod with
        [] -> false
        | first::rest -> if x=first then true else containsP (x, rest) ;;
```

The literal "true" in the second pattern clause should hint to us that we could maybe simplify it by writing it in terms of logic. The result should be true if x=first or if rest contains x, so let’s use a logical or, which is written ||.

```ocaml
let rec containsP : 'a * 'a list -> bool =
    function (x, alod) ->
    match alod with
        [] -> false
        | first::rest -> x=first || containsP (x, rest)
```

Just like or in Racket, || in OCaml is short-circuiting: if the left-hand expression is true, OCaml does not evaluate the right-hand expression. The corresponding and is written &&.

What if we want a procedure that specifically searches for the string "doctor" in a string list? And then another for searching for 17 in a list of ints. Instead of writing each one, we could use currying.

input: x
output: procedure that takes in a list and returns true if x is in the list
let rec curried_containsP : ('a -> 'a list -> bool) =
function x -> function alod ->
match alod with
    [] -> false
  | first::rest -> x=first || (curried_containsP x) rest

# (curried_containsP 17) [1; 2; 3; 4];;
- : bool = false

# let contains_doctor = curried_containsP "doctor";;
val contains_doctor : string list -> bool = <fun>

Note the type of contains_doctor.

When discussing precedence, I said that in evaluating expressions, in case of ties, left binds tighter. Thus (containsP 17) [1; 2; 3; 4] can be rewritten as containsP 17 [1; 2; 3; 4]. That looks like containsP takes two arguments!

1.2 map

Let’s write the map procedure.

let rec map :
    ('a -> 'b) * 'a list -> 'b list =
function (f, alod) ->
match alod with
    [] -> []
  | first::rest -> f first::map (f, rest)

What is the type of this procedure? It takes in a procedure \( f \) and a list. What is the type of the procedure \( f \)? It can be any procedure type. What is the element type of the list? It can be anything.

Not quite—the procedure’s input type must match the element type of the list. We can represent that by using the same type variable twice, once in the type of the procedure \( f \) \( 'a \rightarrow 'b \) and once in the element type of the list \( 'a \) list

Finally, the output type is a list, whose element type is the output type of the procedure \( f \), which is represented by the type variable \( 'b \) list

Putting all that together, the type of map is

\( ('a \rightarrow 'b) \times 'a \) list \rightarrow 'b \) list

Putting this into type annotations for our map procedure, we now get:

let rec map : ('a -> 'b) * 'a list -> 'b list =
function (f, alod) ->
match alod with
    [] -> []
  | first::rest -> f first::map (f, rest)
We can use `map` like this:

```
# map ((function x -> x+1), [10; 20; 30]);;
- : int list = [11; 21; 31]
```

It would be nice to be able to use `map` to construct a procedure that specifically added one to each element of a list. We can do that by writing a curried version of `map`.

`curried_map` takes a procedure \( f \) and produces a procedure that, when applied to a list, returns the list obtained by applying \( f \) to each element.
let rec curried_map = 
  function f -> function alod ->
  match alod with
    [] -> []
  | a::rest -> f a::curried_map f rest

# let one_adder = curried_map (function x->1+x);;
val one_adder : int list -> int list = <fun>
# one_adder [10; 20; 30];;
- : int list = [11; 21; 31]

Quiz: What is the type of curried_map?

Answer: ('a -> 'b) -> 'a list -> 'b list

Rewriting our curried_map procedure with type annotations, we now get:

let rec curried_map : ('a -> 'b) -> 'a list -> 'b list = 
  function f -> function alod ->
  match alod with
    [] -> []
  | a::rest -> f a::curried_map f rest

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or
any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)