

Lecture Notes: OCaml Functions

CS 6371: Advanced Programming Languages

August 26, 2008

<pre>#let solve w x y z = let prod a b = a*b in (prod w x)+(prod y z);; solve : int -> int -> int -> int -> int = <fun> #prod;; Toplevel input: >prod;; >^^^^ The value identifier prod is unbound.</pre>	<p>Top-level and nested “let” operations are different: A top-level “let” defines a global variable (usually a function). It must be followed by “;;”. An inner “let” declares a local variable that can be used only within a subexpression. It must be followed by “in”. When variable names conflict, the innermost declaration “shadows” the others.</p>
<pre>#let solve x = let prod y = x*y in (prod 2)+(prod 3);; solve : int -> int = <fun></pre>	<p>You can use variables defined in an outer scope within locally defined functions. In this example, x is defined by solve but used in prod.</p>
<pre>#type foo = int;; #type foo = bool;; #type foo = string;; #type foo = int list;; #type foo = int * string;; #type btree = BLeaf BNode of (btree * btree);; #type ntree = NLeaf NNode of (ntree list);;</pre>	<p>User-defined types can be primitive types (int, bool, string, etc.), or they can be lists, tuples, or variants that include any of the above.</p>
<pre>#let identity x = x;; identity : 'a -> 'a = <fun> #identity 3;; - : int = 3 #identity "foo";; - : string = "foo"</pre>	<p>When possible, OCaml gives functions a polymorphic type. Polymorphic functions can be applied to arguments of any type.</p>
<pre>#let apply f x = (f x);; apply : ('a -> 'b) -> 'a -> 'b = <fun> #let add (x,y) = x+y;; add : int * int -> int = <fun> #apply add (1,2);; - : int = 3 #apply add "foo";; Toplevel input: >apply add "foo";; > ^^^^^ This expression has type string, but is used with type int * int.</pre>	<p>However, there must be some consistent way to instantiate each type variable. Here we see an example where no such instantiation exists and the compiler therefore rejects the code.</p>
<pre>#let rec map f l = (match l with [] -> [] x::tail -> (f x)::(map f tail));; map : ('a -> 'b) -> 'a list -> 'b list = <fun> #let addone n = n+1;; addone : int -> int = <fun> #map addone [23;42;64];; - : int list = [24; 43; 65]</pre>	<p>Lists can also have polymorphic type.</p>
<pre>#map (fun n -> n+1) [23;42;64];; - : int list = [24; 43; 65]</pre>	<p>Use “fun” to create anonymous (i.e., unnamed) functions. “fun ... -> ...” is the same as if you</p>

<pre> #(fun n -> n+1);; - : int -> int = <fun> #(fun n -> n+1) 2;; - : int = 3 </pre>	<p>typed “let foo ... = ...;” and then used “foo”.</p>
<pre> #let compose f g = (fun x -> f (g x));; compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun> #let cool = (compose (fun n -> n+1) (fun n -> n*2));; cool : int -> int = <fun> #cool 1;; - : int = 3 </pre>	<p>Using anonymous functions, you can build and return functions as values at runtime.</p>
<pre> #let addx x = (fun y -> x+y);; addx : int -> int -> int = <fun> #addx 1;; - : int -> int = <fun> #(addx 1) 2;; - : int = 3 #addx 1 2;; - : int = 3 </pre>	<p>An anonymous function may refer to variables declared in outer scopes.</p>
<pre> #let add = (fun x -> (fun y -> x+y));; add : int -> int -> int = <fun> #add 1 2;; - : int = 3 #add 1;; - : int -> int = <fun> #let add x y = x+y;; add : int -> int -> int = <fun> #map (add 3) [1;2;3];; - : int list = [4; 5; 6] </pre>	<p>Actually “let foo x y = ...” is just an abbreviation for “let foo = (fun x -> (fun y -> ...))”. If you give such a function fewer arguments than it expects, it yields a function from the remaining arguments to the original value. Functions written this way are called “curried functions”. Applying fewer arguments is called “partial evaluation”.</p>
<pre> #let rec map2 f l1 l2 = (match (l1,l2) with [],x -> x x,[], -> x (h1::t1, h2::t2) -> (f h1 h2)::(map2 f t1 t2));; map2 : ('a -> 'a -> 'a) -> 'a list -> 'a list -> 'a list = <fun> #map2 add [1;2;3] [10;20;30];; - : int list = [11; 22; 33] #map2 (+) [1;2;3] [10;20;30];; - : int list = [11; 22; 33] </pre>	<p>Binary operators can be used in prefix rather than infix syntax by enclosing the operator in parentheses. This allows you to pass a binary operator as a function argument.</p>
<pre> #let rec addpairs l = (match l with x::y::tail -> (add x)::(addpairs tail) [] -> [] [x] -> [x]);; Toplevel input: > [x] -> [x];; > ^^^ This expression has type int list, but is used with type (int -> int) list. # let rec addpairs (l:int list) : int list = (match l with x::y::tail -> (add x)::(addpairs tail) [] -> [] [x] -> [x]);; Toplevel input: > x::y::tail -> (add x)::(addpairs tail) > ^^^^^^^^^^^^^^^^^^^^^ </pre>	<p>Typing annotations are almost never necessary, but they can help you debug.</p>

<pre>This expression has type int list but is here used with type (int -> int) list</pre>	
<pre>#let intident (x:int) = x;; intident : int -> int = <fun></pre>	<p>You can also use a typing annotation to restrict the type of a function that would otherwise be polymorphic.</p>
<pre>#let apply (f:'inp->'out) (x:'inp) = (f x);; apply : ('a -> 'b) -> 'a -> 'b = <fun></pre>	<p>Typing annotations may contain type variables.</p>
<pre>#let main () = (print_string "hello\n"; print_int 3; print_newline (); 12);; main : unit -> int = <fun> #main ();; hello 3 - : int = 12</pre>	<p>A ;-separated sequence of expressions is evaluated in order. The last expression is returned as the result of the sequence expression. Use sequence expressions with print statements to debug.</p>
<pre>"foo" = "foo";; #- : bool = true #(3,"foo") = (3,"foo");; - : bool = true #[1;2;3] = [1;2;3];; - : bool = true #Dark (Dark Red) = Dark (Dark Red);; - : bool = true</pre>	<p>OCaml's equality operator (=) tests structural equality. This means that you can use it with ints, booleans, strings, tuples, lists, and variants.</p>