CMSC 330: Organization of Programming Languages

OCaml Features

OCaml Functions Take One Argument

Recall this example

\[
\text{let plus } (x, y) = x + y;;
\]
\[
\text{plus } (3, 4);;
\]
- It looks like you’re passing in two arguments
- Actually, you’re passing in a tuple instead

\[
\text{let plus } t = \text{match } t \text{ with}
\]
\[
(x, y) = x + y;;
\]
\[
\text{plus } (3, 4);;
\]
- And using pattern matching to extract its contents

Tuples

- **Constructed** using \((e_1, \ldots, e_n)\)
- **Deconstructed** using pattern matching
- Tuples are like C structs
  - But without field labels
  - Allocated on the heap
- Tuples can be heterogenous
  - Unlike lists, which must be homogenous
  - \((1, \text{["string1"; "string2"]})\) is a valid tuple

Tuples – Examples

- let plusThree \((x, y, z) = x+y+z\)
  - let addOne \((x, y, z) = (x+1, y+1, z+1)\)
    - plusThree \((\text{addOne } (3,4,5)) = 15\)
- let sum \(((a, b), c) = (a+c, b+c)\)
  - sum \(((1, 2), 3) = (4,5)\)
- let plusFirstTwo \((x::y::_, a) = (x+a, y+a)\)
  - plusFirstTwo \(((1; 2; 3), 4) = (5,6)\)

Tuples – More Examples

- let tls (_::xs, _::ys) = (xs, ys)
  - tls \(((1;2;3);[4;5;6;7]) = ([2;3],[5;6;7])\)

Another Tuple Example

- Given
  - let \(f l = \text{match } l \text{ with } x::(_::y) -> (x,y)\)
- What is the value of
  - \(f [1;2;3;4]\)
- Possibilities
  - \(((1,3))\)
  - \((1,3))\)
  - \((1,4))\)
  - \((1,[3;4])\)}
List and Tuple Types

- Tuple types use * to separate components

Examples
- (1,2) : int * int
- (1,"string",3.5) : int * string * float
- (1, ['a';"b";'c']) : int * string list * char
- [[1,2]] : (int * int) list
- [(1,2);(3,4)] : (int * int) list
- [(1,2);(1,2,3)] : error

Type declarations

- type can be used to create new names for types
  - Useful for combinations of lists and tuples

Examples
- type my_type = int * (int list)
  - (3, [1; 2]) : my_type
- type my_type2 = int * char * (int * float)
  - (3, 'a', (5, 3.0)) : my_type2

Polymorphic Functions

- Some functions require specific list types
  - let plusFirstTwo (x::y::_, a) = (x+a, y+a)
  - plusFirstTwo : int list * int -> (int * int)

- But other functions work for a list of any type
  - let hd (h::_) = h
  - hd [1; 2; 3] (* returns 1 *)
  - hd ['a'; "b"; "c"] (* returns "a" *)

These functions are polymorphic

Polymorphic Types

- OCaml gives such functions polymorphic types
  - hd : 'a list -> 'a
    - Read as
      - Function takes a list of any element type 'a
      - And returns something of that type

Example
- let tl (_::_t) = t
  - tl : 'a list -> 'a list

Polymorphic Types (cont.)

- More Examples
  - let swap (x, y) = (y, x)
    - swap : 'a * 'b -> 'b * 'a
  - let tls (::_:xs, _::_:ys) = (xs, ys)
    - tls : 'a list * 'b list -> 'a list * 'b list

Tuples Are a Fixed Size

- This OCaml definition
  - let foo x = match x with
    - (a, b) -> a + b
    - (a, b, c) -> a + b + c;

- Would yield this error message
  - This pattern matches values of type 'a * 'b * 'c
    but is here used to match values of type 'd * 'e

- Tuples of different size have different types
  - Thus never more than one match case with tuples
Conditionals

- Use `if...then...else` just like C/Java
  - No parentheses and no end
  ```ocaml
  if grade >= 90 then
      print_string "You got an A"
  else if grade >= 70 then
      print_string "You got a C"
  else
      print_string "You're not doing so well"
  ```

Conditionals (cont.)

- In OCaml, conditionals return a result
  - The value of whichever branch is true/false
  ```ocaml
  # if 7 > 42 then "hello" else goodbye;;
  - : string = "goodbye"
  # let x = if true then 3 else 4;;
  x : int = 3
  # if false then 3 else 3.0;;
  This expression has type float but is here used with type int
  ```

The Factorial Function

- Using conditionals & functions
  - Can you write `fact`, the factorial function?
    ```ocaml
    let rec fact n =
      if n = 0 then
        1
      else
        n * fact (n-1);
    ```
  - Notice no return statements
    - This is pretty much how it needs to be written

Let Rec

- The `rec` part means "define a recursive function"
- Let vs. let rec
  ```ocaml
  let fact n =
    if n = 0 then 1
    else n * fact (n-1) in e1
  ```
  Fact is not bound here!

Examples – Semicolon

- Definition
  - `e1 ; e2` (* evaluate e1, evaluate e2, return e2)
  ```ocaml
  1 ; 2 ;;
  ```
  - (* 2 – value of 2nd expression is returned *)
  ```ocaml
  (1 + 2) ; 4 ;;
  ```
  - (* 4 – value of 2nd expression is returned *)
  ```ocaml
  1 + (2 ; 4) ;;
  ```
  - (* 5 – value of 2nd expression is returned to 1 + *)
  ```ocaml
  1 + 2 ; 4 ;;
  ```
  - (* 4 – because + has higher precedence than ; *)

Examples – Let

- `x;;`
  ```ocaml
  (* Unbound value x *)
  ```
- Let
  ```ocaml
  let x = 1 in x + 1;;
  ```
  ```ocaml
  (* 2 *)
  ```
- Let
  ```ocaml
  let x = x in x + 1;;
  ```
  ```ocaml
  (* Unbound value x *)
Examples – Let

- let x = 1 in (x + 1 ; x) ;;  
  • (* 1 – ; has higher precedence than let … in *)

- (let x = 1 in x + 1) ; x;;
  • (* Unbound value x *)

- let x = 4 in (let x = x + 1 in x);
  • (* 5 *)

Let – More Examples

- let f n = 10;;
  let f n = if n = 0 then 1 else n * f (n – 1);;

  • f 0;; (* 1 *)
  • f 1;; (* 10 *)

- let f x = f x;;
  • (* Unbound value f *)