Benjamin Fenster
CS-103 (Skalka)
28 September 2003

Notes – Patterns and Pattern Matching

I. Patterns
   a. One of the cool things about OCaml.
   b. The idea is that you can define a pattern in the way things are bound.
   c. Allows precise case analysis when deconstructing datatypes.
   d. Let p range over patterns.
   e. Patterns
      i. x (simple, basic pattern – a variable)
      ii. (p_1, …, p_n) tuples.
      iii. c constants
         1. let (1, y) = (1, 2)
         2. *Match* left-hand element, *bind* right-hand element.
         3. let (1, y) = (2, 1) $\Downarrow$ raise match failure exception.
         4. NB: Match failure raised whenever a *required* match fails.
   f. Patterns used in declarations
      i. We can use a pattern to declare multiple variables simultaneously
      ii. let (x, y) = (1, 2);;
      iii. Introduces bindings x : int, y : int, x = 1, y = 2
      iv. # x;; -: int = 1
      v. # x+y;; -: int = 3
      vi. FYI: let z as (x, y) = 1, 2);;
   g. `let thrd = (fun (x, y, z) -> z);;`
      i. We accept the pattern (x, y, z) and return the third element.
      ii. This is a polymorphic function, polymorphic type!
      iii. There's nothing to indicate what datatypes are involved except that the third element in the input must match the output.
      iv. It will work for (int, int, float) -> float
      v. (string, int, int) -> int
      vi. Et cetera.
      vii. Has type $\forall\alpha,\beta,\chi\ \alpha*\beta*\chi \rightarrow \chi$
      viii. Much more about this in the second half of the semester.
   h. Wildcard Pattern
      i. _
      ii. Matches any pattern
      iii. Introduces no bindings.
      iv. let _ = 5   never raises match failure, but introduces no bindings.
      v. Used in conjunction with pattern matching (with clauses)

II. Pattern Matching
   a. Good for deconstructing data structures.
   b. let ((x, y), z) = ((1, 2), 5) *Note the nested pattern*
   c. Implement case matching.
      i. `let rec fact = match x with 0 -> 1 | x -> x * fact(x – 1)`
      ii. Less verbose, but equivalent to what we had before.
      d. Example
         i. Want to write passing_grade, returns true iff a given letter grade is passing.
         ii.
```
let passing_grade grade = match grade with
                                  "A" -> true
                                | "B" -> true
                                | "C" -> true
                                | _ -> false
```
         iii. Only use _ when you don't need the value on the right side.
      iv. if..then..else is really sugar for match statement.

v. if p then e_1 else e_2 really means match p with true -> e_1 | false -> e_2

e. Formal Definitions

i. `match e with` p_1 `->` e_1 `| … |` p_n `->` e_n `:` τ `iff for all 1` $\leq$ `I` $\leq$ `n,` e_i `:` τ

ii. Match evaluation

1. Match e with p_1 -> e_1 | … | p_n -> $e_n \Downarrow$ v iff e $\Downarrow$ v' and $e_i \Downarrow$ v in an environment extended with bindings resulting from matching e with p_i where p_i is the *first* match for v' taken in order p_1, …, p_n
2. Example
   a. match (1, (2, 3)) with (0, (x, y)) -> x * y
      | (1, (x, y)) -> x + y
      | (1, (x, y)) -> x – y

   b. x – y will never be the result! The *first* match is taken.

iii. Redundancy

1. let rec fact x = match x with x -> x * fact(x – 1) | 0 -> 1
2. The base case will never be reached!
3. This diverges on any input.

iv. Exhaustiveness

1. let encode_bool x = match x with 1 -> true | 0 -> false
2. Provides compiler / interpreter warning: match not exhaustive.
3. Could make an explicit wildcard case (perhaps raise an exception if it's reached)
4. Could specify in comment: In: x $\in$ {0, 1}
5. Still get the warning if you use a comment, of course, but now it's acceptable from a programming standpoing.

f. NB: Pattern matching clauses introduce new bindings

i. Now we have three ways to introduce bindings. Need to redefine our notion of scooping.

ii. Given match e with p_1 -> e_1 | … | p_n -> e_n, $\forall$ 1 $\leq$ I $\leq$ n, let x_{i1}, …, x_{ij} be all the variables in p_i. Then the scope of x_{i1}, …, x_{ij} is e_i

iii. Example

1. let x = "fred"
2. let y = 5;;
3. match (1, 2) with (x, y) -> x + y $\Downarrow$ 3
4. Even though we already had x, y bound, the new bindings shadow the originals.
5. x $\Downarrow$ "fred" *after the match statement is done executing*

III. Type Polymorphism

a. Very cool. One of the triumphs of programming language research.

b. The idea is that when you define a function it can take on a variety of forms.

c. Example

i. let third (_, _, x) -> x

ii. third(1, 2, 3) $\Downarrow$ 3

iii. third("word", 1.0, (fun x -> x)) $\Downarrow$ (fun x-> x)

iv. third: ‘a * ‘b * ‘c -> ‘c

v. Interpret ‘a, ‘b, ‘c, etc as greek letters. These are type variables!

vi. Types with quantified variables are called type schemes, which can be instantiated to yield types via *consistent* substitution of types for type variables.

vii. So you can substitute int for ‘a, ‘b, ‘c to get an instance of the type scheme.

viii. third: int * int * int -> int

ix. third: int * string * float -> float

x. NO GOOD: third: int * string * float -> int *not consistent*

xi. These are all instances of the type scheme

IV. Polymorphic Lists
- a. Act like stacks.
- b. Recursively defined data structures.
- c. Ho9mogeneous (every element must have the same type τ for a particular list)
- d. Polymorphic in the element type τ
- e. [2; 4; 6; 8]
- f. All OCaml lists are finite.
- g. (fun x-> x)] : (‘a -> ‘a) list
- h. Lists are constructed inductively on the *basis* of the empty list and the cons operation ::
    - i. Example `1::[]` $\Downarrow$ `[1]`
    - ii. $\forall$ τ the constant [ ] : τ list
    - iii. In other words, [ ] : $\forall$ ‘a, ‘a list
    - iv. If v : τ and v’ :τ list then v :: v’ : τ list
- i. Operations
    - i. Constructing
        - 1. e_1 :: e_2 $\Downarrow$ [v; v_1; v_2; …; v_n] iff e_1 $\Downarrow$ v and e_2 $\Downarrow$ [v_1; v_2; …; v_n]
        - 2. Always cons onto the end. Acts like a stack!
    - ii. Deconstructing
        - 1. New pattern p_1::p_2 (matches only non-empty lists)
        - 2. Idea is that p_1 gets the head, p_2 gets the tail of the list.
        - 3. Example
            - a. let head(x, _) = x : ‘a list -> ‘a
            - b. let tail(_, x) + x : ‘a list -> ‘a list
        - 4. Example
            - a. `head [1; 2; 3]` $\Downarrow$ `1`
            - b. `tail [1; 2; 3]` $\Downarrow$ `[2; 3]`
        - 5. Note that head doesn’t address empty lists
        - 6. New pattern [ ] will represent the empty list
- j. Lisp = List Processing Language. Just to give an idea of how important lists are to functional programming.
- k. `let rec lengh l = match l with [] -> 0 | _::xs -> 1 + length(xs) : ‘a list -> int`
- l. Some people use h::t (head, tail). Skalka uses x::x

V. Higher Order Functions
- a. These take functions as arguments, and can return functions as results.
- b. `f o g = f’` ∍ $\forall$ `x f’(x) = f(g(x))`
- c. `let compose = (fun f -> (fun g -> (fun x -> f(g(x)))))`
- d. `((compose : (fun x -> x + 1))(fun x -> x + 2))1` $\Downarrow$ `4`
- e. `compose : (‘a -> ‘b) -> (‘c -> ‘a) -> ‘c -> ‘b`
- f. Syntactic Sugar
    - i. `let f` x_1 … x_n `= e    let f = (fun` x_1 `-> fun` x_2 … `fun` x_n `-> e)`
    - ii. `let f(`x_1, …, x_n`)` is not the same!
    - iii. `let f g x = f(g(x))`
- g. Examples
    - i. `let add1 x = x + 1`
    - ii. `let add2 = compose add1 add1`
    - iii. `add2 2` $\Downarrow$ `4`
    - iv. `let add3 = compose add1 add2`
- h. Partial Composition
    - i. See above
    - ii. `add1 : int -> int`
    - iii. `compose add1 : (‘a -> int) -> ‘c -> int`

        iv. `compose add1 add1 : int -> int`

i. Currying
    i. After Haskell Curry
    ii. Define functions that go between curried and uncurried form
    iii. `let curry f = (fun x -> fun y -> fun(x, y))`
    iv. `curry : (‘a * ‘b -> ‘g) -> ‘a -> b -> ‘c`
    v. `let uncurry f = (fun (x, y) -> f x y)`
    vi. `uncurry : (‘a -> ‘b -> ‘c) -> (‘a * ‘b) -> ‘c`
    vii. Example
        1. `let f = compose add1`
        2. `let f’ = uncurry f`
        3. `f add1` is the same as `add2`
        4. `f’(add1, 2)` $\Downarrow$ `4`
        5. `let f’’ = curry f’ (* back to f again! *)`

ERROR: undefinedfilename
OFFENDING COMMAND: </FONT>

STACK: