Programming Languages and Techniques (CIS120)

Lecture 5

January 25th, 2016

Nested Pattern Matching

Datatypes

What is the type of this expression?

```
[ (1,true); (0, false) ]
```

1. int * bool
2. int list * bool list
3. (int * bool) list
4. (int * bool) list list
5. *none (expression is ill typed)*

Answer: 3

What is the type of this expression?

```
(1 :: [], 2 :: [], 3 :: [])
```

1. int
2. int list
3. int list list
4. int list * int list * int list
5. int * int list * int list list
6. (int * int * int) list
7. *none (expression is ill typed)*

Answer: 4

Announcements

- Submit HW1 by midnight Tuesday
  - Late policy: 10pt penalty for up to 24 hours
    20pt penalty for 24-48 hours
- Register your clicker ID number on course website
  - You should start seeing “Quizzes” on the submission page
  - Name of quiz is lecture date: TP160125 is Today
  - If you have “Not submitted” then we don’t have an ID number for your data
  - No way for me to "excuse" absences in the system. But, send CAR anyways.
- Read Chapters 5 and 6 of the course notes

Nested Patterns

- So far, we've seen simple patterns:

`[]`	*matches empty list*
`x::tl`	*matches nonempty list*
`(a,b)`	*matches pairs (tuples with 2 elts)*
`(a,b,c)`	*matches triples (tuples with 3 elts)*

- Like expressions, patterns can *nest*:

`x :: []`	*matches lists with 1 element*
`[x]`	*matches lists with 1 element*
`x::(y::tl)`	*matches lists of length at least 2*
`(x::xs, y::ys)`	*matches pairs of non-empty lists*

What is the value of this expression?

```
let l = [1; 2] in

begin match l with
  | x :: y :: t  -> 1
  | x :: []      -> 2
  | x :: t       -> 3
  | []           -> 4
end
```

Answer: 1

```
let l = [1; 2] in
begin match l with
   | x :: y :: t  -> 1
   | x :: []      -> 2
   | x :: t       -> 3
   | []           -> 4
end
```

```
let l = 1 :: 2 :: [] in
begin match l with
   | x :: y :: t  -> 1
   | x :: []      -> 2
   | x :: t       -> 3
   | []           -> 4
end
```

```
begin match 1::2::[] with
   | x :: y :: t  -> 1
   | x :: []      -> 2
   | x :: t       -> 3
   | []           -> 4
end
```

```
1
```

What is the value of this expression?

```
let l = [(2,true); (3,false)] in

begin match l with
  | (x,false) :: tl      -> 1
  | w :: (x,y) :: z      -> x
  | x                    -> 4
end
```

Answer: 3

Programming with Lists and Tuples

see zip.ml

Wildcard Pattern

- Another handy pattern is the wildcard pattern: _

  `_::tl` *matches a non-empty list, but only names tail*

  `(_,x)` *matches a pair, but only names the 2nd part*

- A wildcard pattern indicates that the value of the corresponding subcomponent is irrelevant.
  - And hence needs no name.

Unused Branches

- The branches in a match expression are considered in order from top to bottom.
- If you have "redundant" matches, then some later branches might not be reachable.
  - OCaml will give you a warning

```
let bad_cases (l : int list) : int =
  begin match l with
  | [] -> 0
  | x::_ -> x
  | x::y::tl -> x + y     (* unreachable *)
  end
```

This case matches more lists than that one does.

Exhaustive Matches

- Pattern matching is *exhaustive* if there is a pattern for every possible value
- Example of a *non-exhaustive* match:

```
let sum_two (l : int list) : int =
  begin match l with
  | x::y::_ -> x+y
  | _ -> failwith "l must have >= 2 elts"
  end
```

- OCaml will give you a warning and show an example of what isn't covered by your cases
- The wildcard pattern and failwith are useful tools for ensuring match coverage

Recursive Function Pattern

Recursive function patterns

Recursive functions over lists follow a general pattern:

```
let rec length (l : string list) : int =
  begin match l with
  | [] -> 0
  | ( x :: rest ) -> 1 + length rest
  end
```

```
let rec contains (l:string list) (s:string) : bool =
  begin match l with
  | [] -> false
  | ( x :: rest ) -> s = x || contains rest s
  end
```

Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …
  | ( hd :: rest ) -> … f rest …
  end
```

The branch for `[ ]` calculates the value (`f [ ]`) directly.

- this is the *base case* of the recursion

The branch for `hd::rest` calculates

(`f (hd::rest)`) given `hd` and (`f rest`).

- this is the *inductive case* of the recursion

Design Pattern for Recursion

1. Understand the problem

   What are the relevant concepts and how do they relate?
2. Formalize the interface

   How should the program interact with its environment?
3. Write test cases
   - If the main input to the program is an immutable list, make sure the tests cover both empty and non-empty cases
4. Implement the required behavior
   - If the main input to the program is an immutable list, look for a recursive solution…
     - Is there a direct solution for the empty list?
     - Suppose someone has given us a partial solution that works for lists up to a certain size. Can we use it to build a better solution that works for lists that are one element larger?

Example: zip

- zip takes two lists of the same length and returns a single list of pairs:

```
zip [1; 2; 3] [“a”; “b”; “c”] ⇒
  [(1,”a”); (2,”b”); (3,”c”)]
```

```
let rec zip (l1: int list)
            (l2: string list) : (int * string) list =
  begin match (l1, l2) with
  | ([], []) -> []
  | (x::xs, y::ys) -> (x, y)::(zip xs ys)
  | _ -> failwith "zip: unequal length lists"
  end
```