OCaml Lists

CSC 310 - Programming Languages

Lists in OCaml

- The basic data structure in OCaml is the list
  - Lists can be of arbitrary length
  - Lists must be homogeneous (all elements have the same type)
- Operations
  - Construct lists with cons and nil
  - Destruct lists with pattern matching

Constructing Lists

- Syntax
  - `[]` is empty list ("nil")
  - `e1 :: e2` prepends element `e1` to list `e2` ("cons")
    - `e1` is called the head and `e2` is called the tail
  - `[e1;e2;...en]` is syntactic sugar for `e1::e2::...::en::[]`
- Examples:

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

Constructing Lists: Evaluation

- `[]` is a value
- To evaluate `[e1; ...; en]`
  - evaluate `e1` to a value `v1`
  - ...
  - evaluate `en` to a value `vn`
  - return `[v1; ... vn]`
- Desugaring: evaluate `e1::e2`
  - evaluate `e1` to a value `v1`
  - evaluate `e2` to a value `v2`
  - return `v1::v2`

Constructing Lists: Examples

```
# let a = [1; 1+1; 1+1+1];;
val a : int list = [1; 2; 3]
# let b = 0::a;;
val b : int list = [0; 1; 2; 3]
# let c = "a"::"b"::"c"::[];;
val c : string list = ["a"; "b"; "c"]
```

Constructing Lists: Typing

- Nil:
  - `[]: 'a list`
  - An empty list has type `t list` for any type `t`
  - `'a` is a polymorphic type; similar to a template in C++ or generic in Java
- Cons:
  - If `e1:t` and `e2:t list` then `e1::e2 : t list`
  - With parens: if `e1:t` and `e2:(t list)` then `(e1::e2) : (t list)`

List Typing Examples

```
# let x = [1; "a"];;
Error: This expression has type string but an expression
       was expected of type int

# let y = [[1];[2;3]];;
val y : int list list = [[1]; [2; 3]]

# let z = 0::[1;2;3];;
val z : int list = [0; 1; 2; 3]

# let w = [1;2]::z;;
Error: This expression has type int list
       but an expression was expected of type int list list
       Type int is not compatible with type int list
```

List Structure

- A list in OCaml is represented as linked list
  - A non-empty list is a pair (element, rest of list)
  - The element is the head of the list
  - The pointer is the tail of the list (which itself is a list)
- This is an inductively defined data structure
  - The empty list is `[]`
  - Or a pair consisting of an element and a list

List Immutability

- Lists in OCaml are immutable
  - There is no way to mutate an element of a list
  - Instead, build up a new list from an old list
- Example:

```
let x = [1;2;3;4]
let y = 5::x
let z = 6::x
```

Pattern Matching

- The `match` construct is used to pull lists apart
- Syntax

```
match e with
| p1 -> e1
| ...
| pn -> en
```

- `p1 ... pn` are patterns made up of `[]`, `::`, constants, and pattern variables (normal OCaml variables)
- `e1 ... en` are branch expressions in which pattern variables in the corresponding pattern are bound

Pattern Matching: Evaluation

- Syntax

```
match e with
| p1 -> e1
| ...
| pn -> en
```

- Evaluate `e` to a value `v`
- If `p1` matches `v`, evaluate `e1` to ‘`v1` and return it
- . . .
- Else, if `pn` matches `v`, evaluate `en` to `vn` and return it
- Else, no patterns match: raise `Match_failure` exception

Pattern Matching Examples

```
let is_empty lst =
  match lst with
  | [] -> true
  | (h::t) -> false

let hd lst =
  match lst with
    (h::t) -> h
```

“Deep” Pattern Matching

- Patterns can be nested for more precise matches
  - `a::b` matches lists with at least one element
  - `a::[]` matches lists with exactly one element
  - `a::b::[]` matches lists with exactly two elements
  - `a::b::c::d` matches lists with at least three elements

Pattern Matching: Wildcards

- The underscore is a wildcard pattern
  - Matches anything
  - But does not add any bindings
  - Useful to indicate a value will not be used
- Example

```
let hd lst =
  match lst with
    (h::_) -> h
```

Pattern Matching Typing

- Syntax

```
match e with
| p1 -> e1
| ...
| pn -> en
```

- If `e` and `p1`, ..., `pn` each have type `t1`
- and `e1`, ..., `en` each have type `t2`
- then the entire `match` expression has type `t2`

Polymorphic Types

- A function like `length` works for any type of list
- Polymorphic functions have polymorphic types
  - Example: `length: 'a list -> int`
  - This says the function takes a list of any element type `'a`, and returns something of type `int`

Missing Cases

- Exceptions for inputs that do not match any pattern
  - OCaml will warn you about non-exhaustive matches
- Example:

```
# let head lst = match lst with (h::_) -> h;;
Warning 8 [partial-match]: this pattern-matching is not
Here is an example of a case that is not matched:
[]
val head : 'a list -> 'a = <fun>

# head [];;
Exception: Match_failure ...
```

Pattern Matching Helps Make Code Robust

- You cannot forget a case
  - The compiler issues a non-exhaustive pattern match warning
- You cannot duplicate a case
  - The compiler issues an unused match case warning
- You cannot get an exception
  - Cannot do something like `List.hd []`

Lists and Recursion

- Lists have a recursive structure
  - so, most functions over lists will be recursive
- Example

```
let rec length lst = match lst with
  | [] -> 0
  | (_::t) -> 1 + (length t)
```

- This is similar to an inductive definition
  - The length of the empty list is zero
  - The length of a nonempty list is one plus the length of the tail

List Recursion Examples

```
let rec sum lst = match lst with
  | [] -> 0
  | (x::xs) -> x + (sum xs)

let rec last lst = match lst with
  | [x] -> x
  | (x::xs) -> last xs

let rec append lst1 lst2 = match lst1 with
  | [] -> lst2
  | (x::xs) -> x::append(xs lst2)
```