Recursive types & Lists

Ziyan Maraikar

July 21, 2016

Lecture Outline

1. Inductive type definitions
2. Lists in Ocaml
3. List operations
4. Sorting

Data structures and algorithms

- ★ Data structures are containers that store our data in a particular fashion.

Data structures and algorithms

- Data structures are containers that store our data in a particular fashion.
- We can build *algorithms* that do useful operations on data structure, e.g., search for an item, or list items in sorted order.

Data structures and algorithms

- Data structures are containers that store our data in a particular fashion.
- We can build *algorithms* that do useful operations on data structure, e.g., search for an item, or list items in sorted order.
- The data structure determines what algorithms can be *efficiently* implemented,
  e.g., how quickly can we search and sorted items.

Data structures and algorithms

- Data structures are containers that store our data in a particular fashion.
- We can build *algorithms* that do useful operations on data structure, e.g., search for an item, or list items in sorted order.
- The data structure determines what algorithms can be *efficiently* implemented,
  e.g., how quickly can we search and sorted items.
- Languages usually have some data structures built-in, e.g., arrays in imperative languages.

The list type [1]

Functional programming relies on lists as the primitive type of data collection.

```
type list =
| Cons of int * list
| Empty
```

[1] Cons is the traditional name given to the list constructor

The list type [1]

Functional programming relies on lists as the primitive type of data collection.

```
type list =
| Cons of int * list
| Empty
```

This is an *inductive type definition* — the type being defined occurs within the type definition. Observe the similarity with recursive functions.

[1] Cons is the traditional name given to the list constructor

Constructing lists

How do we construct an instance of this type?

```
type list =
| Cons of int * list
| Empty
```

Constructing lists

How do we construct an instance of this type?

```
type list =
| Cons of int * list
| Empty

let l1 = Empty ;;
val l1 : list = Empty
```

Constructing lists

How do we construct an instance of this type?

```
type list =
| Cons of int * list
| Empty

let l1 = Empty ;;
val l1 : list = Empty
let l2 = Cons(1, l1) ;;
val l2 : list = Cons (1, Empty)
```

Exercise: write down an expression for a list containing $1, 2, 3$.

Traversing a list

To perform an operation Example: write a function `contains` that checks whether a list contains a given item.

Traversing a list

To perform an operation Example: write a function `contains` that checks whether a list contains a given item.

```
let rec contains (l:list) (i:int) :bool =
  match l with
  | Cons(hd, tl) -> if hd=i then true else contains tl i
  | Empty -> false
```

Inductive types are variants, so we have a match case corresponding to each constructor.

What are the types of the `hd` and `tl` variables?

Construction vs. traversal

	Type constructor	Traversal

Example	`type 'a list =`
	`| Empty`
	`| Cons of 'a * 'a list`	

Construction vs. traversal

<table>
<tr><th></th><th>Type constructor</th><th>Traversal</th></tr>
<tr><td>Example</td><td><pre>type 'a list =
| Empty
| Cons of 'a * 'a list</pre></td><td><pre>let copy l =
match l with
| Empty -> Empty
| Cons(hd, tl) ->
    Cons(hd, copy tl)</pre></td></tr>
</table>

Construction vs. traversal

	Type constructor	Traversal

Example	`type 'a list =`
	`| Empty`
	`| Cons of 'a * 'a list`
		`let copy l =`
		`match l with`
		`| Empty -> Empty`
		`| Cons(hd, tl) ->`
		`    Cons(hd, copy tl)`

Inductive	*constructs* the DS recursively	*Traverses* the DS item by item
Terminal	Denotes the end of the DS	Terminates traversal

Exercise

```
let rec contains (l:list) (i:int) :bool =
  match l with
  | Cons(hd, tl) -> if hd=i then true else contains tl i
  | Empty -> false
;;
let l1 = Cons( 1, Cons(2, Empty))
```

Show the evaluation the following using substitution,

1. `contains l1 2`

Exercise

```
let rec contains (l:list) (i:int) :bool =
  match l with
  | Cons(hd, tl) -> if hd=i then true else contains tl i
  | Empty -> false
;;
let l1 = Cons( 1, Cons(2, Empty))
```

Show the evaluation the following using substitution,

1. `contains l1 2`
2. `contains l1 0`

Lecture Outline

Ocaml's list syntax

Ocaml has built in support for lists because they are used so often. The only difference is that,

- `Cons` is replaced by the `::` infix operator
- `Empty` is replaced by `[]`.

Ocaml's list syntax

Ocaml has built in support for lists because they are used so often. The only difference is that,

- ★ `Cons` is replaced by the `::` infix operator
- ★ `Empty` is replaced by `[]`.

```
let l1 = 1 :: 2 :: 3 :: [] ;;
let l2 = [ 1 ; 2 ; 3 ]
```

`l2` shows the shorthand syntax for defining lists. Note that the element separator is a semicolon (not a comma!)

Pattern matching on lists

Pattern matching on lists use the `::` and `[]` constructors,

```
let rec contains l i = match l with
| hd::tl -> if hd=i then true else contains tl i
| [] -> false
```

List polymorphism

Ocaml lists are polymorphic,

```
let l1 = [ 1 ; 2 ; 3 ] ;;
val l1 : int list = [1; 2; 3]
let l2 = [ "hello" ; "world" ] ;;
val l2 : string list = ["hello"; "world"]
```

List polymorphism

Ocaml lists are polymorphic,

```
let l1 = [ 1 ; 2 ; 3 ] ;;
val l1 : int list = [1; 2; 3]
let l2 = [ "hello" ; "world" ] ;;
val l2 : string list = ["hello"; "world"]
```

List functions are polymorphic too

```
contains;;
- : 'a list -> 'a -> bool = <fun>
contains l2 "world" ;;
- : bool = true
```

Exercise

1. Make our list type definiton polymorphic

```
type ______ list =
  | Cons of ______ * list
  | Empty
```

Exercise

1. Make our list type definiton polymorphic

```
type _____ list =
  | Cons of _____ * list
  | Empty
```

2. Write a function `replace l i j` which will replace every occurrence of `i` in list `l` with `j`.

Exercise

1. Make our list type definiton polymorphic

```
type _____ list =
  | Cons of _____ * list
  | Empty
```

2. Write a function `replace l i j` which will replace every occurrence of `i` in list `l` with `j`.
3. Is the `replace` function polymorphic?

Lecture Outline

LENGTH

How do we count the number of elements in a list?

```
let length l =
  match l with
  | [] -> 0
  | hd::tl -> 1 + length tl
```

Exercise: write a tail recursive `length` function.

Append

Combine the items in two lists one after another.

The append operation demonstrates how to deal with multiple lists.

Append

Combine the items in two lists one after another.

The append operation demonstrates how to deal with multiple lists.

```
let rec append l1 l2 =
  match (l1, l2) with
```

Append

Combine the items in two lists one after another.

The append operation demonstrates how to deal with multiple lists.

```
let rec append l1 l2 =
  match (l1, l2) with
  | ([], []) -> []
  | ([], hd::tl) -> hd :: append [] tl
  | (hd::tl, []) ->  hd :: append tl []
  | (hd::tl, l2) -> hd :: append tl l2
```

Exercise: evaluate the expression `append [1] [2; 3]`

Append

Combine the items in two lists one after another.

The append operation demonstrates how to deal with multiple lists.

```
let rec append l1 l2 =
  match (l1, l2) with
  | ([], []) -> []
  | ([], hd::tl) -> hd :: append [] tl
  | (hd::tl, []) ->  hd :: append tl []
  | (hd::tl, l2) -> hd :: append tl l2
```

Exercise: evaluate the expression `append [1] [2; 3]` Ocaml provides the `@` operator to append to lists in pervasives. e.g. `[1;2] @ [3;4]`.

Maximum

What is the maximum of an empty list?

Maximum

What is the maximum of an empty list?

```
let rec list_max l =
  match l with
  | [] -> failwith "No maximum in empty list"
  | [hd] -> hd
  | hd::tl -> max hd (list_max tl)
```

`failwith` causes an *exception* terminating evaluation of the function with the error message given.

Reverse

Reverse the order of items in a list.

Reverse

Reverse the order of items in a list.

```
let rec reverse l =
  match l with
  | [] -> []
  | hd::tl -> (reverse tl) @ [hd]
```

A tail recursive reverse is more efficient than this

Lecture Outline

Insertion sort

★ Maintain a partial list of sorted items `sl` (initially empty.)

Insertion sort

- Maintain a partial list of sorted items `sl` (initially empty.)
- Repeatedly insert items from original list into `sl` **ensuring that it remains sorted.**

Insertion sort

```
let rec isort l =
  (* inserts an item x into list sl
   * PRE: sl is sorted *)
  let rec insert x sl =
    match sl with
    |[] -> [x]
    |hd::tl -> if x>hd then hd :: insert x tl
    else x::hd::tl in
  (* insert items one by one into the sorted list *)
```

Insertion sort

```
let rec isort l =
  (* inserts an item x into list sl
   * PRE: sl is sorted *)
  let rec insert x sl =
    match sl with
    |[] -> [x]
    |hd::tl -> if x>hd then hd :: insert x tl
    else x::hd::tl in
  (* insert items one by one into the sorted list *)
  match l with
  | [] -> []
  | hd::tl -> insert hd (isort tl)
```