Inductive Types II: Examples

CAS CS 320: Principles of Programming Languages

Thursday, February 8, 2024

Administrivia

- Homework 2 is due today by 11:59 pm.
- Homework 3 is posted today and due on Thursday, Feb 15, by 11:59 pm.

Administrivia

- Homework 2 is due today by 11:59 pm.
- Homework 3 is posted today and due on Thursday, Feb 15, by 11:59 pm.

Reading Assignment

- OCP, Section 3.9.5, 3.9.7: **Algebraic Data Types**
- OCP, Section 3.11: **Trees**

recursive variant types (OCP 3.9.4)

instead of using type constructor **list** to define intlist1, as in:

```
type intlist1 = int list ;;
let rec sum1 (lst : intlist1) : int =
  match lst with
  | [] -> 0
  | h :: t -> h + sum1 t
```

recursive variant types (OCP 3.9.4)

instead of using type constructor **list** to define intlist1, as in:

```
type intlist1 = int list ;;
let rec sum1 (lst : intlist1) : int =
  match lst with
  | [] -> 0
  | h :: t -> h + sum1 t
```

we can use **recursive variant types**, as in:

```
type intlist2 = Nil | Cons of int * intlist2
let rec sum2 (lst : intlist2) : int =
  match lst with
  | Nil -> 0
  | Cons (h, t) -> h + sum2 t
```

parametrized recursive variant types (OCP 3.9.5)

instead of **recursive variant types**, as in:

```
type intlist2 = Nil | Cons of int * intlist2
let rec length2 (lst : intlist2) : int =
  match lst with
  | Nil -> 0
  | Cons (_, t) -> 1 + length2 t
```

parametrized recursive variant types (OCP 3.9.5)

instead of **recursive variant types**, as in:

```
type intlist2 = Nil | Cons of int * intlist2
let rec length2 (lst : intlist2) : int =
  match lst with
  | Nil -> 0
  | Cons (_, t) -> 1 + length2 t
```

we can use **parametrized recursive variant types**, as in:

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
let rec length3 (lst : 'a mylist) : int =
  match lst with
  | Nil -> 0
  | Cons (_, t) -> 1 + length3 t
```

types with polymorphic variants (OCP 3.9.6)

Skip – this section is not part of the posted schedule.

built-in parametrized recursive variant types (OCP 3.9.7)

OCaml's **list** datatype is an example of a built-in **parametrized recursive variant type**, defined as follows:

```
type 'a list = [] | ( :: ) of 'a * 'a list
```

Note how our definition of `'a mylist` (two slides earlier) mimics that of `'a list`.

built-in parametrized recursive variant types (OCP 3.9.7)

OCaml's **list** datatype is an example of a built-in **parametrized recursive variant type**, defined as follows:

```
type 'a list = [] | ( :: ) of 'a * 'a list
```

Note how our definition of 'a mylist (on the previous slide) mimics that of 'a list.

Another of Ocaml's built-in **parametrized (non-recursive) variant types** is the **option** datatype:

```
type 'a option = None | Some of 'a
```

example: trees (OCP 3.11)

once again, user-defined 'a mylist followed by 'a tree:

```
type 'a mylist =
   | Nil
   | Cons of 'a * 'a mylist
type 'a tree =
   | Leaf
   | Node of 'a * 'a tree * 'a tree
```

example: trees (OCP 3.11)

once again, user-defined `'a mylist` followed by `'a tree`:

```
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

```
(* how to represent
       2
      / \
     1   3  *)
let t =
  Node (2,
    Node (1, Leaf, Leaf),
    Node (3, Leaf, Leaf)
  )
```

example: trees (OCP 3.11)

once again, user-defined `'a mylist` followed by `'a tree`:

```
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

function **preord** constructs a list of tree nodes in preorder:

```
let rec myapp (l1 : 'a mylist) (l2 : 'a mylist) : 'a mylist =
 match l1 with
 | Nil -> l2
 | Cons (hd,tl) -> Cons (hd, myapp tl l2)
let rec preord (t : 'a tree) : 'a mylist =
 match t with
 | Leaf x -> Cons (x, Nil)
 | Node (x,lt,rt) -> Cons(x, myapp (preord lt) (preord rt))
```

example: trees (OCP 3.11)

once again, user-defined 'a mylist followed by 'a tree:

```
type 'a mylist =
   | Nil
   | Cons of 'a * 'a mylist
type 'a tree =
   | Leaf
   | Node of 'a * 'a tree * 'a tree
```

function **postord** constructs a list of tree nodes in postorder:

```
let rec myapp (l1 : 'a mylist) (l2 : 'a mylist) : 'a mylist =
 match l1 with
  |Nil -> l2
  |Cons (hd,tl) -> Cons (hd, myapp tl l2)
let rec postord (t : 'a tree) : 'a mylist =
 match t with
  |Leaf x -> Cons (x,Nil)
  |Node(x,lt,rt) -> myapp(myapp(preord lt)(preord rt))(Cons(x,Nil))
```

example: trees (OCP 3.11)

using record types to represent binary trees:

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {value : 'a; left : 'a tree; right : 'a tree}
```

example: trees (OCP 3.11)

using record types to represent binary trees:

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {value : 'a; left : 'a tree; right : 'a tree}
```

```
(* how to represent
       2
      / \
     1   3   *)

let t =
  Node {
    value = 2;
    left  = Node {value = 1; left = Leaf; right = Leaf};
    right = Node {value = 3; left = Leaf; right = Leaf}
  }
```

example: trees (OCP 3.11)

using record types to represent binary trees:

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {value : 'a; left : 'a tree; right : 'a tree}
```

membership **mem** function using record-type representation:

```
(* (mem x t) is true/false x is/is not a value in t. *)
let rec mem x = function
 |Leaf -> false
 |Node {value;left;right} -> value = x || mem x left || mem x right
```

example: trees (OCP 3.11)

using record types to represent binary trees:

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {value : 'a; left : 'a tree; right : 'a tree}
```

function **preord** constructs a list of tree nodes in preorder:

```
let rec preord = function
  | Leaf -> []
  | Node {value;left;right} ->
          [value] @ preord left @ preord right
```

example: trees (OCP 3.11)

using record types to represent binary trees:

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {value : 'a; left : 'a tree; right : 'a tree}
```

function **preord_lin** constructs a list of nodes in preorder:

```
let preord_lin t =
  let rec pre_acc acc = function
    | Leaf -> acc
    | Node {value; left; right} ->
        value :: (pre_acc (pre_acc acc right) left)
  in pre_acc [] t
```

(THIS PAGE INTENTIONALLY LEFT BLANK)