1 Lecture Plan

1. Recursive types in OCaml, defining the list type
2. Examples with defined lists
3. Implementing lists as a recursive type
   - intlist example – Kozen 2011 Lecture 4
   - Polymorphic case – this lecture
   - Note – this topic informs us about canonical expressions for lists
   - Recursive types are a BIG IDEA
4. Polymorphic recursive types and defined lists
5. Implementing maps and folds – details of the “map reduce paradigm,”
   Kozen 2011 Lecture 5. This is a GOOD IDEA and a “buzz word.”

2 Review, Overview, and Comments

We have seen how OCaml defines lists, both monomorphic (one type such as int list) and polymorphic such as α lists. We have looked at recursive
procedures, map, and fold on built-in lists. We saw questions about the canonical form of list values.

We will now look at how lists, both monomorphic and polymorphic, can be defined using more general recursive types. This will show us what the canonical values are and how to implement the map and fold operations.

Recursive types are an example of a big idea in type theory. Some of the “Big Ideas” in this course are close to Open Research Problems. This is the case with recursive types. Later in the course we will look at some of the deep mathematical problems that OCaml can avoid because it implements partial types and does not attempt to guarantee totality.

One of the creative tensions in this course is between the need to introduce a number of fundamental concepts in computer science and the need to teach a particular programming language and use it to gain experience in specifying, writing, and verifying very small “systems” in the sense of organized code modules. The second goal requires that you write a lot of lines of code (“locs”) and the first goal requires that you absorb several new concepts (“cepts”). Our plan is to make these mutually reinforcing goals with their own evaluation methods – programming assignments on the one hand and exams on the other hand.

3 Reading and Sources

Almost all of the material for this lecture is from Professor Kozen’s notes which were mentioned in Lecture 4. The material in this lecture is taken almost entirely from Lectures 3, 4, and 5 from 2011 spring.
4 Implementing lists as a Recursive Type

When we implement lists as a recursive type, we can see what the right canonical form is for list values. We discovered this form by using the match function to decompose the list \([1;2;3;4]\). We found that it has the internal structure \(1::2::3::4::[]\). We will see a variant of this structure in the recursive implementation.

4.1 Implementing integer lists

```
type intlist = Nil | Cons of (int * intlist)
```

The type is recursive because it is defined in terms of itself. Notes, it uses a disjoint union.

It is tempting to just “unwind” the definition.

```
intlist = int * (int * (int * Nil))
```

More systematically

```
intlist = Nil
intlist = int * Nil
intlist = int * (int * Nil)
intlist = int * (int * (int * Nil))
```

We can imagine \(\text{intlist}\) as the “limit” of this process. An element is any element of one of these “unrollings,” e.g. elements are

- \(\text{Nil}\)
- \((1, \text{Nil})\)
- \((1, (2, \text{Nil}))\)
- \((1, (2, (3, \text{Nil})))\)

etc.

! This is almost right, but we need the \text{Cons} constructor.
If we introduced a new constant nil, these approximations are exactly Lisp’s lists. The OCaml lists are actually:

- Nil
- Cons(1, Nil)
- Cons(1, Cons(2, Nil))
- Cons(1, Cons(2, Cons(3, Nil)))

This gives us insight to the “real” canonical values of the int list type.

We display them as [1;2;3;4] but the more basic canonical value is

- 4::[]
- 3::4::[]
- 2::3::4::[]
- 1::2::3::4::[]

We saw this “truth” from applying the match operator in Lecture 4.

**Defining functions on intlist**

**length**

```ocaml
let rec length (lst : intlist) : int =
    match lst with
    | Nil -> 0
    | Cons (h, t) -> (length t) + 1
```

**mth element**

```ocaml
let rec mth_intlist (lst : intlist) (n : int) : _int_ =
    match lst with
    | Nil -> Msg "empty"
    | Cons (h, t) -> if n = 1 then Int h
                    else mth_intlist t (n - 1)

type _int_ = Int of int | Msg of string
```
5 Polymorphic Recursive Types

Here is code for implementing a version of polymorphic lists as a recursive type.

```ml
# type 'a glist = Nil | Dcons of 'a * 'a glist ;;
type 'a glist = Nil | Dcons of 'a * 'a glist

# type 'a glist_or_msg = Val of 'a | Msg of string ;;
type 'a glist_or_msg = Val of 'a | Msg of string

# let rec mth_gen ( lst : 'a glist ) (n: int ) : 'a glist_or_msg =
  ( if n <= 0 then Msg "out-of-bounds"
    else ( match lst with
        | Nil -> Msg "empty"
        | Dcons (h, t) -> (if n = 1 then Val h
                          else mth_gen t (n - 1)) ) ));;
val mth_gen : 'a glist -> int -> 'a glist_or_msg = <fun>

# mth_gen ( Dcons (1.0 , Dcons (2.0 , Dcons (3.0 , Nil ))) ) 2;;
- : float glist_or_msg = Val 2.
```

Here is a very interesting recursive type that we will discuss later. For fun you might try to build some interesting values.

```ml
# type 'a rftype = Base of (‘a -> ‘a)
  | Rfun of (‘a rftype -> ‘a rftype) ;;
type ‘a rftype = Base of (‘a -> ‘a) | Rfun of (‘a rftype -> ‘a rftype)
```
let rec length (lst : intlist) : int =
  match lst with
  | Nil -> 0
  | Cons (h, t) -> length t + 1

(* is the list empty? *)
let rec is_empty (lst : intlist) : bool =
  match lst with
  | Nil -> true
  | Cons _ -> false

(* Notice that the match expressions for lists all have the same
  form -- a case for the empty list (Nil) and a case for a Cons. 
  Also notice that for most functions, the Cons case involves a
  recursive function call. *)

(* Return the sum of the elements in the list *)
let rec sum (lst : intlist) : int =
  match lst with
  | Nil -> 0
  | Cons (i, t) -> i + sum t

(* Create a string representation of a list *)
let rec to_string (lst : intlist) : string =
  match lst with
  | Nil -> ""
  | Cons (i, Nil) -> string_of_int i
  | Cons (i, Cons (j, t)) ->
    string_of_int i ^ "," ^ to_string (Cons (j, t))

(* Return the head (first element) of the list *)
let head (lst : intlist) : int =
  match lst with
  | Nil -> failwith "empty list"
  | Cons (i, t) -> i

(* Return the tail (rest of the list after the head) *)
let tail (lst : intlist) : intlist =
  match lst with
  | Nil -> failwith "empty list"
  | Cons (i, t) -> t

(* Return the last element of the list (if any) *)
let rec last (lst : intlist) : int =
  match lst with
  | Nil -> failwith "empty list"
  | Cons (i, Nil) -> i
let rec nth (lst : intlist) (n : int) : int =  
match lst with  
| Nil -> failwith "index out of bounds"  
| Cons (i, t) ->  
  if n = 0 then i  
  else nth t (n - 1)

let rec append (l1 : intlist) (l2 : intlist) : intlist =  
match l1 with  
| Nil -> l2  
| Cons (i, t) -> Cons (i, append t l2)

let rec reverse (lst : intlist) : intlist =  
match lst with  
| Nil -> Nil  
| Cons (h, t) -> append (reverse t) (Cons (h, Nil))

let inc (x : int) : int = x + 1  
let square (x : int) : int = x * x

let rec addone_to_all (lst : intlist) : intlist =  
match lst with  
| Nil -> Nil  
| Cons (h, t) -> Cons (inc h, addone_to_all t)

let rec square_all (lst : intlist) : intlist =  
match lst with  
| Nil -> Nil
Cons (h, t) -> Cons (square h, square_all t)

(* Here is a more general method. *)

(* Given a function f and [il; ...; in], return [f il; ...; f in].
* Notice how we factored out the common parts of addone_to_all
* and square_all. *)

let rec do_function_to_all (f : int -> int) (lst : intlist) : intlist =
  match lst with
  | Nil -> Nil
  | Cons (h, t) -> Cons (f h, do_function_to_all f t)

let addone_to_all (lst : intlist) : intlist =
  do_function_to_all (fun x -> x + 1) lst

let square_all (lst : intlist) : intlist =
  do_function_to_all (fun x -> x * x) lst

(* Even better: use anonymous functions. *)

let addone_to_all (lst : intlist) : intlist =
  do_function_to_all (fun x -> x + 1) lst

let square_all (lst : intlist) : intlist =
  do_function_to_all (fun x -> x * x) lst

(* Equivalently, we can partially evaluate by applying
* do_function_to_all just to the first argument. *)

let addone_to_all : intlist -> intlist =
  do_function_to_all (fun x -> x + 1)

let square_all : intlist -> intlist =
  do_function_to_all (fun x -> x * x)

(* Say we want to compute the sum and product of integers
* in a list. *)

(* Explicit versions *)

let rec sum (lst : intlist) : int =
  match lst with
  | Nil -> 0
  | Cons (i, t) -> i + sum t

let rec product (lst : intlist) : int =
  match lst with
  | Nil -> 1
| Cons \((h, t)\) \(\rightarrow h * \text{product } t\) |

(* Better: use a general function \texttt{collapse} that takes an operation and an identity element for that operation.* *)

(* Given \(f, b,\) and \([i_1; i_2; ...; i_n]\), return \(f(i_1, f(i_2, ..., f(i_n, b)))\). Again, we factored out the common parts of sum and product.* *)

\[
\text{let rec collapse } (f : \text{int } \rightarrow \text{int } \rightarrow \text{int}) \ (b : \text{int}) \ (lst : \text{intlist}) : \text{int } = \\
\quad \text{match lst with} \\
\quad | \text{Nil } \rightarrow b \\
\quad | \text{Cons } (h, t) \rightarrow f \ h \ (\text{collapse } f \ b \ t)
\]

(* Now we can define \text{sum} and \text{product} in terms of \text{collapse} *)

\[
\text{let sum } (lst : \text{intlist}) : \text{int } = \\
\quad \text{let add } (i_1 : \text{int}) \ (i_2 : \text{int}) : \text{int } = i_1 + i_2 \text{ in} \\
\quad \text{collapse add 0 lst}
\]

\[
\text{let product } (lst : \text{intlist}) : \text{int } = \\
\quad \text{let mul } (i_1 : \text{int}) \ (i_2 : \text{int}) : \text{int } = i_1 * i_2 \text{ in} \\
\quad \text{collapse mul 1 lst}
\]

(* Here, we use anonymous functions instead of defining add and mul.
* After all, what's the point of giving those functions names if all we're going to do is pass them to \text{collapse}? *)

\[
\text{let sum } (lst : \text{intlist}) : \text{int } = \\
\quad \text{collapse } (\text{fun } i_1 \ i_2 \rightarrow i_1 + i_2) \ 0 \ \text{lst}
\]

\[
\text{let product } (lst : \text{intlist}) : \text{int } = \\
\quad \text{collapse } (\text{fun } i_1 \ i_2 \rightarrow i_1 * i_2) \ 1 \ \text{lst}
\]

(* Trees of integers *)

\[
\text{type inttree } = \text{Empty } | \text{Node of node and node } = \{ \text{value : int; left : inttree; right : inttree} \}
\]

(* Return true if the tree contains \(x\). *)

\[
\text{let rec search } (t : \text{inttree}) \ (x : \text{int}) : \text{bool } = \\
\quad \text{match } t \text{ with} \\
\quad | \text{Empty } \rightarrow \text{false} \\
\quad | \text{Node } \{\text{value} = v; \text{left} = l; \text{right} = r\} \rightarrow \\
\quad \quad v = x \text{ || search } l \ x \text{ || search } r \ x
\]

\[
\text{let treel } = \\
\quad \text{Node } \{\text{value} = 2; \text{left} = \text{Node } \{\text{value} = 1; \text{left} = \text{Empty}; \text{right} = \text{Empty} \}; \\
\quad \text{right} = \text{Node } \{\text{value} = 3; \text{left} = \text{Empty}; \text{right} = \text{Empty} \}
\]

9/8/2013 1:35 PM
Nil -> Nil
| Cons (h, t) -> Cons (f h, map f t)

The type of map is

('a -> 'b) -> 'a list_ -> 'b list_

The parameter \( f \) is a function from the element type of the input list \( 'a \) to the element type of the output list \( 'b \).

Using map with an anonymous function, we can define a function to make a copy of a list:

```ocaml
let copy = map (fun x -> x)
```

(This is the same as saying

```ocaml
let copy lst = map (fun x -> x) lst
```

but we don't really need to include the second argument in the definition; the \( copy \) function is of type \( 'a list_ -> 'a list_ \) and is already well defined without it.)

Similarly, we can create a \( string \) list_ from an \( int \) list_:

```ocaml
# let string_list_of_int_list = map string_of_int;;
val string_list_of_int_list : int list_ -> string list_ = <fun>
# string_list_of_int_list (Cons (1, Cons (2, Cons (3, Nil))));;
- : string list_ = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

Now let's consider the \( reduce \) operation, which like map applies a function to every element of a list, but in doing so accumulates a result rather than just producing another list. In comparison with map, the reduce operator takes an additional argument of an accumulator. As with map, we will consider the curried form of reduce.

There are two versions of reduce, based on the nesting of the applications of the function \( f \) in creating the resulting value. In OCaml there are built-in reduce functions that operate on lists are called \( List.fold_right \) and \( List.fold_left \). These functions produce the following values:

```
fold_right \( f \) [a; b; c] \( x \) = \( f \) a (\( f \) b (\( f \) c \( x \)))
fold_left \( f \) \( x \) [a; b; c] = \( f \) (\( f \) (\( f \) \( x \) a) b) c
```

From the forms of the two results, it can be seen why the functions are called \( fold_right \), which uses a right-parenthesesization of the applications of \( f \), and \( fold_left \), which uses a left-parenthesesization. Note that the formal parameters of the two functions are in different orders: in \( fold_right \) the accumulator is to the right of the list and in \( fold_left \) the accumulator is to the left.

Again using the \( list_ \) type, we can define these two functions as follows:
let rec fold_right (f : 'a -> 'b -> 'b) (lst : 'a list) (r : 'b) : 'b = 
  match lst with 
  | Nil -> r 
  | Cons (hd, tl) -> f hd (fold_right f tl r)

let rec fold_left (f : 'a -> 'b -> 'a) (r : 'a) (lst : 'b list) : 'a = 
  match lst with 
  | Nil -> r 
  | Cons (hd, tl) -> fold_left f (f hd r) tl

The types of fold_right and fold_left are

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

respectively.

The parameter f in both functions is a function from the element type of the input list and the type of the accumulator to the type of the accumulator. The types of the input list and the accumulator do not have to be the same.

Given these definitions, operations such as summing all of the elements of a list of integers can be defined naturally using either fold_right or fold_left.

let sum_right_to_left il = fold_right (+) il 0
let sum_left_to_right = fold_left (+) 0

Here (+) is the same as fun x y -> x + y. Note that we don't need the il in the second case because of the ordering of the arguments in fold_left.

The power of fold

Folding is a very powerful operation. We can write many other list functions in terms of fold. In fact map, while it initially sounded quite different from fold, can be defined naturally using fold_right by accumulating a result that is a list. Continuing with our list_type,

let map f lst = fold_right (fun x y -> Cons (f x, y)) lst Nil

The accumulator function simply applies f to each element and builds up the resulting list, starting from the empty list.

The entire map-reduce paradigm can thus be implemented using fold_left and fold_right. However, it is often conceptually useful to think of map as producing a list and of reduce as producing a value.

What about using fold_left instead of fold_right to define map? In this case we get a function that does the map, but produces an output that is in reverse order, because fold_left processes the elements of the input list left-to-right, whereas fold_right