1 Lecture Plan

1. Specifying problems about lists

2. List induction and proofs of list algorithms
   - list_induction rule, informal
   - list_induction rule, formal
   - Computational meaning
   - Simple example, list reverse
   - Proving list reverse algorithms correct

3. Introduction to search trees

4. Binary search trees as recursive types

5. Operations on binary search trees
   - insertion code
   - look up code
   - max finding code
   - deletion explanation
2 Review, Overview, and Comments

We have written several algorithms for processing lists and we have studied the powerful map reduce paradigm. Now we want to see how to reason about these algorithms. The first step is to investigate how we can precisely define the tasks these algorithms perform and the problems that they solve. We will do this first using informal language, then we will try to define the task more precisely using concepts from OCaml itself and from the theory of types that OCaml implements.

The second topic of the lecture is like “forestry,” we will study important varieties of trees and learn their properties. In this lecture we examine binary search trees (BST’s). In the next lecture we will look at a particular kind called red black trees.

3 Reading and Sources

There is a great deal of literature on how to prove properties of lists, starting with early papers by John McCarthy [3, 4]. Sir Tony Hoare’s papers on data types [1, 2] also deal with this issue. The modern formalized type theories implemented by proof assistants include hundreds of theorems about lists. We will make available some of these proofs as a resource for the course. One of the research staff on our PRL project, Anne Trostle, will be creating web pages accessible to the class that include some of these proofs. She will also post proofs that she has done for other problems we will study in the course.

Professor Kozen’s notes cover red black trees, the topic of Lecture 7.
4 Specifying Computational Problems

4.1 A very simple property

Every $\alpha$ list is either $\text{Nil}$ or has a head element $h$ in $\alpha$ and a tail list $t$ in $\alpha$ list.

$$\forall l : \alpha \text{ list}. \ (l = \text{Nil}) \lor \exists h : \alpha. \exists t : \alpha \text{ list}. (l = h :: t)$$

We can prove this by list induction. It’s so simple that it’s perplexing, so we start with something less simple, characterizing the reverse operation.

How can we specify list reversal independent of the code?

$$\forall l : \alpha \text{ list}. \exists rl : \alpha \text{ list}. \text{Reverse}(l, rl).$$

How about that? How to define $\text{Reverse}(l, rl)$?

4.2 Specification for list reversal

How do we pose the problem of reversing a list without simply giving an algorithm or examples of what the algorithm should do. The starting point for specifications can be simple examples as we illustrate.

Reversing the list of numbers $[1;2;3;4;5]$ means to produce the list $[5;4;3;2;1]$. What we immediately notice is that the reverse of a list $\text{lst}$ has the same length as $\text{lst}$. We can say this very precisely and express it with logical operators. The start of a logical specification is saying $\text{len(lst)} = \text{len}(\text{rev(lst)})$. Here is a precise statement using quantifiers.

$$\forall l : \alpha \text{ list}. \ (\text{len}(l) = \text{len}(\text{rev}(l))).$$

This statement mentions the algorithm, so it is not necessarily a good specification that we can give in advance of writing the code. A good
specification will guide writing the code. Here is a property of functions on lists that leads to an adequate explanation that does not use \textit{rev}.

We notice that this property also relates the reversed list to the original.

\begin{align*}
l &= [1; 2; 3; 4; 5] & \text{len}(l) &= 5 \\
rl &= [5; 4; 3; 2; 1] & \text{len}(rl) &= 5 \\
\text{mth} [1; 2; 3; 4; 5] 1 &= \text{mth} [5; 4; 3; 2; 1] 5 \text{ and } 5 = \text{len}(l) - 0 \\
\text{mth} [1; 2; 3; 4; 5] 2 &= \text{mth} [5; 4; 3; 2; 1] 4 \text{ and } 4 = \text{len}(l) - 1 \\
\text{mth} [1; 2; 3; 4; 5] 3 &= \text{mth} [5; 4; 3; 2; 1] 3 \text{ and } 3 = \text{len}(l) - 2 \\
\text{mth} [1; 2; 3; 4; 5] 4 &= \text{mth} [5; 4; 3; 2; 1] 2 \text{ and } 2 = \text{len}(l) - 3 \\
\text{mth} [1; 2; 3; 4; 5] 5 &= \text{mth} [5; 4; 3; 2; 1] 1 \text{ and } 1 = \text{len}(l) - 4 \\
1 \leq i \leq \text{len}(l) \Rightarrow \\
\text{mth \ } l \ i &= \text{mth \ } rl \ (\text{len}(l) - (i - 1)) \text{ or equivalently, } \\
\text{mth \ } l \ i &= \text{mth \ } rl \ ((\text{len}(l) + 1) - i).
\end{align*}

Thus

\[ \forall l : \alpha \text{ list.} \left( \text{mth \ } l \ i = \text{mth \ } rl \ (\text{len}(l) - (i - 1)) \right). \]

\textit{Reverse}(l, rl) \iff (\text{len}(l) = \text{len}(rl)) \&
\forall i : 1 \leq i \leq \text{len}(l). \left( \text{mth \ } l \ i = \text{mth \ } rl \ ((\text{len}(l) + 1) - i) \right)

We can prove this by list induction on \( l \), using a small bit of arithmetic, for all of the common definitions of \( \text{rev}(l) \).

Now we can see how to specify precisely what reverse should do. We can say that reverse should be a function \( f \) of type \( \alpha \text{ list} \to \alpha \text{ list} \) such that

\[ \forall l : \alpha \text{ list.} \left( \text{len}(l) = \text{len}(f \ l) \right) \& \\
\forall i \text{ where } 1 \leq i \leq \text{len}(l). \left( \text{mth \ } l \ i = \text{mth} \ (f \ l) \ (\text{len}(l) + 1 - i) \right). \]

We could call such a function \( f \) a \textbf{list reverser}, e.g. \textit{ListReverser}(\( f \)). This object is called a \textbf{relation} or a \textbf{predicate} on functions from lists to lists. We can prove these facts about it.
Theorem: \( \exists f : \alpha \text{ list } \to \alpha \text{ list}. \text{ListReverser}(f) \).

### 4.3 Proving reverse correct for various definitions of reverse

We know how to define at least two kinds of list reversers. One is the conceptually simple function given in previous lecture notes. It is an \( n^2 \) algorithm. We gave a more efficient linear one that uses an accumulator. Both are easy to prove correct.

**simple list reverse**
- moderately inefficient \( (n^2) \), conceptually trivial
- the idea: \( a :: t \mapsto \text{rev}(t) @ [a] \)
  \( \mapsto \text{push a rev}(t) \)

```ocaml
# let rec revs (lst : 'a list) : 'a list =
  match lst with
  | [] -> []
  | h::t -> revs t @ [h] ;;
```

**fast reverse with an accumulator** (like reversing a deck of cards by putting them on the table one by one)

```ocaml
# let rec reva (l : 'a list) (acc : 'a list) : 'a list =
  match l with
  | [] -> acc
  | h::t -> reva t (h:: acc) ;;

val reva : 'a list -> 'a list -> 'a list = <fun>

# reva [1;2;3;4;5] [] ;;
- : int list = [5; 4; 3; 2; 1]
```

Once we have one correct program, \( \text{rev} \), we can show other ones correct by proving \( f(l) = \text{rev}(l) \) by induction on \( l \).
5 List induction

To prove ∀l : α list. P(l) by induction (by list induction or list_ind) we must prove:

base case

\[ P([]) \]

induction case

assume given \( h : \alpha, t : \alpha \) list, \( i : P(t) \),
prove that \( P(h :: t) \) has evidence \( p(h, t, i) \)

rule format used by proof assistants in the “ML family”

\[
\begin{align*}
\text{base case} & \quad \cdots \quad \vdash P([]) \text{ by } p_0 \\
\text{induction case} & \quad h : \alpha, t : \alpha \text{ list}, i : P(t) \quad \vdash P(h :: t) \text{ by } p(h, t, i)
\end{align*}
\]

5.1 Correctness of list reverse

induction hypothesis

\[
\begin{align*}
t & \quad \text{rev}(t) \\
1 \leq i \leq \text{len}(t) \\
mth(t)(i) & = mth(\text{rev}(t))(\text{len}(t) + 1 - i)
\end{align*}
\]
5.2 List Reverse Theorem

\[ \forall l : \alpha \text{ list}, \forall i : \text{int}, \ 1 \leq i \leq \text{len}(l). (mth \ l \ i = mth \ (\text{rev}(l)) \ (\text{len}(l) + 1 - i)) \]

Proof by induction on \( l \)

**base case** \( l = [] \)

\( \text{rev}([]) = [] \) and the length condition is vacuous, there are no \( i \) in the range.

**induction case**

assume \( l = h :: t \) and the result holds for \( t \), i.e. \( 1 \leq i \leq \text{len}(t) \Rightarrow mth \ t \ i = mth \ (\text{rev}(t)) \ (\text{len}(t) + 1 - i) \)

where \( \text{rev}(h :: t) = \text{rev}(t) @ [h] \)

We need to show for \( 1 \leq i \leq \text{len}(h :: t) \)

\[ mth \ (h :: t) \ i = mth \ (\text{rev}(t) @ [h]) \ (\text{len}(h :: t) + 1 - i) \]

We see that

\[ mth \ (h :: t) \ 1 = h = mth \ (\text{rev}(t) @ [h]) \ k \] for \( k = \text{len}(h :: t) \)

By the induction hypothesis,

\[ mth \ t \ i = mth \ (\text{rev}(t)) \ (\text{len}(t) + 1 - i) \]

This covers the entire range of \( i \).

QED

5.3 Concluding remark on induction

Now you might see how to prove by induction the character of lists.

List Structure Theorem

\[ \forall l : \alpha \text{ list}. (l = [] \lor \exists h : \alpha. \exists t : \alpha \text{ list}. (l = h :: t)) \]

By proving this you’ll see that
(a) list induction is a form of

``` OCaml
let rec f (l : 'a list) 'b =
  match l with
  | [] -> base
  | h::t -> ind(h,t)
```

that terminates

(b) it characterizes the structure of lists

6 Binary search trees (BST’s)

Here is one

We need a type such as `int` with a linear order relation `<`. The tree has the property that at each node \( n \) the left subtree has elements less than \( n \) and the right subtree has elements greater.
These trees are designed to store elements of the type \( \alpha \) with order \(<\) so that look up is faster than in a list depth \( \log_2 \) rather than \( n \).

Definition of the “skeletal structure”

```plaintext
type int_bst = Nil | Node of (int * int_bst * int_bst)
```

The key operation is insertion, it creates the binary structure.

```plaintext
let rec insrt_int_bst (bst : int_bst) (x : int) =
  match bst with
  | Nil -> Node (x, Nil, Nil)
  | Node (v, t1, t2) ->
    if x < v then Node (v, insrt_int_bst t1 x, t2)
    else (if x > v then Node (v, t1, insrt_int_bst t2 x)
           else bst) (* x = v *)
```

References


Lecture 6 BST code

BINARY SEARCH TREE (BST) CODE

(* BST over int data type definition *)

type int_bst = Nil | Node of (int * int_bst * int_bst)

(* Example of BST over int *)

# let t1 = Node (82, Node (70, Nil, Node (71, Nil, Nil)), Node (87, Node (86, Nil, Nil), Node (90, Nil, Nil))) ;;
val t1 : bi_int_tr = Node
  (82, Node (70, Nil, Node (71, Nil, Nil)),
   Node (87, Node (86, Nil, Nil), Node (90, Nil, Nil)))

(* code for inserting integers into int_bst *)

# let rec insrt_int_bst (bst : int_bst) (x:int) = match bst with Nil -> Node (x,Nil,Nil)
| Node (v, t1, t2) ->
  if x < v then Node (v, insrt_int_bst t1 x, t2)
  else (if x > v then Node (v, t1, insrt_int_bst t2 x ) else bst) ;;
val insrt_int_bst : int_bst -> int -> int_bst = <fun>

(* example of insert_int_bst in action *)

# let t2 = insrt_int_bst t1 42;;
val t2 : int_bst =
  Node
  (82, Node (70, Node (42, Nil, Nil), Node (71, Nil, Nil)),
   Node (87, Node (86, Nil, Nil), Node (90, Nil, Nil)))

(* code for finding an element of an int_bst *)

let rec find (bst: int_bst) (x:int) : int list = match bst with Nil -> [] | Node (v,tlft,trgt) ->
  (if x = v then [v] else (if x < v then find tlft x else find trgt x )) ;;

# find (Node (42, Nil, Nil)) 42;;
- : int list = [42]
# find t2 42 ;;
- : int list = [42]

(* finding the maximum element in a bst *)

# let rec max_bst (bst: int_bst) (start:int) : int = match bst with Nil -> start |
  Node (v,tlft,rgt) ->
  if v = start then match rgt with Nil -> start |
  Node (w,tlft2,rgt2) -> (if start < w then (max_bst rgt2 w) else start) else (max_bst rgt v) ;;;
val max_bst : int_bst -> int -> int = <fun>
# max_bst t2 82 ;;
- : int = 90