Outline of Lecture 8

1. Lists (to represent finite sequences).
2. More induction
Isabelle’s lists are descended from the Lisp language, they are built up using two operations:

1. The empty list: `[]`
2. If $x$ is an object, and $ls$ is a list of objects, then $x \# ls$ is a new list with $x$ at the front and the rest being the same as $ls$.

Also, lists can be created from a comma-separated list enclosed in brackets: `[1, 2, 3, 4]`.

All the objects in a list must have the same type.
You can write primitive recursive functions over lists:

```plaintext
primrec app :: "'a list ⇒ 'a list ⇒ 'a list" where
  "app [] ys = ys" |
  "app (x#xs) ys = x # (app xs ys)"

lemma "app [1,2] [3,4] = [1,2,3,4]" by simp

primrec reverse :: "'a list ⇒ 'a list" where
  "reverse [] = []" |
  "reverse (x#xs) = app (reverse xs) [x]"

lemma "reverse [1,2,3,4] = [4,3,2,1]" by simp
```
Induction on Lists and the Theorem Proving Process

theorem rev_rev_id: "reverse (reverse xs) = xs"
proof (induct xs)
  show "reverse (reverse []) = []" by simp
next
  fix a xs assume IH: "reverse (reverse xs) = xs"
  — We can expand the LHS of the goal as follows
  have "reverse (reverse (a # xs))
           = reverse (app (reverse xs) [a])" by simp
  — But then we’re stuck. How can we use the IH?
  — Can we push the outer reverse under the app?
  show "reverse (reverse (a # xs)) = a # xs"
    oops
Reverse-Append Lemma

reverse(app(xs,ys)) = app(reverse(ys), reverse(xs))
lemma rev_app:
  "reverse (app xs ys) = app (reverse ys) (reverse xs)"
proof (induct xs)
  have 1: "reverse (app [] ys) = reverse ys" by simp
  have 2: "app (reverse ys) (reverse []) = app (reverse ys) []"
    by simp
    — but no we’re stuck
  show "reverse (app [] ys) = app (reverse ys) (reverse [])"
    oops

Exercise: what additional lemma do we need? Prove the additional lemma.
The Append-Nil Lemma

lemma app_nil: "(app xs []) = xs"
proof (induct xs)
  show "app [] [] = []" by simp
next
  fix a xs assume IH: "app xs [] = xs"
  have "app (a # xs) [] = a # (app xs [])" by simp
  also from IH have "... = a # xs" by simp
  finally show "app (a # xs) [] = a # xs" .
qed
lemma rev_app:
  "reverse (app xs ys) = app (reverse ys) (reverse xs)"
proof (induct xs)
  show "reverse (app [] ys) = app (reverse ys) (reverse [])"
    using app_nil[of "reverse ys"] by simp
next
  fix a xs assume IH: "reverse (app xs ys) = app (reverse ys) (reverse xs)"
  have "reverse (app (a # xs) ys) = reverse (a # (app xs ys))" by simp
  also have "... = app (reverse (app xs ys) ) [a]" by simp
  also have "... = app (app (reverse ys) (reverse xs)) [a]"
    using IH by simp
  — We’re stuck again! What lemma do we need this time?
  show "reverse (app (a # xs) ys) = app (reverse ys) (reverse (a # xs))"
  oops
Associativity of Append

lemma app_assoc: "app (app xs ys) zs = app xs (app ys zs)"

oops
lemma app_assoc: "app (app xs ys) zs = app xs (app ys zs)"
proof (induct xs)
  show "app (app [] ys) zs = app [] (app ys zs)" by simp
next
  fix a xs assume IH: "app (app xs ys) zs = app xs (app ys zs)"
  from IH
  show "app (app (a # xs) ys) zs = app (a # xs) (app ys zs)"
    by simp
qed
Back to the Reverse-Append Lemma, Again

lemma rev_app:
  "reverse (app xs ys) = app (reverse ys) (reverse xs)"
proof (induct xs)
  show "reverse (app [] ys) = app (reverse ys) (reverse [])"
    using app_nil[of "reverse ys"] by simp
next
  fix a xs assume IH: "reverse (app xs ys)
    = app (reverse ys) (reverse xs)"
  have "reverse (app (a # xs) ys)
    = reverse (a # (app xs ys))" by simp
  also have "... = app (reverse (app xs ys)) [a]" by simp
  also have "... = app (app (reverse ys) (reverse xs)) [a]"
    using IH by simp
  also have "... = app (reverse ys) (app (reverse xs) [a])"
    using app_assoc[of "reverse ys" "reverse xs" "[a]""] by simp
  also have "... = app (reverse ys) (reverse (a # xs))" by simp
  finally show "reverse (app (a # xs) ys)
    = app (reverse ys) (reverse (a # xs))".
qed
Finally, Back to the Theorem!

**Theorem** rev_rev_id: "reverse (reverse xs) = xs"

**Proof** (induct xs)

- show "reverse (reverse []) = []" by simp

next

- fix a xs assume IH: "reverse (reverse xs) = xs"
  - We can expand the LHS of the goal as follows
  - have "reverse (reverse (a # xs))" 
    = reverse (app (reverse xs) [a])" by simp
  - also have "... = app (reverse [a]) (reverse (reverse xs))"
    using rev_app[of "reverse xs" "[a]"] by simp
  - also from IH have "... = app (reverse [a]) xs" by simp
  - also have "... = a # xs" by simp
  - finally show "reverse (reverse (a # xs)) = a # xs" .

qed
More on Lists and the Theorem Proving Process

- When proving something about a recursive function, induct on the argument that is decomposed by the recursive function (e.g., the first argument of append).
- The pattern of getting stuck and then proving lemmas is normal.
- Isabelle provides many functions and theorems regarding lists. See Isabelle/src/HOL/List.thy for more details.
Use lists to represent finite sequences.
Isabelle provides many functions and theorems regarding lists. See Isabelle/src/HOL/List.thy for more details.
Proofs often require several lemmas.
Generalize your lemmas to make the induction go through.