INDUCTIVE PROOF

PRIMITIVE RECURSION

HOW TO DEFINE DATATYPES ?

datatype — the general case

constructor/type — arguments

datatype $(\alpha_1, \ldots, \alpha_n)t \;=\; C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \mid \ldots \mid C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)t$
- *Distinctness:* $C_i \ldots \neq C_j \ldots$ if $i \neq j$
- *Injectivity:* $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \cdots \wedge x_{n_i} = y_{n_i})$

Distinctness and injectivity are applied automatically
Induction must be applied explicitly

Recursive Functions on Inductively Defined Data

Functions can defined by recursion on "structurally smaller" data.

primrec length :: "*'a list ⇒ nat*"
where
"length Nil = Zero" |
"length (Cons x xs) = Succ (length xs)"

primrec append :: "*'a list ⇒ 'a list ⇒ 'a list*"
where
"append Nil ys = ys" |
"append (Cons x xs) ys = Cons x (append xs ys)"

primrec reverse :: "*'a list ⇒ 'a list*"
where
"reverse Nil = Nil" |
"reverse (Cons x xs) = append (reverse xs) (Cons x Nil)"

Proof by Structural Induction

Properties of structurally recursive functions can be proved by **structural induction**.

To show $\forall xs.\ P\ xs$:

$\begin{cases} \text{prove } P\ \text{Nil} \\ \text{for all } x, xs, \text{ assume } P\ xs \text{ to prove } P\ (\text{Cons}\ x\ xs) \end{cases}$

P holds for empty list

To prove: append xs (append ys zs) = append (append xs ys) zs:

(base)

$$\begin{aligned} \text{append Nil (append } ys\ zs) &= \text{append } ys\ zs \\ &= \text{append (append Nil } ys)\ zs \end{aligned}$$

(step)

$$\begin{aligned} & \text{append (Cons } x\ xs)\ (\text{append } ys\ zs) \\ =\ & \text{Cons } x\ (\text{append } xs\ (\text{append } ys\ zs)) \\ =\ & \text{Cons } x\ (\text{append (append } xs\ ys)\ zs) \quad \text{by IH} \\ =\ & \text{append (Cons } x\ (\text{append } xs\ ys))\ zs \\ =\ & \text{append (append (Cons } x\ xs)\ ys)\ zs \end{aligned}$$

In practice: start with the equation to be proved as the goal, and rewrite both sides to be equal.

Structural induction for *list*

This is analogous to the one for natural numbers (see the lecture on Isar).

```
show P(xs)
proof (induction xs)
  case Nil
  ⋮
  show ?case
next
  case (Cons x xs)
  ⋮
  show ?case
qed
```

Struct. Induction for our type nat

```
show P(n)
proof (induction n)
  case Zero
  ⋮
  show ?case
next
  case (Succ n)
  ⋮
  show ?case
qed
```

Well-Founded Induction

→ Let $<$ be an ordering on a set such that, for all x, there are no infinite downward chains:

Not allowed: $\ldots < \ldots < x_3 < x_2 < x_1 < x$

Such an ordering is called *well-founded* (or *noetherian*)

→ Then, to prove $\forall x.\ P\ x$, it suffices to prove:

$$\forall y.\ (\forall z.\ z < y \rightarrow P\ z) \rightarrow P\ y$$

$z<y$ implies $P(z)$

Specialised to the natural numbers, with the usual less-than ordering, this is usually called **Complete Induction**.

The Need for Intermediate Lemmas

Practically, the lack of a guarantee of a proof with the sub-formula property means that we need *creative generalisation* during proofs, or we need to *speculate new lemmas.*

To prove: reverse (reverse xs) = xs

(base) reverse (reverse Nil) = reverse Nil = Nil

(step) IH: reverse (reverse xs) = xs
Attempt: reverse (reverse (Cons x xs))
= reverse (append (reverse xs) (Cons x Nil))
???? We're stuck, so...
= Cons x xs

We need to *speculate* a new lemma.

Theoretical Limitations of Automated Inductive Proof

Recall L-systems, with left- and right-introduction rules:

$$\frac{\Gamma, P, Q \vdash R}{\Gamma, P \wedge Q \vdash R}\ (\text{e conjE}) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ (\text{disjI1}) \qquad \frac{\Gamma \vdash P \qquad \Gamma, P \vdash Q}{\Gamma \vdash Q}\ (\text{cut})$$

→ This system has two nice properties:

1. *Cut elimination:* the cut rule is unnecessary
2. *Sub-formula property:* every cut-free proof only contains formulas which are sub-formulas of the original goal
   ($Q(t)$ is a sub-formula of $\forall x.\ Q(x)$ and $\exists x.\ Q(x)$, for any t)

So can do complete (but possibly non-terminating) proof search.
↳ If smthg is provable, bc. of these 2 properties, it is possible to find a proof

→ If we add an induction rule:

$$\frac{\underbrace{\Gamma \vdash P(0)}_{\text{Base case}} \qquad \overbrace{\Gamma, P(n) \vdash P(n+1)}^{\text{step case}} \qquad n \notin fv(\Gamma, P)}{\Gamma \vdash \forall n. P(n)}$$

Then Cut elimination fails!

There are variant rules that bring it back, but sub-formula property still fails

Isabelle generates the induction rule automatically!

A New Lemma (LEMMA SPECULATION!)

In this case, it turns out that we need:

reverse (append xs ys) = append (reverse ys) (reverse xs)

(which is proved by induction, and needs *another* lemma)

Now we can proceed:

(step) IH: reverse (reverse xs) = xs
Attempt:
reverse (reverse (Cons x xs))
= reverse (append (reverse xs) (Cons x Nil))
= append (Cons x Nil) (reverse (reverse xs)) by lemma
= Cons x (append Nil (reverse (reverse xs)))
= Cons x (reverse (reverse xs))
= Cons x xs by IH

Another approach

We got stuck trying to prove:

$$\text{reverse (append (reverse } xs\text{) (Cons } x \text{ Nil))} = \text{Cons } x\ xs$$

under the assumption that reverse (reverse xs) = xs

What if we rewrite the RHS *backwards* by the IH, to get the new goal:

$$\text{reverse (append (reverse } xs\text{) (Cons } x \text{ Nil))} = \text{Cons } x \text{ (reverse (reverse } xs\text{))}$$

you have reverse xs on both sides

Maybe this can be proved by induction? GENERALISATION!

Not quite (try it and see!); need to *generalise* and prove:

$$\text{reverse (append } xs \text{ (Cons } x \text{ Nil))} = \text{Cons } x \text{ (reverse } xs\text{)}$$

(A special case of the lemma speculated earlier)

Challenges in Automating Inductive Proofs

Theoretically, and practically, to do inductive proofs, we need:

- Lemma speculation
- Generalisation

Techniques (other than "Get the user to do it"):

- Boyer-Moore approach
  roughly the approach described here (implemented in ACL2)
- Rippling, "Productive Use of Failure" (Bundy and Ireland, 1996)
- Up-front speculation:
  e.g. "maybe this binary function is associative?"
- Cyclic proofs
  (search for a circular proof, and afterwards prove it is well-founded)
- Only doing a few cases $(0, 1, ..., 6)$
- Special purpose techniques (e.g., generating functions)

Consider recursive defn of addition over natural no.

$$0 + X = X$$

$$s(X) + Y = s(X + Y)$$

where s is the successor fn. What is the lemma needed to be speculated in the step case to prove "$X+Y=Y+X$" by induction?

$$\boxed{s(Y+X) = Y + s(X)}$$

1) Pick a var. in the goal to do induction on. Here, X is most appropriate.

2) (BASE) $0 + Y = Y$

3) (STEP) Assume $X+Y=Y+X$, show $s(X)+Y = Y+s(X)$

$$s(X) + Y = s(X+Y)$$
$$= s(Y+X)$$
$$= ???$$

$\boxed{s(Y+X) = Y+s(X)}$

$Y + s(X)$

INDUCTIVELY DEFINED DATA

→ Inductive datatypes are <u>freely generated</u> by some constructors:

datatype nat = Zero | Succ nat

datatype 'a list = Nil | Cons "'a" "'a list"

datatype 'a tree = Leaf "'a" | Node "'a" "'a tree" "'a tree"

→ $\boxed{\text{Free datatypes}}$ are those for which terms are only equal if they are <u>syntactically identical</u> (e.g. Succ (Succ Zero) ≠ Succ Zero)

some values: { Succ (Succ Zero) i.e. "2"
Cons Zero (Cons Zero Nil) i.e. "[0, 0]"

→ $\boxed{\text{Non-freely generated datatypes.}}$

Contrast the above w/ the integers, e.g., defined w/ the constructors <u>Zero</u>, <u>Succ</u> and <u>Pred</u> where Zero and Succ are as for the natural num. but Pred is the predecessor fn:

∴ In this case, Pred (Succ n) = Succ (Pred n) = n

EQUAL! even if they are not syntactically identical.