Automated Reasoning

Lecture 9: Inductive Proof (in Isabelle)

Jacques Fleuriot
jdf@inf.ed.ac.uk

Overview

- Proof by Induction (in Isabelle)
  - Inductive Datatypes
  - Mathematical Induction
  - Structural Recursion and Induction
  - Challenges in Inductive Proof Automation

A Summation Problem

What is

$$1 + 2 + 3 + \ldots + 999 + 1000 \quad ?$$

Is there a general formula for any n?

A Summation Problem

What is

$$1 + 2 + 3 + \ldots + 999 + 1000 \quad ?$$

Is there a general formula for any n?

Gauss came up with the solution:

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

A Summation Problem

What is

$$1 + 2 + 3 + \ldots + 999 + 1000 \quad ?$$

Is there a general formula for any n?

Gauss came up with the solution:

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

How can we prove this? (Automatically?)

- First-order proof search is (generally) unable to prove this

Proof by Induction

To prove $\forall n.\ P\ n$:

$$\begin{cases} \text{(base) prove } P\ 0 \\ \text{(step) for all } n\text{, assume } P\ n \text{ and prove } P\ (n+1) \end{cases}$$

Proof by Induction

To prove $\forall n.\ P\ n$:

$$\begin{cases} \text{(base) prove } P\ 0 \\ \text{(step) for all } n\text{, assume } P\ n \text{ and prove } P\ (n+1) \end{cases}$$

To prove $\forall n.\ 1 + 2 + \ldots + n = \dfrac{n(n+1)}{2}$:

Proof by Induction

To prove $\forall n.\ P\ n$:

$$\begin{cases} \text{(base) prove } P\ 0 \\ \text{(step) for all } n\text{, assume } P\ n \text{ and prove } P\ (n+1) \end{cases}$$

To prove $\forall n.\ 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$:

(base): $0 = \frac{0 * 1}{2}$, by computation.

Proof by Induction

To prove $\forall n.\ P\ n$:

$$\begin{cases} \text{(base) prove } P\ 0 \\ \text{(step) for all } n \text{, assume } P\ n \text{ and prove } P\ (n+1) \end{cases}$$

To prove $\forall n.\ 1 + 2 + \ldots + n = \dfrac{n(n+1)}{2}$:

(base): $0 = \dfrac{0 * 1}{2}$, by computation.

(step): assume the formula holds for n, and:

$$\begin{array}{cll} & 1 + 2 + \ldots + n + (n+1) & \\ = & (1 + 2 + \ldots + n) + (n+1) & \\ = & \dfrac{n(n+1)}{2} + (n+1) & \text{(apply induction hypothesis)} \\ = & \ldots & \\ = & \dfrac{(n+1)(n+2)}{2} & \end{array}$$

as required.

Inductively Defined Data

Induction is especially useful for dealing with **Inductive Datatypes**

Inductively Defined Data

Induction is especially useful for dealing with **Inductive Datatypes**

Inductive Datatypes are *freely generated* by some *constructors*:

$$\begin{array}{l}
\textbf{datatype}\ \mathit{nat} = \mathsf{Zero} \mid \mathsf{Succ}\ \mathit{nat} \\
\textbf{datatype}\ {}'a\ \mathit{list} = \mathsf{Nil} \mid \mathsf{Cons}\ \texttt{"}{}'a\texttt{"}\ \texttt{"}{}'a\ \mathit{list}\texttt{"} \\
\textbf{datatype}\ {}'a\ \mathit{tree} = \mathsf{Leaf}\ \texttt{"}{}'a\texttt{"} \mid \mathsf{Node}\ \texttt{"}{}'a\texttt{"}\ \texttt{"}{}'a\ \mathit{tree}\texttt{"}\ \texttt{"}{}'a\ \mathit{tree}\texttt{"}
\end{array}$$

Inductively Defined Data

Induction is especially useful for dealing with **Inductive Datatypes**

Inductive Datatypes are *freely generated* by some *constructors*:

$$\textbf{datatype } \mathit{nat} = \mathsf{Zero} \mid \mathsf{Succ}\ \mathit{nat}$$
$$\textbf{datatype } {}'a\ \mathit{list} = \mathsf{Nil} \mid \mathsf{Cons}\ \texttt{"}{}'a\texttt{"}\ \texttt{"}{}'a\ \mathit{list}\texttt{"}$$
$$\textbf{datatype } {}'a\ \mathit{tree} = \mathsf{Leaf}\ \texttt{"}{}'a\texttt{"} \mid \mathsf{Node}\ \texttt{"}{}'a\texttt{"}\ \texttt{"}{}'a\ \mathit{tree}\texttt{"}\ \texttt{"}{}'a\ \mathit{tree}\texttt{"}$$

Free datatypes are those for which terms are only equal if they are **syntactically identical** e.g. $\mathsf{Succ}\ (\mathsf{Succ}\ \mathsf{Zero}) \neq \mathsf{Succ}\ \mathsf{Zero}$.

$$\text{Some values: } \begin{cases} \mathsf{Succ}\ (\mathsf{Succ}\ \mathsf{Zero}) & \text{i.e. "}2\text{"} \\ \mathsf{Cons}\ \mathsf{Zero}\ (\mathsf{Cons}\ \mathsf{Zero}\ \mathsf{Nil}) & \text{i.e. "}[0,0]\text{"} \end{cases}$$

Inductively Defined Data

Induction is especially useful for dealing with **Inductive Datatypes**

Inductive Datatypes are *freely generated* by some *constructors*:

$$\textbf{datatype } nat = \text{Zero} \mid \text{Succ } nat$$
$$\textbf{datatype } 'a\ list = \text{Nil} \mid \text{Cons "}'a\text{" "}'a\ list\text{"}$$
$$\textbf{datatype } 'a\ tree = \text{Leaf "}'a\text{"} \mid \text{Node "}'a\text{" "}'a\ tree\text{" "}'a\ tree\text{"}$$

Free datatypes are those for which terms are only equal if they are **syntactically identical** e.g. Succ (Succ Zero) $\neq$ Succ Zero.

$$\text{Some values: } \begin{cases} \text{Succ (Succ Zero)} & \text{i.e. "}2\text{"} \\ \text{Cons Zero (Cons Zero Nil)} & \text{i.e. "}[0,0]\text{"} \end{cases}$$

Non-freely generated datatypes. Contrast the above with the integers, for example, defined with the constructors Zero, Succ and Pred, where Zero and Succ are as for the natural numbers but Pred is the predecessor function.

In this case, Pred (Succ n) = Suc (Pred n) = n, for instance.

datatype — the general case

$$
\begin{array}{lcl}
\textbf{datatype } (\alpha_1, \ldots, \alpha_n)t & = & C_1 \ \tau_{1,1} \ldots \tau_{1,n_1} \\
 & \mid & \ldots \\
 & \mid & C_k \ \tau_{k,1} \ldots \tau_{k,n_k}
\end{array}
$$

datatype — the general case

$$\begin{array}{lcl} \textbf{datatype } (\alpha_1, \ldots, \alpha_n)t & = & C_1 \ \tau_{1,1} \ldots \tau_{1,n_1} \\ & \mid & \ldots \\ & \mid & C_k \ \tau_{k,1} \ldots \tau_{k,n_k} \end{array}$$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)t$

datatype — the general case

$$\begin{array}{lcl} \textbf{datatype}\ (\alpha_1,\ldots,\alpha_n)t & = & C_1\ \tau_{1,1}\ldots\tau_{1,n_1} \\ & \mid & \ldots \\ & \mid & C_k\ \tau_{k,1}\ldots\tau_{k,n_k} \end{array}$$

- *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 $i \neq j$

datatype — the general case

$$\begin{array}{lcl} \textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)t & = & C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \\ & \mid & \ldots \\ & \mid & C_k\ \tau_{k,1} \ldots \tau_{k,n_k} \end{array}$$

- *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:* $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) =$ $(x_1 = y_1 \wedge \cdots \wedge x_{n_i} = y_{n_i})$

datatype — the general case

$$\begin{array}{lcl}\textbf{datatype}\ (\alpha_1,\ldots,\alpha_n)t & = & C_1\ \tau_{1,1}\ldots\tau_{1,n_1} \\ & \mid & \ldots \\ & \mid & C_k\ \tau_{k,1}\ldots\tau_{k,n_k}\end{array}$$

- *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* $\Rightarrow$ *nat*"
where
"length Nil = Zero" |
"length (Cons x xs) = Succ (length xs)"

Recursive Functions on Inductively Defined Data

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

primrec length :: "$'a\ list \Rightarrow nat$"
where
"length Nil = Zero" |
"length (Cons x xs) = Succ (length xs)"

primrec append :: "$'a\ list \Rightarrow\ 'a\ list \Rightarrow\ 'a\ list$"
where
"append Nil ys = ys" |
"append (Cons x xs) ys = Cons x (append xs ys)"

Recursive Functions on Inductively Defined Data

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

primrec length :: "$'a\ list \Rightarrow nat$"
where
"length Nil = Zero" |
"length (Cons x xs) = Succ (length xs)"

primrec append :: "$'a\ list \Rightarrow\ 'a\ list \Rightarrow\ 'a\ list$"
where
"append Nil ys = ys" |
"append (Cons x xs) ys = Cons x (append xs ys)"

primrec reverse :: "$'a\ list \Rightarrow\ 'a\ list$"
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 $\forall xs.\ P\ xs$:

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

Proof by Structural Induction

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

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

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

To prove: $\text{append}\ xs\ (\text{append}\ ys\ zs) = \text{append}\ (\text{append}\ xs\ ys)\ zs$:

Proof by Structural Induction

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

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

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

To prove: $\mathrm{append}\ xs\ (\mathrm{append}\ ys\ zs) = \mathrm{append}\ (\mathrm{append}\ xs\ ys)\ zs$:

(base)

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

Proof by Structural Induction

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

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

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

To prove: $\mathrm{append}\ xs\ (\mathrm{append}\ ys\ zs) = \mathrm{append}\ (\mathrm{append}\ xs\ ys)\ zs$:

(base)

$$\begin{array}{rcl} \mathrm{append}\ \mathsf{Nil}\ (\mathrm{append}\ ys\ zs) & = & \mathrm{append}\ ys\ zs \\ & = & \mathrm{append}\ (\mathrm{append}\ \mathsf{Nil}\ ys)\ zs \end{array}$$

(step)

$$\begin{array}{rll} & \mathrm{append}\ (\mathsf{Cons}\ x\ xs)\ (\mathrm{append}\ ys\ zs) & \\ = & \mathsf{Cons}\ x\ (\mathrm{append}\ xs\ (\mathrm{append}\ ys\ zs)) & \\ = & \mathsf{Cons}\ x\ (\mathrm{append}\ (\mathrm{append}\ xs\ ys)\ zs) & \text{by IH} \\ = & \mathrm{append}\ (\mathsf{Cons}\ x\ (\mathrm{append}\ xs\ ys))\ zs & \\ = & \mathrm{append}\ (\mathrm{append}\ (\mathsf{Cons}\ x\ xs)\ ys)\ zs & \end{array}$$

Proof by Structural Induction

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

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

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

To prove: $\text{append}\ xs\ (\text{append}\ ys\ zs) = \text{append}\ (\text{append}\ xs\ ys)\ zs$:

(base)

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

(step)

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

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

Structural induction for our type *nat*

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

Structural induction for our type *nat*

show *P(n)*
proof *(induction n)*
  case *Zero* $\equiv$ **let** *?case = P(Zero)*
  $\vdots$
  show *?case*
next
  case *(Succ n)*
  $\vdots$
  $\vdots$
  show *?case*
qed

Structural induction for our type *nat*

show $P(n)$
proof $(induction\ n)$
  case $Zero$ $\equiv$ **let** $?case = P(Zero)$
  $\vdots$
  show $?case$
next
  case $(Succ\ n)$ $\equiv$ **fix** n **assume** $Succ\text{:}\ P(n)$
  $\vdots$ **let** $?case = P(Succ\ n)$
  $\vdots$
  show $?case$
qed

Structural induction for *list*

This is analogous to the one for natural numbers.

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

Well-Founded Induction

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

$$\text{Not allowed:} \qquad \ldots < \ldots < x_3 < x_2 < x_1 < x$$

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

Well-Founded Induction

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

$$\text{Not allowed:} \qquad \ldots < \ldots < x_3 < x_2 < x_1 < x$$

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

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

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

Well-Founded Induction

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

$$\text{Not allowed:} \qquad \ldots < \ldots < x_3 < x_2 < x_1 < x$$

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

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

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

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

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.

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 we add an induction rule:

$$\frac{\Gamma \vdash P(0) \qquad \Gamma, P(n) \vdash P(n+1) \qquad n \notin fv(\Gamma, P)}{\Gamma \vdash \forall n. P(n)}$$

Then Cut elimination fails!

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

The Need for Intermediate Lemmas

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

The Need for Intermediate Lemmas

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

To prove: $\mathrm{reverse}\ (\mathrm{reverse}\ xs) = xs$

The Need for Intermediate Lemmas

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

To prove: $\mathrm{reverse}\ (\mathrm{reverse}\ xs) = xs$

(base) $\mathrm{reverse}\ (\mathrm{reverse}\ \mathsf{Nil}) = \mathrm{reverse}\ \mathsf{Nil} = \mathsf{Nil}$

The Need for Intermediate Lemmas

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

To prove: $\mathrm{reverse}\ (\mathrm{reverse}\ xs) = xs$

(base) $\mathrm{reverse}\ (\mathrm{reverse}\ \mathsf{Nil}) = \mathrm{reverse}\ \mathsf{Nil} = \mathsf{Nil}$

(step) IH: $\mathrm{reverse}\ (\mathrm{reverse}\ xs) = xs$

Attempt:

$$\begin{array}{rl} & \mathrm{reverse}\ (\mathrm{reverse}\ (\mathsf{Cons}\ x\ xs)) \\ = & \mathrm{reverse}\ (\mathrm{append}\ (\mathrm{reverse}\ xs)\ (\mathsf{Cons}\ x\ \mathsf{Nil})) \\ & ???? \\ = & \mathsf{Cons}\ x\ xs \end{array}$$

The Need for Intermediate Lemmas

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

To prove: $\text{reverse}\ (\text{reverse}\ xs) = xs$

(base) $\text{reverse}\ (\text{reverse}\ \mathsf{Nil}) = \text{reverse}\ \mathsf{Nil} = \mathsf{Nil}$

(step) IH: $\text{reverse}\ (\text{reverse}\ xs) = xs$

Attempt:
$$\begin{array}{rl} & \text{reverse}\ (\text{reverse}\ (\mathsf{Cons}\ x\ xs)) \\ = & \text{reverse}\ (\text{append}\ (\text{reverse}\ xs)\ (\mathsf{Cons}\ x\ \mathsf{Nil})) \\ & ???? \\ = & \mathsf{Cons}\ x\ xs \end{array}$$

We need to *speculate* a new lemma.

A New Lemma

In this case, it turns out that we need:

$$\text{reverse (append } xs\ ys) = \text{append (reverse } ys) \text{ (reverse } xs)$$

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

A New Lemma

In this case, it turns out that we need:

$$\text{reverse } (\text{append } xs \ ys) = \text{append } (\text{reverse } ys) \ (\text{reverse } xs)$$

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

Now we can proceed:

(step) IH: $\text{reverse } (\text{reverse } xs) = xs$

Attempt:

$$\begin{array}{lll}
 & \text{reverse } (\text{reverse } (\mathsf{Cons}\ x\ xs)) & \\
= & \text{reverse } (\text{append } (\text{reverse } xs)\ (\mathsf{Cons}\ x\ \mathsf{Nil})) & \\
= & \text{append } (\text{reverse}(\mathsf{Cons}\ x\ \mathsf{Nil}))\ (\text{reverse } (\text{reverse } xs)) & \text{by lemma} \\
= & \text{append } (\text{append } (\text{reverse } \mathsf{Nil})\ (\mathsf{Cons}\ x\ \mathsf{Nil}))\ (\text{reverse } (\text{reverse } xs)) & \\
= & \text{append } (\text{append } \mathsf{Nil}\ (\mathsf{Cons}\ x\ \mathsf{Nil}))\ (\text{reverse } (\text{reverse } xs)) & \\
= & \text{append } (\mathsf{Cons}\ x\ \mathsf{Nil})\ (\text{reverse } (\text{reverse } xs)) & \\
= & \mathsf{Cons}\ x\ (\text{append } \mathsf{Nil}\ (\text{reverse } (\text{reverse } xs)) & \\
= & \mathsf{Cons}\ x\ (\text{reverse } (\text{reverse } xs)) & \\
= & \mathsf{Cons}\ x\ xs & \text{by IH}
\end{array}$$

Another approach

We got stuck trying to prove:

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

under the assumption that $\text{reverse } (\text{reverse } xs) = xs$

Another approach

We got stuck trying to prove:

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

under the assumption that $\text{reverse}\ (\text{reverse}\ xs) = xs$

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

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

Maybe this can be proved by induction?

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 $\text{reverse (reverse } xs) = xs$

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

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

Maybe this can be proved by induction?

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

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

(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), Higher Order Rippling in IsaPlanner (Dixon and Fleuriot, 2004)
- 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 $(0, 1, ..., 6)$
- Can machine learning help?
  e.g. Machine Learning for Automated Inductive Theorem Proving (Jiang, Papapanagiotou and Fleuriot, 2019)

Summary

- Proof by Induction (in Isabelle)
  - Natural number induction
  - Inductive Datatypes and Structural Induction (H&R 1.4.2)
  - The automation of Mathematical Induction by Bundy (see AR webpage).
  - The need for generalisation and lemma speculation