

# Automated Reasoning

## Lecture 17: Inductive Proof (in Isabelle)

Jacques Fleuriot  
`jdf@inf.ed.ac.uk`

# Recap

- ▶ Previously:
  - ▶ Unification and Rewriting
- ▶ This time: Proof by Induction (in Isabelle)
  - ▶ Proof by Mathematical Induction
  - ▶ Structural Recursion and Induction
  - ▶ Challenges in Inductive Proof Automation

# A Summation Problem

What is

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

Is there a general formula for any  $n$ ?

# A Summation Problem

What is

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

Is there a general formula for any  $n$ ?

Gauss's solution:

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

# A Summation Problem

What is

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

Is there a general formula for any  $n$ ?

Gauss's solution:

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

How can we prove this? (Automatically?)

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

# Proof by Induction

To prove  $\forall n. P n$ :

- { (base) prove  $P 0$
- { (step) for all  $n$ , assume  $P n$  and prove  $P (n + 1)$

# Proof by Induction

To prove  $\forall n. P n$ :

- { (base) prove  $P 0$
- { (step) for all  $n$ , assume  $P n$  and prove  $P (n + 1)$

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

# Proof by Induction

To prove  $\forall n. P n$ :

- { (base) prove  $P 0$
- { (step) for all  $n$ , assume  $P n$  and prove  $P (n + 1)$

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

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

# Proof by Induction

To prove  $\forall n. P n$ :

- { (base) prove  $P 0$
- { (step) for all  $n$ , assume  $P n$  and prove  $P (n + 1)$

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

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

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

$$\begin{aligned} & 1 + 2 + \dots + n + (n + 1) \\ = & (1 + 2 + \dots + n) + (n + 1) \\ = & \frac{n(n+1)}{2} + (n + 1) \quad (\text{apply induction hypothesis}) \\ = & \dots \\ = & \frac{(n+1)(n+2)}{2} \end{aligned}$$

as required.

# Inductively Defined Data

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

Inductive Datatypes are *freely generated* by some constructors.

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

**datatype**  $nat = \text{Zero} \mid \text{Succ } nat$

**datatype**  $'a\ list = \text{Nil} \mid \text{Cons } "'a" \ "'a\ list"$

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

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

In this case,  $\text{Pred}(\text{Succ } n) = \text{Suc}(\text{Pred } n) = n$ , for instance.

## datatype — the general case

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

## datatype — the general case

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

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

## datatype — the general case

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

- ▶ *Types:*  $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_n)t$
- ▶ *Distinctness:*  $C_i \dots \neq C_j \dots$  if  $i \neq j$

## datatype — the general case

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

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

## datatype — the general case

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

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

Distinctness and injectivity are applied automatically  
Induction must be applied explicitly

# Recursive Functions on Inductively Defined Data

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

`primrec length :: "'a list ⇒ nat"`

`where`

`"length Nil = Zero" |`

`"length (Cons x xs) = Succ (length xs)"`

# Recursive Functions on Inductively Defined Data

Functions can be 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 be 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  $\forall xs. P xs$ :

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

## Proof by Structural Induction

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

To show  $\forall xs. P xs$ :

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

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  $\forall xs. P xs$ :

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

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

(base)

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

# Proof by Structural Induction

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

To show  $\forall xs. P xs$ :

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

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

(base)

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

(step)

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

## Proof by Structural Induction

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

To show  $\forall xs. P xs$ :

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

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

(base)

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

(step)

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

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 (that we saw in the in the lecture on Isar).

```
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:

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

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:

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

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:

Not allowed:

$$\dots < \dots < x_3 < x_2 < x_1 < x$$

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

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

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

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:

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

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \text{ (disjI1)}$$

$$\frac{\Gamma \vdash P \quad \Gamma, P \vdash Q}{\Gamma \vdash Q} \text{ (cut)}$$

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:

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

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \text{ (disjI1)}$$

$$\frac{\Gamma \vdash P \quad \Gamma, P \vdash Q}{\Gamma \vdash Q} \text{ (cut)}$$

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:

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

Then Cut elimination fails!

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

## 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*.

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

## The Need for Intermediate Lemmas

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

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

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

## The Need for Intermediate Lemmas

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

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

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

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

Attempt:

$$\begin{aligned} & \text{reverse}(\text{reverse}(\text{Cons } x \text{ } xs)) \\ &= \text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \text{ Nil})) \\ &\quad \text{????} \\ &= \text{Cons } x \text{ } xs \end{aligned}$$

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

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

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

Attempt:

$$\begin{aligned} & \text{reverse}(\text{reverse}(\text{Cons } x \text{ xs})) \\ &= \text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \text{ Nil})) \\ &\quad \text{????} \\ &= \text{Cons } x \text{ xs} \end{aligned}$$

We need to *speculate* a new 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)

## 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:

$$\begin{aligned} & \text{reverse}(\text{reverse}(\text{Cons } x \ xs)) \\ = & \text{reverse}(\text{append}(\text{reverse } xs)(\text{Cons } x \ \text{Nil})) \\ = & \text{append}(\text{Cons } x \ \text{Nil})(\text{reverse}(\text{reverse } xs)) \quad \text{by lemma} \\ = & \text{Cons } x (\text{append } \text{Nil}(\text{reverse}(\text{reverse } xs))) \\ = & \text{Cons } x (\text{reverse}(\text{reverse } xs)) \\ = & \text{Cons } x \ xs \qquad \qquad \qquad \text{by IH} \end{aligned}$$

## Another approach

We got stuck trying to prove:

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

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

## Another approach

We got stuck trying to prove:

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

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:

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

Maybe this can be proved by induction?

## Another approach

We got stuck trying to prove:

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

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:

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

Maybe this can be proved by induction?

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

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

(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)

# 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