NICTA Advanced Course

Theorem Proving

Principles, Techniques, Applications

Recursion

CONTENT

- Intro & motivation, getting started with Isabelle
- Foundations & Principles
  - Lambda Calculus
  - Higher Order Logic, natural deduction
  - Term rewriting
- **Proof & Specification Techniques**
  - Inductively defined sets, rule induction
  - **Datatypes, recursion, induction**
  - Calculational reasoning, mathematics style proofs
  - Hoare logic, proofs about programs

Last Time

- ➜ Sets in Isabelle

LAST TIME

- ➜ Sets in Isabelle
- ➜ Inductive Definitions

Last Time

- ➜ Sets in Isabelle
- ➜ Inductive Definitions
- ➜ Rule induction

Last Time

- ➜ Sets in Isabelle
- ➜ Inductive Definitions
- ➜ Rule induction
- ➜ Fixpoints

LAST TIME

- ➜ Sets in Isabelle
- ➜ Inductive Definitions
- ➜ Rule induction
- ➜ Fixpoints
- ➜ Isar: induct and cases

DATATYPES

Example:

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

Properties:

DATATYPES

Example:

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

Properties:

➜ Constructors:

Nil	::	'a list
Cons	::	'a $\Rightarrow$ 'a list $\Rightarrow$ 'a list

Example:

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

Properties:

- ➜ Constructors:

Nil	::	'a list
Cons	::	'a ⇒ 'a list ⇒ 'a list

- ➜ Distinctness: Nil ≠ Cons x xs

DATATYPES

Example:

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

Properties:

➜ Constructors:

Nil	::	'a list
Cons	::	'a ⇒ 'a list ⇒ 'a list

➜ Distinctness: Nil $\neq$ Cons x xs

➜ Injectivity: (Cons x xs $=$ Cons y ys) $=$ (x $=$ y $\wedge$ xs $=$ ys)

The General Case

$$\begin{array}{lcl} \textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau & = & \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1} \\ & \mid & \ldots \\ & \mid & \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k} \end{array}$$

The General Case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$$
$$\quad | \quad \ldots$$
$$\quad | \quad \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$$

➜ Constructors: $\mathsf{C}_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$

THE GENERAL CASE

datatype $(\alpha_1, \ldots, \alpha_n)\,\tau$ = $\mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$
| $\ldots$
| $\mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$

- ➜ Constructors: $\mathsf{C}_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\,\tau$
- ➜ Distinctness: $\mathsf{C}_i\ \ldots \neq \mathsf{C}_j\ \ldots$ if $i \neq j$

The General Case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$$
$$\quad | \quad \ldots$$
$$\quad | \quad \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$$

- ➜ Constructors: $\mathsf{C}_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$
- ➜ Distinctness: $\mathsf{C}_i\ \ldots \neq \mathsf{C}_j\ \ldots$ if $i \neq j$
- ➜ Injectivity: $(\mathsf{C}_i\ x_1 \ldots x_{n_i} = \mathsf{C}_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

THE GENERAL CASE

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \tau \quad = \quad \mathsf{C}_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$$
$$\mid \quad \ldots$$
$$\mid \quad \mathsf{C}_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$$

- ➜ Constructors: $\mathsf{C}_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau$
- ➜ Distinctness: $\mathsf{C}_i\ \ldots \neq \mathsf{C}_j\ \ldots$ if $i \neq j$
- ➜ Injectivity: $(\mathsf{C}_i\ x_1 \ldots x_{n_i} = \mathsf{C}_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity applied automatically

HOW IS THIS TYPE DEFINED?

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

➜ internally defined using typedef

HOW IS THIS TYPE DEFINED?

datatype ’a list = Nil | Cons ’a ”’a list”

➜ internally defined using typedef

➜ hence: describes a set

How is this Type Defined?

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

- ➜ internally defined using typedef
- ➜ hence: describes a set
- ➜ set = trees with constructors as nodes

HOW IS THIS TYPE DEFINED?

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

- ➜ internally defined using typedef
- ➜ hence: describes a set
- ➜ set = trees with constructors as nodes
- ➜ inductive definition to characterize which trees belong to datatype

How is this Type Defined?

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

- ➜ internally defined using typedef
- ➜ hence: describes a set
- ➜ set = trees with constructors as nodes
- ➜ inductive definition to characterize which trees belong to datatype

More detail: Datatype_Universe.thy

Datatype Limitations

Must be definable as set.

Datatype Limitations

Must be definable as set.

➜ Infinitely branching ok.

DATATYPE LIMITATIONS

Must be definable as set.

➜ Infinitely branching ok.

➜ Mutually recursive ok.

DATATYPE LIMITATIONS

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.
- ➜ Stricly positive (left of function arrow) occurence ok.

Datatype Limitations

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.
- ➜ Stricly positive (left of function arrow) occurence ok.

Not ok:

$$\textbf{datatype}\ \mathsf{t} \quad = \quad \mathsf{C}\ (\mathsf{t} \Rightarrow \mathsf{bool})$$

DATATYPE LIMITATIONS

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.
- ➜ Stricly positive (left of function arrow) occurence ok.

Not ok:

$$\textbf{datatype}\ t \quad = \quad C\ (t \Rightarrow bool)$$
$$\qquad\qquad\qquad\quad | \quad D\ ((bool \Rightarrow t) \Rightarrow bool)$$

Datatype Limitations

Must be definable as set.

- ➜ Infinitely branching ok.
- ➜ Mutually recursive ok.
- ➜ Stricly positive (left of function arrow) occurence ok.

Not ok:

$$
\begin{array}{lcl}
\textbf{datatype}\ \mathsf{t} & = & \mathsf{C}\ (\mathsf{t} \Rightarrow \mathsf{bool}) \\
 & | & \mathsf{D}\ ((\mathsf{bool} \Rightarrow \mathsf{t}) \Rightarrow \mathsf{bool}) \\
 & | & \mathsf{E}\ ((\mathsf{t} \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool})
\end{array}
$$

Because: Cantor's theorem (α set is larger than α)

CASE

Every datatype introduces a **case** construct, e.g.

$$(\mathsf{case}\ xs\ \mathsf{of}\ [] \Rightarrow \ldots \mid y\ \#ys \Rightarrow \text{...}\ y\ \text{...}\ ys\ \text{...})$$

CASE

Every datatype introduces a **case** construct, e.g.

$$(\textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow \ldots \mid y\ \#ys \Rightarrow ...\ y\ ...\ ys\ ...)$$

In general: one case per constructor

CASE

Every datatype introduces a **case** construct, e.g.

$$(\textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow \ldots \mid y\ \#ys \Rightarrow \text{...}\ y\ \text{...}\ ys\ \text{...})$$

In general: one case per constructor

➜ Same order of cases as in datatype

CASE

Every datatype introduces a **case** construct, e.g.

$$(\textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow \ldots \mid y\ \#ys \Rightarrow \text{...}\ y\ \text{...}\ ys\ \text{...})$$

In general: one case per constructor

- ➜ Same order of cases as in datatype
- ➜ No nested patterns (e.g. $x\#y\#zs$)

Case

Every datatype introduces a **case** construct, e.g.

$$(\textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow \ldots \mid y\ \#ys \Rightarrow \text{...}\ y\ \text{...}\ ys\ \text{...})$$

In general: one case per constructor

- ➜ Same order of cases as in datatype
- ➜ No nested patterns (e.g. $x\#y\#zs$)
  (But nested cases)

CASE

Every datatype introduces a **case** construct, e.g.

$$(\textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow \ldots \mid y\ \#ys \Rightarrow \text{...}\ y\ \text{...}\ ys\ \text{...})$$

In general: one case per constructor

- ➜ Same order of cases as in datatype
- ➜ No nested patterns (e.g. $x\#y\#zs$)
  (But nested cases)
- ➜ Needs $()$ in context

CASES

apply (case_tac t)

CASES

apply (case_tac t)

creates k subgoals

$$\llbracket t = C_i \; x_1 \ldots x_p; \ldots \rrbracket \Longrightarrow \ldots$$

one for each constructor C_i

DEMO

RECURSION

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow$$

$$0 = 1$$

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow$$
$$0 = 1$$

! All functions in HOL must be total !

PRIMITIVE RECURSION

primrec guarantees termination structurally

Example primrec def:

primrec guarantees termination structurally

Example primrec def:

consts app :: "'a list ⇒ 'a list ⇒ 'a list"

primrec

"app Nil ys = ys"

"app (Cons x xs) ys = Cons x (app xs ys)"

THE GENERAL CASE

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then $f :: \tau \Rightarrow \tau'$ can be defined by **primitive recursion**:

$$
\begin{array}{lcl}
f\ (C_1\ y_{1,1}\ \ldots\ y_{1,n_1}) & = & r_1 \\
\vdots & & \\
f\ (C_k\ y_{k,1}\ \ldots\ y_{k,n_k}) & = & r_k
\end{array}
$$

The General Case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then $f :: \tau \Rightarrow \tau'$ can be defined by **primitive recursion**:

$$
\begin{array}{lcl}
f\ (C_1\ y_{1,1}\ \ldots\ y_{1,n_1}) & = & r_1 \\
\vdots & & \\
f\ (C_k\ y_{k,1}\ \ldots\ y_{k,n_k}) & = & r_k
\end{array}
$$

The recursive calls in r_i must be **structurally smaller**
(of the form $f\ a_1\ \ldots\ y_{i,j}\ \ldots\ a_p$)

HOW DOES THIS WORK?

primrec just fancy syntax for a **recursion operator**

Example:

HOW DOES THIS WORK?

primrec just fancy syntax for a **recursion operator**

Example:

list_rec :: "'b ⇒ ('a ⇒ 'a list ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b"

list_rec f_1 f_2 Nil $=$ f_1

list_rec f_1 f_2 (Cons x xs) $=$ $f_2\ x\ xs$ $($list_rec $f_1\ f_2\ xs)$

How does this Work?

primrec just fancy syntax for a **recursion operator**

Example:

list_rec :: "'b ⇒ ('a ⇒ 'a list ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b"

$$\begin{array}{lcl} \text{list\_rec}\ f_1\ f_2\ \text{Nil} & = & f_1 \\ \text{list\_rec}\ f_1\ f_2\ (\text{Cons}\ x\ xs) & = & f_2\ x\ xs\ (\text{list\_rec}\ f_1\ f_2\ xs) \end{array}$$

$$\text{app} \equiv \text{list\_rec}\ (\lambda ys.\ ys)\ (\lambda x\ xs\ xs'.\ \lambda ys.\ \text{Cons}\ x\ (xs'\ ys))$$

How does this Work?

primrec just fancy syntax for a **recursion operator**

Example:

list_rec :: "'b ⇒ ('a ⇒ 'a list ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b"

list_rec f_1 f_2 Nil $\quad = \quad f_1$

list_rec f_1 f_2 (Cons x xs) $\quad = \quad f_2\ x\ xs\ (\text{list\_rec}\ f_1\ f_2\ xs)$

app ≡ list_rec $(\lambda ys.\ ys)\ (\lambda x\ xs\ xs'.\ \lambda ys.\ \text{Cons}\ x\ (xs'\ ys))$

Defined: automatically, first inductively (set), then by epsilon

How does this Work?

primrec just fancy syntax for a **recursion operator**

Example:

list_rec :: "'b ⇒ ('a ⇒ 'a list ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b"

$$\begin{array}{lcl} \text{list\_rec}\ f_1\ f_2\ \text{Nil} & = & f_1 \\ \text{list\_rec}\ f_1\ f_2\ (\text{Cons}\ x\ xs) & = & f_2\ x\ xs\ (\text{list\_rec}\ f_1\ f_2\ xs) \end{array}$$

$$\text{app} \equiv \text{list\_rec}\ (\lambda ys.\ ys)\ (\lambda x\ xs\ xs'.\ \lambda ys.\ \text{Cons}\ x\ (xs'\ ys))$$

Defined: automatically, first inductively (set), then by epsilon

$$\frac{}{(\text{Nil}, f_1) \in \text{list\_rel}\ f_1\ f_2} \qquad \frac{(xs, xs') \in \text{list\_rel}\ f_1\ f_2}{(\text{Cons}\ x\ xs, f_2\ x\ xs\ xs') \in \text{list\_rel}\ f_1\ f_2}$$

$$\text{list\_rec}\ f_1\ f_2\ xs \equiv \text{SOME}\ y.\ (xs, y) \in \text{list\_rel}\ f_1\ f_2$$

Predefined Datatypes

NAT IS A DATATYPE

datatype nat $=$ 0 $|$ Suc nat

NAT IS A DATATYPE

datatype nat $=$ 0 $|$ Suc nat

Functions on nat definable by primrec!

primrec

$$
\begin{aligned}
f\ 0 &= \ldots \\
f\ (\mathsf{Suc}\ n) &= \ldots\ f\ n\ \ldots
\end{aligned}
$$

Option

datatype 'a option = None | Some 'a

Important application:

'b $\Rightarrow$ 'a option $\quad\sim\quad$ partial function:

Option

datatype 'a option = None | Some 'a

Important application:

'b $\Rightarrow$ 'a option	$\sim$	partial function:
None	$\sim$	no result
Some a	$\sim$	result a

Example:

consts lookup :: 'k $\Rightarrow$ ('k $\times$ 'v) list $\Rightarrow$ 'v option

primrec

Option

datatype 'a option = None | Some 'a

Important application:

'b $\Rightarrow$ 'a option	$\sim$	partial function:
None	$\sim$	no result
Some a	$\sim$	result a

Example:

consts lookup :: 'k $\Rightarrow$ ('k $\times$ 'v) list $\Rightarrow$ 'v option

primrec

lookup k [] = None

lookup k (x #xs) =

OPTION

datatype 'a option = None | Some 'a

Important application:

'b $\Rightarrow$ 'a option	$\sim$	partial function:
None	$\sim$	no result
Some a	$\sim$	result a

Example:

conssts lookup :: 'k $\Rightarrow$ ('k $\times$ 'v) list $\Rightarrow$ 'v option

primrec

lookup k [] = None

lookup k (x #xs) = (if fst x = k then Some (snd x) else lookup k xs)

Demo: primrec

INDUCTION

STRUCTURAL INDUCTION

$P\ xs$ holds for all lists xs if

- ➜ P Nil
- ➜ and for arbitrary x and xs, $P\ xs \Longrightarrow P\ (x \# xs)$

STRUCTURAL INDUCTION

$P\ xs$ holds for all lists xs if

- ➜ P Nil
- ➜ and for arbitrary x and xs, $P\ xs \Longrightarrow P\ (x\#xs)$

Induction theorem **list.induct:**
$[\![P\ [\,];\ \bigwedge a\ list.\ P\ list \Longrightarrow P\ (a\#list)]\!] \Longrightarrow P\ list$

STRUCTURAL INDUCTION

$P\ xs$ holds for all lists xs if

- ➜ P Nil
- ➜ and for arbitrary x and xs, $P\ xs \Longrightarrow P\ (x\#xs)$

  Induction theorem **list.induct:**
  $[\![P\ [];\ \bigwedge a\ list.\ P\ list \Longrightarrow P\ (a\#list)]\!] \Longrightarrow P\ list$

- ➜ General proof method for induction: **(induct x)**
  - x must be a free variable in the first subgoal.
  - type of x must be a datatype.

Theorems about recursive functions are proved by induction

Induction on argument number i of f
if f is defined by recursion on argument number i

EXAMPLE

A tail recursive list reverse:

consts itrev :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ 'a list

primrec

itrev $[]$ $\quad\quad ys =$

EXAMPLE

A tail recursive list reverse:

consts itrev :: 'a list ⇒ 'a list ⇒ 'a list

primrec

itrev [] $ys = ys$

itrev $(x\#xs)$ $ys =$

A tail recursive list reverse:

consts itrev :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ 'a list

primrec

itrev $[]$ $ys = ys$

itrev $(x\#xs)$ $ys =$ itrev xs $(x\#ys)$

EXAMPLE

A tail recursive list reverse:

consts itrev :: 'a list $\Rightarrow$ 'a list $\Rightarrow$ 'a list

primrec

itrev $[]$ $\quad ys = ys$

itrev $(x\#xs)$ $\quad ys =$ itrev $xs\ (x\#ys)$

lemma itrev $xs\ [] =$ rev xs

DEMO: PROOF ATTEMPT

Replace constants by variables

lemma itrev $xs\ ys =$ rev $xs @ ys$

Replace constants by variables

lemma itrev $xs\ ys =$ rev $xs@ys$

Quantify free variables by $\forall$

(except the induction variable)

Replace constants by variables

lemma itrev $xs\ ys =$ rev $xs@ys$

Quantify free variables by $\forall$

(except the induction variable)

lemma $\forall ys.$ itrev $xs\ ys =$ rev $xs@ys$

ISAR

DATATYPE CASE DISTINCTION

proof (cases $term$)

case Constructor$_1$

⋮

next

⋮

next

case (Constructor$_k$ $\vec{x}$)

$\ldots$ $\vec{x}$ $\ldots$

qed

DATATYPE CASE DISTINCTION

proof (cases $term$)

  case Constructor$_1$

  ⋮

next

⋮

next

  case (Constructor$_k$ $\vec{x}$)

  $\cdots$ $\vec{x}$ $\cdots$

qed

case (Constructor$_i$ $\vec{x}$) $\equiv$

fix $\vec{x}$ **assume** Constructor$_i$: "$term =$ Constructor$_i$ $\vec{x}$"

STRUCTURAL INDUCTION FOR TYPE NAT

show $P\ n$

proof (induct n)

  case 0 $\equiv$ **let** $?case$ = $P\ 0$

  $\ldots$

  show $?case$

next

  case (Suc n) $\equiv$ **fix** n **assume** Suc: $P\ n$

  $\ldots$ **let** $?case$ = P (Suc n)

  $\ldots\ n\ \ldots$

  show $?case$

qed

STRUCTURAL INDUCTION WITH $\Longrightarrow$ AND $\bigwedge$

show "$\bigwedge x.\ A\ n \Longrightarrow P\ n$"
proof (induct n)
  case 0 $\equiv$ **fix** x **assume** 0: "$A\ 0$"
  ... **let** $?case$ = "$P\ 0$"
  show $?case$
next
  case (Suc n) $\equiv$ **fix** n and x
  ... **assume** Suc: "$\bigwedge x.\ A\ n \Longrightarrow P\ n$"
  $\cdots\ n\ \cdots$ "A (Suc n)"
  ... **let** $?case$ = "P (Suc n)"
  show $?case$
qed

Demo

We have seen today ...

- ➜ Datatypes

WE HAVE SEEN TODAY ...

➜ Datatypes

➜ Primite Recursion

WE HAVE SEEN TODAY ...

- ➜ Datatypes
- ➜ Primite Recursion
- ➜ Case distinction

We have seen today ...

- ➜ Datatypes
- ➜ Primite Recursion
- ➜ Case distinction
- ➜ Induction

EXERCISES

- look at `http://isabelle.in.tum.de/library/HOL/Datatype_Universe.html`
- define a primitive recursive function **listsum** :: nat list $\Rightarrow$ nat that returns the sum of the elements in a list.
- show "$2 *$ listsum $[0..n] = n * (n + 1)$"
- show "listsum (replicate n a) $= n * a$"
- define a function **listsumT** using a tail recursive version of listsum.
- show that the two functions are equivalent: listsum xs = listsumT xs

Nicolas Magaud

on

The Coq System

Monday 15:00 – 16:30