Slide 1

NICTA Advanced Course

Theorem Proving
Principles, Techniques, Applications

Recursion

Slide 2

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

Slide 3

LAST TIME

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

Slide 4

DATATYPES

Example:

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

Properties:

- ➜ Constructors:

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

- ➜ Distinctness: Nil $\neq$ Cons x xs
- ➜ Injectivity: (Cons x xs $=$ Cons y ys) $=$ (x $=$ y $\wedge$ xs $=$ ys)

Slide 5

The General Case

datatype $(\alpha_1,\ldots,\alpha_n)\,\tau$ = $C_1\ \tau_{1,1}\ \ldots\ \tau_{1,n_1}$
| $\ldots$
| $C_k\ \tau_{k,1}\ \ldots\ \tau_{k,n_k}$

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

Distinctness and Injectivity applied automatically

Slide 6

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

Slide 7

Datatype Limitations

Must be definable as set.

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

Not ok:

datatype t = C (t ⇒ bool)
| D ((bool ⇒ t) ⇒ bool)
| E ((t ⇒ bool) ⇒ bool)

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

Slide 8

Case

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

(case xs of [] ⇒ $\ldots$ | $y\ \#ys$ ⇒ ... y ... ys ...)

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

Slide 9

Cases

apply (case_tac t)

creates k subgoals

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

one for each constructor C_i

Slide 10

Demo

Slide 11

Recursion

Slide 12

Why nontermination can be harmful

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:

Slide 13

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**:

Slide 14

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

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:

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

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

Slide 15

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

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

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

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

Slide 16

Predefined Datatypes

Slide 17

NAT IS A DATATYPE

datatype nat $=$ 0 $|$ Suc nat

Functions on nat definable by primrec!

primrec

$f\ 0 \quad = \quad ...$

$f\ (\text{Suc}\ n) \quad = \quad ...\ f\ n\ ...$

Slide 18

OPTION

datatype 'a option = None | Some 'a

Important application:

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

Example:

consts lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option

primrec

lookup k [] = None

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

Slide 19

DEMO: PRIMREC

Slide 20

INDUCTION

Structural induction

Slide 21

$P\ xs$ 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.

Basic heuristics

Slide 22

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

Slide 23

A tail recursive list reverse:

consts itrev :: 'a list ⇒ 'a list ⇒ 'a list
primrec
itrev [] $ys = ys$
itrev $(x\#xs)$ $ys =$ itrev xs $(x\#ys)$

lemma itrev xs $[\,] =$ rev xs

Slide 24

Demo: Proof Attempt

Slide 25

GENERALISATION

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 $xs@ys$

Slide 26

ISAR

Slide 27

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$_i$ $\vec{x}$"

Slide 28

STRUCTURAL INDUCTION FOR TYPE NAT

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

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

Slide 29

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

Slide 30

DEMO

WE HAVE SEEN TODAY ...

Slide 31

➜ Datatypes
➜ Primite Recursion
➜ Case distinction
➜ Induction

EXERCISES

Slide 32

➜ look at `http://isabelle.in.tum.de/library/HOL/Datatype_Universe.html`
➜ define a primitive recursive function **listsum** :: nat list ⇒ 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

NEXT LECTURE

Slide 33

Nicolas Magaud

on

The Coq System

Monday 15:00 – 16:30