



# Interactive Theorem Proving using Isabelle/HOL

## Session 7

René Thiemann

Department of Computer Science

- Inductive Definitions
- Rule Inversion and Rule Induction
- Sets in Isabelle

RT (DCS @ UIBK)

session 7

2/18

Inductive Definitions

## Inductive Definitions

### Definition Principles so Far

- **definition**
  - non-recursive definitions
  - no pattern matching on left-hand sides, form:
  - no simp-rules, but obtain defining equation:
- **fun** or **function**
  - recursive functions definitions including pattern matching on lhs
  - functions have to be terminating
  - obtain simp-rules and induction scheme

$$f\ x_1 \dots x_n = rhs$$
$$f\_def: f\ x_1 \dots x_n = rhs$$

RT (DCS @ UIBK)

session 7

4/18

## Purpose of Definition

- **definition** is the most primitive definition principle
- **definition** can be used formalize certain concepts
- after having derived interface-lemmas to concept, one might hide internal definition (in particular the defining equation is by default not added to simpset)
- many higher-level definition principles internally are based on **definition**
  - example: **function** uses some internal **definitions** which are hidden to user (demo)

## Example: Injectivity

```
definition injective :: "('a ⇒ 'b) ⇒ bool" where
  "injective f = (∀x y. f x = f y → x = y)"

lemma injectiveI: "(∧ x y. f x = f y ⇒ x = y) ⇒ injective f"
  unfolding injective_def by auto

lemma injectiveD: "injective f ⇒ f x = f y ⇒ x = y"
  unfolding injective_def by auto (* hide injective_def at this point *)
```

## Limits of definition and function

- restriction of **definition** and **function**: no capability to conveniently model potentially non-terminating processes
- consider datatype prog, modelling simple programming language with while-loops
- aim: define eval function, e.g., of type prog ⇒ state ⇒ state option, that returns state after complete evaluation of program or fails
- attempt 1: define eval via **function**
  - not possible, since termination is not provable (some programs are non-terminating)
- attempt 2: **fuel-based** approach (introduce some bounded resource to ensure termination)
  - first define eval\_b :: nat ⇒ prog ⇒ state ⇒ state option, a bounded version of eval that restricts the number of loop-iterations
  - eval\_b can be defined via **fun**
  - eval p s = (if ∃ n. eval\_b n p s ≠ None then eval\_b (SOME n. eval\_b n p s ≠ None) p s else None)
  - reasoning with this fuel-based-approach is at least tedious

## Solution: Inductive Predicates

model eval as **inductive predicate** of type prog ⇒ state ⇒ state ⇒ bool that correspond to standard inference rules of a big-step semantics

$$\frac{c \text{ is not satisfied in } s}{(while\ c\ P)\ s \xrightarrow{eval} s} \text{ (while-false)}$$

$$\frac{c \text{ is satisfied in } s \quad P\ s \xrightarrow{eval} t \quad (while\ c\ P)\ t \xrightarrow{eval} u}{(while\ c\ P)\ s \xrightarrow{eval} u} \text{ (while-true)}$$

⋮

(further rules for assignment, sequential composition, etc.)

## Demo

modeling programming language semantics

## Inductive Predicates in More Detail

- constant P :: 'a<sub>1</sub> ⇒ ... ⇒ 'a<sub>n</sub> ⇒ bool is **n-ary predicate**
- **inductive predicate** P is inductively defined, that is, by inference rules
- meaning: input satisfies P iff witnessed by arbitrary (finite) application of inference rules
- syntax
  - inductive** P :: "'a<sub>1</sub> ⇒ ... ⇒ 'a<sub>n</sub> ⇒ bool" where ...
  - followed by |-separated list of propositions (inference rules)
- generated facts
 

P.intros	inference rules
P.cases	case analysis ( <b>rule inversion</b> )
P.induct	induction ( <b>rule induction</b> )
P.simps	equational definition

## Odd Numbers, Inductively

- textual description
  - 1 is odd
  - if  $n$  is odd, then also  $n + 2$  is odd
- inference rules

$$\frac{}{1 \text{ odd}} \quad \frac{n \text{ odd}}{n + 2 \text{ odd}}$$

- `inductive is_odd :: "nat  $\Rightarrow$  bool"`
  - `where`
    - `"is_odd 1"`
    - `| "is_odd n  $\Rightarrow$  is_odd (n + 2)"`

## Special Case – Inductively Defined Sets

- given set  $S$ , let  $\chi_S$  be **characteristic function** such that  $\chi_S(x)$  is true iff  $x \in S$
- characteristic function is obviously predicate
- inductive sets are common special case and come with special syntax
 

```
inductive_set S :: "'a1  $\Rightarrow$  ... 'an  $\Rightarrow$  'a set" for c1 ... cn where
```

## Example – Reflexive Transitive Closure

- (binary) relations encoded by type `('a  $\times$  'b) set`
- given relation  $R$ , reflexive transitive closure, often written  $R^*$ , given by  $(x, y) \in R^*$  iff  $x R x_1 R x_2 R \dots R x_n R y$  for arbitrary  $x_1, x_2, \dots, x_n$  (think: path in graph)
- `inductive_set star :: "('a  $\times$  'a) set  $\Rightarrow$  ('a  $\times$  'a) set" for  $R$ 
  - where
    - refl [simp]: "(x, x)  $\in$  star R"
    - | step: "(x, y)  $\in$  R  $\Rightarrow$  (y, z)  $\in$  star R  $\Rightarrow$  (x, z)  $\in$  star R"`
- remark: one can label individual inference rules; these names will then be used for case-analyses, inductions, and as names of introduction rules (`star.step`)

## Rule Inversion and Rule Induction

### Rule Inversion

- reasoning backwards “which rule could have been used to derive some fact”
- case analysis according to inference rules
- if inductive predicate/set is first of current facts, cases applies **rule inversion** implicitly
- otherwise, use “cases rule: c.cases” for inductively defined constant  $c$

### Demo – Zero is Not Odd

```
lemma is_odd0: "is_odd 0 = False" sorry
```

## Rule Induction

- induction according to inference rules
- if inductive predicate/set is first of current facts, induction applies **rule induction** implicitly
- otherwise, use “`induction rule: c.induct`” for inductively defined constant `c`
- **case** names are taken from names of inference rules (if any, otherwise numbered)

## Demo – If Number is Odd it’s Odd

- `lemma is_odd_odd: assumes "is_odd x" shows "odd x" sorry`
- remarks
  - `odd x` is just an abbreviation of `x` not being divisible by 2
  - in lemma-command one can explicitly assume facts (`assumes`) which are accessible by implicit label `assms`, before the goal statement is written after `shows`
  - further examples on `assumes` and `shows` are provided in lemmas `is_odd_odd3` and `star_trans1` in the demo theory

## Demo – Reflexive Transitive Closure is Transitive

- `lemma star_trans:`  
`assumes "(x, y) ∈ star R" and "(y, z) ∈ star R"`  
`shows "(x, z) ∈ star R"`  
`sorry`

## More Information on Inductive Definitions

`isabelle doc isar-ref`

(chapter 11.1)

## Sets in Isabelle

### Sets in Isabelle

- type ‘`a set`’ for sets with elements of type ‘`a`’

### Set Basics

- $x \in A$  – membership
- $A \cap B$  – intersection
- $A \cup B$  – union
- $\neg A$  – complement
- $A - B$  – difference
- $A \subseteq B$  and  $A \subset B$  – subset
- $\{\}$  – empty set
- $\text{UNIV}$  – universal set (all elements of specific type)
- $\{x\}$  – singleton set
- `insert x A` – insertion of single elements (`insert x A = {x} ∪ A`)
- $f ` A$  – image of function with respect to set (“map `f` over elements of `A`”)

## Demo – Example Proof

```
lemma "A ∩ (B ∪ C) ⊆ (A ∩ B) ∪ (A ∩ C)"
```

### No New Primitives Required

- several of the basic set operations could be defined inductively
- examples

```
inductive_set intersection :: "'a set ⇒ 'a set ⇒ 'a set" for A B where  
  "x ∈ A ⇒ x ∈ B ⇒ x ∈ intersection A B"
```

```
inductive_set union :: "'a set ⇒ 'a set ⇒ 'a set" for A B where  
  "x ∈ A ⇒ x ∈ union A B"  
| "x ∈ B ⇒ x ∈ union A B"
```

```
inductive_set empty :: "'a set"
```

```
inductive_set Univ :: "'a set" where  
  "x ∈ Univ"
```

## Further Operations on Sets

- `set` – convert list to set
- `Collect p` – convert predicate  $p :: 'a \Rightarrow \text{bool}$  to set of type  $'a \text{ set}$
- `finite A` – is set finite?
- `card A :: nat` – cardinality of set (note:  $\text{card } A = 0$  whenever  $A$  is infinite)
- `sum f A` –  $\sum_{x \in A} f(x)$  (note:  $\text{sum } f A = 0$  whenever  $A$  is infinite)
- `prod f A` – similar to `sum`, just product
- `Ball A p` – do all elements of  $A$  satisfy predicate  $p$ ?
- `Bex A p` – does some element of  $A$  satisfy predicate  $p$ ?
- `{x .. y}` – all elements between  $x$  and  $y$

### Syntax for Set Comprehension

- `{x . p x}` – same as `Collect p`
- `{t | x y. p x y}` – same as  $\{z. \exists x y. t = z \wedge p x y\}$
- example: `{(x + 5, y) | x y. x < 7 ∧ odd y}`