



Interactive Theorem Proving using Isabelle/HOL

Session 3

René Thiemann

Department of Computer Science

Outline

- Natural Deduction Revisited
- Case Analysis and Structural Induction for Data Types

Natural Deduction Revisited

Last Lecture: Natural Deduction in Isabelle

- typical proof step: `from this more_facts have label: term by (rule thm)`
- three problems
 - finding names of theorems such as *thm*
 - repetitive long commands, e.g., `from this have`
 - management of labels (tedious, not informative, ...)

Use the Isabelle Library

- Isabelle already provides several theorems, e.g., inference rules of natural deduction, properties of numbers, properties of lists, ...
- to increase efficiency, these theorems should be re-used, not re-proved
- problem: how to know the name of all these theorems, e.g.,
`thm excluded_middle disjI1 exE ccontr`
`thm add.commute add_le_cancel_right`
- solution: use **search engine** to quickly find
 - already proven **theorems**
 - already defined **constants**, e.g., algorithms on lists, numbers, sets, ...

Finding Existing Theorems

- enter query in “Query/Find Theorems” panel or after `find_theorems` command
- scope: search is restricted to accessible content in current theory, including imports

Search Criteria

- name: `foo` – search for facts whose name contains substring “foo”
- `"pattern"` – search for facts that match *pattern*
- prefix criterion by “-” to exclude facts that match
- combine several criteria by juxtaposition

Search Patterns

HOL terms with schematic variables `?x`, `?y`, ... or `_` instead of free variables

Examples

query	finds facts mentioning	query	finds facts mentioning
<code>"_ + _"</code>	addition	<code>2 "(+)"</code>	2 and addition function
<code>"?x + ?x"</code>	addition of same value	<code>"_ * (_ + _) = _"</code>	distributive law

Finding Existing Constants

- enter query in “Query/Find Constants” panel or after `find_consts` command
- scope: search is restricted to accessible content in current theory, including imports

Search Criteria

- `name: foo` – search for constants whose name contains substring “foo”
- `"type"` – search for constants that match a specific *type*
- combine several criteria by juxtaposition

Search Types

HOL types with schematic type variables `?'a, ?'b, ...` or `_` instead of free type variables

Example

```
find_consts "?'a => ?'a => _ list" name: "List"
```

searches for all binary functions where first and second argument have the same type, that return a list, and whose names includes "List" (e.g., as theory-prefix of a long name)

Abbreviations of Statements

- **then** = **from this**
(unlike to **from**, after **then** no further facts may be stated)
- **hence** = **then have**
- **thus** = **then show**
- **with facts** = **from facts this**

Passing Auxiliary Facts

- instead of passing facts before the property to be proven, one can also state facts after the property via **using**:

from facts have *proposition* $\langle proof \rangle$

is equivalent to

have *proposition using facts* $\langle proof \rangle$

- style: state important facts before, and auxiliary facts after *proposition*
- caution: label this **is not available after using**

Avoiding Labels: `moreover` and `ultimately`

- often proofs are of the form that auxiliary properties 1, ..., n are proven and then one can conclude
- manually labeling all these properties is tedious, in particular if labels are somehow sorted and one needs to insert something in the middle
- use `moreover` and `ultimately` to write these proofs without explicit labels
- example

with labels

```

have 1: A <proof>
have B <proof>
hence 2: C <proof>
have D <proof>
hence 3: E <proof>
from 1 2 3 show ?thesis

```

without labels

```

have A <proof>
moreover (* store A *)
have B <proof>
hence C <proof>
moreover (* store C *)
have D <proof>
hence E <proof>
ultimately show ?thesis (* A C E are avail. *)

```

Case Analysis on Booleans

- Isabelle provides special syntax to perform proofs by case analysis
- this slide: case analysis on Booleans (general case: later)
- structure is as follows, where *term* is of type `bool` (copy outline from output panel)

```
proof (cases term) (* here outline is displayed in output panel *)
  case True
  ... (* label True refers to fact "term" *)
  show ?thesis <proof>
next
  case ownLabel: False
  ... (* label ownLabel refers to fact "~ term" *)
  show ?thesis <proof>
qed
```

- order of cases is irrelevant, separation of cases via `next`
- user-defined labels become important in nested case analyses
- omitted case(s) can be solved via final method, e.g., `qed auto`

The rule Method – Revisited

- rule *fact* – if provided facts are empty, apply *fact* as **introduction rule** (last week)
- otherwise, apply *fact* as elimination rule
- introduction rule: conclusion introduces connective $(\dots \implies A \wedge B)$
- elimination rule: premise contains connective that is eliminated $(A \wedge B \implies \dots)$

Rule Application

- given rule $P_1 \implies \dots \implies P_n \implies C$
- intro – **unify C with conclusion of current subgoal** and add correspondingly instantiated premises $P_1\sigma, \dots, P_n\sigma$ as new subgoals
- elim – **unify major premise P_1 of rule with first of current facts**; unify remaining current facts with remaining premises; add rest of premises correspondingly instantiated as new subgoals

Beyond rule – intro and elim

- the rule method applies **exactly one** rule (intro or elim)
- the intro method applies **several** introduction rules exhaustively
- the elim method applies **several** elimination rules exhaustively

Example

```
lemma "A ∧ (∃ x :: nat. B x ∧ (C ∨ D x))"
proof (intro conjI exI)
  - ⟨three subgoals: A, B ?x, C ∨ D ?x⟩
    show A ⟨proof⟩
    show "B 5" ⟨proof⟩      (* here we choose witness 5 *)
    show "C ∨ D 5" ⟨proof⟩ (* no choice of witness anymore *)
qed
```

Case Analysis and Structural Induction for Data Types

Data Type Definitions

- whenever a data type `ty` is defined, in the background several theorems are proven
 - they can be inspected via `print_theorems` directly after the definition
 - simplification rules: `ty.simps` (automatically used by `auto`)
 - case analysis rule: `ty.exhaust` (used by `cases "term :: ty"`)
 - induction rule: `ty.induct` (used by induction `"variable :: ty"`)

Example

- consider Isabelle's lists: `datatype 'a list = Nil | Cons 'a "'a list"`
- special syntax: `[]` is the same as `Nil`, `#` is an infix operator for `Cons`, and there is syntax such as `[x, y, z]`
- `list.simps` contains among others $(x \# xs = y \# ys) = (x = y \wedge xs = ys)$
 $(\text{case } x \# xs \text{ of } [] \Rightarrow e \mid y \# ys \Rightarrow f \ y \ ys) = f \ x \ xs$
- `list.exhaust`: $(ys = [] \implies P) \implies (\bigwedge x \ xs. \ ys = x \# \ xs \implies P) \implies P$
- `list.induct`: $P \ [] \implies (\bigwedge x \ xs. \ P \ xs \implies P \ (x \# \ xs)) \implies P \ ys$

Function Definitions

- whenever a function `f` is defined, in the background several theorems are proven
 - they can be inspected via `print_theorems` directly after the definition
 - simplification rules: `f.simps` (automatically used by `auto`)
 - induction rule: `f.induct` (details in upcoming lecture)

Example

- consider append function:


```
fun app :: "'a list ⇒ 'a list ⇒ 'a list" where
  "app [] ys = ys"
| "app (x # xs) ys = x # (app xs ys)"
```
- `app.simps` are the two defining equations as theorems

The induction Method

- induction `x` – induction on parameter `x` (rule chosen according to type of `x`)
- use `case` to start case
 - syntax: `case (CName x1 ... xn)` where
 - `CName` is name of constructor
 - `x1, ..., xn` are freely chosen variable names that represent the arguments of `CName`
 - `CName` is also label that contains the IHs;
 - e.g., for binary tree with constructor `Node`, the fact `Node(1)` would be the first IH (left subtree) and `Node(2)` would be the second IH (right subtree)
- `?case` abbreviates goal of current case, separate cases by `next`
- outline of induction proof is available in output panel for `induction x` method

The cases Method

- `cases term` – case analysis on parameter `term` (rule chosen according to type of `term`)
- same structure as induction method, with two differences
 - goals of current case are still `?thesis`, not `?case`
 - no IHs are available as facts, but equalities `term = CName x1 ... xn`

Demo – List Reversal

```
fun app :: "'a list ⇒ 'a list ⇒ 'a list" where
  "app [] ys = ys"
| "app (x # xs) ys = x # (app xs ys)"
```

```
fun reverse :: "'a list ⇒ 'a list" where
  "reverse [] = []"
| "reverse (x # xs) = app (reverse xs) ([x])"
```

```
lemma rev_rev: "reverse (reverse xs) = xs"
```

Proof Strategies

1. perform induction on suitable variable (more on that next week)
2. copy proof outline by click in blue part of output panel; adjust variable names on demand
3. handle each case, replace `sorry` by `proof auto`
 - if successful, replace `proof auto` by `by auto`
 - if not, either
 - perform proof manually (natural deduction, add intermediate statements, ...)
 - or `identify required lemma` to make progress and first prove that lemma
4. cleanup proof, e.g., drop trivial cases and replace final `qed` by `qed auto`

Auxiliary Lemmas

- currently: assume auxiliary lemmas are just equations $lhs = rhs$
- formulate lemmas such that lhs is larger than rhs , so that terms get smaller
- activate lemma globally via `[simp]-attribute`: `lemma useful [simp]: "lhs = rhs"`
- activate lemmas locally: `proof (auto simp: useful ...)`
- warning: if the activated equations do not terminate, then `auto` might not terminate