Chapter 4

A Proof System for Higher-Order Logic

Overview of Chapter

Overview

1. Formulas, sequents, and rules revisited
2. Application of rules
3. Fundamental methods of Isabelle/HOL
4. Logical rules and theory Main
5. Rewriting and simplification
6. Case analysis and structural induction
7. Proof automation
8. More proof methods

» Chapter 5 of Isabelle/HOL Tutorial til page 100

Section 4.1

Methods and Rules

Formulas, sequents, and rules revisited

We need to represent:

- formulas, generalized sequents: lemmas/theorems to be proven
- rules: to be applied in a proof step
- proof (sub-)goals, i.e., open leaves in a proof tree

Examples: from Lecture.thy

- SPEC, SCHEMATIC (not allowed)
- ARULE
- GOAL

A proven lemma/theorem is automatically transformed into a rule. That is, the set of rules is not fixed in Isabelle/HOL (e.g. ARULE).

Variables

Six kinds of variables:

- (logical) variables bound by the logic-quantifiers
- (logical) variables bound by the meta-quantifier
- free (logical) variables
- schematic variables (in rules and proofs)
- type variables
- schematic type variables

Format of **goals**

- $\bigwedge x_1 \ldots x_k.\ [\![A_1; \ldots; A_m]\!] \Longrightarrow C$
- x_i are variables local to the subgoal (possibly none)
- A_i are called the assumptions (possibly none)
- C is called the conclusion
- usually no schematic variables

Format of **rules**

- $[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$
- P_i are called the premises (possibly none)
- P_1 is called the major premise
- Q is called the consequent (not standard)
- Schematic variables in P_i, Q.

Proofs and methods

Proof state

A *proof state* is characterized by the list of *open* subgoals:

- at the beginning: proof goal
- during the proof: not yet proven subgoals
- at the end: empty

Methods

Methods are commands working on the proof state.
In particular, they allow to apply rules and to do simplification.

- Isabelle/HOL provides a fixed set of basic methods.
- New methods can only be defined based on the basic methods.
- Set of rules is not fixed, i.e., new rules can be derived.

Methods overview

Format of method application

`apply` (<*method_name*> <*arguments*>)
where the number and type of arguments depends on the method
(proof state is implicit).

Kinds of methods:

- [edf]rule: different methods for rule application
- assumption: proving the subgoal from the assumptions
- induct/cases: do a proof by induction/case analysis
- unfold/simp: unfolding definitions/simplification

Depending on method, arguments, and proof state the application can fail.

Method “rule”: Basic idea

Rule application

The application of rules is based on unification:

- Unification is done w.r.t. the schematic variables.
- The unifier is applied to the complete proof state!
- Unification may involve renaming of bound variables.

Example

Applying rule $[\![P_1; P_2]\!] \Longrightarrow Q$ with method rule to subgoal $A \Longrightarrow C$:

- If σ unifies C and Q, then replace subgoal by two new subgoals:
  - $\sigma(A) \Longrightarrow \sigma(P1)$
  - $\sigma(A) \Longrightarrow \sigma(P2)$

Method “rule”

apply (rule <rule_name>)

- let R be the name of rule $[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$
- let $\bigwedge x_1 \ldots x_k.\ [\![A_1; \ldots; A_m]\!] \Longrightarrow C$ be the current subgoal
- `apply` (rule R) unifies Q with C;
  fails if no unifier exists; otherwise unifier σ
- new subgoals: For $i = 1, ..., n$:

$$\bigwedge x_1 \ldots x_k.\ \sigma([\![A_1; \ldots; A_m]\!] \Longrightarrow P_i)$$

- Example: SPEC

Method “erule”

apply (erule <rule_name>)

- let R be the name of rule $[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$
- let $\bigwedge x_1 \ldots x_k.\ [\![A_1; \ldots; A_m]\!] \Longrightarrow C$ be the current subgoal
- `apply` (erule R) unifies Q with C and simultaneously P_1 with some A_j;
  fails if no A_j and unifier can be found; otherwise unifier σ
- new subgoals: For $i = 2, ..., n$:

$$\bigwedge x_1 \ldots x_k.\ \sigma([\![A_1; \ldots; A_m \setminus \{A_j\}]\!] \Longrightarrow P_i)$$

- helpful for applying elimination rules
- Example: ARULE

Method “drule”

apply (drule <rule_name>)

- let R be the name of rule $[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$
- let $\bigwedge x_1 \ldots x_k.\ [\![A_1; \ldots; A_m]\!] \Longrightarrow C$ be the current subgoal
- `apply` (drule R) unifies P_1 with some A_j;
  fails if no A_j and unifier can be found; otherwise unifier σ
- new subgoals: For $i = 2, ..., n$:

$$\bigwedge x_1 \ldots x_k.\ \sigma([\![A_1; \ldots; A_m \setminus \{A_j\}]\!] \Longrightarrow P_i)$$

$$\bigwedge x_1 \ldots x_k.\ \sigma([\![A_1; \ldots; A_m \setminus \{A_j\};\ Q]\!] \Longrightarrow C)$$

- helpful for applying destruction rules
- Example: C1

Method “frule”

apply (frule <rule_name>)

Like drule, but assumption is not eliminated

- let R be the name of rule $[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$
- let $\bigwedge x_1 \ldots x_k.\ [\![A_1; \ldots; A_m]\!] \Longrightarrow C$ be the current subgoal
- `apply` (frule R) unifies P_1 with some A_j;
  fails if no A_j and unifier can be found; otherwise unifier σ
- new subgoals: For $i = 2, ..., n$:

$$\bigwedge x_1 \ldots x_k.\ \sigma([\![A_1; \ldots; A_m]\!] \Longrightarrow P_i)$$

$$\bigwedge x_1 \ldots x_k.\ \sigma([\![A_1; \ldots; A_m;\ Q]\!] \Longrightarrow C)$$

- Example: C1

Method “rule_tac”-versions

apply ([edf]rule_tac x =<term> in <rule_name>)

similar to [edf]rule, but allow to refine unification

- Example: Isabelle/HOL Tutorial, 5.8.2, p. 79/80 (*erule_tac*)
- NOFIX

Method “assumption”

apply (assumption)

- let $\bigwedge x_1 \ldots x_k.\ [\![A_1; \ldots; A_m]\!] \Longrightarrow C$ be the current subgoal
- `apply` (assumption) unifies C with some A_j;
  fails if no A_j and unifier can be found; otherwise:
- subgoal is closed, i.e., eliminated from proof state.
- Example: GOAL

Methods “induct”, “unfold”

apply (induct[_tac] <variable_name>)

- uses the inductive definition of a function
- generates the corresponding subgoals

apply (unfold <name_def>)

- unfolds the definition of a constant in all subgoals
- Example: SPEC

Logical rules of Isabelle/HOL

The logical rules are defined in theory `Main`
(see IsabelleHOLMain, Sect. 2.2)

Remark

Distinguish between *safe* and *unsafe* rules:

- Safe rules preserve provability:
  e.g. conjI, impI, notI, iffI, refl, ccontr, classical, conjE, disjE
- Unsafe rules can turn a provable goal into an unprovable one:
  e.g. disjI1, disjI2, impE, iffD1, iffD2, notE
- $\rightsquigarrow$ Apply safe rules before unsafe ones

Applying logical rules

Example

- lemma UNSAFE: “$A \vee \neg A$”
- apply (rule disjI1)
- sorry

Remark

Working with theory Main is similar to programming with large libraries:

- The programmer cannot know the complete library
- The “verificator” cannot know all rules.

Support for finding rules is important in practice.

Section 4.2

Rewriting and simplification

Rewriting and simplification

Content:
» `Isabelle/HOL Tutorial, Section 3.1`
Slides:
» `slides from Prof. Nipkow`
Examples:
» `ExSimp.thy`

Usage:

apply (simp add: <eq_1> ... <eq_n>)
Powerful method for rewriting and simplification

Overview

- Term rewriting foundations
- Term rewriting in Isabelle/HOL
  - Basic simplification
  - Extensions

Term rewriting foundations

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\leadsto$ *rewrite rule*

An example

Equations:

$$\begin{array}{rcll} 0 + n & = & n & (1)\\ (Suc\ m) + n & = & Suc\ (m + n) & (2)\\ (Suc\ m \leq Suc\ n) & = & (m \leq n) & (3)\\ (0 \leq m) & = & True & (4) \end{array}$$

Rewriting:

$$\begin{array}{rcll} 0 + Suc\ 0 & \leq & Suc\ 0 + x & \overset{(1)}{=}\\ Suc\ 0 & \leq & Suc\ 0 + x & \overset{(2)}{=}\\ Suc\ 0 & \leq & Suc\ (0 + x) & \overset{(3)}{=}\\ 0 & \leq & 0 + x & \overset{(4)}{=}\\ True & & & \end{array}$$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

$\sigma = \{n \mapsto b + c\}$

Result: $a + (b + c)$

Extension: conditional rewriting

Rewrite rules can be conditional:

$$[\![P_1 \ldots P_n]\!] \Longrightarrow l = r$$

is *applicable* to term $t[s]$ with σ if

- $\sigma(l) = s$ and
- $\sigma(P_1), \ldots, \sigma(P_n)$ are provable (again by rewriting).

Interlude: Variables in Isabelle

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Schematic variables:

- Logically: free = schematic
- Operationally:
  - free variables are fixed
  - schematic variables are instantiated by substitutions

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: xs @ [] = xs*

⋮

done

After the proof: Isabelle changes *xs* to *?xs* (internally):

?xs @ [] = ?xs

Now usable with arbitrary values for *?xs*

Example: rewriting

rev(a @ []) = rev a

using *app_Nil2* with $\sigma = \{?xs \mapsto a\}$

Term rewriting in Isabelle

Basic simplification

Goal: *1.* $[\![P_1; \ldots ; P_m]\!] \Longrightarrow C$

apply*(simp add:* $eq_1 \ldots eq_n$*)*

Simplify $P_1 \ldots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec**, **fun** and **datatype**
- additional lemmas $eq_1 \ldots eq_n$
- assumptions $P_1 \ldots P_m$

Variations:

- *(simp ... del: ...)* removes *simp*-lemmas
- *add* and *del* are optional

auto versus simp

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Example: $f(x) = g(x),\ g(x) = f(x)$

$$[\![P_1 \ldots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is "bigger" than r and each P_i

n < m $\Longrightarrow$ *(n < Suc m) = True* YES
Suc n < m $\Longrightarrow$ *(n < m) = True* NO

Rewriting with definitions

Definitions do not have the *simp* attribute.

They must be used explicitly: *(simp add: f_def ...)*

Extensions of rewriting

Local assumptions

Simplification of $A \longrightarrow B$:

1. Simplify A to A'
2. Simplify B using A'

Case splitting with simp

P(if A then s else t)
=
(A $\longrightarrow$ *P(s))* $\wedge$ *(*$\neg$*A* $\longrightarrow$ *P(t))*

Automatic

P(case e of 0 $\Rightarrow$ *a | Suc n* $\Rightarrow$ *b)*
=
(e = 0 $\longrightarrow$ *P(a))* $\wedge$ *(*$\forall$ *n. e = Suc n* $\longrightarrow$ *P(b))*

By hand: *(simp split: nat.split)*

Similar for any datatype *t*: *t.split*

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \rightsquigarrow a + b$ but not $a + b \rightsquigarrow b + a$.

For types *nat*, *int* etc:

- lemmas *add_ac* sort any sum ($+$)
- lemmas *times_ac* sort any product ($*$)

Example: *(simp add: add_ac)* yields

$$(b + c) + a \rightsquigarrow \cdots \rightsquigarrow a + (b + c)$$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$\begin{array}{rcl} \neg A & \mapsto & A = False \\ A \longrightarrow B & \mapsto & A \Longrightarrow B \\ A \wedge B & \mapsto & A,\ B \\ \forall x.A(x) & \mapsto & A(?x) \\ A & \mapsto & A = True \end{array}$$

Example:

$$(p \longrightarrow q \wedge \neg\, r) \wedge s \quad \mapsto \quad \left\{ \begin{array}{r} p \Longrightarrow q = True \\ p \Longrightarrow r = False \\ s = True \end{array} \right\}$$

98

When everything else fails: Tracing

Set trace mode on/off in Proof General:

Isabelle → Settings → Trace simplifier

Output in separate `trace` buffer

99

Section 4.3

Case analysis and structural induction

Case analysis and structural induction

Content:
» `Isabelle/HOL Tutorial, Sections 3.2 and 3.5`

Examples:
» `ExCaseInduct.thy`

Case analysis

Properties of datatype values are often proved using case analysis:

```
datatype 'a tree = Tip | Node "'a tree" 'a "'a tree"


lemma
"(1::nat) ≤ (case t of Tip ⇒ 1 | Node l x r ⇒ x+1)"

apply (case_tac t)
apply simp
apply simp
done
```


Induction heuristics

Theorems about recursive functions are proved by induction.

Remark

- In general, induction proofs can go wrong.
- In the following, we consider some heuristics that might help.

Heuristic 1

If argument number k is the argument in which the function is recursive, do the induction on argument number k.

Example for Heuristic 1

```
primrec app::"'a list=>'a list=>'a list"  (infixr "@" 65)
where
   "[]       @ bs =   bs" |
   "(a # as) @ bs =   a # (as @ bs)"


lemma "(xs @ ys) @ zs = xs @ (ys @ zs)"
```

- `@` is recursive in first argument
- `xs` only occurs in first argument of `@`
- both `ys` and `zs` occur as second argument

Hence, do induction on `xs`

Goal generalization

Heuristic 2

Generalize the goal before induction:

- Replace constants by variables
- Universally quantify all free variables except the induction variable.

Remark

Heuristics should not be applied blindly.

Examples:
» `ExCaseInduct.thy`

Recursion induction

Function definitions do not always enable to prove properties about the structure of one argument.
In such cases, a more general inductive scheme has to be used.
Isabelle/HOL calls this *recursion induction*.

Isabelle/HOL supports recursion induction by generating an inductive rule:

- The rule exploits the structure on the argument domain induced by the function definition.
- It allows to do induction over several arguments simultaneously.

Example for recursion induction

Example

```
fun sep :: "'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

Generated induction rule `sep.induct`:

```
⟦ ⋀ a. ?P a [];
  ⋀ a x. ?P a [x];
  ⋀ a x y zs. ?P a (y # zs)   ⟹   ?P a (x # y # zs)  ⟧
⟹  ?P ?a0.0 ?a1.0
```

```
lemma "map f (sep a xs) = sep (f a) (map f xs)"
```


More about proof development

Forward proof step in backward proof:

- apply rules to assumptions

Forward proofs (Hilbert style proofs):

- directly prove a theorem from proven theorems

Directives/attributes:

- of: instantiates the variables of a rule to a list of terms
- OF: applies a rule to a list of theorems
- THEN: gives a theorem to named rule and returns the conclusion
- simplified: applies the simplifier to a theorem

Example for forward proof steps

Forward proof steps by “frule” and “drule”

- Consider subgoal: $[\![B_1; \ldots; B_n]\!] \Longrightarrow C$
- Work on assumption B_1 towards C using rule R: $A_1 \Longrightarrow A$
- Unifier σ with: $\sigma(B_1) = \sigma(A_1)$
- Command `apply (frule R)` yields new subgoal:

$$\sigma([\![B_1; \ldots; B_n; A]\!] \Longrightarrow C)$$

- Command `apply (drule R)` would also delete B_1

More proof methods

Method insert:

- inserts a theorem as a new assumption into current subgoal

Method subgoal_tac:

- inserts an arbitrary formula F as assumption
- F becomes additional subgoal

`print_methods` lists all supported methods

Section 4.4

Proof automation

Proof search automation

Content: » `Isabelle/HOL Tutorial, Sections 5.12 and 5.13`

Examples: » `ExProofAutomation.thy`

Proof automation tries to apply rules either

- to finish the proof of the (sub-)goal(s)
- to simplify the subgoals

We call this the success criterion.

Proof search automation (2)

Methods for proof automation are different in

- the success criterion
- the rules they use
- the way in which these rule are applied

Terminology

- *Simplification* applies rewrite rules repeatedly as long as possible.
- *Classical reasoning* uses search and backtracking with rules from predicate logic.

General methods (Tactics)

blast:

- tries to finish proof of (sub-)goal
- classical reasoner

clarify:

- tries to perform obvious proof steps on current subgoal
- classical reasoner (only safe rules, no splitting of subgoal)

safe:

- tries to perform obvious proof steps on all subgoals
- classical reasoner (only safe rules, splitting of subgoal)

General methods (continued)

clarsimp:

- tries to finish proof of subgoal
- classical reasoner interleaved with simplification (only safe rules, no splitting)

force:

- tries to finish proof of subgoal
- classical reasoner and simplification

auto:

- tries to perform proof steps on all subgoals
- classical reasoner and simplification (splitting)

Questions

1. A natural deduction proof system distinguishes between formulas, sequents, and rules. What are the differences?
2. Isabelle/HOL has nor clear distinction between sequents and rules. Why?
3. Explain the different kinds of variables.
4. What is a proof state?
5. What is the distinction between a rule and a method?
6. Explain the method “rule” and show in detail how it can be applied in a proof state?
7. What is an elimination rule?
8. Here is a proof state (shown on the screen). What is a rule that can be applied?

Questions (2)

9. Name some rule and their uses.
10. What does it mean that a rule is safe?
11. Why is verification in Isabelle/HOL usually based on theory Main and not directly on the HOL axioms?
12. What is rewriting and simplification?
13. How can an Isabelle/HOL user influence the simplification process?
14. What is case analysis?
15. How differ methods for proof automation?
16. Explain a method for proof automation.
17. What is a forward proof step?