universität freiburg

Isabelle and Program Verification

Mathias Fleury

Tutorial Vienna, 2024-1-12

Chapter 1

Introduction

A theorem looks like:

```
lemma
  assumes ‹P› and ‹Q›
  shows ‹conclusion›
proof – — or tactic
  show ?thesis
oops
```

People call everything a lemma, but you can also use theorem, corollary, or proposition.

A theorem looks like:

lemma
  shows ‹*add m 0 = m*›

If there are assumptions: using assms

proof (*induction m*)
  case *0*
  then show *?case* by *simp*

$add\ 0\ 0 = 0$ by definition

next
  case (*Suc m*)
  then show *?case* by *simp*

$add\ (Suc\ m)\ 0 = Suc\ (add\ m\ 0)$ by definition.
$add\ (Suc\ m)\ 0 = Suc\ m$ by $add\ m\ 0 = m$

qed

Let's have a look at List_Demo.thy.

Proofs

State

Induction Principles

The Proof State

lemma
  shows ‹$rev\ (xs\ @\ ys) = rev\ ys\ @\ rev\ xs$›
proof ($induction\ ys$) — Look at the output panel!
oops

  term ‹$\bigwedge x_1\ x_2\ x_n.\ A \Longrightarrow B \Longrightarrow B$›

where

- $x_1\ x_2\ x_n$ are the fixed local variables
- A and B are local assumptions
- C is the (actual) subgoal

Proof Methods

- *induct* performs structural induction on some variables
- *auto* solves as many goals as possible, mainly by simplification

Proofs

State

Induction Principles

Generalization

By default, for better automation, induct keeps constant unchanged.

But sometimes you need to generalize over it.

```
lemma ‹rev (xs @ ys) = rev ys @ rev xs›
  by (induction xs arbitrary: ys) (auto simp del: rev_append)
```

Adapted Induction Principles

fun generates an adapted induction principle by default:

fun *div2* :: ‹*nat* ⇒ *nat*› where
‹*div2 0* = *0*› |
‹*div2* (*Suc 0*) = *0*› |
‹*div2* (*Suc* (*Suc n*)) = *Suc* (*div2 n*)›

thm *div2.induct*

Theorems

Proofs

Simplifying Rules

More Term Structure: Sets

More Tactics

Conclusion

Simplifying Rules

Simplification

Splitting

Datatypes

Inductive Definitions

Simplifier

lemma ‹$(Suc\ n \le Suc\ m) = (n + 2 \le m + 2)$›
  supply [[*simp_trace_new*]]
  apply (*simp add: diff_right_mono*)
  oops

Simplification (II)

$0 < ?n \Longrightarrow Suc\ (?n - Suc\ 0) = ?n$ is conditional rewriting:

```
lemma
  fixes n m :: nat
  assumes ‹n > 0›
  shows ‹(n - 1 < m) = (n ≤ m)›
proof -
  show ?thesis
    supply [[simp_trace]]
    using assms
    apply (simp add: less_eq_Suc_le)
    done
qed
```

You can also delete rules with del

Unfolding Definitions

```
definition square :: ‹nat ⇒nat› where
  ‹square n = n * n›

lemma shows ‹square 3 = 9›
proof –
  show ?thesis
    — simp: does nothing here
    apply (simp add: square_def)
    done
qed
```

Good Simplification Rules

A theorem $P \Longrightarrow s = t$ is a good simplification rule if:

1. t is simpler than s
2. P is simpler than s
3. the rewrite rules *should* be confluent and not looping
   Simpler also means simpler operators, shorter term, more primitive definitions.

Simplification rules are applied blindly:

lemma
  shows ‹$\exists\, xs\ ys\ zs.\ xs\ @\ ys\ @\ zs = xs'\ @\ [a]\ @\ zs'$›
  apply *auto*
  oops

lemma
  shows ‹$\exists\, xs\ ys\ zs.\ xs\ @\ ys\ @\ zs = xs'\ @\ a\ @\ zs'$›
  apply *auto*
  oops

Simplifying Rules

lemma ‹$P\ (\mathit{if}\ b\ \mathit{then}\ s\ \mathit{else}\ t) = ((b \longrightarrow P\ s) \wedge (\neg b \longrightarrow P\ t))$›
by *simp*

For splitting over cases, you need to specify the rule:

lemma ‹$P\ (\mathit{case}\ x\ \mathit{of}\ [] \Rightarrow s\ x \mid \_ \# \_ \Rightarrow t\ x) = ((x = [] \longrightarrow P\ (s\ [])) \wedge (\forall a\ b.\ x = a \# b \longrightarrow P\ (t\ x)))$›
apply (*simp split: list.splits*)
done

thm *option.splits*

Simplifying Rules

Simplification

Splitting

Datatypes

Inductive Definitions

Datatype

datatype $('a,\ 'b)\ d = A\ 'a\ 'b \mid B\ 'a\ 'b$

Injectivity and surjectivity are applied automatically by the simplifier.

Case expression

term ‹
  case x of
    A a b ⇒ *f a b*
  | *B a _* ⇒ *g a* — wildcards are also allowed
›

Most of the time you actually want parenthesis:

definition *is_Nil* :: ‹*'a list* ⇒ *bool*› where
 ‹*is_Nil x* = (*case x of* [] ⇒ *True* | _ # _ ⇒ *False*)›

Natural numbers

Natural numbers are *not* transformed into *Suc*, except for 1!
You need:

thm *numeral_eq_Suc*

This is a heuristic to avoid explosion of goal size.
Not clear if this was a good idea or not. Mathematician often want 2 to be transformed too.

Simplifying Rules

Definitions

If you want to talk about processes:

inductive *ev* where
ev0: ‹*ev 0*› |
evSuc: ‹*ev* (*Suc* (*Suc n*))›
if ‹*ev n*› and
‹$n \geq 0$›

Example

We get also a proper induction if it is an assumption:

lemma ‹$even\ n \longleftrightarrow ev\ n$› (is ‹$?A \longleftrightarrow ?B$›)

Sets vs Fun/Definition

Inductive predicates are:

- not determistic
- not terminating
- minimal (it is not true if there is no reason to!)
- can be always false

The set version also exists.

Set Version

```
inductive_set ev_set where
ev_set0:  ‹0 ∈ ev_set› |
ev_setSuc: ‹Suc (Suc n) ∈ ev_set›
  if ‹n ∈ ev_set›
```

term ‹$(\{\}, \{a, b\})$›

Set comprehension:

term ‹$\{x.\ P\ x\}$›

term ‹$A \cup B \cap C$›

But you cannot do: $\{f\ s.\ P\ s\}$

Instead you can do:

term ‹$\{f\ s \mid s.\ P\ s\}$›

short for $\{t.\ \exists\, s.\ t = f\ s \land P\ s\}$

Or nicer for proofs:

term ‹f ‘ $\{s.\ P\ s\}$›

What Non-Proving Tactics?

- *induction*
- *cases*

What Proving Tactics Exist?

- *blast*: decision procedure for predicate logic and set theory
- *fastforce* and *force*: like *auto* but solve the goal or fail (DFS or BFS on the search space)
- *metis*: Metis-based (ordered paramodulation prover, with encoding of types for HO)
- *smt*: Z3/veriT-based (SMT solver)
- and many more (*best*, *slow*, *slowsimp*, ...)

What Proving Tactics Should I Use?

- *auto*
- *simp* (if auto is doing something weird)
- try0: try various tactics directly without additional facts
- nitpick: counter-example finder
- sledgehammer: selects relevant facts and calls ATPs. Returns a tactic in the best case. Warning: sledgehammer is *not* magic, at some point you have to write a proof.

What Proving Tactics Should I Use?

- *auto*
- *simp* (if auto is doing something weird)
- try0: try various tactics directly without additional facts
- nitpick: counter-example finder
- sledgehammer: selects relevant facts and calls ATPs. Returns a tactic in the best case.
  Warning: sledgehammer is *not* magic, at some point you have to write a proof.

What Proving Tactics Should I Use?

- *auto*
- *simp* (if auto is doing something weird)
- try0: try various tactics directly without additional facts
- nitpick: counter-example finder
- sledgehammer: selects relevant facts and calls ATPs. Returns a tactic in the best case. Warning: sledgehammer is *not* magic, at some point you have to write a proof.

Theorems

Proofs

Simplifying Rules

More Term Structure: Sets

More Tactics

Conclusion

We have seen the basics on how to write a proof.
We will see more on how to write proofs tomorrow.