How to Prove it in Isabelle/HOL

Tobias Nipkow

September 11, 2023

Abstract

How does one perform induction on the length of a list? How are numerals converted into *Suc* terms? How does one prove equalities in rings and other algebraic structures?

This document is a collection of practical hints and techniques for dealing with specific frequently occurring situations in proofs in Isabelle/HOL. Not arbitrary proofs but proofs that refer to material that is part of *Main* or *Complex_Main*.

This is *not* an introduction to

- proofs in general; for that see mathematics or logic books.
- Isabelle/HOL and its proof language; for that see the tutorial [1] or the reference manual [3].
- the contents of theory *Main*; for that see the overview [2].

Contents

Chapter 1

Main

1.1 Natural numbers

Induction rules

In addition to structural induction there is the induction rule *less_induct*:

$$(\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$$

This is often called "complete induction". It is applied like this:

(*induction n rule*: *less_induct*)

In fact, it is not restricted to *nat* but works for any wellfounded order $<$.

There are many more special induction rules. You can find all of them via the Find button (in Isabelle/jedit) with the following search criteria:

```
name: Nat name: induct
```

How to convert numerals into *Suc* terms

Solution: simplify with the lemma *numeral_eq_Suc*.
Example:

lemma fixes x :: *int* **shows** "x ^ $3 = x * x * x$"
by (*simp add*: *numeral_eq_Suc*)

This is a typical situation: function "^" is defined by pattern matching on *Suc* but is applied to a numeral.

Note: simplification with *numeral_eq_Suc* will convert all numerals. One can be more specific with the lemmas *numeral_2_eq_2* ($2 = Suc\ (Suc\ 0)$) and *numeral_3_eq_3* ($3 = Suc\ (Suc\ (Suc\ 0))$).

1.2 Lists

Induction rules

In addition to structural induction there are a few more induction rules that come in handy at times:

- Structural induction where the new element is appended to the end of the list (*rev_induct*):

  $[\![P\ [];\ \bigwedge x\ xs.\ P\ xs \Longrightarrow P\ (xs\ @\ [x])]\!] \Longrightarrow P\ xs$

- Induction on the length of a list (*length_induct*):

  $(\bigwedge xs.\ \forall ys.\ length\ ys < length\ xs \longrightarrow P\ ys \Longrightarrow P\ xs) \Longrightarrow P\ xs$

- Simultaneous induction on two lists of the same length (*list_induct2*):

  $$\begin{array}{l}[\![length\ xs = length\ ys;\ P\ []\ [];\\ \ \bigwedge x\ xs\ y\ ys.\\ \quad [\![length\ xs = length\ ys;\ P\ xs\ ys]\!] \Longrightarrow P\ (x\ \#\ xs)\ (y\ \#\ ys)]\!]\\ \Longrightarrow P\ xs\ ys\end{array}$$

1.3 Algebraic simplification

On the numeric types *nat*, *int* and *real*, proof method *simp* and friends can deal with a limited amount of linear arithmetic (no multiplication except by numerals) and method *arith* can handle full linear arithmetic (on *nat*, *int* including quantifiers). But what to do when proper multiplication is involved? At this point it can be helpful to simplify with the lemma list *algebra_simps*. Examples:

lemma fixes x :: *int*
  shows "$(x + y) * (y - z) = (y - z) * x + y * (y-z)$"
by(*simp add*: *algebra_simps*)

lemma fixes x :: "'a :: *comm_ring*"
  shows "$(x + y) * (y - z) = (y - z) * x + y * (y-z)$"
by(*simp add*: *algebra_simps*)

Rewriting with *algebra_simps* has the following effect: terms are rewritten into a normal form by multiplying out, rearranging sums and products into some canonical order. In the above lemma the normal form will be something like $x * y + y * y - x * z - y * z$. This works for concrete types like *int* as well as for classes like *comm_ring* (commutative rings). For some classes (e.g. *ring* and *comm_ring*) this yields a decision procedure for equality.

Additional function and predicate symbols are not a problem either:

lemma fixes f :: "$int \Rightarrow int$" **shows** "$2 * f(x*y) - f(y*x) < f(y*x) + 1$"
by(*simp add*: *algebra_simps*)

Here *algebra_simps* merely has the effect of rewriting $y * x$ to $x * y$ (or the other way around). This yields a problem of the form $2 * t - t < t + 1$ and we are back in the realm of linear arithmetic.

Because *algebra_simps* multiplies out, terms can explode. If one merely wants to bring sums or products into a canonical order it suffices to rewrite with *ac_simps*:

lemma fixes f :: "$int \Rightarrow int$" **shows** "$f(x*y*z) - f(z*x*y) = 0$"
by(*simp add*: *ac_simps*)

The lemmas *algebra_simps* take care of addition, subtraction and multiplication (algebraic structures up to rings) but ignore division (fields). The lemmas *field_simps* also deal with division:

lemma fixes x :: *real* **shows** "$x+z \neq 0 \Longrightarrow 1 + y/(x+z) = (x+y+z)/(x+z)$"
by(*simp add*: *field_simps*)

Warning: *field_simps* can blow up your terms beyond recognition.

Bibliography

[1] Tobias Nipkow. *Programming and Proving in Isabelle/HOL*. https://isabelle.in.tum.de/doc/prog-prove.pdf.

[2] Tobias Nipkow. *What's in Main*. https://isabelle.in.tum.de/doc/main.pdf.

[3] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. https://isabelle.in.tum.de/doc/isar-ref.pdf.