A Tutorial Introduction to Structured Isar Proofs

Tobias Nipkow

Institut für Informatik, TU München
http://www.in.tum.de/~nipkow/

1 Introduction

This is a tutorial introduction to structured proofs in Isabelle/HOL. It introduces the core of the proof language Isar by example. Isar is an extension of the `apply`-style proofs introduced in the Isabelle/HOL tutorial [4] with structured proofs in a stylised language of mathematics. These proofs are readable for both human and machine.

1.1 A first glimpse of Isar

Below you find a simplified grammar for Isar proofs. Parentheses are used for grouping and $^?$ indicates an optional item:

proof ::= **proof** *method*$^?$ *statement** **qed**
  | **by** *method*

statement ::= **fix** *variables*
  | **assume** *propositions*
  | (**from** *fact**)$^?$ (**show** | **have**) *propositions proof*

proposition ::= (*label*:)$^?$ *string*

fact ::= *label*

A proof can be either compound (**proof** – **qed**) or atomic (**by**). A *method* is a proof method.

This is a typical proof skeleton:

proof
  assume "*the-assm*"
  have "..." — intermediate result
  ⋮
  have "..." — intermediate result
  show "*the-concl*"
qed

It proves *the-assm* $\Longrightarrow$ *the-concl*. Text starting with "—" is a comment. The intermediate **have**s are only there to bridge the gap between the assumption and the conclusion and do not contribute to the theorem being proved. In contrast, **show** establishes the conclusion of the theorem.

1.2 Background

Interactive theorem proving has been dominated by a model of proof that goes back to the LCF system [2]: a proof is a more or less structured sequence of commands that manipulate an implicit proof state. Thus the proof text is only suitable for the machine; for a human, the proof only comes alive when he can see the state changes caused by the stepwise execution of the commands. Such proofs are like uncommented assembly language programs. Their Isabelle incarnation are sequences of `apply`-commands.

In contrast there is the model of a mathematics-like proof language pioneered in the Mizar system [5] and followed by Isar [7]. The most important arguments in favour of this style are *communication* and *maintainance*: structured proofs are immensely more readable and maintainable than `apply`-scripts.

For reading this tutorial you should be familiar with natural deduction and the basics of Isabelle/HOL [4] although we summarize the most important aspects of Isabelle below. The definitive Isar reference is its manual [6]. For an example-based account of Isar's support for reasoning by chains of (in)equations see [1].

1.3 Bits of Isabelle

Isabelle's meta-logic comes with a type of *propositions* with implication $\Longrightarrow$ and a universal quantifier $\bigwedge$ for expressing inference rules and generality. Iterated implications $A_1 \Longrightarrow \dots A_n \Longrightarrow A$ may be abbreviated to $[\![A_1; \dots; A_n]\!] \Longrightarrow A$. Applying a theorem $A \Longrightarrow B$ (named `T`) to a theorem A (named `U`) is written `T[OF U]` and yields theorem B.

Isabelle terms are simply typed. Function types are written $\tau_1 \Rightarrow \tau_2$.

Free variables that may be instantiated ("logical variables" in Prolog parlance) are prefixed with a `?`. Typically, theorems are stated with ordinary free variables but after the proof those are automatically replaced by `?`-variables. Thus the theorem can be used with arbitrary instances of its free variables.

Isabelle/HOL offers all the usual logical symbols like $\longrightarrow$, $\wedge$, $\forall$ etc. HOL formulae are propositions, e.g. $\forall$ can appear below $\Longrightarrow$, but not the other way around. Beware that $\longrightarrow$ binds more tightly than $\Longrightarrow$: in $\forall x.P \longrightarrow Q$ the $\forall x$ covers $P \longrightarrow Q$, whereas in $\forall x.P \Longrightarrow Q$ it covers only P.

Proof methods include `rule` (which performs a backwards step with a given rule, unifying the conclusion of the rule with the current subgoal and replacing the subgoal by the premises of the rule), `simp` (for simplification) and `blast` (for predicate calculus reasoning).

1.4 Advice

A word of warning for potential writers of Isar proofs. It is easier to write obscure rather than readable texts. Similarly, `apply`-style proofs are sometimes easier to write than readable ones: structure does not emerge automatically but needs to be understood and imposed. If the precise structure of the proof is unclear at beginning, it can be useful to start with *apply* for exploratory purposes until one has found a proof which can be converted into a structured text in a second step. Top down conversion is possible because Isar allows *apply*-style proofs as components of structured ones.

Finally, do not be mislead by the simplicity of the formulae being proved, especially in the beginning. Isar has been used very successfully in large applications, for example the formalisation of Java dialects [3].

The rest of this tutorial is divided into two parts. Section 2 introduces proofs in pure logic based on natural deduction. Section 3 is dedicated to induction.

2 Logic

2.1 Propositional logic

Introduction rules We start with a really trivial toy proof to introduce the basic features of structured proofs.

```
lemma "A ⟶ A"
proof (rule impI)
  assume a: "A"
  show "A" by(rule a)
qed
```

The operational reading: the **assume-show** block proves $A \Longrightarrow A$ (a is a degenerate rule (no assumptions) that proves A outright), which rule *impI* ($(?P \Longrightarrow ?Q) \Longrightarrow ?P \longrightarrow ?Q$) turns into the desired $A \longrightarrow A$. However, this text is much too detailed for comfort. Therefore Isar implements the following principle:

> *Command* **proof** *automatically tries to select an introduction rule based on the goal and a predefined list of rules.*

Here *impI* is applied automatically:

```
lemma "A ⟶ A"
proof
  assume a: A
  show A by(rule a)
qed
```

Single-identifier formulae such as A need not be enclosed in double quotes. However, we will continue to do so for uniformity.

Trivial proofs, in particular those by assumption, should be trivial to perform. Proof "." does just that (and a bit more). Thus naming of assumptions is often superfluous:

```
lemma "A ⟶ A"
proof
  assume "A"
  show "A" .
```

qed

To hide proofs by assumption further, **by**`(method)` first applies `method` and then tries to solve all remaining subgoals by assumption:

```
lemma "A ⟶ A ∧ A"
proof
  assume "A"
  show "A ∧ A" by(rule conjI)
qed
```

Rule `conjI` is of course $[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$. A drawback of implicit proofs by assumption is that it is no longer obvious where an assumption is used.

Proofs of the form **by**`(rule` *name*`)` can be abbreviated to ".." if *name* refers to one of the predefined introduction rules (or elimination rules, see below):

```
lemma "A ⟶ A ∧ A"
proof
  assume "A"
  show "A ∧ A" ..
qed
```

This is what happens: first the matching introduction rule `conjI` is applied (first "."), then the two subgoals are solved by assumption (second ".").

Elimination rules A typical elimination rule is `conjE`, ∧-elimination:

$$[\![?P \wedge ?Q;\ [\![?P;\ ?Q]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$$

In the following proof it is applied by hand, after its first (*major*) premise has been eliminated via `[OF AB]`:

```
lemma "A ∧ B ⟶ B ∧ A"
proof
  assume AB: "A ∧ B"
  show "B ∧ A"
  proof (rule conjE[OF AB])  — conjE[OF AB]: ([A; B] ⟹ ?R) ⟹ ?R
    assume "A" "B"
    show ?thesis ..
  qed
qed
```

Note that the term `?thesis` always stands for the "current goal", i.e. the enclosing **show** (or **have**) statement.

This is too much proof text. Elimination rules should be selected automatically based on their major premise, the formula or rather connective to be eliminated. In Isar they are triggered by facts being fed *into* a proof. Syntax:

from *fact* **show** *proposition proof*

where *fact* stands for the name of a previously proved proposition, e.g. an assumption, an intermediate result or some global theorem, which may also be modified with `OF` etc. The *fact* is "piped" into the *proof*, which can deal with it how it chooses. If the *proof* starts with a plain **proof**, an elimination rule (from a predefined list) is applied whose first premise is solved by the *fact*. Thus the proof above is equivalent to the following one:

```
lemma "A ∧ B ⟶ B ∧ A"
proof
  assume AB: "A ∧ B"
  from AB show "B ∧ A"
  proof
    assume "A" "B"
    show ?thesis ..
  qed
qed
```

Now we come to a second important principle:

> *Try to arrange the sequence of propositions in a UNIX-like pipe, such that the proof of each proposition builds on the previous proposition.*

The previous proposition can be referred to via the fact `this`. This greatly reduces the need for explicit naming of propositions:

```
lemma "A ∧ B ⟶ B ∧ A"
proof
  assume "A ∧ B"
  from this show "B ∧ A"
  proof
    assume "A" "B"
    show ?thesis ..
  qed
qed
```

Because of the frequency of **from** `this`, Isar provides two abbreviations:

then	=	**from** `this`
thus	=	**then show**

Here is an alternative proof that operates purely by forward reasoning:

```
lemma "A ∧ B ⟶ B ∧ A"
proof
  assume ab: "A ∧ B"
  from ab have a: "A" ..
  from ab have b: "B" ..
  from b a show "B ∧ A" ..
qed
```

It is worth examining this text in detail because it exhibits a number of new concepts. For a start, it is the first time we have proved intermediate propositions (**have**) on the way to the final **show**. This is the norm in nontrivial proofs where one cannot bridge the gap between the assumptions and the conclusion in one step. To understand how the proof works we need to explain more Isar details:

– Method `rule` can be given a list of rules, in which case `(rule` *rules*`)` applies the first matching rule in the list *rules*.
– Command **from** can be followed by any number of facts. Given **from** $f_1 \ldots f_n$, the proof step `(rule` *rules*`)` following a **have** or **show** searches *rules* for a rule whose first n premises can be proved by $f_1 \ldots f_n$ in the given order.
– ".." is short for `by(rule` *elim-rules intro-rules*`)`[1], where *elim-rules* and *intro-rules* are the predefined elimination and introduction rule. Thus elimination rules are tried first (if there are incoming facts).

Hence in the above proof both **have**s are proved via `conjE` triggered by **from** `ab` whereas in the **show** step no elimination rule is applicable and the proof succeeds with `conjI`. The latter would fail had we written **from** `a b` instead of **from** `b a`.

A plain **proof** with no argument is short for **proof** `(rule` *elim-rules intro-rules*`)`[1]. This means that the matching rule is selected by the incoming facts and the goal exactly as just explained.

Although we have only seen a few introduction and elimination rules so far, Isar's predefined rules include all the usual natural deduction rules. We conclude our exposition of propositional logic with an extended example — which rules are used implicitly where?

```
lemma "¬ (A ∧ B) ⟶ ¬ A ∨ ¬ B"
proof
  assume n: "¬ (A ∧ B)"
  show "¬ A ∨ ¬ B"
  proof (rule ccontr)
    assume nn: "¬ (¬ A ∨ ¬ B)"
    have "¬ A"
    proof
      assume "A"
      have "¬ B"
      proof
        assume "B"
        have "A ∧ B" ..
        with n show False ..
      qed
      hence "¬ A ∨ ¬ B" ..
      with nn show False ..
    qed
    hence "¬ A ∨ ¬ B" ..
    with nn show False ..
  qed
qed
```

Rule `ccontr` ("classical contradiction") is $(\neg P \Longrightarrow False) \Longrightarrow P$. Apart from demonstrating the strangeness of classical arguments by contradiction, this example also introduces two new abbreviations:

hence	=	**then have**
with *facts*	=	**from** *facts* `this`

[1] or merely `(rule` *intro-rules*`)` if there are no facts fed into the proof

2.2 Avoiding duplication

So far our examples have been a bit unnatural: normally we want to prove rules expressed with $\Longrightarrow$, not $\longrightarrow$. Here is an example:

lemma "$A \wedge B \Longrightarrow B \wedge A$"
proof
  assume "$A \wedge B$" **thus** "B" ..
next
  assume "$A \wedge B$" **thus** "A" ..
qed

The **proof** always works on the conclusion, $B \wedge A$ in our case, thus selecting $\wedge$-introduction. Hence we must show B and A; both are proved by $\wedge$-elimination and the proofs are separated by **next**:

next deals with multiple subgoals. For example, when showing $A \wedge B$ we need to show both A and B. Each subgoal is proved separately, in *any* order. The individual proofs are separated by **next**. [2]
Strictly speaking **next** is only required if the subgoals are proved in different assumption contexts which need to be separated, which is not the case above. For clarity we have employed **next** anyway and will continue to do so.

This is all very well as long as formulae are small. Let us now look at some devices to avoid repeating (possibly large) formulae. A very general method is pattern matching:

lemma "large_A ∧ large_B ⟹ large_B ∧ large_A"
      (**is** "?AB ⟹ ?B ∧ ?A")
proof
  assume "?AB" **thus** "?B" ..
next
  assume "?AB" **thus** "?A" ..
qed

Any formula may be followed by (**is** *pattern*) which causes the pattern to be matched against the formula, instantiating the ?-variables in the pattern. Subsequent uses of these variables in other terms causes them to be replaced by the terms they stand for.

We can simplify things even more by stating the theorem by means of the **assumes** and **shows** elements which allow direct naming of assumptions:

lemma assumes AB: "large_A ∧ large_B"
  shows "large_B ∧ large_A" (**is** "?B ∧ ?A")
proof
  from AB **show** "?B" ..
next
  from AB **show** "?A" ..

[2] Each **show** must prove one of the pending subgoals. If a **show** matches multiple subgoals, e.g. if the subgoals contain ?-variables, the first one is proved. Thus the order in which the subgoals are proved can matter — see §3.1 for an example.

qed

Note the difference between *?AB*, a term, and *AB*, a fact.

Finally we want to start the proof with ∧-elimination so we don't have to perform it twice, as above. Here is a slick way to achieve this:

lemma assumes *AB: "large_A ∧ large_B"*
  shows *"large_B ∧ large_A"* (**is** *"?B ∧ ?A"*)
using *AB*
proof
  assume *"?A" "?B"* **show** *?thesis* ..
qed

Command **using** can appear before a proof and adds further facts to those piped into the proof. Here *AB* is the only such fact and it triggers ∧-elimination. Another frequent idiom is as follows:

from *major-facts* **show** *proposition* **using** *minor-facts* *proof*

Sometimes it is necessary to suppress the implicit application of rules in a **proof**. For example **show** *A ∨ B* would trigger ∨-introduction, requiring us to prove *A*. A simple "-" prevents this *faux pas*:

lemma assumes *AB: "A ∨ B"* **shows** *"B ∨ A"*
proof -
  from *AB* **show** *?thesis*
  proof
    assume *A* **show** *?thesis* ..
  next
    assume *B* **show** *?thesis* ..
  qed
qed

Alternatively one can feed *A ∨ B* directly into the proof, thus triggering the elimination rule:

lemma assumes *AB: "A ∨ B"* **shows** *"B ∨ A"*
using *AB*
proof
  assume *A* **show** *?thesis* ..
next
  assume *B* **show** *?thesis* ..
qed

Remember that eliminations have priority over introductions.

2.3 Avoiding names

Too many names can easily clutter a proof. We already learned about *this* as a means of avoiding explicit names. Another handy device is to refer to a fact not by name but by contents: for example, writing *‘A ∨ B‘* (enclosing the formula in back quotes) refers to the fact `A ∨ B` without the need to name it. Here is a simple example, a revised version of the previous proof

```
lemma assumes "A ∨ B" shows "B ∨ A"
using ‘A ∨ B‘
```

which continues as before.

Clearly, this device of quoting facts by contents is only advisable for small formulae. In such cases it is superior to naming because the reader immediately sees what the fact is without needing to search for it in the preceding proof text.

The assumptions of a lemma can also be referred to via their predefined name *assms*. Hence the ‘*A* ∨ *B*‘ in the previous proof can also be replaced by *assms*. Note that *assms* refers to the list of *all* assumptions. To pick out a specific one, say the second, write *assms(2)*.

This indexing notation *name*(.) works for any *name* that stands for a list of facts, for example *f.simps*, the equations of the recursively defined function *f*. You may also select sublists by writing $name(2-3)$.

Above we recommended the UNIX-pipe model (i.e. *this*) to avoid the need to name propositions. But frequently we needed to feed more than one previously derived fact into a proof step. Then the UNIX-pipe model appears to break down and we need to name the different facts to refer to them. But this can be avoided:

```
lemma assumes "A ∧ B" shows "B ∧ A"
proof -
  from ‘A ∧ B‘ have "B" ..
  moreover
  from ‘A ∧ B‘ have "A" ..
  ultimately show "B ∧ A" ..
qed
```

You can combine any number of facts *A1 ... An* into a sequence by separating their proofs with **moreover**. After the final fact, **ultimately** stands for **from** *A1 ... An*. This avoids having to introduce names for all of the sequence elements.

2.4 Predicate calculus

Command **fix** introduces new local variables into a proof. The pair **fix**-**show** corresponds to $\bigwedge$ (the universal quantifier at the meta-level) just like **assume**-**show** corresponds to $\Longrightarrow$. Here is a sample proof, annotated with the rules that are applied implicitly:

```
lemma assumes P: "∀x. P x" shows "∀x. P(f x)"
proof                          — allI: (⋀x. ?P x) ⟹ ∀x. ?P x
  fix a
  from P show "P(f a)" ..      — allE: ⟦∀x. ?P x; ?P ?x ⟹ ?R⟧ ⟹ ?R
qed
```

Note that in the proof we have chosen to call the bound variable `a` instead of `x` merely to show that the choice of local names is irrelevant.

Next we look at ∃ which is a bit more tricky.

```
lemma assumes Pf: "∃x. P(f x)" shows "∃y. P y"
proof -
  from Pf show ?thesis
  proof                — exE: ⟦∃x. ?P x; ⋀x. ?P x ⟹ ?Q⟧ ⟹ ?Q
    fix x
    assume "P(f x)"
    show ?thesis ..   — exI: ?P ?x ⟹ ∃x. ?P x
  qed
qed
```

Explicit ∃-elimination as seen above can become cumbersome in practice. The derived Isar language element **obtain** provides a more appealing form of generalised existence reasoning:

```
lemma assumes Pf: "∃x. P(f x)" shows "∃y. P y"
proof -
  from Pf obtain x where "P(f x)" ..
  thus "∃y. P y" ..
qed
```

Note how the proof text follows the usual mathematical style of concluding $P(x)$ from $\exists x.P(x)$, while carefully introducing x as a new local variable. Technically, **obtain** is similar to **fix** and **assume** together with a soundness proof of the elimination involved.

Here is a proof of a well known tautology. Which rule is used where?

```
lemma assumes ex: "∃x. ∀y. P x y" shows "∀y. ∃x. P x y"
proof
  fix y
  from ex obtain x where "∀y. P x y" ..
  hence "P x y" ..
  thus "∃x. P x y" ..
qed
```

2.5 Making bigger steps

So far we have confined ourselves to single step proofs. Of course powerful automatic methods can be used just as well. Here is an example, Cantor's theorem that there is no surjective function from a set to its powerset:

```
theorem "∃S. S ∉ range (f :: 'a ⇒ 'a set)"
proof
  let ?S = "{x. x ∉ f x}"
  show "?S ∉ range f"
  proof
    assume "?S ∈ range f"
    then obtain y where "?S = f y" ..
    show False
    proof cases
```

```
      assume "y ∈ ?S"
      with `?S = f y` show False by blast
    next
      assume "y ∉ ?S"
      with `?S = f y` show False by blast
    qed
  qed
qed
```

For a start, the example demonstrates two new constructs:

- **let** introduces an abbreviation for a term, in our case the witness for the claim.
- Proof by *cases* starts a proof by cases. Note that it remains implicit what the two cases are: it is merely expected that the two subproofs prove $P \Longrightarrow$ *?thesis* and $\neg P \Longrightarrow$ *?thesis* (in that order) for some P.

If you wonder how to **obtain** y: via the predefined elimination rule $[\![b \in range\ f;\ \bigwedge x.\ b = f\ x \Longrightarrow P]\!] \Longrightarrow P$.

Method *blast* is used because the contradiction does not follow easily by just a single rule. If you find the proof too cryptic for human consumption, here is a more detailed version; the beginning up to **obtain** stays unchanged.

```
theorem "∃S. S ∉ range (f :: 'a ⇒ 'a set)"
proof
  let ?S = "{x. x ∉ f x}"
  show "?S ∉ range f"
  proof
    assume "?S ∈ range f"
    then obtain y where "?S = f y" ..
    show False
    proof cases
      assume "y ∈ ?S"
      hence "y ∉ f y"   by simp
      hence "y ∉ ?S"    by(simp add: `?S = f y`)
      thus False        by contradiction
    next
      assume "y ∉ ?S"
      hence "y ∈ f y"   by simp
      hence "y ∈ ?S"    by(simp add: `?S = f y`)
      thus False        by contradiction
    qed
  qed
qed
```

Method *contradiction* succeeds if both P and $\neg P$ are among the assumptions and the facts fed into that step, in any order.

As it happens, Cantor's theorem can be proved automatically by best-first search. Depth-first search would diverge, but best-first search successfully navigates through the large search space:

```
theorem "∃S. S ∉ range (f :: 'a ⇒ 'a set)"
by best
```

2.6 Raw proof blocks

Although we have shown how to employ powerful automatic methods like *blast* to achieve bigger proof steps, there may still be the tendency to use the default introduction and elimination rules to decompose goals and facts. This can lead to very tedious proofs:

```
lemma "∀x y. A x y ∧ B x y ⟶ C x y"
proof
  fix x show "∀y. A x y ∧ B x y ⟶ C x y"
  proof
    fix y show "A x y ∧ B x y ⟶ C x y"
    proof
      assume "A x y ∧ B x y"
      show "C x y" sorry
    qed
  qed
qed
```

Since we are only interested in the decomposition and not the actual proof, the latter has been replaced by **sorry**. Command **sorry** proves anything but is only allowed in quick and dirty mode, the default interactive mode. It is very convenient for top down proof development.

Luckily we can avoid this step by step decomposition very easily:

```
lemma "∀x y. A x y ∧ B x y ⟶ C x y"
proof -
  have "⋀x y. ⟦ A x y; B x y ⟧ ⟹ C x y"
  proof -
    fix x y assume "A x y" "B x y"
    show "C x y" sorry
  qed
  thus ?thesis by blast
qed
```

This can be simplified further by *raw proof blocks*, i.e. proofs enclosed in braces:

```
lemma "∀x y. A x y ∧ B x y ⟶ C x y"
proof -
  { fix x y assume "A x y" "B x y"
    have "C x y" sorry }
  thus ?thesis by blast
qed
```

The result of the raw proof block is the same theorem as above, namely $\bigwedge x\ y.\ [\![A\ x\ y;\ B\ x\ y]\!] \Longrightarrow C\ x\ y$. Raw proof blocks are like ordinary proofs except that they do not prove some explicitly stated property but that the property emerges directly out of the **fix**es, **assume**s and **have** in the block. Thus they again serve to avoid duplication. Note that the conclusion of a raw proof block is stated with **have** rather than **show** because it is not the conclusion of some pending goal but some independent claim.

The general idea demonstrated in this subsection is very important in Isar and distinguishes it from `apply`-style proofs:

> *Do not manipulate the proof state into a particular form by applying proof methods but state the desired form explicitly and let the proof methods verify that from this form the original goal follows.*

This yields more readable and also more robust proofs.

General case distinctions As an important application of raw proof blocks we show how to deal with general case distinctions — more specific kinds are treated in §3.1. Imagine that you would like to prove some goal by distinguishing n cases $P_1, \ldots, P_n$. You show that the n cases are exhaustive (i.e. $P_1 \vee \ldots \vee P_n$) and that each case P_i implies the goal. Taken together, this proves the goal. The corresponding Isar proof pattern (for $n = 3$) is very handy:

```
proof -
  have "P1 ∨ P2 ∨ P3"  ...
  moreover
  { assume P1
    ...
    have ?thesis  ...  }
  moreover
  { assume P2
    ...
    have ?thesis  ...  }
  moreover
  { assume P3
    ...
    have ?thesis  ...  }
  ultimately show ?thesis by blast
qed
```

2.7 Further refinements

This subsection discusses some further tricks that can make life easier although they are not essential.

and Propositions (following **assume** etc) may but need not be separated by **and**. This is not just for readability (**from** `A` **and** `B` looks nicer than **from** `A B`) but for structuring lists of propositions into possibly named blocks. In

$$\textbf{assume } \texttt{A:}\ A_1\ A_2\ \textbf{and } \texttt{B:}\ A_3\ \textbf{and } A_4$$

label `A` refers to the list of propositions A_1 A_2 and label `B` to A_3.

note If you want to remember intermediate fact(s) that cannot be named directly, use **note**. For example the result of raw proof block can be named by following it with **note** *some_name = this*. As a side effect, *this* is set to the list of facts on the right-hand side. You can also say *note some_fact*, which simply sets *this*, i.e. recalls *some_fact*, e.g. in a **moreover** sequence.

fixes Sometimes it is necessary to decorate a proposition with type constraints, as in Cantor's theorem above. These type constraints tend to make the theorem less readable. The situation can be improved a little by combining the type constraint with an outer $\bigwedge$:

theorem *"⋀f :: 'a ⇒ 'a set. ∃S. S ∉ range f"*

However, now *f* is bound and we need a **fix** *f* in the proof before we can refer to *f*. This is avoided by **fixes**:

theorem fixes *f :: "'a ⇒ 'a set"* **shows** *"∃S. S ∉ range f"*

Even better, **fixes** allows to introduce concrete syntax locally:

```
lemma comm_mono:
  fixes r :: "'a ⇒ 'a ⇒ bool" (infix ">" 60) and
        f :: "'a ⇒ 'a ⇒ 'a"    (infixl "++" 70)
  assumes comm: "⋀x y::'a. x ++ y = y ++ x" and
          mono: "⋀x y z::'a. x > y ⟹ x ++ z > y ++ z"
  shows "x > y ⟹ z ++ x > z ++ y"
by(simp add: comm mono)
```

The concrete syntax is dropped at the end of the proof and the theorem becomes

```
⟦⋀x y. ?f x y = ?f y x;
 ⋀x y z. ?r x y ⟹ ?r (?f x z) (?f y z); ?r ?x ?y⟧
⟹ ?r (?f ?z ?x) (?f ?z ?y)
```

obtain The **obtain** construct can introduce multiple witnesses and propositions as in the following proof fragment:

```
lemma assumes A: "∃x y. P x y ∧ Q x y" shows "R"
proof -
  from A obtain x y where P: "P x y" and Q: "Q x y"  by blast
```

Remember also that one does not even need to start with a formula containing ∃ as we saw in the proof of Cantor's theorem.

Combining proof styles Finally, whole *apply*-scripts may appear in the leaves of the proof tree, although this is best avoided. Here is a contrived example:

```
lemma "A ⟶ (A ⟶ B) ⟶ B"
proof
```

```
  assume a: "A"
  show "(A ⟶ B) ⟶ B"
    apply(rule impI)
    apply(erule impE)
    apply(rule a)
    apply assumption
    done
qed
```

You may need to resort to this technique if an automatic step fails to prove the desired proposition.

When converting a proof from *apply*-style into Isar you can proceed in a top-down manner: parts of the proof can be left in script form while the outer structure is already expressed in Isar.

3 Case distinction and induction

Computer science applications abound with inductively defined structures, which is why we treat them in more detail. HOL already comes with a datatype of lists with the two constructors *Nil* and *Cons*. *Nil* is written *[]* and *Cons x xs* is written *x # xs*.

3.1 Case distinction

We have already met the *cases* method for performing binary case splits. Here is another example:

```
lemma "¬ A ∨ A"
proof cases
  assume "A" thus ?thesis ..
next
  assume "¬ A" thus ?thesis ..
qed
```

The two cases must come in this order because *cases* merely abbreviates *(rule case_split_thm)* where *case_split_thm* is $[\![?P \Longrightarrow ?Q; \neg ?P \Longrightarrow ?Q]\!] \Longrightarrow ?Q$. If we reverse the order of the two cases in the proof, the first case would prove $\neg A \Longrightarrow \neg A \lor A$ which would solve the first premise of *case_split_thm*, instantiating $?P$ with $\neg A$, thus making the second premise $\neg \neg A \Longrightarrow \neg A \lor A$. Therefore the order of subgoals is not always completely arbitrary.

The above proof is appropriate if *A* is textually small. However, if *A* is large, we do not want to repeat it. This can be avoided by the following idiom

```
lemma "¬ A ∨ A"
proof (cases "A")
  case True thus ?thesis ..
next
  case False thus ?thesis ..
```

```
qed
```

which is like the previous proof but instantiates *?P* right away with *A*. Thus we could prove the two cases in any order. The phrase **case** *True* abbreviates **assume** *True: A* and analogously for *False* and $\neg$ *A*.

The same game can be played with other datatypes, for example lists, where *tl* is the tail of a list, and *length* returns a natural number (remember: $0-1=0$):

```
lemma "length(tl xs) = length xs - 1"
proof (cases xs)
  case Nil thus ?thesis by simp
next
  case Cons thus ?thesis by simp
qed
```

Here **case** *Nil* abbreviates **assume** *Nil: xs = []* and **case** *Cons* abbreviates **fix** *? ??* **assume** *Cons: xs = ? # ??*, where *?* and *??* stand for variable names that have been chosen by the system. Therefore we cannot refer to them. Luckily, this proof is simple enough we do not need to refer to them. However, sometimes one may have to. Hence Isar offers a simple scheme for naming those variables: replace the anonymous *Cons* by *(Cons y ys)*, which abbreviates **fix** *y ys* **assume** *Cons: xs = y # ys*. In each **case** the assumption can be referred to inside the proof by the name of the constructor. In Section 3.4 below we will come across an example of this.

3.2 Structural induction

We start with an inductive proof where both cases are proved automatically:

```
lemma "2 * (∑i::nat≤n. i) = n*(n+1)"
by (induct n) simp_all
```

The constraint *::nat* is needed because all of the operations involved are overloaded. This proof also demonstrates that **by** can take two arguments, one to start and one to finish the proof — the latter is optional.

If we want to expose more of the structure of the proof, we can use pattern matching to avoid having to repeat the goal statement:

```
lemma "2 * (∑i::nat≤n. i) = n*(n+1)" (is "?P n")
proof (induct n)
  show "?P 0" by simp
next
  fix n assume "?P n"
  thus "?P(Suc n)" by simp
qed
```

We could refine this further to show more of the equational proof. Instead we explore the same avenue as for case distinctions: introducing context via the **case** command:

```
lemma "2 * (∑i::nat ≤ n. i) = n*(n+1)"
```

```
proof (induct n)
  case 0 show ?case by simp
next
  case Suc thus ?case by simp
qed
```

The implicitly defined `?case` refers to the corresponding case to be proved, i.e. `?P 0` in the first case and `?P(Suc n)` in the second case. Context **case** `0` is empty whereas **case** `Suc` assumes `?P n`. Again we have the same problem as with case distinctions: we cannot refer to an anonymous `n` in the induction step because it has not been introduced via **fix** (in contrast to the previous proof). The solution is the one outlined for `Cons` above: replace `Suc` by `(Suc i)`:

```
lemma fixes n::nat shows "n < n*n + 1"
proof (induct n)
  case 0 show ?case by simp
next
  case (Suc i) thus "Suc i < Suc i * Suc i + 1" by simp
qed
```

Of course we could again have written **thus** `?case` instead of giving the term explicitly but we wanted to use `i` somewhere.

3.3 Generalization via `arbitrary`

It is frequently necessary to generalize a claim before it becomes provable by induction. The tutorial [4] demonstrates this with `itrev xs ys = rev xs @ ys`, where `ys` needs to be universally quantified before induction succeeds.[3] But strictly speaking, this quantification step is already part of the proof and the quantifiers should not clutter the original claim. This is how the quantification step can be combined with induction:

```
lemma "itrev xs ys = rev xs @ ys"
by (induct xs arbitrary: ys) simp_all
```

The annotation `arbitrary:` *vars* universally quantifies all *vars* before the induction. Hence they can be replaced by *arbitrary* values in the proof.

The nice thing about generalization via `arbitrary:` is that in the induction step the claim is available in unquantified form but with the generalized variables replaced by `?`-variables, ready for instantiation. In the above example the induction hypothesis is `itrev xs ?ys = rev xs @ ?ys`.

For the curious: `arbitrary:` introduces $\bigwedge$ behind the scenes.

3.4 Inductive proofs of conditional formulae

Induction also copes well with formulae involving $\Longrightarrow$, for example

[3] `rev [] = []`, `rev (x # xs) = rev xs @ [x]`,
`itrev [] ys = ys`, `itrev (x # xs) ys = itrev xs (x # ys)`

```
lemma "xs ≠ [] ⟹ hd(rev xs) = last xs"
by (induct xs) simp_all
```

This is an improvement over that style the tutorial [4] advises, which requires $\longrightarrow$. A further improvement is shown in the following proof:

```
lemma  "map f xs = map f ys ⟹ length xs = length ys"
proof (induct ys arbitrary: xs)
  case Nil thus ?case by simp
next
  case (Cons y ys)  note Asm = Cons
  show ?case
  proof (cases xs)
    case Nil
    hence False using Asm(2) by simp
    thus ?thesis ..
  next
    case (Cons x xs')
    with Asm(2) have "map f xs' = map f ys" by simp
    from Asm(1)[OF this] `xs = x#xs'` show ?thesis by simp
  qed
qed
```

The base case is trivial. In the step case Isar assumes (under the name `Cons`) two propositions:

```
map f ?xs = map f ys ⟹ length ?xs = length ys
map f xs = map f (y # ys)
```

The first is the induction hypothesis, the second, and this is new, is the premise of the induction step. The actual goal at this point is merely `length xs = length (y # ys)`. The assumptions are given the new name `Asm` to avoid a name clash further down. The proof procedes with a case distinction on `xs`. In the case `xs = []`, the second of our two assumptions (`Asm(2)`) implies the contradiction `0 = Suc(...)`. In the case `xs = x # xs'`, we first obtain `map f xs' = map f ys`, from which a forward step with the first assumption (`Asm(1)[OF this]`) yields `length xs' = length ys`. Together with `xs = x # xs` this yields the goal `length xs = length (y # ys)`.

3.5 Induction formulae involving $\bigwedge$ or $\Longrightarrow$

Let us now consider abstractly the situation where the goal to be proved contains both $\bigwedge$ and $\Longrightarrow$, say $\bigwedge x.\ P\ x \Longrightarrow Q\ x$. This means that in each case of the induction, `?case` would be of the form $\bigwedge x.\ P'\ x \Longrightarrow Q'\ x$. Thus the first proof steps will be the canonical ones, fixing `x` and assuming `P' x`. To avoid this tedium, induction performs the canonical steps automatically: in each step case, the assumptions contain both the usual induction hypothesis and `P' x`, whereas `?case` is only `Q' x`.

3.6 Rule induction

HOL also supports inductively defined sets. See [4] for details. As an example we define our own version of the reflexive transitive closure of a relation — HOL provides a predefined one as well.

```
inductive_set
  rtc :: "('a × 'a)set ⇒ ('a × 'a)set"   ("_*" [1000] 999)
  for r :: "('a × 'a)set"
where
  refl:  "(x,x) ∈ r*"
| step:  "⟦ (x,y) ∈ r; (y,z) ∈ r* ⟧ ⟹ (x,z) ∈ r*"
```

First the constant is declared as a function on binary relations (with concrete syntax `r*` instead of `rtc r`), then the defining clauses are given. We will now prove that `r*` is indeed transitive:

```
lemma assumes A: "(x,y) ∈ r*" shows "(y,z) ∈ r* ⟹ (x,z) ∈ r*"
using A
proof induct
  case refl thus ?case .
next
  case step thus ?case by(blast intro: rtc.step)
qed
```

Rule induction is triggered by a fact $(x_1, \ldots, x_n) \in R$ piped into the proof, here **using** `A`. The proof itself follows the inductive definition very closely: there is one case for each rule, and it has the same name as the rule, analogous to structural induction.

However, this proof is rather terse. Here is a more readable version:

```
lemma assumes "(x,y) ∈ r*" and "(y,z) ∈ r*" shows "(x,z) ∈ r*"
using assms
proof induct
  fix x assume "(x,z) ∈ r*"  — B[y := x]
  thus "(x,z) ∈ r*" .
next
  fix x' x y
  assume 1: "(x',x) ∈ r" and
         IH: "(y,z) ∈ r* ⟹ (x,z) ∈ r*" and
         B:  "(y,z) ∈ r*"
  from 1 IH[OF B] show "(x',z) ∈ r*" by(rule rtc.step)
qed
```

This time, merely for a change, we start the proof with by feeding both assumptions into the inductive proof. Only the first assumption is "consumed" by the induction. Since the second one is left over we don't just prove `?thesis` but `(y,z) ∈ r* ⟹ ?thesis`, just as in the previous proof. The base case is trivial. In the assumptions for the induction step we can see very clearly how things fit together and permit ourselves the obvious forward step `IH[OF B]`.

The notation **case** `(`*constructor vars*`)` is also supported for inductive definitions. The *constructor* is the name of the rule and the *vars* fix the free variables in the rule; the order of the *vars* must correspond to the left-to-right order of the variables as they appear in the rule. For example, we could start the above detailed proof of the induction with **case** *(step x' x y)*. In that case we don't need to spell out the assumptions but can refer to them by *step(.)*, although the resulting text will be quite cryptic.

3.7 More induction

We close the section by demonstrating how arbitrary induction rules are applied. As a simple example we have chosen recursion induction, i.e. induction based on a recursive function definition. However, most of what we show works for induction in general.

The example is an unusual definition of rotation:

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

This yields, among other things, the induction rule *rot.induct*:

$$[\![P\ [];\ \bigwedge x.\ P\ [x];\ \bigwedge x\ y\ zs.\ P\ (x\ \#\ zs) \Longrightarrow P\ (x\ \#\ y\ \#\ zs)]\!] \Longrightarrow P\ a0$$

The following proof relies on a default naming scheme for cases: they are called 1, 2, etc, unless they have been named explicitly. The latter happens only with datatypes and inductively defined sets, but (usually) not with recursive functions.

```
lemma "xs ≠ [] ⟹ rot xs = tl xs @ [hd xs]"
proof (induct xs rule: rot.induct)
  case 1 thus ?case by simp
next
  case 2 show ?case by simp
next
  case (3 a b cs)
  have "rot (a # b # cs) = b # rot(a # cs)" by simp
  also have "... = b # tl(a # cs) @ [hd(a # cs)]" by(simp add:3)
  also have "... = tl (a # b # cs) @ [hd (a # b # cs)]" by simp
  finally show ?case .
qed
```

The third case is only shown in gory detail (see [1] for how to reason with chains of equations) to demonstrate that the **case** *(constructor vars)* notation also works for arbitrary induction theorems with numbered cases. The order of the *vars* corresponds to the order of the $\bigwedge$-quantified variables in each case of the induction theorem. For induction theorems produced by **fun** it is the order in which the variables appear on the left-hand side of the equation.

The proof is so simple that it can be condensed to

```
by (induct xs rule: rot.induct) simp_all
```

References

1. Gertrud Bauer and Markus Wenzel. Calculational reasoning revisited — an Isabelle/Isar experience. In R. Boulton and P. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lect. Notes in Comp. Sci.*, pages 75–90. Springer-Verlag, 2001.
2. M.C.J. Gordon, Robin Milner, and C.P. Wadsworth. *Edinburgh LCF: a Mechanised Logic of Computation*, volume 78 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1979.
3. Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
4. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.
5. P. Rudnicki. An overview of the Mizar project. In *Workshop on Types for Proofs and Programs*. Chalmers University of Technology, 1992.
6. Markus Wenzel. *The Isabelle/Isar Reference Manual*. Technische Universität München, 2002. http://isabelle.in.tum.de/dist/Isabelle2002/doc/isar-ref.pdf.
7. Markus Wenzel and Freek Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *J. Automated Reasoning*, pages 389–411, 2002.