Structured Proofs in Isar/HOL

Tobias Nipkow

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

Abstract. Isar is an extension of the theorem prover Isabelle with a language for writing human-readable structured proofs. This paper is an introduction to the basic constructs of this language.

1 Introduction

Isabelle is a generic proof assistant. Isar is an extension of Isabelle with structured proofs in a stylised language of mathematics. These proofs are readable for both a human and a machine. Isabelle/HOL [4] is a specialisation of Isabelle with higher-order logic (HOL). This paper is a compact introduction to structured proofs in Isar/HOL, an extension of Isabelle/HOL. We intentionally do not present the full language but concentrate on the essentials. Neither do we give a formal semantics of Isar. Instead we introduce Isar by example. We believe that the language "speaks for itself" in the same way that traditional mathematical proofs do, which are also introduced by example rather than by teaching students logic first. A detailed exposition of Isar can be found in Markus Wenzel's PhD thesis [6] (which also discusses related work) and the Isar reference manual [7].

1.1 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. We call them *tactic-style* proofs because LCF proof commands were called tactics.

A radically different approach was taken by the Mizar system [5] where proofs are written in a stylised language akin to that used in ordinary mathematics texts. The most important argument in favour of a mathematics-like proof language is communication: as soon as not just the theorem but also the proof becomes an object of interest, it should be readable. From a system development point of view there is a second important argument against tactic-style proofs: they are much harder to maintain when the system is modified.

For these reasons the Isabelle system, originally firmly in the LCF-tradition, was extended with a language for writing structured proofs in a mathematics-like style. As the name already indicates, Isar was certainly inspired by Mizar.

However, there are many differences. For a start, Isar is generic: only a few of the language constructs described below are specific to HOL; many are generic and thus available for any logic defined in Isabelle, e.g. ZF. Furthermore, we have Isabelle's powerful automatic proof procedures at our disposal. A closer comparison of Isar and Mizar can be found elsewhere [8].

1.2 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** *facts*)$^?$ (**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 (tactic) offered by the underlying theorem prover. Thus this grammar is generic both w.r.t. the logic and the theorem prover.

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.

1.3 Bits of Isabelle

We recall some basic notions and notation from Isabelle. For more details and for instructions how to run examples see elsewhere [4].

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 Overview of the paper

The rest of the paper is divided into two parts. Section 2 introduces proofs in pure logic based on natural deduction. Section 3 is dedicated to induction, the key reasoning principle for computer science applications.

There are two further areas where Isar provides specific support, but which we do not document here. Reasoning by chains of (in)equations is described elsewhere [1]. Reasoning about axiomatically defined structures by means of so called "locales" was first described in [3] but has evolved much since then.

Finally, a word of warning for potential writers of Isar proofs. It has always been easier to write obscure rather than readable texts. Similarly, tactic-style proofs are often (though by no means always!) 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 in a tactic-based style for exploratory purposes until one has found a proof which can be converted into a structured text in a second step.

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 "$A \longrightarrow A$"
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 "$A \longrightarrow A$"
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 "$A \longrightarrow A \wedge A$"
proof
  assume "A"
  show "$A \wedge A$" **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 "$A \longrightarrow A \wedge A$"
proof
  assume "A"
  show "$A \wedge A$" ..
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*, $\wedge$-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. Finally one needs to know that ".." is short for *by(rule elim-rules intro-rules)* (or *by(rule intro-rules)* if there are no facts fed into the proof), i.e. elimination rules are tried before introduction rules.

Thus 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*.

Proofs starting with a plain *proof* behave the same because the latter is short for *proof (rule elim-rules intro-rules)* (or *proof (rule intro-rules)* if there are no facts fed into the proof).

2.2 More constructs

In the previous proof of $A \wedge B \longrightarrow B \wedge A$ we needed to feed more than one fact into a proof step, a frequent situation. 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 "A ∧ B ⟶ B ∧ A"
proof
  assume ab: "A ∧ B"
  from ab have "B" ..
  moreover
  from ab 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.

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 `(¬ P ⟹ False) ⟹ 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`

2.3 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**. [1]
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 "$large\_A \wedge large\_B \Longrightarrow large\_B \wedge large\_A$"
    (**is** "$?AB \Longrightarrow ?B \wedge ?A$")
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: "$large\_A \wedge large\_B$"
  shows "$large\_B \wedge large\_A$" (**is** "$?B \wedge ?A$")
proof
  from AB **show** "$?B$" ..

[1] 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.

```
next
  from AB show "?A" ..
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
```

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 fy: "?S = f y" ..
```

```
    show False
    proof cases
      assume "y ∈ ?S"
      with fy show False by blast
    next
      assume "y ∉ ?S"
      with fy 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 fy: "?S = f y" ..
    show False
    proof cases
      assume "y ∈ ?S"
      hence "y ∉ f y"   by simp
      hence "y ∉ ?S"    by(simp add:fy)
      thus False        by contradiction
    next
      assume "y ∉ ?S"
      hence "y ∈ f y"   by simp
      hence "y ∈ ?S"    by(simp add:fy)
      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 **fixe**s, **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 tactic-style proofs:

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

This yields more readable and also more robust proofs.

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

assume *A:* $A_1\ A_2$ **and** *B:* A_3 **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 "$\bigwedge f$ :: 'a ⇒ 'a set. $\exists S.\ S \notin$ 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 "scripts" (tactic-based proofs in the style of [4]) 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 tactic-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 3.3 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<n+1. i) = n*(n+1)"
by (induct n, simp_all)
```

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<n+1. 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<n+1. 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 Induction formulae involving $\bigwedge$ or $\Longrightarrow$

Let us now consider the situation where the goal to be proved contains $\bigwedge$ or $\Longrightarrow$, say $\bigwedge x.\ P\ x \Longrightarrow Q\ x$ — motivation and a real example follow shortly. 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 these steps automatically: for example in case *(Suc n)*, *?case* is only *Q' x* whereas the assumptions (named *Suc*!) contain both the usual induction hypothesis *and* *P' x*. It should be clear how this generalises to more complex formulae.

As an example we will now prove complete induction via structural induction.

```
lemma assumes A: "(⋀n. (⋀m. m < n ⟹ P m) ⟹ P n)"
  shows "P(n::nat)"
proof (rule A)
  show "⋀m. m < n ⟹ P m"
  proof (induct n)
    case 0 thus ?case by simp
  next
    case (Suc n)   — fix m assume Suc: "?m < n ⟹ P ?m" "m < Suc n"
    show ?case     — P m
    proof cases
      assume eq: "m = n"
      from Suc and A have "P n" by blast
      with eq show "P m" by simp
    next
      assume "m ≠ n"
      with Suc have "m < n" by arith
      thus "P m" by(rule Suc)
    qed
  qed
qed
```

Given the explanations above and the comments in the proof text (only necessary for novices), the proof should be quite readable.

The statement of the lemma is interesting because it deviates from the style in the Tutorial [4], which suggests to introduce $\forall$ or $\longrightarrow$ into a theorem to strengthen it for induction. In Isar proofs we can use $\bigwedge$ and $\Longrightarrow$ instead. This simplifies the proof and means we do not have to convert between the two kinds of connectives.

Note that in a nested induction over the same data type, the inner case labels hide the outer ones of the same name. If you want to refer to the outer ones inside, you need to name them on the outside, e.g. **note** *outer_IH = Suc*.

3.4 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.

```
consts rtc :: "('a × 'a)set ⇒ ('a × 'a)set"   ("_*" [1000] 999)
inductive "r*"
intros
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 A: "(x,y) ∈ r*" and B: "(y,z) ∈ r*"
  shows "(x,z) ∈ r*"
proof -
  from A B show ?thesis
  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
qed
```

We start the proof with **from** *A B*. Only *A* is "consumed" by the induction step. Since *B* is left over we don't just prove *?thesis* but *B* ⟹ *?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 *alphabetical 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)*. However, we can then only refer to the assumptions named *step* collectively and not individually, as the above proof requires.

3.5 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:

```
consts rot :: "'a list ⇒ 'a list"
recdef rot "measure length"  — for the internal termination proof
"rot [] = []"
"rot [x] = [x]"
"rot (x#y#zs) = y # rot(x#zs)"
```

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

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

In the following proof we rely 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 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 **recdef** 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)
```

Acknowledgement I am deeply indebted to Markus Wenzel for conceiving Isar. Clemens Ballarin, Gertrud Bauer, Stefan Berghofer, Gerwin Klein, Norbert Schirmer, Markus Wenzel and Freek Wiedijk commented on and improved this paper.

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. Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales: A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics, TPHOLs'99*, volume 1690 of *Lect. Notes in Comp. Sci.*, pages 149–165. Springer-Verlag, 1999.
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. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents.* PhD thesis, Institut für Informatik, Technische Universität München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.
7. Markus Wenzel. *The Isabelle/Isar Reference Manual.* Technische Universität München, 2002. http://isabelle.in.tum.de/dist/Isabelle2002/doc/isar-ref.pdf.
8. Markus Wenzel and Freek Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *J. Automated Reasoning*, 2003. To appear.