Interactive Proof: Introduction to Isabelle/HOL

Tobias NIPKOW

Technische Universität München

Abstract. This paper introduces interactive theorem proving with the Isabelle/HOL system [3]. The following topics are covered:

- Verified functional programming: The logic HOL contains an ML-style functional programming language. It is shown how to verify functional programs in this language by induction and simplification.
- Predicate logic: Formulas of predicate logic and set theory are introduced, together with methods for proof automation.
- Inductively defined predicates.
- Structured proofs: We introduce the proof language Isar and show how to write structured proofs that are readable by both the machine and the human.

We assume basic familiarity with some functional programming language of the ML or Haskell family, in particular with recursive data types and pattern matching. No specific background in logic is necessary beyond the ability to read predicate logic formulas.

1. Introduction

Isabelle is a generic system for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, which abbreviates Higher-Order Logic. We introduce HOL step by step following the equation

$$\text{HOL} = \text{Functional Programming} + \text{Logic}.$$

We assume that the reader is familiar with the basic concepts of functional programming and is used to logical and set theoretic notation.

Section §3 introduces HOL as a functional programming language and explains how to write simple inductive proofs of mostly equational properties of recursive functions. Section §4 covers the full logic, sets and automatic proof tools. Section §5 explains inductive definitions. Section §6 introduces Isabelle's full language for writing structured proofs.

Further material (slides, demos etc) is found online at www.in.tum.de/~nipkow.

2. Basics

2.1. Types, Terms and Formulae

HOL is a typed logic whose type system resembles that of functional programming languages. Thus there are

base types, in particular `bool`, the type of truth values, `nat`, the type of natural numbers, and `int`, the type of integers.

type constructors, in particular `list`, the type of lists, and `set`, the type of sets. Type constructors are written postfix, e.g. `nat list` is the type of lists whose elements are natural numbers.

function types, denoted by $\Rightarrow$.

type variables, denoted by `'a`, `'b` etc., just like in ML.

Terms are formed as in functional programming by applying functions to arguments. If `f` is a function of type $\tau_1 \Rightarrow \tau_2$ and `t` is a term of type τ_1 then `f t` is a term of type τ_2. We write `t :: `τ to mean that term `t` has type τ.

There are many predefined infix symbols like `+` and $\leq$. The name of the corresponding binary function is `op +`, not just `+`. That is, `x + y` is syntactic sugar for `op + x y`.

HOL also supports some basic constructs from functional programming:

```
(if b then t1 else t2)
(let x = t in u)
(case t of pat1 => t1 | ... | patn => tn)
```

The above three constructs must always be enclosed in parentheses if they sit inside other constructs.

Terms may also contain λ-abstractions. For example, `%x. x` is the identity function.

Formulae are terms of type `bool`. There are the basic constants `True` and `False` and the usual logical connectives (in decreasing order of precedence): $\neg$, $\wedge$, $\vee$, $\longrightarrow$.

Equality is available in the form of the infix function = of type `'a => 'a => bool`. It also works for formulas, where it means "if and only if".

Quantifiers are written $\forall$`x. P` and $\exists$`x. P`.

Isabelle automatically computes the type of each variable in a term. This is called *type inference*. Despite type inference, it is sometimes necessary to attach explicit *type constraints* (or *type annotations*) to a variable or term. The syntax is `t :: `τ as in `m < (n::nat)`. Type constraints may be needed to disambiguate terms involving overloaded functions such as `+`, $*$ and $\leq$.

Finally there are the universal quantifier $\bigwedge$ and the implication $\Longrightarrow$. They are part of the Isabelle framework, not the logic HOL. Logically, they agree with their HOL counterparts $\forall$ and $\longrightarrow$, but operationally they behave differently. This will become clearer as we go along.

Right-arrows of all kinds always associate to the right. In particular, $A_1 \Longrightarrow A_2 \Longrightarrow A_3$ means $A_1 \Longrightarrow (A_2 \Longrightarrow A_3)$. The notation $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ is short for the iterated implication $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A$. Sometimes we also employ inference rule notation: $\dfrac{A_1 \quad \ldots \quad A_n}{A}$

2.2. Theories

Roughly speaking, a *theory* is a named collection of types, functions, and theorems, much like a module in a programming language. The general format of a theory `T` is

theory `T`
imports T_1 ... T_n
begin
definitions, theorems and proofs
end

where T_1 ... T_n are the names of existing theories that `T` is based on. The T_i are the direct *parent theories* of `T`. Everything defined in the parent theories (and their parents, recursively) is automatically visible. Each theory `t` must reside in a *theory file* named `T.thy`.

HOL contains a theory `Main`, the union of all the basic predefined theories like arithmetic, lists, sets, etc. Unless you know what you are doing, always include `Main` as a direct or indirect parent of all your theories.

In addition to the theories that come with the Isabelle/HOL distribution (see isabelle.in.tum.de/library/HOL/) there is also the *Archive of Formal Proofs* at afp.sourceforge.net, a growing collection of Isabelle theories that everybody can contribute to.

3. Programming and Proving

This section introduces HOL as a functional programming language and shows how to prove properties of functional programs by induction. Our approach is mostly example-based: We concentrate on examples that cover the typical cases and do not explain the general case if it can be inferred from the examples.

We start by examining the the most important predefined types.

3.1. Types `bool`, `nat` and `list`

3.1.1. Type `bool`

The type of boolean values is a predefined data type

```
datatype bool = True | False
```

with many predefined functions: $\neg$, $\wedge$, $\vee$, $\longrightarrow$ etc. Here is how conjunction could be defined by pattern matching:

```
fun conj :: "bool ⇒ bool ⇒ bool" where
"conj True True = True" |
"conj _ _ = False"
```

Both the data type and function definitions roughly follow the syntax of functional programming languages.

3.1.2. Type `nat`

Natural numbers are another predefined data type:

```
datatype nat = 0 | Suc nat
```

with many predefined functions: $+$, $*$, $\leq$, etc. And this is how you could define your own addition:

```
fun add :: "nat ⇒ nat ⇒ nat" where
"add 0 n = n" |
"add (Suc m) n = Suc(add m n)"
```

And here is a proof of the fact that `add m 0 = m`:

```
lemma add_02: "add m 0 = m"
apply(induct m)
apply(auto)
done
```

The **`lemma`** command starts the proof and gives the lemma a name, `add_02`. Properties of recursively defined functions need to be established by induction in most cases. Command **`apply`**`(induct m)` instructs Isabelle to start a proof by induction on `m`. In response, it will show the following proof state:

```
1. add 0 0 = 0
2. ⋀m. add m 0 = m ⟹ add (Suc m) 0 = Suc m
```

The numbered lines are known as *subgoals*. The first subgoal is the base case, the second one the induction step. The prefix $\bigwedge m.$ is Isabelle's way of saying "for an arbitrary but fixed `m`". The $\Longrightarrow$ separates assumptions from the conclusion. The command **`apply`**`(auto)` instructs Isabelle to try and prove all subgoals automatically, essentially by simplifying them. Because both subgoals are easy, Isabelle can do it. The base case `add 0 0 = 0` holds by definition of `add`, and the induction step is almost as simple: `add (Suc m) 0 = Suc(add m 0) = Suc m` using first the definition of `add` and then the induction hypothesis. In summary, both proofs rely on simplification with function definitions and the induction hypothesis. The final **`done`** is like a "QED" (and would fail if there were unproved subgoals). As a result of that final **`done`**, Isabelle associates the lemma just proved with its name. You can now inspect the lemma with the command

```
thm add_02
```

which displays

```
add ?m 0 = ?m
```

The free variable `m` has been replaced by the *unknown* `?m`. There is no logical difference between the two but an operational one: unknowns can be instantiated, which is what you want after some lemma has been proved.

Terminology: We use *lemma*, *theorem* and *rule* interchangeably for propositions that have been proved.

Numerals (*0*, *1*, *2*, ...) and most of the standard arithmetic operations (*+*, *−*, ***, $\leq$, $<$ etc) are overloaded: they are available not just for natural numbers but for other types as well. For example, given the goal `x + 0 = x`, there is nothing to indicate that you are talking about natural numbers. Hence Isabelle can only infer that `x` is of some arbitrary type where `0` and `+` exist. As a consequence, you will be unable to prove the goal. To alert you to such pitfalls, Isabelle flags numerals without a fixed type in its output: `x + (0::'a) = x`. In this particular example, you need to include an explicit type constraint, for example `x+0 = (x::nat)`. If there is enough contextual information this may not be necessary: `Suc x = x` automatically implies `x::nat` because `Suc` is not overloaded.

3.1.3. Type `list`

Although lists are already predefined, we define our own copy just for demonstration purposes:

```
datatype 'a list = Nil | Cons "'a" "'a list"
```

Similarly for two standard functions, append and reverse:

```
fun app :: "'a list ⇒ 'a list ⇒ 'a list" where
"app Nil ys = ys" |
"app (Cons x xs) ys = Cons x (app xs ys)"

fun rev :: "'a list ⇒ 'a list" where
"rev Nil = Nil" |
"rev (Cons x xs) = app (rev xs) (Cons x Nil)"
```

Command **value** evaluates a term. For example,

```
value "rev(Cons True (Cons False Nil))"
```

yields the result `Cons False (Cons True Nil)`. This works symbolically, too:

```
value "rev(Cons a (Cons b Nil))"
```

yields `Cons b (Cons a Nil)`.

Proofs by induction on lists are essentially proofs over the length of the list, although the length remains implicit. To prove that some property `P` holds for all lists `xs`, i.e. `P xs`, you need to prove

1. the base case `P Nil` and
2. the inductive case `P (Cons x xs)` under the assumption `P xs`, for some arbitrary but fixed `xs`.

This is often called *structural induction*.

We will now demonstrate the typical proof process, which involves the formulation and proof of auxiliary lemmas.

3.1.4. Theorem `rev (rev xs) = xs`

Our goal is to show that reversing a list twice produces the original list.

```
theorem rev_rev [simp]: "rev(rev xs) = xs"
```

Commands **theorem** and **lemma** are interchangeable and merely indicate the importance we attach to a proposition. Via the bracketed attribute `simp` we also tell Isabelle to make the eventual theorem a *simplification rule*: future proofs involving simplification will replace occurrences of `rev (rev xs)` by `xs`. The proof is by induction:

```
apply(induct xs)
```

As explained above, we obtain two subgoals, namely the base case (`Nil`) and the induction step (`Cons`):

```
1. rev (rev Nil) = Nil
2. ⋀a xs. rev (rev xs) = xs ⟹ rev (rev (Cons a xs)) = Cons a xs
```

Let us try to solve both goals automatically:

```
apply(auto)
```

Subgoal 1 is proved, and disappears; the simplified version of subgoal 2 becomes the new subgoal 1:

```
1. ⋀a xs.
      rev (rev xs) = xs ⟹
      rev (app (rev xs) (Cons a Nil)) = Cons a xs
```

In order to simplify this subgoal further, a lemma suggests itself.

3.1.5. Lemma `rev (app xs ys) = app (rev ys) (rev xs)`

We insert the following lemma in front of the main theorem:

```
lemma rev_app [simp]: "rev(app xs ys) = app (rev ys) (rev xs)"
```

There are two variables that we could induct on: `xs` and `ys`. Because `app` is defined by recursion on the first argument, `xs` is the correct one:

```
apply(induct xs)
```

This time not even the base case is solved automatically:

```
apply(auto)
```

```
1. rev ys = app (rev ys) Nil
```

Again, we need to abandon this proof attempt and prove another simple lemma first.

3.1.6. Lemma `app xs Nil = xs`

We again try the canonical proof procedure:

```
lemma app_Nil2 [simp]: "app xs Nil = xs"
apply(induct xs)
apply(auto)
done
```

Thankfully, this worked. Now we can continue with our stuck proof attempt of the first lemma:

```
lemma rev_app [simp]: "rev(app xs ys) = app (rev ys) (rev xs)"
apply(induct xs)
apply(auto)
```

We find that this time `auto` solves the base case, but the induction step merely simplifies to

```
1. ⋀a xs.
      rev (app xs ys) = app (rev ys) (rev xs) ⟹
      app (app (rev ys) (rev xs)) (Cons a Nil) =
      app (rev ys) (app (rev xs) (Cons a Nil))
```

The the missing lemma is associativity of `app`, which we insert in front of the failed lemma `rev_app`.

3.1.7. Associativity of `app`

The canonical proof procedure succeeds without further ado:

```
lemma app_assoc [simp]: "app (app xs ys) zs = app xs (app ys zs)"
apply(induct xs)
apply(auto)
done
```

Finally the proofs of `rev_app` and `rev_rev` succeed, too.

3.1.8. Predefined lists

Isabelle's predefined lists are the same as the ones above, but with

- more syntactic sugar:
  - `[]` for `Nil`,
  - `x # xs` for `Cons x xs`,
  - `[`x_1`, ..., `x_n`]` for x_1 `# ... #` x_n `# []`, and
  - `xs @ ys` for `app xs ys`,
- and a large library of functions like `length`, `map`, `filter` etc.

3.1.9. Exercises

Exercise 3.1 Define your own addition, multiplication, and exponentiation functions on type `nat`. Prove as many of the standard equational rules as possible, e.g. associativity, commutativity and distributivity.

Exercise 3.2 Define your own sorting function on the predefined lists. Prove that the result is sorted and that every element occurs as many times in the output as in the input.

3.2. Type and function definitions

3.2.1. Type synonyms

Type synonyms are abbreviations for existing types, for example

```
type_synonym string = "char list"
```

Type synonyms are expanded after parsing and are not present in internal representation and output. They are mere conveniences for the reader.

3.2.2. Datatypes

The general form of a datatype definition looks like this:

$$\begin{array}{lcl} \texttt{datatype } (\texttt{'}a_1,\ldots,\texttt{'}a_n)t & = & C_1 \texttt{ "}\tau_{1,1}\texttt{"} \ldots \texttt{"}\tau_{1,n_1}\texttt{"} \\ & | & \ldots \\ & | & C_k \texttt{ "}\tau_{k,1}\texttt{"} \ldots \texttt{"}\tau_{k,n_k}\texttt{"} \end{array}$$

The double quotes are not not necessary for simple identifiers like `nat` or `'a`.

This definition introduces the constructors $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow$ `('`a_1`,...,'`a_n`)t` which obey

- *Distinctness:* $C_i \ldots \neq C_j \ldots$ if $i \neq j$
- *Injectivity:* $(C_i\, x_1 \ldots x_{n_i} = C_i\, y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

The datatype also comes with a structural induction rule: to show Px for all x of type `('`a_1`,...,'`a_n`)t`, one needs to show $P(C_i\, x_1 \ldots x_{n_i})$ (for each i) assuming $P(x_j)$ for all j where $\tau_{i,j} =$ `('`a_1`,...,'`a_n`)t`. Distinctness and injectivity are applied automatically by `auto` and other proof methods. Induction must be applied explicitly.

Datatypes values can be taken apart with case-expressions, for example

```
(case xs of [] ⇒ 0 | x # _ ⇒ Suc x)
```

just like in functional programming languages. Case expressions must be enclosed in parentheses.

As an example, consider binary trees:

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

with a mirror function:

```
fun mirror :: "'a tree ⇒ 'a tree" where
"mirror Tip = Tip" |
"mirror (Node l a r) = Node (mirror r) a (mirror l)"
```

The following lemma illustrates induction:

```
lemma "mirror(mirror t) = t"
apply(induct t)
```

yields

```
 1. mirror (mirror Tip) = Tip
 2. ⋀t1 a t2.
       ⟦mirror (mirror t1) = t1; mirror (mirror t2) = t2⟧
       ⟹ mirror (mirror (Node t1 a t2)) = Node t1 a t2
```

The induction step contains two induction hypotheses, one for each subtree. An application of `auto` finishes the proof.

3.2.3. Definitions

Non recursive functions can be defined as in the following example:

```
definition sq :: "nat ⇒ nat" where
"sq n = n*n"
```

Such definitions do not allow pattern matching but only $f\ x_1\ \ldots\ x_n = t$, where f does not occur in t.

3.2.4. Recursive functions

Recursive functions are defined with **fun** by pattern matching over datatype constructors. The order of equations matters. Just as in functional programming languages. However, all HOL functions must be total. This simplifies the logic—terms are always defined—but means that recursive functions must terminate. Otherwise one could define a function $f\ n = f\ n + 1$ and conclude $0 = 1$ by subtracting $f\ n$ on both sides.

Isabelle automatic termination checker requires that the arguments of recursive calls on the right-hand side must be strictly smaller than the arguments on the left-hand side. In the simplest case, this means that one fixed argument position decreases in size with each recursive call. The size is measured as the number of constructors (excluding 0-ary ones, e.g. `Nil`). Lexicographic combinations are also recognised. In more complicated situations, the user may have to prove termination by hand. For details see [2].

Functions defined with **fun** come with their own induction schema that mirrors the recursion schema and is derived from the termination order. For example,

```
fun div2 :: "nat ⇒ nat" where
"div2 0 = 0" |
"div2 (Suc 0) = Suc 0" |
"div2 (Suc(Suc n)) = Suc(div2 n)"
```

does not just define `div2` but also proves a customised induction rule:

$$\frac{P\ 0 \qquad P\ (Suc\ 0) \qquad \bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ (Suc\ n))}{P\ m}$$

This customised induction rule can simplify inductive proofs. For example,

```
lemma "div2(n+n) = n"
apply(induct n rule: div2.induct)
```

yields the 3 subgoals

```
1. div2 (0 + 0) = 0
2. div2 (Suc 0 + Suc 0) = Suc 0
3. ⋀n. div2 (n + n) = n ⟹
       div2 (Suc (Suc n) + Suc (Suc n)) = Suc (Suc n)
```

An application of `auto` finishes the proof. Had we used ordinary structural induction on `n`, the proof would have needed an additional case distinction in the induction step.

The general case is often called *computation induction*, because the induction follows the (terminating!) computation. For every defining equation

$f(e) = \ldots f(r_1) \ldots f(r_k) \ldots$

where $f(r_i)$, $i=1\ldots k$, are all the recursive calls, the induction rule `f.induct` contains one premise of the form

$P(r_1) \Longrightarrow \ldots \Longrightarrow P(r_k) \Longrightarrow P(e)$

If $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$ then `f.induct` is applied like this:

```
apply(induct x1 ... xn rule: f.induct)
```

where typically there is a call $f\ x_1 \ldots x_n$ in the goal. But note that the induction rule does not mention `f` at all, except in its name, and is applicable independently of `f`.

3.3. Induction heuristics

We have already noted that theorems about recursive functions are proved by induction. In case the function has more than one argument, we have followed the following heuristic in the proofs about the append function:

Perform induction on argument number i
if the function is defined by recursion on argument number i.

The key heuristic, and the main point of this section, is to *generalise the goal before induction*. The reason is simple: if the goal is too specific, the induction hypothesis is too weak to allow the induction step to go through. Let us illustrate the idea with an example.

Function `rev` has quadratic worst-case running time because it calls append for each element of the list and append is linear in its first argument. A linear time version of `rev` requires an extra argument where the result is accumulated gradually, using only `#`:

```
fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where
"itrev []     ys = ys" |
"itrev (x#xs) ys = itrev xs (x#ys)"
```

The behaviour of `itrev` is simple: it reverses its first argument by stacking its elements onto the second argument, and returning that second argument when the first one becomes empty. Note that `itrev` is tail-recursive: it can be compiled into a loop, no stack is necessary for executing it.

Naturally, we would like to show that `itrev` does indeed reverse its first argument provided the second one is empty:

```
lemma "itrev xs [] = rev xs"
```

There is no choice as to the induction variable:

```
apply(induct xs)
apply(auto)
```

Unfortunately, this attempt does not prove the induction step:

```
1. ⋀a xs. itrev xs [] = rev xs ⟹ itrev xs [a] = rev xs @ [a]
```

The induction hypothesis is too weak. The fixed argument, `[]`, prevents it from rewriting the conclusion. This example suggests a heuristic:

Generalise goals for induction by replacing constants by variables.

Of course one cannot do this naïvely: `itrev xs ys = rev xs` is just not true. The correct generalisation is

```
lemma "itrev xs ys = rev xs @ ys"
```

If `ys` is replaced by `[]`, the right-hand side simplifies to `rev xs`, as required. In this instance it was easy to guess the right generalisation. Other situations can require a good deal of creativity.

Although we now have two variables, only `xs` is suitable for induction, and we repeat our proof attempt. Unfortunately, we are still not there:

```
1. ⋀a xs.
       itrev xs ys = rev xs @ ys ⟹
       itrev xs (a # ys) = rev xs @ a # ys
```

The induction hypothesis is still too weak, but this time it takes no intuition to generalise: the problem is that the `ys` in the induction hypothesis is fixed, but the induction hypothesis needs to be applied with `a # ys` instead of `ys`. Hence we prove the theorem for all `ys` instead of a fixed one. We can instruct induction to perform this generalisation for us by adding `arbitrary: ys`.

```
apply(induct xs arbitrary: ys)
```

The induction hypothesis in the induction step is now universally quantified over `ys`:

```
1. ⋀ys. itrev [] ys = rev [] @ ys
2. ⋀a xs ys.
       (⋀ys. itrev xs ys = rev xs @ ys) ⟹
       itrev (a # xs) ys = rev (a # xs) @ ys
```

Thus the proof succeeds:

```
apply auto
done
```

This leads to another heuristic for generalisation:

> *Generalise induction by generalising all free variables*
> (except the induction variable itself).

Generalisation is best performed with `arbitrary:` $y_1 \ldots y_k$. This heuristic prevents trivial failures like the one above. However, it should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that need to be quantified are typically those that change in recursive calls.

3.4. Simplification

So far we have talked a lot about simplifying terms without explaining the concept. *Simplification* means

- using equations $l = r$ from left to right (only),
- as long as possible.

To emphasise the directionality, equations that have been given the `simp` attribute are called *simplification* rules. Logically, they are still symmetric, but proofs by simplification use them only in the left-to-right direction. The proof tool that performs simplifications is called the *simplifier*. It is the basis of `auto` and other related proof methods.

The idea of simplification is best explained by an example. Given the simplification rules

$$\texttt{0 + n} = \texttt{n} \quad (1)$$
$$\texttt{Suc m + n} = \texttt{Suc (m + n)} \quad (2)$$
$$\texttt{(Suc m} \leq \texttt{Suc n)} = \texttt{(m} \leq \texttt{n)} \quad (3)$$
$$\texttt{(0} \leq \texttt{m)} = \texttt{True} \quad (4)$$

the formula $\texttt{0 + Suc 0} \leq \texttt{Suc 0 + x}$ is simplified to `True` as follows:

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

Simplification is often also called *rewriting* and simplification rules *rewrite rules*.

3.4.1. Simplification rules

The attribute `simp` declares theorems to be simplification rules, which the simplifier will use automatically. In addition, **`datatype`** and **`fun`** commands implicitly declare some simplification rules: **`datatype`** the distinctness and injectivity rules, **`fun`** the defining

equations. Definitions are not declared as simplification rules automatically! Nearly any theorem can become a simplification rule. The simplifier will try to transform it into an equation. For example, the theorem `¬ P` is turned into `P = False`.

Only equations that really simplify, like `rev (rev xs) = xs` and `xs @ [] = xs`, should be declared as simplification rules. Equations that may be counterproductive as simplification rules should only be used in specific proof steps (see §3.4.4 below). Distributivity laws, for example, alter the structure of terms and can produce an exponential blow-up.

3.4.2. Conditional simplification rules

Simplification rules can be conditional. Before applying such a rule, the simplifier will first try to prove the preconditions, again by simplification. For example, given the simplification rules

```
p 0 = True
p x ⟹ f x = g x,
```

the term `f 0` simplifies to `g 0` but `f 1` does not simplify because `p 1` is not provable.

3.4.3. Termination

Simplification can run forever, for example if both `f x = g x` and `g x = f x` are simplification rules. It is the user's responsibility not to include simplification rules that can lead to nontermination, either on their own or in combination with other simplification rules. The right-hand side of a simplification rule should always be "simpler" than the left-hand side—in some sense. But since termination is undecidable, such a check cannot be automated completely and Isabelle makes little attempt to detect nontermination.

The case of conditional simplification rules is a bit more complicated. Because the preconditions are proved first, they also need to be simpler than the left-hand side of the conclusion. For example

```
n < m ⟹ (n < Suc m) = True
```

is suitable as a simplification rule: both `n < m` and `True` are simpler than `n < Suc m`. But

```
Suc n < m ⟹ (n < m) = True
```

leads to nontermination: when trying to rewrite `n < m` to `True` one first has to prove `Suc n < m`, which can be rewritten to `True` provided `Suc (Suc n) < m`, *ad infinitum*.

3.4.4. The `simp` proof method

So far we have only used the proof method `auto`. Method `simp` is the key component of `auto`, but `auto` can do much more. In some cases, `auto` is overeager and modifies the proof state too much. In such cases the more predictable `simp` method should be used. Given a goal

```
1. ⟦ P1; ...; Pm ⟧ ⟹ C
```

the command

```
apply(simp add: th1 ... thn)
```

simplifies the assumptions P_i and the conclusion C using

- all simplification rules, including the ones coming from **datatype** and **fun**,
- the additional lemmas th_1 ... th_n, and
- the assumptions.

In addition to or instead of *add* there is also *del* for removing simplification rules temporarily. Both are optional. Method *auto* can be modified similarly:

```
apply(auto simp add: ... simp del: ...)
```

Here the modifiers are *simp add* and *simp del* instead of just *add* and *del* because *auto* does not just perform simplification.

Note that *simp* acts only on subgoal 1, *auto* acts on all subgoals.

3.4.5. Rewriting with definitions

Definitions (**definition**) can be used as simplification rules, but by default they are not: the simplifier does not expand them automatically. Definitions are intended for introducing abstract concepts and not merely as abbreviations. Of course, we need to expand the definition initially, but once we have proved enough abstract properties of the new constant, we can forget its original definition. This style makes proofs more robust: if the definition has to be changed, only the proofs of the abstract properties will be affected.

The definition of a function *f* is a theorem named *f_def* and can be added to a call of *simp* just like any other theorem:

```
apply(simp add: f_def)
```

In particular, let-expressions can be unfolded by making *Let_def* a simplification rule.

3.4.6. Case splitting with simp

Goals containing if-expressions are automatically split into two cases by *simp* using the rule

```
P (if A then s else t) = ((A ⟶ P s) ∧ (¬ A ⟶ P t))
```

For example, *simp* can prove

```
(A ∧ B) = (if A then B else False)
```

because both *A* ⟶ *(A ∧ B) = B* and ¬ *A* ⟶ *(A ∧ B) = False* simplify to *True*.

We can split case-expressions similarly. For *nat* the rule looks like this:

```
P (case e of 0 ⇒ a | Suc n ⇒ b n) =
((e = 0 ⟶ P a) ∧ (∀n. e = Suc n ⟶ P (b n)))
```

Case expressions are not split automatically by *simp*, but *simp* can be instructed to do so:

```
apply(simp split: nat.split)
```

splits all case-expressions over natural numbers. For an arbitrary datatype *t* it is *t.split* instead of *nat.split*. Method *auto* can be modified in exactly the same way.

3.4.7. Exercises

Exercise 3.3 Define arithmetic expressions in one variable over integers (type `int`) as a data type:

```
datatype exp = Var | Const int | Add exp exp | Mul exp exp
```

Define a function `eval` that evaluates an expression at some point.

A polynomial can be represented as a list of coefficients, starting with the constant. For example, `[4, 2, -1, 3]` represents the polynomial $4 + 2x - x^2 + 3x^3$. Define a function `evalp` that evaluates a polynomial at some point. Define a function `coeffs :: exp ⇒ int list` that transforms an expression into a polynomial. This may require auxiliary functions. Prove that `coeffs` preserves the value of the expression: `evalp (coeffs e) x = eval e x`.

4. Logic and Proof Beyond Equality

4.1. Formulas

The basic syntax of formulas (*form* below) provides the standard logical constructs, in decreasing precedence:

$$\begin{aligned} form ::= {} & \texttt{(}form\texttt{)} \mid \texttt{True} \mid \texttt{False} \\ \mid {} & term = term \mid \neg form \mid form \wedge form \mid form \vee form \mid form \longrightarrow form \\ \mid {} & \forall x.\ form \mid \exists x.\ form \end{aligned}$$

Remember that formulas are simply terms of type `bool`. Hence = also works for formulas. Beware that = has a higher precedence than the other logical operators. Hence `s = t ∧ A` means `(s = t) ∧ A`. Logical equivalence can also be written with $\longleftrightarrow$ instead of =, where $\longleftrightarrow$ has the same low precedence as $\longrightarrow$.

Quantifiers need to be enclosed in parentheses if they are nested within other constructs (just like `if`, `case` and `let`).

The most frequent logical symbols have the following ASCII representations:

$\forall$	`\<forall>`	`ALL`
$\exists$	`\<exists>`	`EX`
λ	`\<lambda>`	`%`
$\longrightarrow$	`-->`	
$\longleftrightarrow$	`<-->`	
$\wedge$	`/\`	`&`
$\vee$	`\/`	`|`
$\neg$	`\<not>`	`~`
$\neq$	`\<noteq>`	`~=`

The first column shows the symbols, the second column ASCII representations that Isabelle interfaces convert into the corresponding symbol, and the third column shows ASCII representations that stay fixed.

The implication $\Longrightarrow$ is part of the Isabelle framework. It structures theorems and proof states, separating assumptions from conclusion. The implication $\longrightarrow$ is part of the logic HOL and can occur inside the formulas that make up the assumptions and conclusion. Theorems should be of the form $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$, not $A_1 \wedge \ldots \wedge A_n \longrightarrow A$. Both are logically equivalent but the first one works better when using the theorem in further proofs.

4.2. Sets

Sets are simply predicates, i.e. functions to `bool`:

```
type_synonym 'a set = "'a ⇒ bool"
```

Sets come with the usual notations:

- `{}`, `{`e_1`,...,`e_n`}`, `{x. P}`
- `e ∈ A`, `A ⊆ B`
- `A ∪ B`, `A ∩ B`, `A - B`, `- A`

and much more. Note that set comprehension is written `{x. P}` rather than `{x | P}` (to emphasise the variable binding nature of the construct). Here are the ASCII representations of the mathematical symbols:

∈	\<in>	:
⊆	\<subseteq>	<=
∪	\<union>	Un
∩	\<inter>	Int

Sets also allow bounded quantifications `∀x∈A. P` and `∃x∈A. P`.

4.3. Proof automation

So far we have only seen `simp` and `auto`: Both perform rewriting, both can also prove linear arithmetic facts (no multiplication), and `auto` is also able to prove simple logical or set-theoretic goals:

```
lemma "∀x. ∃y. x = y"
by auto

lemma "A ⊆ B ∩ C ⟹ A ⊆ B ∪ C"
by auto
```

where

by *proof-method*

is short for

apply *proof-method*
done

The key characteristics of both `simp` and `auto` are

- They show you were they got stuck, giving you an idea how to continue.

- They perform the obvious steps but are highly incomplete.

A proof method that is still incomplete but tries harder than `auto` is `fastsimp`. It either succeeds or fails, it acts on the first subgoal only, and it can be modified just like `auto`, e.g. with `simp add`. Here is a typical example of what `fastsimp` can do:

```
lemma "⟦ ∀xs ∈ A. ∃ys. xs = ys @ ys;  us ∈ A ⟧
   ⟹ ∃n. length us = n+n"
by fastsimp
```

This lemma is out of reach for `auto` because of the quantifiers. But `fastsimp` fails when the quantifier structure becomes more complicated. In that case `blast` is the method of choice. In the following example, `T` and `A` are two binary predicates, and it is shown that `T` is total, `A` is antisymmetric and `T` is a subset of `A`, then `A` is a subset of `T`:

```
lemma
  "⟦ ∀x y. T x y ∨ T y x;
     ∀x y. A x y ∧ A y x ⟶ x = y;
     ∀x y. T x y ⟶ A x y ⟧
   ⟹ ∀x y. A x y ⟶ T x y"
by blast
```

This is one of the rare cases where a not completely obvious theorem is proved automatically. Method `blast`

- is (in principle) a complete proof procedure for first-order formulas. In practice there is a search bound.
- does no rewriting and knows very little about equality.
- covers logic, sets and relations.
- either succeeds or fails.

Because of its strength in logic and sets and its weakness in equality reasoning, it complements the earlier proof methods.

4.4. Single step proofs

Although automation is nice, it often fails, at least initially, and you need to find out why. When `fastsimp` or `blast` simply fail, you have no clue why. At this point, the stepwise application of proof rules may be necessary. For example, if `blast` fails on `A ∧ B`, you want to attack the two conjuncts `A` and `B` separately. This can be achieved by applying *conjunction introduction*

$$\frac{?P \qquad ?Q}{?P \wedge ?Q}\ conjI$$

to the proof state. We will now examine the details of this process.

4.4.1. Instantiating unknowns

We had briefly mentioned earlier that after proving some theorem, Isabelle replaces all free variables `x` by so called *unknowns* `?x`. We can see this clearly in rule `conjI`. These unknowns can later be instantiated explicitly or implicitly:

- By hand, using `of`. The expression `conjI[of "a=b" "False"]` instantiates the unknowns in `conjI` from left to right with the two formulas `a=b` and `False`, yielding the rule

  $$\frac{a = b \qquad False}{a = b \wedge False}$$

  In general, *th*`[of` *string*$_1$ ... *string*$_n$`]` instantiates the unknowns in the theorem *th* from left to right with the terms *string*$_1$ to *string*$_n$.
- By unification. *Unification* is the process of making two terms syntactically equal by suitable instantiations of unknowns. For example, unifying `?P ∧ ?Q` with `a = b ∧ False` instantiates `?P` with `a = b` and `?Q` with `False`.

We need not instantiate all unknowns. If we want to skip a particular one we can just write _ instead, for example `conjI[of _ "False"]`. Unknowns can also be instantiated by name, for example `conjI[where ?P = "a=b" and ?Q = "False"]`.

4.4.2. Rule application

Rule application means applying a rule backwards to a proof state. For example, applying rule `conjI` to a proof state

```
1.  ...  ⟹ A ∧ B
```

results in two subgoals, one for each premise of `conjI`:

```
1.  ...  ⟹ A
2.  ...  ⟹ B
```

In general, the application of a rule ⟦ A_1; ...; A_n ⟧ ⟹ A to a subgoal ... ⟹ C proceeds in two steps:

1. Unify A and C, thus instantiating the unknowns in the rule.
2. Replace the subgoal C with n new subgoals A_1 to A_n.

This is the command to apply rule `xyz`:

```
apply(rule xyz)
```

This is also called *backchaining* with rule `xyz`.

4.4.3. Introduction rules

Conjunction introduction (`conjI`) is one example of a whole class of rules known as *introduction rules*. They explain under which premises some logical construct can be introduced. Here are some further useful introduction rules:

$$\frac{?P \Longrightarrow ?Q}{?P \longrightarrow ?Q}\ impI \qquad \frac{\bigwedge x.\ ?P\ x}{\forall x.\ ?P\ x}\ allI$$

$$\frac{?P \Longrightarrow ?Q \qquad ?Q \Longrightarrow ?P}{?P = ?Q}\ iffI$$

These rules are part of the logical system of *natural deduction* (e.g. [1]). Although we intentionally de-emphasise the basic rules of logic in favour of automatic proof methods

that allow you to take bigger steps, these rules are helpful in locating where and why automation fails. When applied backwards, these rules decompose the goal:

- *conjI* and *iffI* split the goal into two subgoals,
- *impI* moves a formula (*?P*) into the list of assumptions,
- and *allI* removes a $\forall$ by turning the quantified variable into a fixed local variable of the subgoal.

Isabelle knows about these and a number of other introduction rules. The command

```
apply rule
```

automatically selects the appropriate rule for the current subgoal.

You can also turn your own theorems into introduction rules by giving them them *intro* attribute, analogous to the *simp* attribute. In that case *blast*, *fastsimp* and (to a limited extent) *auto* will automatically backchain with those theorems. The *intro* attribute should be used with care because it increases the search space and can lead to nontermination. Sometimes it is better to use it only in a particular calls of *blast* and friends. For example, *le_trans*, transitivity of $\leq$ on type *nat*, is not an introduction rule by default because of the disastrous effect on the search space, but can be useful in specific situations:

```
lemma "⟦ (a::nat) ≤ b; b ≤ c; c ≤ d; d ≤ e ⟧ ⟹ a ≤ e"
by(blast intro: le_trans)
```

4.4.4. Forward proof

Forward proof means deriving new theorems from old theorems. We have already seen a very simple form of forward proof: the *of* operator for instantiating unknowns in a theorem. The big brother of *of* is *OF* for applying one theorem to others. Given a theorem *A* $\Longrightarrow$ *B* called *r* and a theorem *A'* called *r'*, the theorem *r[OF r']* is the result of applying *r* to *r'*, where *r* should be viewed as a function taking a theorem *A* and returning *B*. More precisely, *A* and *A'* are unified, thus instantiating the unknowns in *B*, and the result is the instantiated *B*. Of course, unification may also fail.

Application of rules to other rules operates in the forward direction: from the premises to the conclusion of the rule; application of rules to proof states operates in the backward direction, from the conclusion to the premises.

In general *r* can be of the form ⟦ A_1; ...; A_n ⟧ $\Longrightarrow$ *A* and there can be multiple argument theorems r_1 to r_m (with $m \leq n$), in which case *r[OF* r_1 ... r_m*]* is obtained by unifying and thus proving A_i with r_i, $i = 1 \ldots m$. Here is an example, where *refl* is the theorem *?t = ?t*:

```
thm conjI[OF refl[of "a"] refl[of "b"]]
```

yields the theorem *a = a* $\wedge$ *b = b*. The command **thm** merely displays the result.

Forward reasoning does also make sense in connection with proof states. Therefore *blast*, *fastsimp* and *auto* support a modifier *dest* which instructs the proof method to use certain rules in a forward fashion. If *r* is of the form *A* $\Longrightarrow$ *B*, the modifier

`dest: r` allows proof search to reason forward with `r`, i.e. to replace an assumption `A'`, where `A'` unifies with `A`, with the correspondingly instantiated `B`. For example, `Suc_leD` is the theorem `Suc m ≤ n ⟹ m ≤ n`, which works well for forward reasoning:

```
lemma "Suc(Suc(Suc a)) ≤ b ⟹ a ≤ b"
by(blast dest: Suc_leD)
```

In this particular example we could have backchained with `Suc_leD`, too, but because the premise is more complicated than the conclusion this can easily lead to nontermination.

To ease readability we will drop the question marks in front of unknowns from now on.

5. Inductive definitions

Here is a simple example of an inductively defined predicate:

- 0 is even
- If n is even, so is $n + 2$.

The phrase "inductive" implies that these are the only even numbers. In Isabelle you write

```
inductive ev :: "nat ⇒ bool" where
"ev 0" |
"ev n ⟹ ev (n + 2)"
```

To get used to inductive definitions, we will first prove a few properties of `ev` informally before we descend to the Isabelle level.

How do we prove that some number is even, e.g. `ev 4`? Simply by combining the defining rules for `ev`:

```
ev 0 ⟹ ev (0 + 2) ⟹ ev((0 + 2) + 2) = ev 4
```

Showing that all even numbers have some property is more complicated. For example, let us prove that the doubling an even number again yields an even number: `ev m ⟹ ev (m + m)`. This requires a proof by induction on the length or structure of the derivation of `ev m`.

Base case `ev m` is proved by rule `ev 0`:
`⟹ m = 0 ⟹ ev (m + m)`

Induction step `ev m` is proved by rule `ev n ⟹ ev (n + 2)`:
`⟹ m = n + 2` and by induction hypothesis `ev (n + n)`
`⟹ m + m = (n + 2) + (n + 2) = ((n + n) + 2) + 2`
`⟹ ev (m + m)`

What we have just seen is a special case of *rule induction*. Rule induction applies to propositions of this form

```
ev n ⟹ P n
```

That is, we want to prove a property `P n` for all even `n`. But if we assume `ev n`, then there must be some derivation of this assumption using the two defining rules for `ev`. That is, we must prove

Base case `P 0`
Induction step `P n ⟹ P (n + 2)`

The corresponding rule is called `ev.induct` and looks like this:

$$\frac{ev\ n \quad P\ 0 \quad \bigwedge n.\ [\![ev\ n;\ P\ n]\!] \Longrightarrow P\ (n + 2)}{P\ n}$$

Inductive definitions have the following general form:

inductive `I :: "τ ⇒ bool"` **where**

followed by a sequence of (possibly named) rules of the form

$[\![I\ a_1; \ldots; I\ a_n]\!] \Longrightarrow I\ a$

separated by `|`. As usual, `n` can be 0. The corresponding rule induction principle `I.induct` applies to propositions of the form

$I\ x \Longrightarrow P\ x$

Proving such a proposition by rule induction means proving for every rule that `P` is invariant:

$[\![P\ a_1; \ldots; P\ a_n]\!] \Longrightarrow P\ a$

The above format for inductive definitions is simplified in a number of respects. `I` can have any number of arguments and each rule can have additional premises not involving `I`, so-called *side conditions*.

6. Isar: A Language for Structured Proofs

Apply-scripts are unreadable and hard to maintain. The language of choice for larger proofs is *Isar*. The two key features of Isar are:

- It is structured, not linear.
- It is readable without running it because you need to state what you are proving at any given point.

Whereas apply-scripts are like assembly language programs, Isar proofs are like structured programs with comments. A typical Isar proof looks like this:

proof
  assume "$formula_0$"
  have "$formula_1$" **by** `simp`
  ⋮
  have "$formula_n$" **by** `blast`
  show "$formula_{n+1}$" **by** ...
qed

It proves $formula_0 \Longrightarrow formula_{n+1}$ (provided provided each proof step succeeds). The intermediate **have** statements are merely stepping stones on the way towards the **show** statement that proves the actual goal. In more detail, this is the Isar core syntax:

proof = **by** *method*
| **proof** [*method*] *step** **qed**

step = **fix** *variables*
| **assume** *proposition*
| [**from** *fact*$^+$] (**have** | **show**) *proposition proof*

proposition = [*name*:] *"formula"*

fact = *name* | ...

A proof can either be an atomic **by** with a single proof method which must finish off the statement being proved, for example `auto`. Or it can be a **proof**–**qed** block of multiple steps. Such a block can optionally begin with a proof method that indicates how to start off the proof, e.g. `(induct xs)`.

A step either assumes a proposition or states a proposition together with its proof. The optional **from** clause indicates which facts are to be used in the proof. Intermediate propositions are stated with **have**, the overall goal with **show**. A step can also introduce new local variables with **fix**. Logically, **fix** introduces $\bigwedge$-quantified variables, **assume** introduces the assumption of an implication ($\Longrightarrow$) and **have**/**show** the conclusion.

Propositions are optionally named formulas. These names can be referred to in later **from** clauses. In the simplest case, a fact is such a name. But facts can also be composed with `OF` and `of` as shown in §4.4.4—hence the ... in the above grammar. Note that assumptions, intermediate **have** statements and global lemmas all have the same status and are thus collectively referred to as *facts*.

Fact names can stand for whole lists of facts. For example, if `f` is defined by command **fun**, `f.simps` refers to the whole list of recursion equations defining `f`. Individual facts can be selected by writing `f.simps(2)`, whole sublists by `f.simps(2-4)`.

6.1. Isar by example

We show a number of proofs of Cantors theorem that a function from a set to its powerset cannot be surjective, illustrating various features of Isar. The constant `surj` is predefined.

```
lemma "¬ surj(f :: 'a ⇒ 'a set)"
proof
  assume 0: "surj f"
  from 0 have 1: "∀A. ∃a. A = f a" by(simp add: surj_def)
  from 1 have 2: "∃a. {x. x ∉ f x} = f a" by blast
  from 2 show "False" by blast
qed
```

The **proof** command lacks an explicit method how to perform the proof. In such cases Isabelle tries to use some standard introduction rule, in the above case for ¬:

$$\frac{P \Longrightarrow False}{\neg\ P}$$

In order to prove `¬ P`, assume `P` and show `False`. Thus we may assume `surj f`. The proof shows that names of propositions may be (single!) digits—meaningful names are hard to invent and are often not necessary. Both **`have`** steps are obvious. The second one introduces the diagonal set `{x. x ∉ f x}`, the key idea in the proof. If you wonder why `2` directly implies `False`: from `2` it follows that `(a ∉ f a) = (a ∈ f a)`.

6.1.1. `this`, `then`, `hence` *and* `thus`

Labels should be avoided. They interrupt the flow of the reader who has to scan the context for the point where the label was introduced. Ideally, the proof is a linear flow, where the output of one step becomes the input of the next step, piping the previously proved fact into the next proof, just like in a UNIX pipe. In such cases the predefined name `this` can be used to refer to the proposition proved in the previous step. This allows us to eliminate all labels from our proof (we suppress the **`lemma`** statement):

```
proof
  assume "surj f"
  from this have "∃a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
  from this show "False" by blast
qed
```

We have also taken the opportunity to compress the two **`have`** steps into one.

To compact the text further, Isar has a few convenient abbreviations:

then	=	**from** *this*
thus	=	**then show**
hence	=	**then have**

With the help of these abbreviations the proof becomes

```
proof
  assume "surj f"
  hence "∃a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
  thus "False" by blast
qed
```

There are two further linguistic variations:

(**have**|**show**) *prop* **using** *facts*	=	**from** *facts* (**have**|**show**) *prop*
with *facts*	=	**from** *facts* *this*

The **`using`** idiom de-emphasises the used facts by moving them behind the proposition.

6.1.2. Structured lemma statements: **fixes**, **assumes**, **shows**

Lemmas can also be stated in a more structured fashion. To demonstrate this feature with Cantor's theorem, we rephrase ¬ *surj f* a little:

```
lemma
  fixes f :: "'a ⇒ 'a set"
  assumes s: "surj f"
  shows "False"
```

The optional **fixes** part allows you to state the types of variables up front rather than by decorating one of their occurrences in the formula with a type constraint. The key advantage of the structured format is the **assumes** part that allows you to name each assumption. The **shows** part gives the goal. The actual theorem that will come out of the proof is *surj f* ⟹ *False*, but during the proof the assumption *surj f* is available under the name *s* like any other fact.

```
proof -
  have "∃ a. {x. x ∉ f x} = f a" using s
    by(auto simp: surj_def)
  thus "False" by blast
qed
```

In the **have** step the assumption *surj f* is now referenced by its name *s*. The duplication of *surj f* in the above proofs (once in the statement of the lemma, once in its proof) has been eliminated.

Note the dash after the **proof** command. It is the null method that does nothing to the goal. Leaving it out would ask Isabelle to try some suitable introduction rule on the goal *False*—but there is no suitable introduction rule and **proof** would fail.

Stating a lemmas with **assumes**-**shows** implicitly introduces the name *assms* that stands for the list of all assumptions. You can refer to individual assumptions by *assms(1)*, *assms(2)* etc, thus obviating the need to name them individually.

6.2. Proof patterns

We show a number of important basic proof patterns. Many of them arise from the rules of natural deduction that are applied by **proof** by default. The patterns are phrased in terms of **show** but work for **have** and **lemma**, too.

We start with two forms of *case distinction*: starting from a formula *P* we have the two cases *P* and ¬ *P*, and starting from a fact *P* ∨ *Q* we have the two cases *P* and *Q*:

```
show "R"                        have "P ∨ Q"  ...
proof cases                     then show "R"
  assume "P"                    proof
  ⋮                               assume "P"
  show "R"  ...                   ⋮
next                              show "R"  ...
  assume "¬ P"                  next
  ⋮                               assume "Q"
  show "R"  ...                   ⋮
qed                               show "R"  ...
                                qed
```

How to prove a logical equivalence:

```
show "P ⟷ Q"
proof
  assume "P"
  ⋮
  show "Q"  ...
next
  assume "Q"
  ⋮
  show "P"  ...
qed
```

Proofs by contradiction:

```
show "¬ P"                      show "P"
proof                           proof (rule ccontr)
  assume "P"                      assume "¬P"
  ⋮                               ⋮
  show "False"  ...               show "False"  ...
qed                             qed
```

The name `ccontr` stands for "classical contradiction".

How to prove quantified formulas:

```
show "∀x. P(x)"                 show "∃x. P(x)"
proof                           proof
  fix x                           ⋮
  ⋮                               show "P(witness)"  ...
  show "P(x)"  ...              qed
qed
```

In the proof of `ALL x. P(x)`, the step **fix** `x` introduces a locale fixed variable `x` into the subproof, the proverbial "arbitrary but fixed value". Instead of `x` we could have chosen any name in the subproof. In the proof of `EX x. P(x)`, `witness` is some arbitrary term for which we can prove that it satisfies `P`.

How to reason forward from `EX x. P(x)`:

```
have "∃x. P(x)" ...
then obtain x where p: "P(x)" by blast
```

After the **obtain** step, x (we could have chosen any name) is a fixed local variable, and p is the name of the fact $P(x)$. This pattern works for one or more x. As an example of the **obtain** command, here is the proof of Cantor's theorem in more detail:

```
lemma "¬ surj(f :: 'a ⇒ 'a set)"
proof
  assume "surj f"
  hence  "∃a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
  then obtain a where  "{x. x ∉ f x} = f a"  by blast
  hence  "a ∉ f a ⟷ a ∈ f a"  by blast
  thus "False" by blast
qed
```

How to prove set equality and subset relationship:

```
show "A = B"
proof
  show "A ⊆ B"  ...
next
  show "B ⊆ A"  ...
qed
```

```
show "A ⊆ B"
proof
  fix x
  assume "x ∈ A"
  ⋮
  show "x ∈ B"  ...
qed
```

6.3. Streamlining proofs

6.3.1. Pattern matching and quotations

In the proof patterns shown above, formulas are often duplicated. This can make the text harder to read, write and maintain. Pattern matching is an abbreviation mechanism to avoid such duplication. Writing

show *formula* (**is** *pattern*)

matches the pattern against the formula, thus instantiating the unknowns in the pattern for later use. As an example, consider the proof pattern for $\longleftrightarrow$:

```
show "formula₁ ⟷ formula₂" (is "?L ⟷ ?R")
proof
  assume "?L"
  ⋮
  show "?R"  ...
next
  assume "?R"
  ⋮
  show "?L"  ...
qed
```

Instead of duplicating *formula*$_i$ in the text, we introduce the two abbreviations *?L* and *?R* by pattern matching. Pattern matching works wherever a formula is stated, in particular with **have** and **lemma**.

The unknown *?thesis* is implicitly matched against any goal stated by **lemma** or **show**. Here is a typical example:

```
lemma "formula"
proof -
  ⋮
  show ?thesis  ...
qed
```

Unknowns can also be instantiated with **let** commands

```
let ?t = "some-big-term"
```

Later proof steps can refer to *?t*:

```
have "... ?t ..."
```

Names of facts are introduced with *name:* and refer to proved theorems. Unknowns *?X* refer to terms or formulas.

Although abbreviations shorten the text, the reader needs to remember what they stand for. Similarly for names of facts. Names like *1*, *2* and *3* are not helpful and should only be used in short proofs. For longer proof, descriptive names are better. But look at this example:

```
have x_gr_0: "x > 0"
⋮
from x_gr_0 ...
```

The name is longer than the fact it stands for! Short facts do not need names, one can refer to them easily by quoting them:

```
have "x > 0"
⋮
from `x>0` ...
```

Note that the quotes around *x>0* are *back quotes*. They refer to the fact not by name but by value.

6.3.2. **moreover**

Sometimes one needs a number of facts to enable some deduction. Of course one can name these facts individually, as shown on the right, but one can also combine them with **moreover**, as shown on the left:

```
have "P1"  ...                    have lab1: "P1"  ...
moreover have "P2"  ...           have lab2: "P2"  ...
moreover                          ⋮
⋮                                 have labn: "Pn"  ...
moreover have "Pn"  ...           from lab1 lab2 ...
ultimately have "P"   ...         have "P"   ...
```

The **moreover** version is no shorter but expresses the structure more clearly and avoids new names.

6.3.3. Raw proof blocks

Sometimes one would like to prove some lemma locally with in a proof. A lemma that shares the current context of assumptions but that has its own assumptions and is generalised over its locally fixed variables at the end. This is what a *raw proof block* does:

```
{ fix x1 ... xn
  assume A1 ... Am
  ⋮
  have B
}
```

proves $[\![A_1; \ldots; A_m]\!] \Longrightarrow B$ where all x_i have been replaced by unknowns $?x_i$.

The conclusion of a raw proof block is *not* indicated by **show** but is simply the final **have**.

As an example we prove a simple fact about divisibility on integers. The definition of *dvd* is `(b dvd a) = (∃k. a = b * k)`.

```
lemma fixes a b :: int assumes "b dvd (a+b)" shows "b dvd a"
proof -
  { fix k assume k: "a+b = b*k"
    have "∃k'. a = b*k'"
    proof
      show "a = b*(k - 1)" using k by(simp add: algebra_simps)
    qed }
  then show ?thesis using assms by(auto simp add: dvd_def)
qed
```

Note that the result of a raw proof block has no name. In this example it was directly piped (via **then**) into the final proof, but it can also be named for later reference: you simply follow the block directly by a **note** command:

```
note name = this
```

This introduces a new name *name* that refers to *this*, the fact just proved, in this case the preceding block. In general, **note** introduces a new name for one or more facts.

6.4. Case distinction and induction

6.4.1. Datatype case distinction

We have seen case distinction on formulas. Now we want to distinguish which form some term takes: is it `0` or of the form `Suc n`, is it `[]` or of the form `x # xs`, etc. Here is a typical example proof by case distinction on the form of `xs`:

```
lemma "length(tl xs) = length xs - 1"
proof (cases xs)
  assume "xs = []"
  thus ?thesis by simp
next
  fix y ys assume "xs = y#ys"
  thus ?thesis by simp
qed
```

Function `tl` ("tail") is defined by `tl [] = []` and `tl (x # xs) = xs`. Note that the result type of `length` is `nat` and `0 - 1 = 0`.

This proof pattern works for any term `t` whose type is a datatype. The goal has to be proved for each constructor `C`:

```
fix x1 ... xn assume "t = C x1 ... xn"
```

Each case can be written in a more compact form by means of the **`case`** command:

```
case (C x1 ... xn)
```

This is equivalent to the explicit **`fix`**-**`assume`**n line but also gives the assumption `"t = C x1 ... xn"` a name: `C`, like the constructor. Here is the **`case`** version of the proof above:

```
proof (cases xs)
  case Nil
  thus ?thesis by simp
next
  case (Cons y ys)
  thus ?thesis by simp
qed
```

Remember that `Nil` and `Cons` are the alphanumeric names for `[]` and `#`. The names of the assumptions are not used because they are directly piped (via **`thus`**) into the proof of the claim.

6.4.2. Structural induction

We illustrate structural induction with an example based on natural numbers: the sum ($\sum$) of the first `n` natural numbers (`{0..n::nat}`) is equal to `n * (n + 1) div 2`. Never mind the details, just focus on the pattern:

```
lemma "∑{0..n::nat} = n*(n+1) div 2" (is "?P n")
proof (induct n)
```

```
  show "∑{0..0::nat} = 0*(0+1) div 2" by simp
next
  fix n assume "∑{0..n::nat} = n*(n+1) div 2"
  thus "∑{0..Suc n::nat} = Suc n*(Suc n+1) div 2" by simp
qed
```

Except for the rewrite steps, everything is explicitly given. This makes the proof easily readable, but the duplication means it is tedious to write and maintain. Here is how pattern matching can completely avoid any duplication:

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

The first line introduces an abbreviation `?P n` for the goal. This make `?P` a function, and pattern matching instantiates `?P` to λ`n. ∑{0..n} = n * (n + 1) div 2`. Now the proposition to be proved in the base case can be written as `?P 0`, the induction hypothesis as `?P n`, and the conclusion of the induction step as `?P(Suc n)`.

Induction also provides the **case** idiom that abbreviates the **fix**-**assume** step. The above proof becomes

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

The unknown `?case` is set in each case to the required claim, i.e. `?P 0` and `?P(Suc n)` in the above proof, without requiring the user to define a `?P`. The general pattern for induction over `nat` is this:

```
show "P(n)"
proof (induct n)
  case 0                        let ?case = "P(0)"
  ⋮
  show ?case ...
next
  case (Suc n)                  fix n assume Suc: "P(n)"
  ⋮                             let ?case = "P(Suc n)"
  show ?case ...
qed
```

On the right side you can see what the **case** command on the left stands for.

In case the goal is an implication, induction does one more thing: the proposition to be proved in each case is not the whole implication but only its conclusion; the premises of the implication are immediately made assumptions of that case. That is, if in the above proof we replace **show** *"P(n)"* by **show** *"A(n)* $\Longrightarrow$ *P(n)"* then **case** *0* stands for

```
assume 0: "A(0)"
let ?case = "P(0)"
```

and **case** *(Suc n)* stands for

```
fix n
assume Suc:  "A(n) ⟹ P(n)"  "A(Suc n)"
let ?case = "P(Suc n)"
```

The list of assumptions *Suc* is actually subdivided into *Suc.hyps*, the induction hypotheses (here *A(n)* $\Longrightarrow$ *P(n)*) and *Suc.prems*, the premises of the goal being proved (here *A(Suc n)*).

Induction works for any datatype. In general, trying to prove a goal $[\![A_1(x); \ldots; A_k(x)]\!] \Longrightarrow P(x)$ by induction on x generates a proof obligation for each constructor C of the datatype. The command *case (C* $x_1 \ldots x_n$*)* performs the following steps:

1. **fix** $x_1 \ldots x_n$
2. **assume** the induction hypotheses (calling them *C.hyps*) and the premises $A_i(C\ x_1 \ldots x_n)$ (calling them *C.prems*) and calling the whole list *C*
3. **let** *?case = "P(C* $x_1 \ldots x_n$*)"*

6.4.3. Rule induction

Remember the inductive definition of even numbers in §5:

```
inductive ev :: "nat ⇒ bool" where
ev0: "ev 0" |
evSS: "ev n ⟹ ev(Suc(Suc n))"
```

We had given an informal proof of *ev n* $\Longrightarrow$ *ev (n + n)*. On the left-hand side is the structured counterpart.

```
lemma "ev n ⟹ ev(n+n)"
proof(induct rule: ev.induct)
  case ev0                              let ?case = "ev(0+0)"
  show ?case
    by(simp add: ev.ev0)
next
  case evSS                             fix n
                                        assume evSS: "ev n" "ev(n+n)"
                                        let ?case =
                                          "ev(Suc(Suc n)+Suc(Suc n))"
  thus ?case
    by(simp add: ev.evSS)
qed
```

The proof resembles structural induction, but the induction rule is given explicitly and the names of the cases are the names of the rules in the inductive definition. On the right-

hand side you can see the implicit effect of the two **case** commands. Let us examine the two assumptions named *evSS*: *ev n* is the premise of rule *evSS*, which we may assume because we are in the case where that rule was used; *ev (n + n)* is the induction hypothesis.


Because each **case** command introduces a list of assumptions named like the case name, which is the name of a rule of the inductive definition, those rules now need to be accessed with a qualified name, here *ev.ev0* and *ev.evSS*

In the case *evSS* of the proof above we have pretended that the system fixes a variable *n*. But unless the user provides the name *n*, the system will just invent its own name that cannot be referred to. In the above proof, we do not need to refer to it, hence we do not give it a specific name. In case one needs to refer to it one writes

```
case (evSS m)
```

just like **case** *(Suc n)* in earlier structural inductions. The name *m* is an arbitrary choice. As a result, case *evSS* is derived from a renamed version of rule *evSS*: *ev m* $\Longrightarrow$ *ev(m+m)*. Here is an example with an intermediate step that refers to *m*:

```
lemma "ev n ⟹ ev(n+n)"
proof(induct rule: ev.induct)
  case ev0 show ?case by(simp add: ev.ev0)
next
  case (evSS m)
  have "ev(Suc(Suc(Suc(Suc(m+m)))))"
    using `ev(m+m)` by(simp add: ev.evSS)
  thus ?case by(simp)
qed
```

In general, let *I* be an (for simplicity unary) inductively defined predicate and let the rules in the definition of *I* be called $rule_1, \ldots, rule_n$. A proof by rule induction follows this pattern:

```
show "I x ⟹ P x"
proof(induct rule: I.induct)
  case rule₁
  ⋮
  show ?case ...
next
⋮
next
  case ruleₙ
  ⋮
  show ?case ...
qed
```

One may need to provide explicit variable names by writing **case** $(rule_i\ x_1 \ldots x_k)$, thus renaming the first k free variables in rule i to $x_1 \ldots x_k$, going through rule i from left to right.

6.4.4. Assumption naming

In any induction, **case** *name* sets up a list of assumptions also called *name* which is subdivided into two parts:

name.prems contains the (suitably instantiated) premises of the statement being proved;
name.hyps contains all the hypotheses of this case in the induction rule. For structural inductions these are merely the induction hypotheses. For rule inductions these are a mixture of the hypotheses of rule *name* and the induction hypotheses.

More complicated inductive proofs than the ones we have seen so far often need to refer to specific assumptions—just *name* or even *name.prems* and *name.hyps* can be too unspecific. This is where the indexing of fact lists comes in handy, e.g. *name(3)* or *name.prems(1-2)*.

6.4.5. Exercises

Exercise 6.1 Define a recursive function *elems :: "'a list ⇒ 'a set"* and prove

x ∈ elems xs ⟹ ∃ys zs. xs = ys @ x # zs ∧ x ∉ elems ys

Exercise 6.2 A context-free grammar can be seen as an inductive definition where each nonterminal A is an inductively defined predicate on lists of terminal symbols: $A(w)$ mans that w is in the language generated by A. For example, the production $S \to aSb$ can be viewed as the clause *S w ⟹ S (a # w @ [b])* where *a* and *b* are constructors of some datatype of terminal symbols: **datatype** *tsymbs = a | b | ...*

Define the two grammars

$$\begin{array}{lcl} S & \to & \varepsilon \mid a\,S\,b \mid S\,S \\ T & \to & \varepsilon \mid T\,a\,T\,b \end{array}$$

(ε is the empty word) as two inductive predicates and prove *S w = T w*.

References

[1] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2004.

[2] A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. http://isabelle.in.tum.de/doc/functions.pdf.

[3] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002.