Tobias Nipkow

Programming and Proving in Isabelle/HOL

April 13, 2012

Contents

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.

Chapter 2 introduces HOL as a functional programming language and explains how to write simple inductive proofs of mostly equational properties of recursive functions. Chapter 3 introduces the rest of HOL: the language of formulas beyond equality, automatic proof tools, single step proofs, and inductive definitions, an essential specification construct. Chapter 4 introduces Isar, Isabelle's language for writing structured proofs.

This introduction to the core of Isabelle is intentionally concrete and example-based: we concentrate on examples that illustrate the typical cases; we do not explain the general case if it can be inferred from the examples. For a comprehensive treatment of all things Isabelle we recommend the *Isabelle/Isar Reference Manual* [?], which comes with the Isabelle distribution. The tutorial by Nipkow, Paulson and Wenzel [?] (in its updated version that comes with the Isabelle distribution) is still recommended for the wealth of examples and material, but its proof style is outdated. In particular it fails to cover the structured proof language Isar.

2

Programming and Proving

This chapter introduces HOL as a functional programming language and shows how to prove properties of functional programs by induction.

2.1 Basics

2.1.1 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 ($\mathbb{N}$), and *int*, the type of mathematical integers ($\mathbb{Z}$).
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 $t :: \tau$ to mean that term t has type τ.

! There are many predefined infix symbols like $+$ and $\leqslant$. 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:

$(if\ b\ then\ t_1\ \ else\ t_2)$
$(let\ x = t\ in\ u)$
$(case\ t\ of\ pat_1 \Rightarrow t_1 \mid \ldots \mid pat_n \Rightarrow t_n)$

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

Terms may also contain λ-abstractions. For example, $\lambda x.\ x$ 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 $'a \Rightarrow\ 'a \Rightarrow$ *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 $t :: \tau$ as in $m < (n::nat)$. Type constraints may be needed to disambiguate terms involving overloaded functions such as $+$, $*$ and $\leqslant$.

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, the formula $A_1 \Longrightarrow A_2 \Longrightarrow A_3$ means $A_1 \Longrightarrow (A_2 \Longrightarrow A_3)$. The (Isabelle specific) 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: $\dfrac{A_1 \quad \ldots \quad A_n}{A}$

2.1.2 Theories

Roughly speaking, a **theory** is a named collection of types, functions, and theorems, much like a module in a programming language. All the Isabelle text that you ever type needs to go into a theory. The general format of a theory T is

theory T
imports $T_1 \ldots T_n$
begin
definitions, theorems and proofs
end

where $T_1 \ldots T_n$ 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 `http://isabelle.in.tum.de/library/HOL/`) there is also the *Archive of Formal Proofs* at `http://afp.sourceforge.net`, a growing collection of Isabelle theories that everybody can contribute to.

2.1.3 Quotation Marks

The textual definition of a theory follows a fixed syntax with keywords like **begin** and **datatype**. Embedded in this syntax are the types and formulae of HOL. To distinguish the two levels, everything HOL-specific (terms and types) must be enclosed in quotation marks: "...". To lessen this burden, quotation marks around a single identifier can be dropped. When Isabelle prints a syntax error message, it refers to the HOL syntax as the **inner syntax** and the enclosing theory language as the **outer syntax**.

2.2 Types *bool*, *nat* and *list*

These are the most important predefined types. We go through them one by one. Based on examples we learn how to define (possibly recursive) functions and prove theorems about them by induction and simplification.

2.2.1 Type *bool*

The type of boolean values is a predefined datatype

datatype *bool* = *True* | *False*

with the two values *True* and *False* and with many predefined functions: $\neg$, $\wedge$, $\vee$, $\longrightarrow$ etc. Here is how conjunction could be defined by pattern matching:

fun *conj* :: "$bool \Rightarrow bool \Rightarrow bool$" **where**
"*conj True True* = *True*" |
"*conj* _ _ = *False*"

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

2.2.2 Type *nat*

Natural numbers are another predefined datatype:

datatype $nat = 0 \mid Suc\ nat$

All values of type *nat* are generated by the constructors 0 and *Suc*. Thus the values of type *nat* are 0, *Suc* 0, *Suc* (*Suc* 0) etc. There are many predefined functions: $+$, $*$, $\leqslant$, etc. Here is how you could define your own addition:

fun *add* :: "$nat \Rightarrow nat \Rightarrow nat$" **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(*induction 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**(*induction 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. $\bigwedge m.\ add\ m\ 0 = m \Longrightarrow add\ (Suc\ m)\ 0 = Suc\ m$

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 subproofs rely on simplification with function definitions and the induction hypothesis. 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.

Note that there is also a proof method *induct*, which behaves almost like *induction*; the difference is explained in Chapter 4.

! 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 ($+$, $-$, $*$, $\leqslant$, $<$ 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.

An informal proof

Above we gave some terse informal explanation of the proof of $add\ m\ 0 = m$. A more detailed informal exposition of the lemma might look like this:

Lemma $add\ m\ 0 = m$
Proof by induction on m.

- Case 0 (the base case): $add\ 0\ 0 = 0$ holds by definition of *add*.
- Case $Suc\ m$ (the induction step): We assume $add\ m\ 0 = m$, the induction hypothesis (IH), and we need to show $add\ (Suc\ m)\ 0 = Suc\ m$. The proof is as follows:

$$\begin{aligned} add\ (Suc\ m)\ 0 &= Suc\ (add\ m\ 0) && \text{by definition of } add \\ &= Suc\ m && \text{by IH} \end{aligned}$$

Throughout this book, **IH** will stand for “induction hypothesis”.

We have now seen three proofs of $add\ m\ 0 = 0$: the Isabelle one, the terse 4 lines explaining the base case and the induction step, and just now a model of a traditional inductive proof. The three proofs differ in the level of detail given and the intended reader: the Isabelle proof is for the machine, the informal proofs are for humans. Although this book concentrates of Isabelle proofs, it is important to be able to rephrase those proofs as informal text comprehensible to a reader familiar with traditional mathematical proofs. Later on we will introduce an Isabelle proof language that is closer to traditional informal mathematical language and is often directly readable.

2.2.3 Type *list*

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

datatype $'a\ list = Nil \mid Cons\ 'a\ ''\,'a\ list''$

- Type $'a\ list$ is the type of list over elements of type $'a$. Because $'a$ is a type variable, lists are in fact **polymorphic**: the elements of a list can be of arbitrary type (but must all be of the same type).
- Lists have two constructors: *Nil*, the empty list, and *Cons*, which puts an element (of type $'a$) in front of a list (of type $'a\ list$). Hence all lists are of the form *Nil*, or $Cons\ x\ Nil$, or $Cons\ x\ (Cons\ y\ Nil)$ etc.
- **datatype** requires no quotation marks on the left-hand side, but on the right-hand side each of the argument types of a constructor needs to be enclosed in quotation marks, unless it is just an identifier (e.g. *nat* or $'a$).

We also define two standard functions, append and reverse:

fun $app :: ''\,'a\ list \Rightarrow\ 'a\ list \Rightarrow\ 'a\ list''$ **where**
$''app\ Nil\ ys = ys'' \mid$
$''app\ (Cons\ x\ xs)\ ys = Cons\ x\ (app\ xs\ ys)''$

fun $rev :: ''\,'a\ list \Rightarrow\ 'a\ list''$ **where**
$''rev\ Nil = Nil'' \mid$
$''rev\ (Cons\ x\ xs) = app\ (rev\ xs)\ (Cons\ x\ Nil)''$

By default, variables xs, ys and zs are of *list* type.

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 $Cons\ b\ (Cons\ a\ Nil)$.

Figure 2.1 shows the theory created so far.

Structural Induction for Lists

Just as for natural numbers, there is a proof principle of induction for lists. Induction over a list is essentially induction 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

```
theory MyList
imports Main
begin

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

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)"

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

end
```

Fig. 2.1. A Theory of Lists

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**.

2.2.4 The Proof Process

We will now demonstrate the typical proof process, which involves the formulation and proof of auxiliary lemmas. 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(*induction xs*)

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

1. $rev\ (rev\ Nil) = Nil$
2. $\bigwedge a\ xs.\ rev\ (rev\ xs) = xs \Longrightarrow rev\ (rev\ (Cons\ a\ xs)) = Cons\ a\ xs$

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. $\bigwedge a\ xs.$
   $rev\ (rev\ xs) = xs \Longrightarrow$
   $rev\ (app\ (rev\ xs)\ (Cons\ a\ Nil)) = Cons\ a\ xs$

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

A First Lemma

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(*induction 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.

A Second Lemma

We again try the canonical proof procedure:

lemma *app_ Nil2* [*simp*]: "$app\ xs\ Nil = xs$"
apply(*induction 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(*induction xs*)
apply(*auto*)

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

1. $\bigwedge a\ xs.$
 $rev\ (app\ xs\ ys) = app\ (rev\ ys)\ (rev\ xs) \Longrightarrow$
 $app\ (app\ (rev\ ys)\ (rev\ xs))\ (Cons\ a\ Nil) =$
 $app\ (rev\ ys)\ (app\ (rev\ xs)\ (Cons\ a\ Nil))$

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

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(*induction xs*)
apply(*auto*)
done

Finally the proofs of *rev_ app* and *rev_ rev* succeed, too.

Another informal proof

Here is the informal proof of associativity of *app* corresponding to the Isabelle proof above.

Lemma $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$
Proof by induction on *xs*.

- Case *Nil*: $app\ (app\ Nil\ ys)\ zs = app\ ys\ zs = app\ Nil\ (app\ ys\ zs)$ holds by definition of *app*.
- Case *Cons x xs*: We assume

 $$app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs) \qquad \text{(IH)}$$

 and we need to show

 $$app\ (app\ (Cons\ x\ xs)\ ys)\ zs = app\ (Cons\ x\ xs)\ (app\ ys\ zs).$$

 The proof is as follows:

 $app\ (app\ (Cons\ x\ xs)\ ys)\ zs$
 $= app\ (Cons\ x\ (app\ xs\ ys))\ zs$ by definition of *app*
 $= Cons\ x\ (app\ (app\ xs\ ys)\ zs)$ by definition of *app*
 $= Cons\ x\ (app\ xs\ (app\ ys\ zs))$ by IH
 $= app\ (Cons\ x\ xs)\ (app\ ys\ zs)$ by definition of *app*

Didn't we say earlier that all proofs are by simplification? But in both cases, going from left to right, the last equality step is not a simplification at all!

In the base case it is $app\ ys\ zs = app\ Nil\ (app\ ys\ zs)$. It appears almost mysterious because we suddenly complicate the term by appending *Nil* on the left. What is really going on is this: when proving some equality $s = t$, both s and t are simplified to some common term u. This heuristic for equality proofs works well for a functional programming context like ours. In the base case s is $app\ (app\ Nil\ ys)\ zs$, t is $app\ Nil\ (app\ ys\ zs)$, and u is $app\ ys\ zs$.

2.2.5 Predefined lists

Isabelle's predefined lists are the same as the ones above, but with more syntactic sugar:

- $[]$ is *Nil*,
- $x\ \#\ xs$ is $Cons\ x\ xs$,
- $[x_1, \ldots, x_n]$ is $x_1\ \#\ \ldots\ \#\ x_n\ \#\ []$, and
- $xs\ @\ ys$ is $app\ xs\ ys$.

There is also a large library of predefined functions. The most important ones are the length function $length :: 'a\ list \Rightarrow nat$ (with the obvious definition), and the map function that applies a function to all elements of a list:

fun $map :: "('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list"$
$"map\ f\ [] = []"\ |$
$"map\ f\ (x\ \#\ xs) = f\ x\ \#\ map\ f\ xs"$

2.3 Type and function definitions

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.

2.3.1 Datatypes

The general form of a datatype definition looks like this:

$$
\begin{aligned}
\textbf{datatype}\ ('a_1,\ldots,'a_n)t = {} & \mathrm{C}_1\ \ "\tau_{1,1}" \ldots "\tau_{1,n_1}" \\
| {} & \ldots \\
| {} & \mathrm{C}_k\ \ "\tau_{k,1}" \ldots "\tau_{k,n_k}"
\end{aligned}
$$

It introduces the constructors $\mathrm{C}_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow ('a_1,\ldots,'a_n)t$ and expresses that any value of this type is built from these constructors in a unique manner. Uniqueness is implied by the following properties of the constructors:

- *Distinctness:* $C_i \ \ldots \neq C_j \ \ldots \quad$ 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 fact that any value of the datatype is built from the constructors implies the structural induction rule: to show $P\ x$ for all x of type $('a_1,\ldots,'a_n)t$, one needs to show $P(C_i\ x_1 \ldots x_{n_i})$ (for each i) assuming $P(x_j)$ for all j where $\tau_{i,j} = ('a_1,\ldots,'a_n)t$. Distinctness and injectivity are applied automatically by *auto* and other proof methods. Induction must be applied explicitly.

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

$(case\ xs\ of\ [] \Rightarrow 0 \mid x \ \#\ \_ \Rightarrow Suc\ x)$

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(*induction t*)

yields

1. $mirror\ (mirror\ Tip) = Tip$
2. $\bigwedge t1\ a\ t2.$
   $[\![mirror\ (mirror\ t1) = t1;\ mirror\ (mirror\ t2) = t2]\!]$
   $\Longrightarrow mirror\ (mirror\ (Node\ t1\ a\ t2)) = Node\ t1\ a\ t2$

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

A very simple but also very useful datatype is the predefined

datatype *'a option = None | Some 'a*

Its sole purpose is to add a new element *None* to an existing type *'a*. To make sure that *None* is distinct from all the elements of *'a*, you wrap them up in *Some* and call the new type *'a option*. A typical application is a lookup function on a list of key-value pairs, often called an association list:

fun *lookup* :: *"('a * 'b) list ⇒ 'a ⇒ 'b option"* **where**

"lookup [] x = None" |
"lookup ((a,b) # ps) x = (if a = x then Some b else lookup ps x)"

Note that $\tau_1 * \tau_2$ is the type of pairs, also written $\tau_1 \times \tau_2$.

2.3.2 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 $f\ x_1\ \dots\ x_n = t$, where f does not occur in t.

2.3.3 Abbreviations

Abbreviations are similar to definitions:

abbreviation *sq'* :: *"nat ⇒ nat"* **where**
*"sq' n == n * n"*

The key difference is that *sq'* is only syntactic sugar: *sq' t* is replaced by $t * t$ after parsing, and every occurrence of a term $u * u$ is replaced by *sq' u* before printing. Internally, *sq'* does not exist. This is also the advantage of abbreviations over definitions: definitions need to be expanded explicitly (see subsection 2.5.5) whereas abbreviations are already expanded upon parsing. However, abbreviations should be introduced sparingly: if abused, they can lead to a confusing discrepancy between the internal and external view of a term.

2.3.4 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 [?].

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(*induction n rule*: *div2.induct*)

yields the 3 subgoals

1. $div2\ (0 + 0) = 0$
2. $div2\ (Suc\ 0 + Suc\ 0) = Suc\ 0$
3. $\bigwedge n.\ div2\ (n + n) = n \Longrightarrow$
   $div2\ (Suc\ (Suc\ n) + Suc\ (Suc\ n)) = Suc\ (Suc\ n)$

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...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(*induction* $x_1 \ldots x_n$ *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.

2.4 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(*induction xs*)
apply(*auto*)

Unfortunately, this attempt does not prove the induction step:

1. $\bigwedge a\ xs.\ itrev\ xs\ [] = rev\ xs \Longrightarrow itrev\ xs\ [a] = rev\ xs\ @\ [a]$

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. $\bigwedge a\ xs.$
   $itrev\ xs\ ys = rev\ xs\ @\ ys \Longrightarrow$
   $itrev\ xs\ (a\ \#\ ys) = rev\ xs\ @\ a\ \#\ ys$

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(*induction xs arbitrary*: *ys*)

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

1. $\bigwedge ys.\ itrev\ []\ ys = rev\ []\ @\ ys$
2. $\bigwedge a\ xs\ ys.$
   $(\bigwedge ys.\ itrev\ xs\ ys = rev\ xs\ @\ ys) \Longrightarrow$
   $itrev\ (a\ \#\ xs)\ ys = rev\ (a\ \#\ xs)\ @\ ys$

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.

2.5 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

$$
\begin{aligned}
0 + n &= n & (1)\\
Suc\ m + n &= Suc\ (m + n) & (2)\\
(Suc\ m \leqslant Suc\ n) &= (m \leqslant n) & (3)\\
(0 \leqslant m) &= True & (4)
\end{aligned}
$$

the formula $0 + Suc\ 0 \leqslant Suc\ 0 + x$ is simplified to *True* as follows:

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

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

2.5.1 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 $\neg\ P$ 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 §2.5.4 below). Distributivity laws, for example, alter the structure of terms and can produce an exponential blow-up.

2.5.2 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

$$p\ 0 = \mathit{True}$$
$$p\ x \Longrightarrow f\ x = g\ x,$$

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

2.5.3 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.

When conditional simplification rules are applied, their preconditions are proved first. Hence all preconditions need to be simpler than the left-hand side of the conclusion. For example

$$n < m \Longrightarrow (n < \mathit{Suc}\ m) = \mathit{True}$$

is suitable as a simplification rule: both $n < m$ and True are simpler than $n < \mathit{Suc}\ m$. But

$$\mathit{Suc}\ n < m \Longrightarrow (n < m) = \mathit{True}$$

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

2.5.4 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

$$1.\ [\![\ P_1;\ \ldots;\ P_m\ ]\!] \Longrightarrow C$$

the command

apply(*simp add*: $th_1 \ldots th_n$)

simplifies the assumptions P_i and the conclusion C using

- all simplification rules, including the ones coming from **datatype** and **fun**,
- the additional lemmas $th_1 \ldots th_n$, 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. There is also *simp_all*, which applies *simp* to all subgoals.

2.5.5 Rewriting with definitions

Definitions introduced by the command **definition** can also 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.

2.5.6 Case splitting with *simp*

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

$$P\ (\mathit{if}\ A\ \mathit{then}\ s\ \mathit{else}\ t) = ((A \longrightarrow P\ s) \wedge (\neg\ A \longrightarrow P\ t))$$

For example, *simp* can prove

$$(A \wedge B) = (\mathit{if}\ A\ \mathit{then}\ B\ \mathit{else}\ \mathit{False})$$

because both $A \longrightarrow (A \wedge B) = B$ and $\neg A \longrightarrow (A \wedge B) = False$ simplify to *True*.

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

$$P\ (case\ e\ of\ 0 \Rightarrow a \mid Suc\ n \Rightarrow b\ n) =$$
$$((e = 0 \longrightarrow P\ a) \wedge (\forall n.\ e = Suc\ n \longrightarrow P\ (b\ n)))$$

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

Logic

3.1 Logic and Proof Beyond Equality

3.1.1 Formulas

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

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

Terms are the ones we have seen all along, built from constant, variables, function application and λ-abstraction, including all the syntactic sugar like infix symbols, *if*, *case* etc.

! 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 $s = t \wedge A$ means $(s = t) \wedge A$, and $A \wedge B = B \wedge A$ means $A \wedge (B = A) \wedge B$. Logical equivalence can also be written with $\longleftrightarrow$ instead of $=$, where $\longleftrightarrow$ has the same low precedence as $\longrightarrow$. Hence $A \wedge B \longleftrightarrow B \wedge A$ really means $(A \wedge B) \longleftrightarrow (B \wedge A)$.

! 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 $\llbracket A_1; \ldots; A_n \rrbracket \Longrightarrow A$, 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.

3.1.2 Sets

Sets of elements of type $'a$ have type $'a\ set$. They can be finite or infinite. Sets come with the usual notations:

- $\{\}$, $\{e_1,\ldots,e_n\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $-A$

and much more. $UNIV$ is the set of all elements of some type. Set comprehension is written $\{x.\ P\}$ rather than $\{x \mid P\}$, to emphasize the variable binding nature of the construct.

! In $\{x.\ P\}$ the x must be a variable. Set comprehension involving a proper term t must be written $''\{t \mid x\ y\ z.\ P\}''$, where $x\ y\ z$ are the free variables in t. This is just a shorthand for $\{v.\ \exists x\ y\ z.\ v = t \wedge P\}$, where v is a new variable.

Here are the ASCII representations of the mathematical symbols:

$\in$	\<in>	:
$\subseteq$	\<subseteq>	<=
$\cup$	\<union>	Un
$\cap$	\<inter>	Int

Sets also allow bounded quantifications $\forall x \in A.\ P$ and $\exists x \in A.\ P$.

3.1.3 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 "$\forall x.\ \exists y.\ x = y$"
by *auto*

lemma "$A \subseteq B \cap C \Longrightarrow A \subseteq B \cup C$"
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 is **complete** if it can prove all true formulas. There is no complete proof method for HOL, not even in theory. Hence all our proof methods only differ in how incomplete they are.

A proof method that is still incomplete but tries harder than *auto* is *fastforce*. 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 *fastforce* can do:

lemma "$[\![\ \forall xs \in A.\ \exists ys.\ xs = ys\ @\ ys;\ \ us \in A\ ]\!]$
$\Longrightarrow \exists n.\ length\ us = n{+}n$"
by *fastforce*

This lemma is out of reach for *auto* because of the quantifiers. Even *fastforce* fails when the quantifier structure becomes more complicated. In a few cases, its slow version *force* succeeds where *fastforce* fails.

The method of choice for complex logical goals is *blast*. 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
"$[\![\ \forall x\ y.\ T\ x\ y \lor T\ y\ x;$
$\forall x\ y.\ A\ x\ y \land A\ y\ x \longrightarrow x = y;$

$\forall x\ y.\ T\ x\ y \longrightarrow A\ x\ y\ \rrbracket$
$\Longrightarrow \forall x\ y.\ A\ x\ y \longrightarrow T\ x\ y$"
by *blast*

We leave it to the reader to figure out why this lemma is true. Method *blast*

- is (in principle) a complete proof procedure for first-order formulas, a fragment of HOL. 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.

Sledgehammer

Command **sledgehammer** calls a number of external automatic theorem provers (ATPs) that run for up to 30 seconds searching for a proof. Some of these ATPs are part of the Isabelle installation, others are queried over the internet. If successful, a proof command is generated and can be inserted into your proof. The biggest win of **sledgehammer** is that it will take into account the whole lemma library and you do not need to feed in any lemma explicitly. For example,

lemma "$\llbracket$ *xs* @ *ys* = *ys* @ *xs*; *length xs* = *length ys* $\rrbracket \Longrightarrow$ *xs* = *ys*"

cannot be solved by any of the standard proof methods, but **sledgehammer** finds the following proof:

by (*metis append_eq_conv_conj*)

We do not explain how the proof was found but what this command means. For a start, Isabelle does not trust external tools (and in particular not the translations from Isabelle's logic to those tools!) and insists on a proof that it can check. This is what *metis* does. It is given a list of lemmas and tries to find a proof just using those lemmas (and pure logic). In contrast to *simp* and friends that know a lot of lemmas already, using *metis* manually is tedious because one has to find all the relevant lemmas first. But that is precisely what **sledgehammer** does for us. In this case lemma *append_eq_conv_conj* alone suffices:

(*xs* @ *ys* = *zs*) = (*xs* = *take* (*length xs*) *zs* $\wedge$ *ys* = *drop* (*length xs*) *zs*)

We leave it to the reader to figure out why this lemma suffices to prove the above lemma, even without any knowledge of what the functions *take* and *drop* do. Keep in mind that the variables in the two lemmas are independent of each other, despite the same names, and that you can substitute arbitrary values for the free variables in a lemma.

Just as for the other proof methods we have seen, there is no guarantee that **sledgehammer** will find a proof if it exists. Nor is **sledgehammer** superior to the other proof methods. They are incomparable. Therefore it is recommended to apply *simp* or *auto* before invoking **sledgehammer** on what is left.

Arithmetic

By arithmetic formulas we mean formulas involving variables, numbers, $+$, $-$, $=$, $<$, $\leqslant$ and the usual logical connectives $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftrightarrow$. Strictly speaking, this is known as **linear arithmetic** because it does not involve multiplication, although multiplication with numbers, e.g. $2{*}n$ is allowed. Such formulas can be proved by *arith*:

lemma "$[\![\ (a{::}nat) \leqslant x + b;\ 2{*}x < c\ ]\!] \Longrightarrow 2{*}a + 1 \leqslant 2{*}b + c$"
by *arith*

In fact, *auto* and *simp* can prove many linear arithmetic formulas already, like the one above, by calling a weak but fast version of *arith*. Hence it is usually not necessary to invoke *arith* explicitly.

The above example involves natural numbers, but integers (type *int*) and real numbers (type *real*) are supported as well. As are a number of further operators like *min* and *max*. On *nat* and *int*, *arith* can even prove theorems with quantifiers in them, but we will not enlarge on that here.

3.1.4 Single step proofs

Although automation is nice, it often fails, at least initially, and you need to find out why. When *fastforce* 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 $A \wedge B$, 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.

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

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

  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 $?P \wedge ?Q$ with $a = b \wedge \mathit{False}$ 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*"].

Rule application

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

1. $\ldots \Longrightarrow A \wedge B$

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

1. $\ldots \Longrightarrow A$
2. $\ldots \Longrightarrow B$

In general, the application of a rule $[\![\ A_1;\ \ldots;\ A_n\ ]\!] \Longrightarrow A$ to a subgoal $\ldots \Longrightarrow C$ 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*.

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. [?]). Although we intentionally de-emphasize 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 the left-hand side of a HOL implication 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*, *fastforce* 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 $\leqslant$ 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 $''[\![\ (a{::}nat)\ \leqslant\ b;\ b\ \leqslant\ c;\ c\ \leqslant\ d;\ d\ \leqslant\ e\ ]\!] \Longrightarrow a\ \leqslant\ e''$
by(*blast intro*: *le_ trans*)

Of course this is just an example and could be proved by *arith*, too.

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 $\llbracket\ A_1;\ \ldots;\ A_n\ \rrbracket \Longrightarrow A$ and there can be multiple argument theorems r_1 to r_m (with $m \leqslant n$), in which case $r[OF\ r_1 \ldots r_m]$ 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 also makes sense in connection with proof states. Therefore *blast*, *fastforce* 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 $Suc\ m \leqslant n \Longrightarrow m \leqslant n$, which works well for forward reasoning:

lemma "$Suc(Suc(Suc\ a)) \leqslant b \Longrightarrow a \leqslant b$"
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.

3.2 Inductive definitions

Inductive definitions are the third important definition facility, after datatypes and recursive function. In fact, they are the key construct in the definition of operational semantics in the second part of the book.

3.2.1 An example: even numbers

Here is a simple example of an inductively defined predicate:

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

The operative word "inductive" means that these are the only even numbers. In Isabelle we give the two rules the names *ev0* and *evSS* and write

inductive ev :: "$nat \Rightarrow bool$" **where**
$ev0$: "$ev\ 0$" |
$evSS$: "$ev\ n \Longrightarrow ev\ (n + 2)$"

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:

$$ev\ 0 \Longrightarrow ev\ (0 + 2) \Longrightarrow ev((0 + 2) + 2) = ev\ 4$$

Rule induction

Showing that all even numbers have some property is more complicated. For example, let us prove that the inductive definition of even numbers agrees with the following recursive one:

fun $even$ :: "$nat \Rightarrow bool$" **where**
"$even\ 0 = True$" |
"$even\ (Suc\ 0) = False$" |
"$even\ (Suc(Suc\ n)) = even\ n$"

We prove $ev\ m \Longrightarrow even\ m$. That is, we assume $ev\ m$ and by induction on the form of its derivation prove $even\ m$. There are two cases corresponding to the two rules for ev:

Case $ev0$: $ev\ m$ was derived by rule $ev\ 0$:
$\Longrightarrow m = 0 \Longrightarrow even\ m = even\ 0 = True$
Case $evSS$: $ev\ m$ was derived by rule $ev\ n \Longrightarrow ev\ (n + 2)$:
$\Longrightarrow m = n + 2$ and by induction hypothesis $even\ n$
$\Longrightarrow even\ m = even(n + 2) = even\ n = True$

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

$$ev\ n \Longrightarrow P\ n$$

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

Case $ev0$: $P\ 0$

Case $evSS$: $[\![ev\ n;\ P\ n]\!] \Longrightarrow P\ (n + 2)$

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

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

The first premise $ev\ n$ enforces that this rule can only be applied in situations where we know that n is even.

Note that in the induction step we may not just assume $P\ n$ but also $ev\ n$, which is simply the premise of rule $evSS$. Here is an example where the local assumption $ev\ n$ comes in handy: we prove $ev\ m \Longrightarrow ev\ (m - 2)$ by induction on $ev\ m$. Case $ev0$ requires us to prove $ev\ (0 - 2)$, which follows from $ev\ 0$ because $0 - 2 = 0$ on type nat. In case $evSS$ we have $m = n + 2$ and may assume $ev\ n$, which implies $ev\ (m - 2)$ because $m - 2 = (n + 2) - 2 = n$. We did not need the induction hypothesis at all for this proof, it is just a case distinction on which rule was used, but having $ev\ n$ at our disposal in case $evSS$ was essential. This case distinction over rules is also called "rule inversion" and is discussed in more detail in Chapter 4.

In Isabelle

Let us now recast the above informal proofs in Isabelle. For a start, we use Suc terms instead of numerals in rule $evSS$:

$ev\ n \Longrightarrow ev\ (Suc\ (Suc\ n))$

This avoids the difficulty of unifying $n+2$ with some numeral, which is not automatic.

The simplest way to prove $ev\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$ is in a forward direction: $evSS[OF\ evSS[OF\ ev0]]$ yields the theorem $ev\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$. Alternatively, you can also prove it as a lemma in backwards fashion. Although this is more verbose, it allows us to demonstrate how each rule application changes the proof state:

lemma "$ev(Suc(Suc(Suc(Suc\ 0))))$"

1. $ev\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$

apply($rule\ evSS$)

1. $ev\ (Suc\ (Suc\ 0))$

apply($rule\ evSS$)

1. $ev\ 0$

apply(*rule ev0*)
done

Rule induction is applied by giving the induction rule explicitly via the *rule*: modifier:

lemma "$ev\ m \Longrightarrow even\ m$"
apply(*induction rule*: *ev.induct*)
by(*simp_all*)

Both cases are automatic. Note that if there are multiple assumptions of the form $ev\ t$, method *induction* will induct on the leftmost one.

As a bonus, we also prove the remaining direction of the equivalence of *ev* and *even*:

lemma "$even\ n \Longrightarrow ev\ n$"
apply(*induction n rule*: *even.induct*)

This is a proof by computation induction on n (see subsection 2.3.4) that sets up three subgoals corresponding to the three equations for *even*:

1. $even\ 0 \Longrightarrow ev\ 0$
2. $even\ (Suc\ 0) \Longrightarrow ev\ (Suc\ 0)$
3. $\bigwedge n.\ [\![even\ n \Longrightarrow ev\ n;\ even\ (Suc\ (Suc\ n))]\!] \Longrightarrow ev\ (Suc\ (Suc\ n))$

The first and third subgoals follow with *ev0* and *evSS*, and the second subgoal is trivially true because $even\ (Suc\ 0)$ is *False*:

by (*simp_all add*: *ev0 evSS*)

The rules for *ev* make perfect simplification and introduction rules because their premises are always smaller than the conclusion. It makes sense to turn them into simplification and introduction rules permanently, to enhance proof automation:

declare *ev.intros*[*simp*,*intro*]

The rules of an inductive definition are not simplification rules by default because, in contrast to recursive functions, there is no termination requirement for inductive definitions.

Inductive versus recursive

We have seen two definitions of the notion of evenness, an inductive and a recursive one. Which one is better? Much of the time, the recursive one is more convenient: it allows us to do rewriting in the middle of terms, and it expresses both the positive information (which numbers are even) and the negative information (which numbers are not even) directly. An inductive definition only expresses the positive information directly. The negative information, for example, that 1 is not even, has to be proved from it (by induction or rule inversion). On the other hand, rule induction is Taylor made for proving $ev\ n \Longrightarrow P\ n$ because it only asks you to prove the positive cases. In the proof of $even\ n \Longrightarrow P\ n$ by computation induction via *even.induct*, we are also presented with the trivial negative cases. If you want the convenience of both rewriting and rule induction, you can make two definitions and show their equivalence (as above) or make one definition and prove additional properties from it, for example rule induction from computation induction.

But many concepts do not admit a recursive definition at all because there is no datatype for the recursion (for example, the transitive closure of a relation), or the recursion would not terminate (for example, the operational semantics in the second part of this book). Even if there is a recursive definition, if we are only interested in the positive information, the inductive definition may be much simpler.

3.2.2 The reflexive transitive closure

Evenness is really more conveniently expressed recursively than inductively. As a second and very typical example of an inductive definition we define the reflexive transitive closure. It will also be an important building block for some of the semantics considered in the second part of the book.

The reflexive transitive closure, called *star* below, is a function that maps a binary predicate to another binary predicate: if r is of type $\tau \Rightarrow \tau \Rightarrow bool$ then $star\ r$ is again of type $\tau \Rightarrow \tau \Rightarrow bool$, and $star\ r\ x\ y$ means that x and y are in the relation $star\ r$. Think r^* when you see $star\ r$, because $star\ r$ is meant to be the reflexive transitive closure. That is, $star\ r\ x\ y$ is meant to be true if from x we can reach y in finitely many r steps. This concept is naturally defined inductively:

```
inductive star :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"  for r where
refl:  "star r x x" |
step:  "r x y ⟹ star r y z ⟹ star r x z"
```

The base case *refl* is reflexivity: $x = y$. The step case *step* combines an r step (from x to y) and a *star* step (from y to z) into a *star* step (from x to z). The "**for** r" in the header is merely a hint to Isabelle that r is a fixed parameter of *star*, in contrast to the further parameters of *star*, which change. As a result, Isabelle generates a simpler induction rule.

By definition $star\ r$ is reflexive. It is also transitive, but we need rule induction to prove that:

lemma *star_trans*: "$star\ r\ x\ y \Longrightarrow star\ r\ y\ z \Longrightarrow star\ r\ x\ z$"
apply(*induction rule*: *star.induct*)

The induction is over $star\ r\ x\ y$ and we try to prove $star\ r\ y\ z \Longrightarrow star\ r\ x\ z$, which we abbreviate by $P\ x\ y$. These are our two subgoals:

1. $\bigwedge x.\ star\ r\ x\ z \Longrightarrow star\ r\ x\ z$
2. $\bigwedge u\ x\ y.$
   $[\![r\ u\ x;\ star\ r\ x\ y;\ star\ r\ y\ z \Longrightarrow star\ r\ x\ z;\ star\ r\ y\ z]\!]$
   $\Longrightarrow star\ r\ u\ z$

The first one is $P\ x\ x$, the result of case *refl*, and it is trivial.

apply(*assumption*)

Let us examine subgoal 2, case *step*. Assumptions $r\ u\ x$ and $star\ r\ x\ y$ are the premises of rule *step*. Assumption $star\ r\ y\ z \Longrightarrow star\ r\ x\ z$ is $P\ x\ y$, the IH coming from $star\ r\ x\ y$. We have to prove $P\ u\ y$, which we do by assuming $star\ r\ y\ z$ and proving $star\ r\ u\ z$. The proof itself is straightforward: from $star\ r\ y\ z$ the IH leads to $star\ r\ x\ z$ which, together with $r\ u\ x$, leads to $star\ r\ u\ z$ via rule *step*:

apply(*metis step*)
done

3.2.3 The general case

Inductive definitions have approximately the following general form:

inductive I :: "$\tau \Rightarrow bool$" **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

$I\ x \Longrightarrow P\ x$

where P may itself be a chain of implications.

! Rule induction is always on the leftmost premise of the goal. Hence $I\ x$ must be the first premise.

Proving $I\ x \Longrightarrow P\ x$ by rule induction means proving for every rule of I that P is invariant:

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

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**. In rule inductions, these side-conditions appear as additional assumptions. The **for** clause seen in the definition of the reflexive transitive closure merely simplifies the form of the induction rule.

4

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*
  $\vdots$
  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. (*induction 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 §3—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).

4.1 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 $"\neg\ surj(f :: 'a \Rightarrow 'a\ set)"$
proof
  assume 0: $"surj\ f"$
  from 0 **have** 1: $"\forall A.\ \exists a.\ A = f\ a"$ **by**(*simp add*: *surj_def*)
  from 1 **have** 2: $"\exists a.\ \{x.\ x \notin f\ x\} = f\ a"$ **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 $\neg$:

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

In order to prove $\neg\ P$, 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 $\{x.\ x \notin f\ x\}$, the key idea in the proof. If you wonder why 2 directly implies *False*: from 2 it follows that $(a \notin f\ a) = (a \in f\ a)$.

4.1.1 *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** $"\exists\, a.\ \{x.\ x \notin f\ x\} = f\ a"$ **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 $"\exists\, a.\ \{x.\ x \notin f\ x\} = f\ a"$ **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.

4.1.2 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 $\neg$ *surj f* a little:

lemma
  fixes f :: "$'a \Rightarrow 'a\ set$"
  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; multiple assumptions can be separated by **and**. The **shows** part gives the goal. The actual theorem that will come out of the proof is $surj\ f \Longrightarrow False$, but during the proof the assumption $surj\ f$ is available under the name s like any other fact.

proof −
  have "$\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$" **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.

4.2 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 $\neg\ P$, and starting from a fact $P \vee Q$ we have the two cases P and Q:

```
show "R"
proof cases
  assume "P"
    ⋮
  show "R" ...
next
  assume "¬ P"
    ⋮
  show "R" ...
qed
```

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

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

The name *ccontr* stands for “classical contradiction”.

How to prove quantified formulas:

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

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

In the proof of $\forall x.\ P(x)$, 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 $\exists x.\ P(x)$, *witness* is some arbitrary term for which we can prove that it satisfies P.

How to reason forward from $\exists x.\ P(x)$:

have $''\exists x.\ P(x)''$ ...
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 $''\neg\ surj(f :: 'a \Rightarrow 'a\ set)''$
proof
  assume $''surj\ f''$
  hence $''\exists a.\ \{x.\ x \notin f\ x\} = f\ a''$ **by**(*auto simp*: *surj_def*)
  then obtain a **where** $''\{x.\ x \notin f\ x\} = f\ a''$ **by** *blast*
  hence $''a \notin f\ a \longleftrightarrow a \in f\ a''$ **by** *blast*
  thus $''False''$ **by** *blast*
qed

Finally, how to prove set equality and subset relationship:

show $''A = B''$
proof
  show $''A \subseteq B''$ ...
next
  show $''B \subseteq A''$ ...
qed

show $''A \subseteq B''$
proof
  fix x
  assume $''x \in A''$
  $\vdots$
  show $''x \in B''$ ...
qed

4.3 Streamlining proofs

4.3.1 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 *"formula*$_1$ $\longleftrightarrow$ *formula*$_2$*"* (**is** *"?L* $\longleftrightarrow$ *?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 $"x > 0"$
$\vdots$
from ‘$x{>}0$‘ ...

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

4.3.2 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 $"P_1"$ ...	**have** lab_1: $"P_1"$ ...
moreover have $"P_2"$ ...	**have** lab_2: $"P_2"$ ...
moreover	$\vdots$
$\vdots$	**have** lab_n: $"P_n"$ ...
moreover have $"P_n"$ ...	**from** lab_1 lab_2 ...
ultimately have $"P"$ ...	**have** $"P"$ ...

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

4.3.3 Raw proof blocks

Sometimes one would like to prove some lemma locally within 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** x_1 ... x_n
  assume A_1 ... A_m
  $\vdots$
  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 $(b\ dvd\ a) = (\exists\, k.\ a = b * k)$.

```
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.

4.4 Case distinction and induction

4.4.1 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 $x_1 \ldots x_n$ **assume** "$t = C\ x_1 \ldots x_n$"

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

case $(C\ x_1\ \dots\ x_n)$

This is equivalent to the explicit **fix-assumen** line but also gives the assumption "$t = C\ x_1\ \dots\ x_n$" 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.

4.4.2 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 "$\sum\{0..n::nat\} = n*(n+1)\ div\ 2$" (**is** "*?P n*")
proof (*induction n*)
  show "$\sum\{0..0::nat\} = 0*(0+1)\ div\ 2$" **by** *simp*
next
  fix n **assume** "$\sum\{0..n::nat\} = n*(n+1)\ div\ 2$"
  thus "$\sum\{0..Suc\ n::nat\} = Suc\ n*(Suc\ n+1)\ div\ 2$" **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 "$\sum\{0..n::nat\} = n*(n+1)\ div\ 2$" (**is** "*?P n*")
proof (*induction 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. Pattern matching *?P n* with the goal instantiates *?P* to the function $\lambda n.\ \sum\{0..n\} = n * (n +$

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 (*induction 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 shown on the left-hand side:

show "$P(n)$"	
proof (*induction 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*: "$A(n) \Longrightarrow P(n)$"
"$A(Suc\ n)$"
let $?case = "P(Suc\ n)"$

The list of assumptions *Suc* is actually subdivided into *Suc.IH*, 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. Proving 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 $C.IH$) and the premises $A_i(C\ x_1\ \ldots\ x_n)$ (calling them *C.prems*) and calling the whole list C
3. **let** *?case* = "$P(C\ x_1\ \ldots\ x_n)$"

4.4.3 Rule induction

Recall the inductive and recursive definitions of even numbers in Section 3.2:

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

fun even :: "nat ⇒ bool" where
"even 0 = True" |
"even (Suc 0) = False" |
"even (Suc(Suc n)) = even n"
```

We recast the proof of *ev n* $\Longrightarrow$ *even n* in Isar. The left column shows the actual proof text, the right column shows the implicit effect of the two **case** commands:

```
lemma "ev n ⟹ even n"
proof(induction rule: ev.induct)
  case ev0                            let ?case = "even 0"
  show ?case by simp
next
  case evSS                           fix n
                                      assume evSS: "ev n"
                                                   "even n"
                                      let ?case = "even(Suc(Suc n))"
  thus ?case by simp
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. 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; *even 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*: $ev\ m \Longrightarrow ev(Suc(Suc\ m))$. Here is an example with a (contrived) intermediate step that refers to m:

```
lemma "ev n ⟹ even n"
proof(induction rule: ev.induct)
  case ev0 show ?case by simp
next
  case (evSS m)
  have "even(Suc(Suc m)) = even m" by simp
  thus ?case using ‘even m‘ by blast
qed
```

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

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

One can 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.

4.4.4 Assumption naming

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

name.IH contains the induction hypotheses.

name.hyps contains all the other hypotheses of this case in the induction rule. For rule inductions these are the hypotheses of rule *name*, for structural inductions these are empty.

name.prems contains the (suitably instantiated) premises of the statement being proved, i.e. the A_i when proving $[\![\ A_1;\ \ldots;\ A_n\ ]\!] \Longrightarrow A$.

! Proof method *induct* differs from *induction* only in this naming policy: *induct* does not distinguish *IH* from *hyps* but subsumes *IH* under *hyps*.

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.IH* can be too unspecific. This is where the indexing of fact lists comes in handy, e.g. *name.IH*(2) or *name.prems*(1–2).

4.4.5 Rule inversion

Rule inversion is case distinction on which rule could have been used to derive some fact. The name **rule inversion** emphasizes that we are reasoning backwards: by which rules could some given fact have been proved? For the inductive definition of *ev*, rule inversion can be summarized like this:

$$ev\ n \Longrightarrow n = 0 \lor (\exists k.\ n = Suc\ (Suc\ k) \land ev\ k)$$

The realisation in Isabelle is a case distinction. A simple example is the proof that $ev\ n \Longrightarrow ev\ (n - 2)$. We already went through the details informally in subsection 3.2.1. This is the Isar proof:

```
assume "ev n"
from this have "ev(n - 2)"
proof cases
  case ev0 thus "ev(n - 2)" by (simp add: ev.ev0)
next
  case (evSS k) thus "ev(n - 2)" by (simp add: ev.evSS)
qed
```

The key point here is that a case distinction over some inductively defined predicate is triggered by piping the given fact (here: **from** *this*) into a proof by *cases*. Let us examine the assumptions available in each case. In case *ev0* we have $n = 0$ and in case *evSS* we have $n = Suc\ (Suc\ k)$ and *ev k*. In each case the assumptions are available under the name of the case; there is no fine grained naming schema like for induction.

Sometimes some rules could not have beed used to derive the given fact because constructors clash. As an extreme example consider rule inversion applied to $ev\ (Suc\ 0)$: neither rule *ev0* nor rule *evSS* can yield $ev\ (Suc\ 0)$ because $Suc\ 0$ unifies neither with 0 nor with $Suc\ (Suc\ n)$. Impossible cases do not have to be proved. Hence we can prove anything from $ev\ (Suc\ 0)$:

assume "$ev(Suc\ 0)$" **then have** P **by** *cases*

That is, $ev\ (Suc\ 0)$ is simply not provable:

lemma "$\neg\ ev(Suc\ 0)$"
proof
  assume "$ev(Suc\ 0)$" **then show** *False* **by** *cases*
qed

Normally not all cases will be impossible. As a simple exercise, prove that $\neg\ ev\ (Suc\ (Suc\ (Suc\ 0)))$.