

The Tutorial
— DRAFT —

Tobias Nipkow
Technische Universität München
Institut für Informatik
<http://www.in.tum.de/~nipkow/>

6 October 2000

Contents

1	Basic Concepts	1
1.1	Introduction	1
1.2	Theories	1
1.3	Types, terms and formulae	2
1.4	Variables	5
1.5	Interaction and interfaces	5
1.6	Getting started	6
2	Functional Programming in HOL	7
2.1	An introductory theory	7
2.2	An introductory proof	9
2.3	Some helpful commands	13
2.4	Datatypes	15
2.4.1	Lists	15
2.4.2	The general format	15
2.4.3	Primitive recursion	16
2.4.4	Case expressions	16
2.4.5	Structural induction and case distinction	17
2.4.6	Case study: boolean expressions	18
2.5	Some basic types	20
2.5.1	Natural numbers	20
2.5.2	Products	22
2.6	Definitions	22
2.6.1	Type synonyms	23
2.6.2	Constant definitions	23
3	More Functional Programming	24
3.1	Simplification	24
3.2	Induction heuristics	31
3.3	Case study: compiling expressions	33
3.4	Advanced datatypes	35
3.4.1	Mutual recursion	35
3.4.2	Nested recursion	37
3.4.3	The limits of nested recursion	39
3.4.4	Case study: Tries	40
3.5	Total recursive functions	43

3.5.1	Defining recursive functions	43
3.5.2	Proving termination	45
3.5.3	Simplification with recdef	46
3.5.4	Induction	48
4	Advanced Simplification, Recursion, and Induction	50
4.1	Simplification	50
4.1.1	Advanced features	50
4.1.2	How it works	52
4.2	Advanced forms of recursion	52
4.3	Advanced induction techniques	54
4.3.1	Massaging the proposition	55
4.3.2	Beyond structural and recursion induction	56
4.3.3	Derivation of new induction schemas	58
A	Appendix	59

Acknowledgements

This tutorial owes a lot to the constant discussions with and the valuable feedback from Larry Paulson and the Isabelle group at Munich: Olaf Müller, Wolfgang Naraschewski, David von Oheimb, Leonor Prensa Nieto, Cornelia Pusch and Markus Wenzel. Stefan Berghofer and Stephan Merz were also kind enough to read and comment on a draft version.

Basic Concepts

1.1 Introduction

This is a tutorial on how to use Isabelle/HOL as a specification and verification system. 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 both fields. For excellent introductions to functional programming consult the textbooks by Bird and Wadler [3] or Paulson [10]. Although this tutorial initially concentrates on functional programming, do not be misled: HOL can express most mathematical concepts, and functional programming is just one particularly simple and ubiquitous instance.

This tutorial introduces HOL via Isabelle/Isar [11], which is an extension of Isabelle [9] with structured proofs.¹ The most noticeable difference to classical Isabelle (which is the basis of another version of this tutorial) is the replacement of the ML level by a dedicated language for definitions and proofs.

A tutorial is by definition incomplete. Currently the tutorial only introduces the rudiments of Isar's proof language. To fully exploit the power of Isar you need to consult the Isabelle/Isar Reference Manual [11]. If you want to use Isabelle's ML level directly (for example for writing your own proof procedures) see the Isabelle Reference Manual [7]; for details relating to HOL see the Isabelle/HOL manual [6]. All manuals have a comprehensive index.

1.2 Theories

Working with Isabelle means creating theories. Roughly speaking, a **theory** is a named collection of types, functions, and theorems, much like a module

¹Thus the full name of the system should be Isabelle/Isar/HOL, but that is a bit of a mouthful.

in a programming language or a specification in a specification language. In fact, theories in HOL can be either. The general format of a theory T is

```
theory T = B1 + ... + Bn:
  declarations, definitions, and proofs
end
```

where B_1, \dots, B_n are the names of existing theories that T is based on and *declarations, definitions, and proofs* represents the newly introduced concepts (types, functions etc.) and proofs about them. The B_i are the direct **parent theories** of T . Everything defined in the parent theories (and their parents ...) is automatically visible. To avoid name clashes, identifiers can be **qualified** by theory names as in $T.f$ and $B.f$. Each theory T must reside in a **theory file** named $T.thy$.

This tutorial is concerned with introducing you to the different linguistic constructs that can fill *declarations, definitions, and proofs* in the above theory template. A complete grammar of the basic constructs is found in the Isabelle/Isar Reference Manual.

HOL's theory library is available online at

<http://isabelle.in.tum.de/library/>

and is recommended browsing. Note that most of the theories in the library are based on classical Isabelle without the Isar extension. This means that they look slightly different than the theories in this tutorial, and that all proofs are in separate ML files.

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

1.3 Types, terms and formulae

Embedded in a theory are the types, terms and formulae of HOL. HOL is a typed logic whose type system resembles that of functional programming languages like ML or Haskell. Thus there are

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

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. Parentheses around single arguments can be dropped (as in *nat list*), multiple arguments are separated by commas (as in *(bool,nat)ty*).

function types, denoted by \Rightarrow . In HOL \Rightarrow represents *total* functions only. As is customary, $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$ means $\tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$. Isabelle also supports the notation $[\tau_1, \dots, \tau_n] \Rightarrow \tau$ which abbreviates $\tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \tau$.

type variables, denoted by 'a, 'b etc., just like in ML. They give rise to polymorphic types like 'a \Rightarrow 'a, the type of the identity function.

! Types are extremely important because they prevent us from writing nonsense.

- Isabelle insists that all terms and formulae must be well-typed and will print an error message if a type mismatch is encountered. To reduce the amount of explicit type information that needs to be provided by the user, Isabelle infers the type of all variables automatically (this is called **type inference**) and keeps quiet about it. Occasionally this may lead to misunderstandings between you and the system. If anything strange happens, we recommend to set the flag `show_types` that tells Isabelle to display type information that is usually suppressed: simply type

```
ML "set show_types"
```

This can be reversed by ML `reset show_types`. Various other flags can be set and reset in the same manner.

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 . HOL also supports infix functions like $+$ and some basic constructs from functional programming:

`if b then t1 else t2` means what you think it means and requires that b is of type `bool` and t_1 and t_2 are of the same type.

`let x = t in u` is equivalent to u where all occurrences of x have been replaced by t . For example, `let x = 0 in x+x` is equivalent to `0+0`. Multiple bindings are separated by semicolons: `let x1 = t1; ...; xn = tn in u`.

`case e of c1 \Rightarrow e1 | ... | cn \Rightarrow en` evaluates to e_i if e is of the form c_i .

Terms may also contain λ -abstractions. For example, `$\lambda x. x+1$` is the function that takes an argument x and returns $x+1$. Instead of `$\lambda x. \lambda y. \lambda z. t$` we can write `$\lambda x\ y\ z. t$` .

Formulae are terms of type `bool`. There are the basic constants `True` and `False` and the usual logical connectives (in decreasing order of priority): \neg , \wedge , \vee , and \longrightarrow , all of which (except the unary \neg) associate to the right. In particular $A \longrightarrow B \longrightarrow C$ means $A \longrightarrow (B \longrightarrow C)$ and is thus logically equivalent to $A \wedge B \longrightarrow C$ (which is $(A \wedge B) \longrightarrow C$).

Equality is available in the form of the infix function `=` of type 'a \Rightarrow 'a \Rightarrow `bool`. Thus $t_1 = t_2$ is a formula provided t_1 and t_2 are terms of the same type. In case t_1 and t_2 are of type `bool`, `=` acts as if-and-only-if. The formula $t_1 \neq t_2$ is merely an abbreviation for $\neg(t_1 = t_2)$.

The syntax for quantifiers is $\forall x. P$ and $\exists x. P$. There is even $\exists! x. P$, which means that there exists exactly one x that satisfies P . Nested quantifications can be abbreviated: $\forall x y z. P$ means $\forall x. \forall y. \forall z. P$.

Despite type inference, it is sometimes necessary to attach explicit **type constraints** to a term. The syntax is $t :: \tau$ as in $x < (y :: \text{nat})$. Note that $::$ binds weakly and should therefore be enclosed in parentheses: $x < y :: \text{nat}$ is ill-typed because it is interpreted as $(x < y) :: \text{nat}$. The main reason for type constraints are overloaded functions like $+$, $*$ and $<$. (See §?? for a full discussion of overloading.)

! In general, HOL's concrete syntax tries to follow the conventions of functional programming and mathematics. Below we list the main rules that you should be familiar with to avoid certain syntactic traps. A particular problem for novices can be the priority of operators. If you are unsure, use more rather than fewer parentheses. In those cases where Isabelle echoes your input, you can see which parentheses are dropped—they were superfluous. If you are unsure how to interpret Isabelle's output because you don't know where the (dropped) parentheses go, set (and possibly reset) the flag `show_brackets`:

```
ML "set show_brackets"; ...; ML "reset show_brackets";
```

- Remember that $f t u$ means $(f t) u$ and not $f(t u)$!
- Isabelle allows infix functions like $+$. The prefix form of function application binds more strongly than anything else and hence $f x + y$ means $(f x) + y$ and not $f(x+y)$.
- Remember that in HOL if-and-only-if is expressed using equality. But equality has a high priority, as befitting a relation, while if-and-only-if typically has the lowest priority. Thus, $\neg \neg P = P$ means $\neg \neg (P = P)$ and not $(\neg \neg P) = P$. When using $=$ to mean logical equivalence, enclose both operands in parentheses, as in $(A \wedge B) = (B \wedge A)$.
- Constructs with an opening but without a closing delimiter bind very weakly and should therefore be enclosed in parentheses if they appear in subterms, as in $f = (\lambda x. x)$. This includes *if*, *let*, *case*, λ , and quantifiers.
- Never write $\lambda x.x$ or $\forall x.x=x$ because $x.x$ is always read as a single qualified identifier that refers to an item x in theory x . Write $\lambda x. x$ and $\forall x. x=x$ instead.
- Identifiers may contain `_` and `'`.

For the sake of readability the usual mathematical symbols are used throughout the tutorial. Their ASCII-equivalents are shown in figure A.1 in the appendix.

1.4 Variables

Isabelle distinguishes free and bound variables just as is customary. Bound variables are automatically renamed to avoid clashes with free variables. In addition, Isabelle has a third kind of variable, called a **schematic variable** or **unknown**, which starts with a `?`. Logically, an unknown is a free variable. But it may be instantiated by another term during the proof process. For example, the mathematical theorem $x = x$ is represented in Isabelle as `?x = ?x`, which means that Isabelle can instantiate it arbitrarily. This is in contrast to ordinary variables, which remain fixed. The programming language Prolog calls unknowns *logical* variables.

Most of the time you can and should ignore unknowns and work with ordinary variables. Just don't be surprised that after you have finished the proof of a theorem, Isabelle will turn your free variables into unknowns: it merely indicates that Isabelle will automatically instantiate those unknowns suitably when the theorem is used in some other proof. Note that for readability we often drop the `?`s when displaying a theorem.

- ! If you use `?` as an existential quantifier, it needs to be followed by a space.
- Otherwise `?x` is interpreted as a schematic variable.

1.5 Interaction and interfaces

Interaction with Isabelle can either occur at the shell level or through more advanced interfaces. To keep the tutorial independent of the interface we have phrased the description of the interaction in a neutral language. For example, the phrase “to abandon a proof” means to type **oops** at the shell level, which is explained the first time the phrase is used. Other interfaces perform the same act by cursor movements and/or mouse clicks. Although shell-based interaction is quite feasible for the kind of proof scripts currently presented in this tutorial, the recommended interface for Isabelle/Isar is the Emacs-based **Proof General** [1, 2].

Some interfaces (including the shell level) offer special fonts with mathematical symbols. For those that do not, remember that ASCII-equivalents are shown in figure A.1 in the appendix.

Finally, a word about semicolons. Commands may but need not be terminated by semicolons. At the shell level it is advisable to use semicolons to enforce that a command is executed immediately; otherwise Isabelle may wait for the next keyword before it knows that the command is complete.

1.6 Getting started

Assuming you have installed Isabelle, you start it by typing `isabelle -I HOL` in a shell window.² This presents you with Isabelle's most basic ASCII interface. In addition you need to open an editor window to create theory files. While you are developing a theory, we recommend to type each command into the file first and then enter it into Isabelle by copy-and-paste, thus ensuring that you have a complete record of your theory. As mentioned above, Proof General offers a much superior interface. If you have installed Proof General, you can start it with `Isabelle`.

²Simply executing `isabelle -I` starts the default logic, which usually is already `HOL`. This is controlled by the `ISABELLE_LOGIC` setting, see *The Isabelle System Manual* for more details.

Functional Programming in HOL

Although on the surface this chapter is mainly concerned with how to write functional programs in HOL and how to verify them, most of the constructs and proof procedures introduced are general purpose and recur in any specification or verification task.

The dedicated functional programmer should be warned: HOL offers only *total functional programming* — all functions in HOL must be total; lazy data structures are not directly available. On the positive side, functions in HOL need not be computable: HOL is a specification language that goes well beyond what can be expressed as a program. However, for the time being we concentrate on the computable.

2.1 An introductory theory

Functional programming needs datatypes and functions. Both of them can be defined in a theory with a syntax reminiscent of languages like ML or Haskell. As an example consider the theory in figure 2.1. We will now examine it line by line.

```
theory ToyList = PreList:

datatype 'a list = Nil                               ("[]")
                  | Cons 'a "'a list"              (infixr "#" 65)

consts app :: "'a list => 'a list => 'a list"      (infixr "@" 65)
        rev :: "'a list => 'a list"

primrec
"[] @ ys      = ys"
"(x # xs) @ ys = x # (xs @ ys)"

primrec
"rev []      = []"
"rev (x # xs) = (rev xs) @ (x # [])"
```

Figure 2.1: A theory of lists

```
theory ToyList = PreList:
```

HOL already has a predefined theory of lists called *List* — *ToyList* is merely a small fragment of it chosen as an example. In contrast to what is recommended in §1.2, *ToyList* is not based on *Main* but on *PreList*, a theory that contains pretty much everything but lists, thus avoiding ambiguities caused by defining lists twice.

```
datatype 'a list = Nil                               ("[]")
                | Cons 'a "'a list"                 (infixr "#" 65)
```

The datatype *list* introduces two constructors *Nil* and *Cons*, the empty list and the operator that adds an element to the front of a list. For example, the term *Cons True (Cons False Nil)* is a value of type *bool list*, namely the list with the elements *True* and *False*. Because this notation becomes unwieldy very quickly, the datatype declaration is annotated with an alternative syntax: instead of *Nil* and *Cons x xs* we can write *[]* and *x # xs*. In fact, this alternative syntax is the standard syntax. Thus the list *Cons True (Cons False Nil)* becomes *True # False # []*. The annotation **infixr** means that *#* associates to the right, i.e. the term *x # y # z* is read as *x # (y # z)* and not as *(x # y) # z*.

! Syntax annotations are a powerful but completely optional feature. You could

- drop them from theory *ToyList* and go back to the identifiers *Nil* and *Cons*.

However, lists are such a central datatype that their syntax is highly customized. We recommend that novices should not use syntax annotations in their own theories.

Next, two functions *app* and *rev* are declared:

```
consts app :: "'a list ⇒ 'a list ⇒ 'a list"   (infixr "@" 65)
        rev :: "'a list ⇒ 'a list"
```

In contrast to ML, Isabelle insists on explicit declarations of all functions (keyword **consts**). (Apart from the declaration-before-use restriction, the order of items in a theory file is unconstrained.) Function *op @* is annotated with concrete syntax too. Instead of the prefix syntax *app xs ys* the infix *xs @ ys* becomes the preferred form. Both functions are defined recursively:

```
primrec
"[] @ ys      = ys"
"(x # xs) @ ys = x # (xs @ ys)"
```

```
primrec
"rev []      = []"
"rev (x # xs) = (rev xs) @ (x # [])"
```

The equations for *op @* and *rev* hardly need comments: *op @* appends two lists and *rev* reverses a list. The keyword **primrec** indicates that the recursion is of a particularly primitive kind where each recursive call peels off a

datatype constructor from one of the arguments. Thus the recursion always terminates, i.e. the function is **total**.

The termination requirement is absolutely essential in HOL, a logic of total functions. If we were to drop it, inconsistencies would quickly arise: the “definition” $f(n) = f(n) + 1$ immediately leads to $0 = 1$ by subtracting $f(n)$ on both sides.

! As we have indicated, the desire for total functions is not a gratuitously imposed restriction but an essential characteristic of HOL. It is only because of totality that reasoning in HOL is comparatively easy. More generally, the philosophy in HOL is not to allow arbitrary axioms (such as function definitions whose totality has not been proved) because they quickly lead to inconsistencies. Instead, fixed constructs for introducing types and functions are offered (such as **datatype** and **primrec**) which are guaranteed to preserve consistency.

A remark about syntax. The textual definition of a theory follows a fixed syntax with keywords like **datatype** and **end** (see Fig. ?? in Appendix A for a full list). Embedded in this syntax are the types and formulae of HOL, whose syntax is extensible, e.g. by new user-defined infix operators (see ??). To distinguish the two levels, everything HOL-specific (terms and types) should be enclosed in "...". To lessen this burden, quotation marks around a single identifier can be dropped, unless the identifier happens to be a keyword, as in

```
consts "end" :: "'a list ⇒ 'a"
```

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 An introductory proof

Assuming you have input the declarations and definitions of `ToyList` presented so far, we are ready to prove a few simple theorems. This will illustrate not just the basic proof commands but also the typical proof process.

Main goal: $rev(rev\ xs) = xs$

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

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

- establishes a new theorem to be proved, namely $rev\ (rev\ xs) = xs$,
- gives that theorem the name `rev_rev` by which it can be referred to,

- and tells Isabelle (via `[simp]`) to use the theorem (once it has been proved) as a simplification rule, i.e. all future proofs involving simplification will replace occurrences of `rev (rev xs)` by `xs`.

The name and the simplification attribute are optional.

Isabelle’s response is to print

```
proof(prove): step 0
```

```
goal (theorem rev_rev):
rev (rev xs) = xs
1. rev (rev xs) = xs
```

The first three lines tell us that we are 0 steps into the proof of theorem `rev_rev`; for compactness reasons we rarely show these initial lines in this tutorial. The remaining lines display the current proof state. Until we have finished a proof, the proof state always looks like this:

```
G
1. G1
⋮
n. Gn
```

where G is the overall goal that we are trying to prove, and the numbered lines contain the subgoals G_1, \dots, G_n that we need to prove to establish G . At `step 0` there is only one subgoal, which is identical with the overall goal. Normally G is constant and only serves as a reminder. Hence we rarely show it in this tutorial.

Let us now get back to `rev (rev xs) = xs`. Properties of recursively defined functions are best established by induction. In this case there is not much choice except to induct on `xs`:

```
apply(induct_tac xs)
```

This tells Isabelle to perform induction on variable `xs`. The suffix `tac` stands for “tactic”, a synonym for “theorem proving function”. By default, induction acts on the first subgoal. The new proof state contains two subgoals, namely the base case (`Nil`) and the induction step (`Cons`):

```
1. rev (rev []) = []
2.  $\bigwedge a \text{ list. } \text{rev}(\text{rev list}) = \text{list} \implies \text{rev}(\text{rev}(a \# \text{list})) = a \# \text{list}$ 
```

The induction step is an example of the general format of a subgoal:

```
i.  $\bigwedge x_1 \dots x_n. \text{assumptions} \implies \text{conclusion}$ 
```

The prefix of bound variables $\bigwedge x_1 \dots x_n$ can be ignored most of the time, or simply treated as a list of variables local to this subgoal. Their deeper significance is explained in §??. The *assumptions* are the local assumptions for this subgoal and *conclusion* is the actual proposition to be proved. Typical proof steps that add new assumptions are induction or case distinction.

In our example the only assumption is the induction hypothesis $rev (rev list) = list$, where $list$ is a variable name chosen by Isabelle. If there are multiple assumptions, they are enclosed in the bracket pair `[` and `]` and separated by semicolons.

Let us try to solve both goals automatically:

```
apply(auto)
```

This command tells Isabelle to apply a proof strategy called `auto` to all subgoals. Essentially, `auto` tries to “simplify” the subgoals. In our case, subgoal 1 is solved completely (thanks to the equation $rev [] = []$) and disappears; the simplified version of subgoal 2 becomes the new subgoal 1:

```
1. ... rev(rev list) = list  $\implies$  rev(rev list @ a # []) = a # list
```

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

First lemma: $rev(xs @ ys) = (rev ys) @ (rev xs)$

After abandoning the above proof attempt (at the shell level type `oops`) we start a new proof:

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

The keywords **theorem** and **lemma** are interchangeable and merely indicate the importance we attach to a proposition. In general, we use the words *theorem* and *lemma* pretty much interchangeably.

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

```
apply(induct_tac xs)
```

This time not even the base case is solved automatically:

```
apply(auto)
```

```
1. rev ys = rev ys @ []
2. ...
```

Again, we need to abandon this proof attempt and prove another simple lemma first. In the future the step of abandoning an incomplete proof before embarking on the proof of a lemma usually remains implicit.

Second lemma: $xs @ [] = xs$

This time the canonical proof procedure

```
lemma app_Nil2 [simp]: "xs @ [] = xs"
```

```
apply(induct_tac xs)
```

```
apply(auto)
```

leads to the desired message `No subgoals!`:

```
xs @ [] = xs
No subgoals!
```

We still need to confirm that the proof is now finished:

.

As a result of that final dot, Isabelle associates the lemma just proved with its name. Instead of **apply** followed by a dot, you can simply write **by**, which we do most of the time. Notice that in lemma `app_Nil2` (as printed out after the final dot) the free variable `xs` has been replaced by the unknown `?xs`, just as explained in §1.4.

Going back to the proof of the first lemma

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

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

1. $\bigwedge a \text{ list.}$

$$\text{rev (list @ ys)} = \text{rev ys @ rev list} \implies$$

$$(\text{rev ys @ rev list}) @ a \# [] = \text{rev ys @ rev list @ a \# []}$$

Now we need to remember that `@` associates to the right, and that `#` and `@` have the same priority (namely the 65 in their **infixr** annotation). Thus the conclusion really is

$$(\text{rev ys @ rev list}) @ (a \# []) = \text{rev ys @ (rev list @ (a \# []))}$$

and the missing lemma is associativity of `@`.

Third lemma: $(xs @ ys) @ zs = xs @ (ys @ zs)$

Abandoning the previous proof, the canonical proof procedure

```
lemma app_assoc [simp]: "(xs @ ys) @ zs = xs @ (ys @ zs)"
apply(induct_tac xs)
by(auto)
```

succeeds without further ado. Now we can go back and prove the first lemma

```
lemma rev_app [simp]: "rev(xs @ ys) = (rev ys) @ (rev xs)"
apply(induct_tac xs)
by(auto)
```

and then solve our main theorem:

```
theorem rev_rev [simp]: "rev(rev xs) = xs"
apply(induct_tac xs)
by(auto)
```

The final **end** tells Isabelle to close the current theory because we are finished with its development:

end

The complete proof script is shown in figure 2.2. The concatenation of figures 2.1 and 2.2 constitutes the complete theory `ToyList` and should reside in file `ToyList.thy`. It is good practice to present all declarations and definitions at the beginning of a theory to facilitate browsing.

```
lemma app_Nil2 [simp]: "xs @ [] = xs"
  apply(induct_tac xs)
  apply(auto)
  .

lemma app_assoc [simp]: "(xs @ ys) @ zs = xs @ (ys @ zs)"
  apply(induct_tac xs)
  by(auto)

lemma rev_app [simp]: "rev(xs @ ys) = (rev ys) @ (rev xs)"
  apply(induct_tac xs)
  by(auto)

theorem rev_rev [simp]: "rev(rev xs) = xs"
  apply(induct_tac xs)
  by(auto)

end
```

Figure 2.2: Proofs about lists

Review

This is the end of our toy proof. It should have familiarized you with

- the standard theorem proving procedure: state a goal (lemma or theorem); proceed with proof until a separate lemma is required; prove that lemma; come back to the original goal.
- a specific procedure that works well for functional programs: induction followed by all-out simplification via *auto*.
- a basic repertoire of proof commands.

2.3 Some helpful commands

This section discusses a few basic commands for manipulating the proof state and can be skipped by casual readers.

There are two kinds of commands used during a proof: the actual proof commands and auxiliary commands for examining the proof state and controlling the display. Simple proof commands are of the form **apply** (*method*) where **method** is a synonym for “theorem proving function”. Typical examples are *induct_tac* and *auto*. Further methods are introduced throughout the tutorial. Unless stated otherwise you may assume that a method attacks merely the first subgoal. An exception is *auto* which tries to solve all subgoals.

The most useful auxiliary commands are:

Undoing: **undo** undoes the effect of the last command; **undo** can be undone by **redo**. Both are only needed at the shell level and should not occur in the final theory.

Printing the current state: **pr** redisplay the current proof state, for example when it has disappeared off the screen.

Limiting the number of subgoals: **pr** *n* tells Isabelle to print only the first *n* subgoals from now on and redisplay the current proof state. This is helpful when there are many subgoals.

Modifying the order of subgoals: **defer** moves the first subgoal to the end and **prefer** *n* moves subgoal *n* to the front.

Printing theorems: **thm** *name*₁ ... *name*_{*n*} prints the named theorems.

Displaying types: We have already mentioned the flag **show_types** above. It can also be useful for detecting typos in formulae early on. For example, if **show_types** is set and the goal *rev(rev xs) = xs* is started, Isabelle prints the additional output

```
Variables:
  xs :: 'a list
```

which tells us that Isabelle has correctly inferred that *xs* is a variable of list type. On the other hand, had we made a typo as in *rev(re xs) = xs*, the response

```
Variables:
  re :: 'a list => 'a list
  xs :: 'a list
```

would have alerted us because of the unexpected variable *re*.

Reading terms and types: **term** *string* reads, type-checks and prints the given string as a term in the current context; the inferred type is output as well. **typ** *string* reads and prints the given string as a type in the current context.

(Re)loading theories: When you start your interaction you must open a named theory with the line `theory T = ...` `:`. Isabelle automatically loads all the required parent theories from their respective files (which may take a moment, unless the theories are already loaded and the files have not been modified).

If you suddenly discover that you need to modify a parent theory of your current theory you must first abandon your current theory (at the shell level type `kill`). After the parent theory has been modified you go back to your original theory. When its first line `theory T = ...` `:` is processed, the modified parent is reloaded automatically.

The only time when you need to load a theory by hand is when you simply want to check if it loads successfully without wanting to make use of the theory itself. This you can do by typing `use_thy "T"`.

Further commands are found in the Isabelle/Isar Reference Manual.

We now examine Isabelle’s functional programming constructs systematically, starting with inductive datatypes.

2.4 Datatypes

Inductive datatypes are part of almost every non-trivial application of HOL. First we take another look at a very important example, the datatype of lists, before we turn to datatypes in general. The section closes with a case study.

2.4.1 Lists

Lists are one of the essential datatypes in computing. Readers of this tutorial and users of HOL need to be familiar with their basic operations. Theory `ToyList` is only a small fragment of HOL’s predefined theory `List`¹. The latter contains many further operations. For example, the functions `hd` (“head”) and `tl` (“tail”) return the first element and the remainder of a list. (However, pattern-matching is usually preferable to `hd` and `tl`.) Theory `List` also contains more syntactic sugar: `[x1, ..., xn]` abbreviates `x1#...#xn#[]`. In the rest of the tutorial we always use HOL’s predefined lists.

2.4.2 The general format

The general HOL `datatype` definition is of the form

$$\mathbf{datatype} (\alpha_1, \dots, \alpha_n) t = C_1 \tau_{11} \dots \tau_{1k_1} \mid \dots \mid C_m \tau_{m1} \dots \tau_{mk_m}$$

where α_i are distinct type variables (the parameters), C_i are distinct constructor names and τ_{ij} are types; it is customary to capitalize the first letter

¹<http://isabelle.in.tum.de/library/HOL/List.html>

in constructor names. There are a number of restrictions (such as that the type should not be empty) detailed elsewhere [6]. Isabelle notifies you if you violate them.

Laws about datatypes, such as $[] \neq x\#xs$ and $(x\#xs = y\#ys) = (x=y \wedge xs=ys)$, are used automatically during proofs by simplification. The same is true for the equations in primitive recursive function definitions.

Every datatype t comes equipped with a *size* function from t into the natural numbers (see §2.5.1 below). For lists, *size* is just the length, i.e. $size [] = 0$ and $size(x \# xs) = size xs + 1$. In general, *size* returns 0 for all constructors that do not have an argument of type t , and for all other constructors $1 +$ the sum of the sizes of all arguments of type t . Note that because *size* is defined on every datatype, it is overloaded; on lists *size* is also called *length*, which is not overloaded.

2.4.3 Primitive recursion

Functions on datatypes are usually defined by recursion. In fact, most of the time they are defined by what is called **primitive recursion**. The keyword **primrec** is followed by a list of equations

$$f x_1 \dots (C y_1 \dots y_k) \dots x_n = r$$

such that C is a constructor of the datatype t and all recursive calls of f in r are of the form $f \dots y_i \dots$ for some i . Thus Isabelle immediately sees that f terminates because one (fixed!) argument becomes smaller with every recursive call. There must be exactly one equation for each constructor. Their order is immaterial. A more general method for defining total recursive functions is introduced in §3.5.

Exercise 2.4.1 Define the datatype of binary trees

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

and a function *mirror* that mirrors a binary tree by swapping subtrees (recursively). Prove

```
lemma mirror_mirror: "mirror(mirror t) = t"
```

Define a function *flatten* that flattens a tree into a list by traversing it in infix order. Prove

```
lemma "flatten(mirror t) = rev(flatten t)"
```

2.4.4 Case expressions

HOL also features *case*-expressions for analyzing elements of a datatype. For example,

```
case xs of [] => 1 | y # ys => y
```

evaluates to 1 if xs is $[]$ and to y if xs is $y \# ys$. (Since the result in both branches must be of the same type, it follows that y is of type *nat* and hence that xs is of type *nat list*.)

In general, if e is a term of the datatype t defined in §2.4.2 above, the corresponding *case*-expression analyzing e is

```
case e of      C1 x11 ... x1k1 => e1
              ⋮
              | Cm xm1 ... xmkm => em
```

- ! All constructors must be present, their order is fixed, and nested patterns are
- not supported. Violating these restrictions results in strange error messages.

Nested patterns can be simulated by nested *case*-expressions: instead of

```
case xs of [] => 1 | [x] => x | x # (y # zs) => y
```

write

```
case xs of [] => 1
| x # ys => case ys of [] => x | y # zs => y
```

Note that *case*-expressions may need to be enclosed in parentheses to indicate their scope

2.4.5 Structural induction and case distinction

Almost all the basic laws about a datatype are applied automatically during simplification. Only induction is invoked by hand via *induct_tac*, which works for any datatype. In some cases, induction is overkill and a case distinction over all constructors of the datatype suffices. This is performed by *case_tac*. A trivial example:

```
lemma "(case xs of [] => [] | y#ys => xs) = xs"
```

```
apply (case_tac xs)
```

results in the proof state

1. $xs = [] \implies (case\ xs\ of\ [] \Rightarrow [] \mid y \# ys \Rightarrow xs) = xs$
2. $\bigwedge a\ list.\ xs = a \# list \implies (case\ xs\ of\ [] \Rightarrow [] \mid y \# ys \Rightarrow xs) = xs$

which is solved automatically:

```
by (auto)
```

Note that we do not need to give a lemma a name if we do not intend to refer to it explicitly in the future.

- ! Induction is only allowed on free (or \bigwedge -bound) variables that should not occur
- among the assumptions of the subgoal; see §4.3.1 for details. Case distinction (*case_tac*) works for arbitrary terms, which need to be quoted if they are non-atomic.

2.4.6 Case study: boolean expressions

The aim of this case study is twofold: it shows how to model boolean expressions and some algorithms for manipulating them, and it demonstrates the constructs introduced above.

How can we model boolean expressions?

We want to represent boolean expressions built up from variables and constants by negation and conjunction. The following datatype serves exactly that purpose:

```
datatype boolex = Const bool | Var nat | Neg boolex
                | And boolex boolex
```

The two constants are represented by *Const True* and *Const False*. Variables are represented by terms of the form *Var n*, where *n* is a natural number (type *nat*). For example, the formula $P_0 \wedge \neg P_1$ is represented by the term *And (Var 0) (Neg (Var 1))*.

What is the value of a boolean expression?

The value of a boolean expression depends on the value of its variables. Hence the function *value* takes an additional parameter, an *environment* of type $\text{nat} \Rightarrow \text{bool}$, which maps variables to their values:

```
consts value :: "boolex  $\Rightarrow$  (nat  $\Rightarrow$  bool)  $\Rightarrow$  bool"
primrec
"value (Const b) env = b"
"value (Var x) env = env x"
"value (Neg b) env = ( $\neg$  value b env)"
"value (And b c) env = (value b env  $\wedge$  value c env)"
```

If-expressions

An alternative and often more efficient (because in a certain sense canonical) representation are so-called *If-expressions* built up from constants (*CIF*), variables (*VIF*) and conditionals (*IF*):

```
datatype ifex = CIF bool | VIF nat | IF ifex ifex ifex
```

The evaluation if If-expressions proceeds as for *boolex*:

```
consts valif :: "ifex  $\Rightarrow$  (nat  $\Rightarrow$  bool)  $\Rightarrow$  bool"
primrec
"valif (CIF b) env = b"
"valif (VIF x) env = env x"
```

```
"valif (IF b t e) env = (if valif b env then valif t env
                        else valif e env)"
```

Transformation into and of If-expressions

The type *boollex* is close to the customary representation of logical formulae, whereas *ifex* is designed for efficiency. It is easy to translate from *boollex* into *ifex*:

```
consts bool2if :: "boollex  $\Rightarrow$  ifex"
primrec
"bool2if (Const b) = CIF b"
"bool2if (Var x)   = VIF x"
"bool2if (Neg b)   = IF (bool2if b) (CIF False) (CIF True)"
"bool2if (And b c) = IF (bool2if b) (bool2if c) (CIF False)"
```

At last, we have something we can verify: that *bool2if* preserves the value of its argument:

```
lemma "valif (bool2if b) env = value b env"
```

The proof is canonical:

```
apply(induct_tac b)
by(auto)
```

In fact, all proofs in this case study look exactly like this. Hence we do not show them below.

More interesting is the transformation of If-expressions into a normal form where the first argument of *IF* cannot be another *IF* but must be a constant or variable. Such a normal form can be computed by repeatedly replacing a subterm of the form *IF (IF b x y) z u* by *IF b (IF x z u) (IF y z u)*, which has the same value. The following primitive recursive functions perform this task:

```
consts normif :: "ifex  $\Rightarrow$  ifex  $\Rightarrow$  ifex  $\Rightarrow$  ifex"
primrec
"normif (CIF b) t e = IF (CIF b) t e"
"normif (VIF x) t e = IF (VIF x) t e"
"normif (IF b t e) u f = normif b (normif t u f) (normif e u f)"

consts norm :: "ifex  $\Rightarrow$  ifex"
primrec
"norm (CIF b) = CIF b"
"norm (VIF x) = VIF x"
"norm (IF b t e) = normif b (norm t) (norm e)"
```

Their interplay is a bit tricky, and we leave it to the reader to develop an intuitive understanding. Fortunately, Isabelle can help us to verify that the transformation preserves the value of the expression:

```
theorem "valif (norm b) env = valif b env"
```

The proof is canonical, provided we first show the following simplification lemma (which also helps to understand what *normif* does):

```
lemma [simp]:
```

```
" $\forall t e. \text{valif } (\text{normif } b \ t \ e) \ \text{env} = \text{valif } (\text{IF } b \ t \ e) \ \text{env}"$ "
```

Note that the lemma does not have a name, but is implicitly used in the proof of the theorem shown above because of the *[simp]* attribute.

But how can we be sure that *norm* really produces a normal form in the above sense? We define a function that tests If-expressions for normality

```
consts normal :: "ifex  $\Rightarrow$  bool"
```

```
primrec
```

```
"normal(CIF b) = True"
```

```
"normal(VIF x) = True"
```

```
"normal(IF b t e) = (normal t  $\wedge$  normal e  $\wedge$ 
```

```
  (case b of CIF b  $\Rightarrow$  True | VIF x  $\Rightarrow$  True | IF x y z  $\Rightarrow$  False))"
```

and prove *normal (norm b)*. Of course, this requires a lemma about normality of *normif*:

```
lemma[simp]: " $\forall t e. \text{normal}(\text{normif } b \ t \ e) = (\text{normal } t \ \wedge \ \text{normal } e)"$ "
```

How does one come up with the required lemmas? Try to prove the main theorems without them and study carefully what *auto* leaves unproved. This has to provide the clue. The necessity of universal quantification ($\forall t e$) in the two lemmas is explained in §3.2

Exercise 2.4.2 We strengthen the definition of a *normal* If-expression as follows: the first argument of all *IFs* must be a variable. Adapt the above development to this changed requirement. (Hint: you may need to formulate some of the goals as implications (\longrightarrow) rather than equalities (=).)

2.5 Some basic types

2.5.1 Natural numbers

The type *nat* of natural numbers is predefined and behaves like

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

In particular, there are *case*-expressions, for example

```
case n of 0 => 0 | Suc m => m
```

primitive recursion, for example

```
consts sum :: "nat => nat"
primrec "sum 0 = 0"
        "sum (Suc n) = Suc n + sum n"
```

and induction, for example

```
lemma "sum n + sum n = n*(Suc n)"
apply(induct_tac n)
by(auto)
```

The usual arithmetic operations $+$, $-$, $*$, div , mod , min and max are predefined, as are the relations \leq and $<$. There is even a least number operation *LEAST*. For example, $(LEAST\ n.\ 1 < n) = 2$, although Isabelle does not prove this completely automatically. Note that *1* and *2* are available as abbreviations for the corresponding *Suc*-expressions. If you need the full set of numerals, see §??.

! The constant *0* and the operations $+$, $-$, $*$, min , max , \leq and $<$ are overloaded, i.e. they are available not just for natural numbers but at other types as well (see §??). 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 $+$ are declared. As a consequence, you will be unable to prove the goal (although it may take you some time to realize what has happened if `show_types` is not set). 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.

Simple arithmetic goals are proved automatically by both `auto` and the simplification methods introduced in §3.1. For example,

```
lemma "[[  $\neg m < n$ ;  $m < n+1$  ]] ==>  $m = n$ "
```

is proved automatically. The main restriction is that only addition is taken into account; other arithmetic operations and quantified formulae are ignored.

For more complex goals, there is the special method `arith` (which attacks the first subgoal). It proves arithmetic goals involving the usual logical connectives (\neg , \wedge , \vee , \longrightarrow), the relations \leq and $<$, and the operations $+$, $-$, min and max . For example,

```
lemma "min i (max j (k*k)) = max (min (k*k) i) (min i (j::nat))"
by(arith)
```

succeeds because $k*k$ can be treated as atomic. In contrast,

```
lemma "n*n = n ==> n=0  $\vee$  n=1"
```

is not even proved by `arith` because the proof relies essentially on properties of multiplication.

! The running time of `arith` is exponential in the number of occurrences of `-`, `min` and `max` because they are first eliminated by case distinctions. `arith` is incomplete even for the restricted class of formulae described above (known as “linear arithmetic”). If divisibility plays a role, it may fail to prove a valid formula, for example $m + m \neq n + n + 1$. Fortunately, such examples are rare in practice.

2.5.2 Products

HOL also has pairs: (a_1, a_2) is of type $\tau_1 \times \tau_2$ provided each a_i is of type τ_i . The components of a pair are extracted by `fst` and `snd`: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$. Tuples are simulated by pairs nested to the right: (a_1, a_2, a_3) stands for $(a_1, (a_2, a_3))$ and $\tau_1 \times \tau_2 \times \tau_3$ for $\tau_1 \times (\tau_2 \times \tau_3)$. Therefore we have $\text{fst}(\text{snd}(a_1, a_2, a_3)) = a_2$.

It is possible to use (nested) tuples as patterns in abstractions, for example $\lambda(x, y, z). x + y + z$ and $\lambda((x, y), z). x + y + z$. In addition to explicit λ -abstractions, tuple patterns can be used in most variable binding constructs. Typical examples are

```
let (x, y) = f z in (y, x)
case xs of [] => 0 | (x, y) # zs => x + y
```

Further important examples are quantifiers and sets (see §??).

! Abstraction over pairs and tuples is merely a convenient shorthand for a more complex internal representation. Thus the internal and external form of a term may differ, which can affect proofs. If you want to avoid this complication, use `fst` and `snd`, i.e. write $\lambda p. \text{fst } p + \text{snd } p$ instead of $\lambda(x, y). x + y$. See §?? for theorem proving with tuple patterns.

Note that products, like natural numbers, are datatypes, which means in particular that `induct_tac` and `case_tac` are applicable to products (see §??).

Instead of tuples with many components (where “many” is not much above 2), it is far preferable to use record types (see §??).

2.6 Definitions

A definition is simply an abbreviation, i.e. a new name for an existing construction. In particular, definitions cannot be recursive. Isabelle offers definitions on the level of types and terms. Those on the type level are called type synonyms, those on the term level are called (constant) definitions.

2.6.1 Type synonyms

Type synonyms are similar to those found in ML. Their syntax is fairly self explanatory:

```
types number      = nat
      gate        = "bool ⇒ bool ⇒ bool"
      ('a,'b)alist = "('a × 'b)list"
```

Internally all synonyms are fully expanded. As a consequence Isabelle's output never contains synonyms. Their main purpose is to improve the readability of theories. Synonyms can be used just like any other type:

```
consts nand :: gate
      exor  :: gate
```

2.6.2 Constant definitions

The above constants *nand* and *exor* are non-recursive and can therefore be defined directly by

```
defs nand_def: "nand A B ≡ ¬(A ∧ B)"
     exor_def: "exor A B ≡ A ∧ ¬B ∨ ¬A ∧ B"
```

where **defs** is a keyword and *nand_def* and *exor_def* are user-supplied names. The symbol \equiv is a special form of equality that must be used in constant definitions. Declarations and definitions can also be merged

```
constdefs nor :: gate
      "nor A B ≡ ¬(A ∨ B)"
      exor2 :: gate
      "exor2 A B ≡ (A ∨ B) ∧ (¬A ∨ ¬B)"
```

in which case the default name of each definition is *f_def*, where *f* is the name of the defined constant.

Note that pattern-matching is not allowed, i.e. each definition must be of the form $f\ x_1 \dots x_n \equiv t$. Section §3.1 explains how definitions are used in proofs.

! A common mistake when writing definitions is to introduce extra free variables on the right-hand side as in the following fictitious definition:

```
"prime p ≡ 1 < p ∧ (m dvd p → m = 1 ∨ m = p)"
```

where *dvd* means “divides”. Isabelle rejects this “definition” because of the extra *m* on the right-hand side, which would introduce an inconsistency (why?). What you should have written is

```
"prime p ≡ 1 < p ∧ (∀m. m dvd p → m = 1 ∨ m = p)"
```

More Functional Programming

The purpose of this chapter is to deepen the reader’s understanding of the concepts encountered so far and to introduce advanced forms of datatypes and recursive functions. The first two sections give a structured presentation of theorem proving by simplification (§3.1) and discuss important heuristics for induction (§3.2). They can be skipped by readers less interested in proofs. They are followed by a case study, a compiler for expressions (§3.3). Advanced datatypes, including those involving function spaces, are covered in §3.4, which closes with another case study, search trees (“tries”). Finally we introduce **recdef**, a very general form of recursive function definition which goes well beyond what **primrec** allows (§3.5).

3.1 Simplification

So far we have proved our theorems by *auto*, which “simplifies” *all* subgoals. In fact, *auto* can do much more than that, except that it did not need to so far. However, when you go beyond toy examples, you need to understand the ingredients of *auto*. This section covers the method that *auto* always applies first, namely simplification.

Simplification is one of the central theorem proving tools in Isabelle and many other systems. The tool itself is called the **simplifier**. The purpose of this section is introduce the many features of the simplifier. Anybody intending to use HOL should read this section. Later on (§4.1) we explain some more advanced features and a little bit of how the simplifier works. The serious student should read that section as well, in particular in order to understand what happened if things do not simplify as expected.

What is simplification

In its most basic form, simplification means repeated application of equations from left to right. For example, taking the rules for $@$ and applying them to the term $[0, 1] @ []$ results in a sequence of simplification steps:

$$(0\#1\#[\]) @ [\] \rightsquigarrow 0\#((1\#[\]) @ [\]) \rightsquigarrow 0\#(1\#([\] @ [\])) \rightsquigarrow 0\#1\#[\]$$

This is also known as **term rewriting** and the equations are referred to as **rewrite rules**. “Rewriting” is more honest than “simplification” because the terms do not necessarily become simpler in the process.

Simplification rules

To facilitate simplification, theorems can be declared to be simplification rules (with the help of the attribute `[simp]`), in which case proofs by simplification make use of these rules automatically. In addition the constructs **datatype** and **primrec** (and a few others) invisibly declare useful simplification rules. Explicit definitions are *not* declared simplification rules automatically!

Not merely equations but pretty much any theorem can become a simplification rule. The simplifier will try to make sense of it. For example, a theorem $\neg P$ is automatically turned into $P = \text{False}$. The details are explained in §4.1.2.

The simplification attribute of theorems can be turned on and off as follows:

```
declare theorem-name [simp]
declare theorem-name [simp del]
```

As a rule of thumb, equations that really simplify (like `rev (rev xs) = xs` and `xs @ [] = xs`) should be made simplification rules. Those of a more specific nature (e.g. distributivity laws, which alter the structure of terms considerably) should only be used selectively, i.e. they should not be default simplification rules. Conversely, it may also happen that a simplification rule needs to be disabled in certain proofs. Frequent changes in the simplification status of a theorem may indicate a badly designed theory.

! Simplification may not terminate, 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 simplification method

The general format of the simplification method is

```
simp list of modifiers
```

where the list of *modifiers* helps to fine tune the behaviour and may be empty. Most if not all of the proofs seen so far could have been performed with `simp` instead of `auto`, except that `simp` attacks only the first subgoal

and may thus need to be repeated—use `simp_all` to simplify all subgoals. Note that `simp` fails if nothing changes.

Adding and deleting simplification rules

If a certain theorem is merely needed in a few proofs by simplification, we do not need to make it a global simplification rule. Instead we can modify the set of simplification rules used in a simplification step by adding rules to it and/or deleting rules from it. The two modifiers for this are

`add`: *list of theorem names*
`del`: *list of theorem names*

In case you want to use only a specific list of theorems and ignore all others:

`only`: *list of theorem names*

Assumptions

By default, assumptions are part of the simplification process: they are used as simplification rules and are simplified themselves. For example:

```
lemma "[ xs @ zs = ys @ xs; [] @ xs = [] @ [] ] => ys = zs"
by simp
```

The second assumption simplifies to `xs = []`, which in turn simplifies the first assumption to `zs = ys`, thus reducing the conclusion to `ys = ys` and hence to `True`.

In some cases this may be too much of a good thing and may lead to nontermination:

```
lemma "∀ x. f x = g (f (g x)) => f [] = f [] @ []"
```

cannot be solved by an unmodified application of `simp` because the simplification rule `f x = g (f (g x))` extracted from the assumption does not terminate. Isabelle notices certain simple forms of nontermination but not this one. The problem can be circumvented by explicitly telling the simplifier to ignore the assumptions:

```
by(simp (no_asm))
```

There are three options that influence the treatment of assumptions:

`(no_asm)` means that assumptions are completely ignored.

`(no_asm_simp)` means that the assumptions are not simplified but are used in the simplification of the conclusion.

`(no_asm_use)` means that the assumptions are simplified but are not used in the simplification of each other or the conclusion.

Neither `(no_asm_simp)` nor `(no_asm_use)` allow to simplify the above problematic subgoal.

Note that only one of the above options is allowed, and it must precede all other arguments.

Rewriting with definitions

Constant definitions (§2.6.2) can be used as simplification rules, but by default they are not. Hence the simplifier does not expand them automatically, just as it should be: definitions are introduced for the purpose of abbreviating complex concepts. Of course we need to expand the definitions initially to derive enough lemmas that characterize the concept sufficiently for us to forget the original definition. For example, given

```
constdefs exor :: "bool  $\Rightarrow$  bool  $\Rightarrow$  bool"
           "exor A B  $\equiv$  (A  $\wedge$   $\neg$ B)  $\vee$  ( $\neg$ A  $\wedge$  B)"
```

we may want to prove

```
lemma "exor A ( $\neg$ A)"
```

Typically, the opening move consists in *unfolding* the definition(s), which we need to get started, but nothing else:

```
apply (simp only: exor_def)
```

In this particular case, the resulting goal

```
1. A  $\wedge$   $\neg$   $\neg$  A  $\vee$   $\neg$  A  $\wedge$   $\neg$  A
```

can be proved by simplification. Thus we could have proved the lemma outright

```
by (simp add: exor_def)
```

Of course we can also unfold definitions in the middle of a proof.

You should normally not turn a definition permanently into a simplification rule because this defeats the whole purpose of an abbreviation.

- ! If you have defined $f\ x\ y \equiv t$ then you can only expand occurrences of f with at least two arguments. Thus it is safer to define $f \equiv \lambda x\ y. t$.

Simplifying let-expressions

Proving a goal containing *let*-expressions almost invariably requires the *let*-constructs to be expanded at some point. Since *let-in* is just syntactic sugar for a predefined constant (called *Let*), expanding *let*-constructs means rewriting with *Let_def*:

```
lemma "(let xs = [] in xs@ys@xs) = ys"
by(simp add: Let_def)
```

If, in a particular context, there is no danger of a combinatorial explosion of nested *lets* one could even simplify with *Let_def* by default:

```
declare Let_def [simp]
```

Conditional equations

So far all examples of rewrite rules were equations. The simplifier also accepts *conditional* equations, for example

```
lemma hd_Cons_tl[simp]: "xs ≠ [] ⇒ hd xs # tl xs = xs"
by(case_tac xs, simp, simp)
```

Note the use of “,” to string together a sequence of methods. Assuming that the simplification rule $(\text{rev } xs = []) = (xs = [])$ is present as well,

```
lemma "xs ≠ [] ⇒ hd(rev xs) # tl(rev xs) = rev xs"
```

is proved by plain simplification: the conditional equation *hd_Cons_tl* above can simplify $\text{hd } (\text{rev } xs) \# \text{tl } (\text{rev } xs)$ to $\text{rev } xs$ because the corresponding precondition $\text{rev } xs \neq []$ simplifies to $xs \neq []$, which is exactly the local assumption of the subgoal.

Automatic case splits

Goals containing *if*-expressions are usually proved by case distinction on the condition of the *if*. For example the goal

```
lemma "∀xs. if xs = [] then rev xs = [] else rev xs ≠ []"
```

can be split into

1. $\forall xs. (xs = [] \longrightarrow \text{rev } xs = []) \wedge (xs \neq [] \longrightarrow \text{rev } xs \neq [])$

by a degenerate form of simplification

```
apply(simp only: split: split_if)
```

where no simplification rules are included (*only:* is followed by the empty list of theorems) but the rule *split_if* for splitting *ifs* is added (via the modifier *split:*). Because case-splitting on *ifs* is almost always the right proof strategy, the simplifier performs it automatically. Try **apply**(*simp*) on the initial goal above.

This splitting idea generalizes from *if* to *case*:

```
lemma "(case xs of [] ⇒ zs | y#ys ⇒ y#(ys@zs)) = xs@zs"
```

becomes

1. $(xs = [] \longrightarrow zs = xs @ zs) \wedge$
 $(\forall a \text{ list. } xs = a \# \text{ list} \longrightarrow a \# \text{ list} @ zs = xs @ zs)$

by typing

```
apply(simp only: split: list.split)
```

In contrast to *if*-expressions, the simplifier does not split *case*-expressions by default because this can lead to nontermination in case of recursive datatypes. Again, if the *only:* modifier is dropped, the above goal is solved,

```
by(simp split: list.split)
```

which **apply**(simp) alone will not do.

In general, every datatype *t* comes with a theorem *t.split* which can be declared to be a **split rule** either locally as above, or by giving it the *split* attribute globally:

```
declare list.split [split]
```

The *split* attribute can be removed with the *del* modifier, either locally

```
apply(simp split del: split_if)
```

or globally:

```
declare list.split [split del]
```

The above split rules intentionally only affect the conclusion of a subgoal. If you want to split an *if* or *case*-expression in the assumptions, you have to apply *split_if_asm* or *t.split_asm*:

```
lemma "if xs = [] then ys ~= [] else ys = [] ==> xs @ ys ~= []"
```

```
apply(simp only: split: split_if_asm)
```

In contrast to splitting the conclusion, this actually creates two separate subgoals (which are solved by *simp_all*):

1. $\llbracket xs = []; ys \neq [] \rrbracket \Longrightarrow [] @ ys \neq []$
2. $\llbracket xs \neq []; ys = [] \rrbracket \Longrightarrow xs @ [] \neq []$

If you need to split both in the assumptions and the conclusion, use *t.splits* which subsumes *t.split* and *t.split_asm*. Analogously, there is *if_splits*.

! The simplifier merely simplifies the condition of an *if* but not the *then* or *else* parts. The latter are simplified only after the condition reduces to *True* or *False*, or after splitting. The same is true for *case*-expressions: only the selector is simplified at first, until either the expression reduces to one of the cases or it is split.

Arithmetic

The simplifier routinely solves a small class of linear arithmetic formulae (over type `nat` and other numeric types): it only takes into account assumptions and conclusions that are (possibly negated) (in)equalities ($=$, \leq , $<$) and it only knows about addition. Thus

```
lemma "[¬ m < n; m < n+1] ==> m = n"
```

is proved by simplification, whereas the only slightly more complex

```
lemma "¬ m < n ∧ m < n+1 ==> m = n"
```

is not proved by simplification and requires `arith`.

Tracing

Using the simplifier effectively may take a bit of experimentation. Set the `trace_simp` flag to get a better idea of what is going on:

```
ML "set trace_simp"
lemma "rev [a] = []"
apply(simp)
```

produces the trace

```
Applying instance of rewrite rule:
rev (?x1 # ?xs1) == rev ?xs1 @ [?x1]
Rewriting:
rev [x] == rev [] @ [x]
Applying instance of rewrite rule:
rev [] == []
Rewriting:
rev [] == []
Applying instance of rewrite rule:
[] @ ?y == ?y
Rewriting:
[] @ [x] == [x]
Applying instance of rewrite rule:
?x3 # ?t3 = ?t3 == False
Rewriting:
[x] = [] == False
```

In more complicated cases, the trace can be quite lengthy, especially since invocations of the simplifier are often nested (e.g. when solving conditions of rewrite rules). Thus it is advisable to reset it:

```
ML "reset trace_simp"
```

3.2 Induction heuristics

The purpose of this section is to illustrate some simple heuristics for inductive proofs. The first one we have already mentioned in our initial example:

Theorems about recursive functions are proved by induction.

In case the function has more than one argument

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

When we look at the proof of " $(xs @ ys) @ zs = xs @ (ys @ zs)$ " in §2.2 we find (a) $@$ is recursive in the first argument, (b) xs occurs only as the first argument of $@$, and (c) both ys and zs occur at least once as the second argument of $@$. Hence it is natural to perform induction on xs .

The key heuristic, and the main point of this section, is to generalize 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 now illustrate the idea with an example.

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

```
consts itrev :: "'a list ⇒ 'a list ⇒ 'a list"
primrec
  "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, i.e. it can be compiled into a loop.

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, and we immediately simplify:

```
apply(induct_tac xs, simp_all)
```

Unfortunately, this is not a complete success:

```
1. ... itrev list [] = rev list ⇒ itrev list [a] = rev list @ [a]
```

Just as predicted above, the overall goal, and hence the induction hypothesis, is too weak to solve the induction step because of the fixed `[]`. The corresponding heuristic:

Generalize 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 generalization is

lemma " $itrev\ xs\ ys = rev\ xs\ @\ ys$ "

If ys is replaced by $[]$, the right-hand side simplifies to $rev\ xs$, just as required.

In this particular instance it was easy to guess the right generalization, but in more complex situations a good deal of creativity is needed. This is the main source of complications in inductive proofs.

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

1. $\bigwedge a\ list.$
 $itrev\ list\ ys = rev\ list\ @\ ys \implies$
 $itrev\ list\ (a\ \#\ ys) = rev\ list\ @\ a\ \#\ ys$

The induction hypothesis is still too weak, but this time it takes no intuition to generalize: the problem is that ys is fixed throughout the subgoal, 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:

lemma " $\forall ys. itrev\ xs\ ys = rev\ xs\ @\ ys$ "

This time induction on xs followed by simplification succeeds. This leads to another heuristic for generalization:

Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the above and does not change the provability of the goal. Because it is not always required, and may even complicate matters in some cases, this heuristic is often not applied blindly.

In general, if you have tried the above heuristics and still find your induction does not go through, and no obvious lemma suggests itself, you may need to generalize your proposition even further. This requires insight into the problem at hand and is beyond simple rules of thumb. In a nutshell: you will need to be creative. Additionally, you can read §4.3 to learn about some advanced techniques for inductive proofs.

A final point worth mentioning is the orientation of the equation we just proved: the more complex notion ($itrev$) is on the left-hand side, the simpler one (rev) on the right-hand side. This constitutes another, albeit weak heuristic that is not restricted to induction:

The right-hand side of an equation should (in some sense) be simpler than the left-hand side.

This heuristic is tricky to apply because it is not obvious that `rev xs @ ys` is simpler than `itrev xs ys`. But see what happens if you try to prove `rev xs @ ys = itrev xs ys!`

Exercise 3.2.1 In Exercise 2.4.1 we defined a function `flatten` from trees to lists. The straightforward version of `flatten` is based on `@` and is thus, like `rev`, quadratic. A linear time version of `flatten` again requires an extra argument, the accumulator:

```
consts flatten2 :: "'a tree => 'a list => 'a list"
```

Define `flatten2` and prove

```
lemma "flatten2 t [] = flatten t"
```

3.3 Case study: compiling expressions

The task is to develop a compiler from a generic type of expressions (built up from variables, constants and binary operations) to a stack machine. This generic type of expressions is a generalization of the boolean expressions in §2.4.6. This time we do not commit ourselves to a particular type of variables or values but make them type parameters. Neither is there a fixed set of binary operations: instead the expression contains the appropriate function itself.

```
types 'v binop = "'v => 'v => 'v"
datatype ('a, 'v)expr = Cex 'v
                    | Vex 'a
                    | Bex "'v binop" "('a, 'v)expr" "('a, 'v)expr"
```

The three constructors represent constants, variables and the application of a binary operation to two subexpressions.

The value of an expression w.r.t. an environment that maps variables to values is easily defined:

```
consts value :: "('a, 'v)expr => ('a => 'v) => 'v"
primrec
"value (Cex v) env = v"
"value (Vex a) env = env a"
"value (Bex f e1 e2) env = f (value e1 env) (value e2 env)"
```

The stack machine has three instructions: load a constant value onto the stack, load the contents of a certain address onto the stack, and apply a binary operation to the two topmost elements of the stack, replacing them by the result. As for `expr`, addresses and values are type parameters:

```

datatype ('a,'v) instr = Const 'v
                        | Load 'a
                        | Apply "'v binop"

```

The execution of the stack machine is modelled by a function `exec` that takes a list of instructions, a store (modelled as a function from addresses to values, just like the environment for evaluating expressions), and a stack (modelled as a list) of values, and returns the stack at the end of the execution—the store remains unchanged:

```

consts exec :: "('a,'v)instr list  $\Rightarrow$  ('a $\Rightarrow$ 'v)  $\Rightarrow$  'v list  $\Rightarrow$  'v list"
primrec
exec [] s vs = vs"
exec (i#is) s vs = (case i of
  Const v  $\Rightarrow$  exec is s (v#vs)
| Load a  $\Rightarrow$  exec is s ((s a)#vs)
| Apply f  $\Rightarrow$  exec is s ((f (hd vs) (hd(tl vs)))#(tl(tl vs))))"

```

Recall that `hd` and `tl` return the first element and the remainder of a list. Because all functions are total, `hd` is defined even for the empty list, although we do not know what the result is. Thus our model of the machine always terminates properly, although the above definition does not tell us much about the result in situations where `Apply` was executed with fewer than two elements on the stack.

The compiler is a function from expressions to a list of instructions. Its definition is pretty much obvious:

```

consts comp :: "('a,'v)expr  $\Rightarrow$  ('a,'v)instr list"
primrec
comp (Cex v)          = [Const v]"
comp (Vex a)          = [Load a]"
comp (Bex f e1 e2) = (comp e2) @ (comp e1) @ [Apply f]"

```

Now we have to prove the correctness of the compiler, i.e. that the execution of a compiled expression results in the value of the expression:

```

theorem "exec (comp e) s [] = [value e s]"

```

This theorem needs to be generalized to

```

theorem " $\forall$  vs. exec (comp e) s vs = (value e s) # vs"

```

which is proved by induction on `e` followed by simplification, once we have the following lemma about executing the concatenation of two instruction sequences:

```

lemma exec_app[simp]:
  " $\forall$  vs. exec (xs@ys) s vs = exec ys s (exec xs s vs)"
```

This requires induction on `xs` and ordinary simplification for the base cases. In the induction step, simplification leaves us with a formula that contains

two *case*-expressions over instructions. Thus we add automatic case splitting as well, which finishes the proof:

```
by(induct_tac xs, simp, simp split: instr.split)
```

Note that because *auto* performs simplification, it can also be modified in the same way *simp* can. Thus the proof can be rewritten as

```
by(induct_tac xs, auto split: instr.split)
```

Although this is more compact, it is less clear for the reader of the proof.

We could now go back and prove $\text{exec } (\text{comp } e) s [] = [\text{value } e s]$ merely by simplification with the generalized version we just proved. However, this is unnecessary because the generalized version fully subsumes its instance.

3.4 Advanced datatypes

This section presents advanced forms of **datatypes**.

3.4.1 Mutual recursion

Sometimes it is necessary to define two datatypes that depend on each other. This is called **mutual recursion**. As an example consider a language of arithmetic and boolean expressions where

- arithmetic expressions contain boolean expressions because there are conditional arithmetic expressions like “if $m < n$ then $n - m$ else $m - n$ ”, and
- boolean expressions contain arithmetic expressions because of comparisons like “ $m < n$ ”.

In Isabelle this becomes

```
datatype 'a aexp = IF "'a bexp" "'a aexp" "'a aexp"
  | Sum "'a aexp" "'a aexp"
  | Diff "'a aexp" "'a aexp"
  | Var 'a
  | Num nat
and      'a bexp = Less "'a aexp" "'a aexp"
  | And "'a bexp" "'a bexp"
  | Neg "'a bexp"
```

Type *aexp* is similar to *expr* in §3.3, except that we have fixed the values to be of type *nat* and that we have fixed the two binary operations *aexp.Sum* and *Diff*. Boolean expressions can be arithmetic comparisons, conjunctions and negations. The semantics is fixed via two evaluation functions

```

consts evala :: "'a aexp  $\Rightarrow$  ('a  $\Rightarrow$  nat)  $\Rightarrow$  nat"
          evalb :: "'a bexp  $\Rightarrow$  ('a  $\Rightarrow$  nat)  $\Rightarrow$  bool"

```

that take an expression and an environment (a mapping from variables 'a to values nat) and return its arithmetic/boolean value. Since the datatypes are mutually recursive, so are functions that operate on them. Hence they need to be defined in a single **primrec** section:

primrec

```

"evala (IF b a1 a2) env =
  (if evalb b env then evala a1 env else evala a2 env)"
"evala (Sum a1 a2) env = evala a1 env + evala a2 env"
"evala (Diff a1 a2) env = evala a1 env - evala a2 env"
"evala (Var v) env = env v"
"evala (Num n) env = n"

"evalb (Less a1 a2) env = (evala a1 env < evala a2 env)"
"evalb (And b1 b2) env = (evalb b1 env  $\wedge$  evalb b2 env)"
"evalb (Neg b) env = ( $\neg$  evalb b env)"

```

In the same fashion we also define two functions that perform substitution:

```

consts subst a :: "'a  $\Rightarrow$  'b aexp)  $\Rightarrow$  'a aexp  $\Rightarrow$  'b aexp"
          subst b :: "'a  $\Rightarrow$  'b bexp)  $\Rightarrow$  'a bexp  $\Rightarrow$  'b bexp"

```

The first argument is a function mapping variables to expressions, the substitution. It is applied to all variables in the second argument. As a result, the type of variables in the expression may change from 'a to 'b. Note that there are only arithmetic and no boolean variables.

primrec

```

"subst a s (IF b a1 a2) =
  IF (subst b s b) (subst a s a1) (subst a s a2)"
"subst a s (Sum a1 a2) = Sum (subst a s a1) (subst a s a2)"
"subst a s (Diff a1 a2) = Diff (subst a s a1) (subst a s a2)"
"subst a s (Var v) = s v"
"subst a s (Num n) = Num n"

"subst b s (Less a1 a2) = Less (subst a s a1) (subst a s a2)"
"subst b s (And b1 b2) = And (subst b s b1) (subst b s b2)"
"subst b s (Neg b) = Neg (subst b s b)"

```

Now we can prove a fundamental theorem about the interaction between evaluation and substitution: applying a substitution s to an expression a and evaluating the result in an environment env yields the same result as evaluation a in the environment that maps every variable x to the value of $s(x)$ under env . If you try to prove this separately for arithmetic or boolean expressions (by induction), you find that you always need the other theorem

in the induction step. Therefore you need to state and prove both theorems simultaneously:

```
lemma "evala (subst s a) env = evala a (λx. evala (s x) env) ∧
      evalb (subst s b) env = evalb b (λx. evala (s x) env)"
apply (induct_tac a and b)
```

The resulting 8 goals (one for each constructor) are proved in one fell swoop: by `simp_all`

In general, given n mutually recursive datatypes τ_1, \dots, τ_n , an inductive proof expects a goal of the form

$$P_1(x_1) \wedge \dots \wedge P_n(x_n)$$

where each variable x_i is of type τ_i . Induction is started by

```
apply (induct_tac x_1 and ... and x_n)
```

Exercise 3.4.1 Define a function *norma* of type `'a aexp ⇒ 'a aexp` that replaces *IFs* with complex boolean conditions by nested *IFs* where each condition is a *Less* — *And* and *Neg* should be eliminated completely. Prove that *norma* preserves the value of an expression and that the result of *norma* is really normal, i.e. no more *And*s and *Neg*s occur in it. (*Hint*: proceed as in §2.4.6).

3.4.2 Nested recursion

So far, all datatypes had the property that on the right-hand side of their definition they occurred only at the top-level, i.e. directly below a constructor. This is not the case any longer for the following model of terms where function symbols can be applied to a list of arguments:

```
datatype ('a,'b)"term" = Var 'a | App 'b "('a,'b)term list"
```

Note that we need to quote *term* on the left to avoid confusion with the command `term`. Parameter `'a` is the type of variables and `'b` the type of function symbols. A mathematical term like $f(x, g(y))$ becomes `App f [Var x, App g [Var y]]`, where f, g, x, y are suitable values, e.g. numbers or strings.

What complicates the definition of *term* is the nested occurrence of *term* inside *list* on the right-hand side. In principle, nested recursion can be eliminated in favour of mutual recursion by unfolding the offending datatypes, here *list*. The result for *term* would be something like

```
datatype ('a,'b)"term" = Var 'a | App 'b "('a,'b)term_list"
and ('a,'b)term_list = Nil | Cons "('a,'b)term" "('a,'b)term_list"
```

Although we do not recommend this unfolding to the user, it shows how to simulate nested recursion by mutual recursion. Now we return to the initial definition of `term` using nested recursion.

Let us define a substitution function on terms. Because terms involve term lists, we need to define two substitution functions simultaneously:

consts

```
subst :: "('a ⇒ ('a, 'b) term) ⇒ ('a, 'b) term      ⇒ ('a, 'b) term"
subst :: "('a ⇒ ('a, 'b) term) ⇒ ('a, 'b) term list ⇒ ('a, 'b) term list"
```

primrec

```
"subst s (Var x) = s x"
subst_App:
"subst s (App f ts) = App f (subst s ts)"

"subst s [] = []"
"subst s (t # ts) = subst s t # subst s ts"
```

(Please ignore the label `subst_App` for the moment.)

Similarly, when proving a statement about terms inductively, we need to prove a related statement about term lists simultaneously. For example, the fact that the identity substitution does not change a term needs to be strengthened and proved as follows:

```
lemma "subst Var t = (t :: ('a, 'b) term) ∧
      subst Var ts = (ts :: ('a, 'b) term list)"
by(induct_tac t and ts, simp_all)
```

Note that `Var` is the identity substitution because by definition it leaves variables unchanged: `subst Var (Var x) = Var x`. Note also that the type annotations are necessary because otherwise there is nothing in the goal to enforce that both halves of the goal talk about the same type parameters `('a, 'b)`. As a result, induction would fail because the two halves of the goal would be unrelated.

Exercise 3.4.2 The fact that substitution distributes over composition can be expressed roughly as follows:

$$\text{subst } (f \circ g) \ t = \text{subst } f \ (\text{subst } g \ t)$$

Correct this statement (you will find that it does not type-check), strengthen it, and prove it. (Note: `o` is function composition; its definition is found in theorem `o_def`).

Exercise 3.4.3 Define a function `trev` of type `('a, 'b) term ⇒ ('a, 'b) term` that recursively reverses the order of arguments of all function symbols in a term. Prove that `trev (trev t) = t`.

The experienced functional programmer may feel that our above definition of *subst* is unnecessarily complicated in that *substs* is completely unnecessary. The *App*-case can be defined directly as

```
subst s (App f ts) = App f (map (subst s) ts)
```

where *map* is the standard list function such that $\text{map } f [x_1, \dots, x_n] = [f x_1, \dots, f x_n]$. This is true, but Isabelle insists on the above fixed format. Fortunately, we can easily *prove* that the suggested equation holds:

```
lemma [simp]: "subst s (App f ts) = App f (map (subst s) ts)"
by(induct_tac ts, simp_all)
```

What is more, we can now disable the old defining equation as a simplification rule:

```
declare subst_App [simp del]
```

The advantage is that now we have replaced *substs* by *map*, we can profit from the large number of pre-proved lemmas about *map*. Unfortunately inductive proofs about type *term* are still awkward because they expect a conjunction. One could derive a new induction principle as well (see §4.3.3), but turns out to be simpler to define functions by **recdef** instead of **primrec**. The details are explained in §4.2 below.

Of course, you may also combine mutual and nested recursion. For example, constructor *Sum* in §3.4.1 could take a list of expressions as its argument: *Sum* "'a aexp list".

3.4.3 The limits of nested recursion

How far can we push nested recursion? By the unfolding argument above, we can reduce nested to mutual recursion provided the nested recursion only involves previously defined datatypes. This does not include functions:

```
datatype t = C "t  $\Rightarrow$  bool"
```

is a real can of worms: in HOL it must be ruled out because it requires a type *t* such that *t* and its power set $t \Rightarrow \text{bool}$ have the same cardinality—an impossibility. For the same reason it is not possible to allow recursion involving the type *set*, which is isomorphic to $t \Rightarrow \text{bool}$.

Fortunately, a limited form of recursion involving function spaces is permitted: the recursive type may occur on the right of a function arrow, but never on the left. Hence the above can of worms is ruled out but the following example of a potentially infinitely branching tree is accepted:

```
datatype ('a, 'i)bigtree = Tip | Branch 'a "'i  $\Rightarrow$  ('a, 'i)bigtree"
```

Parameter *'a* is the type of values stored in the *Branches* of the tree, whereas *'i* is the index type over which the tree branches. If *'i* is instantiated to *bool*, the result is a binary tree; if it is instantiated to *nat*, we have an

infinitely branching tree because each node has as many subtrees as there are natural numbers. How can we possibly write down such a tree? Using functional notation! For example, the term

```
Branch 0 (λi. Branch i (λn. Tip))
```

of type (nat, nat) *bigtree* is the tree whose root is labeled with 0 and whose i th subtree is labeled with i and has merely *Tips* as further subtrees.

Function *map_bt* applies a function to all labels in a *bigtree*:

```
consts map_bt :: "('a ⇒ 'b) ⇒ ('a,'i)bigtree ⇒ ('b,'i)bigtree"
primrec
"map_bt f Tip          = Tip"
"map_bt f (Branch a F) = Branch (f a) (λi. map_bt f (F i))"
```

This is a valid **primrec** definition because the recursive calls of *map_bt* involve only subtrees obtained from F , i.e. the left-hand side. Thus termination is assured. The seasoned functional programmer might have written $map_bt\ f \circ F$ instead of $\lambda i. map_bt\ f\ (F\ i)$, but the former is not accepted by Isabelle because the termination proof is not as obvious since *map_bt* is only partially applied.

The following lemma has a canonical proof

```
lemma "map_bt (g ∘ f) T = map_bt g (map_bt f T)"
by(induct_tac T, simp_all)
```

but it is worth taking a look at the proof state after the induction step to understand what the presence of the function type entails:

1. $map_bt\ (g \circ f)\ Tip = map_bt\ g\ (map_bt\ f\ Tip)$
2. $\bigwedge a\ F.$
 $\forall x. map_bt\ (g \circ f)\ (F\ x) = map_bt\ g\ (map_bt\ f\ (F\ x)) \implies$
 $map_bt\ (g \circ f)\ (Branch\ a\ F) = map_bt\ g\ (map_bt\ f\ (Branch\ a\ F))$

If you need nested recursion on the left of a function arrow, there are alternatives to pure HOL: LCF [8] is a logic where types like

```
datatype lam = C "lam → lam"
```

do indeed make sense (but note the intentionally different arrow \rightarrow). There is even a version of LCF on top of HOL, called HOLCF [5].

3.4.4 Case study: Tries

Tries are a classic search tree data structure [4] for fast indexing with strings. Figure 3.1 gives a graphical example of a trie containing the words “all”, “an”, “ape”, “can”, “car” and “cat”. When searching a string in a trie, the letters of the string are examined sequentially. Each letter determines which

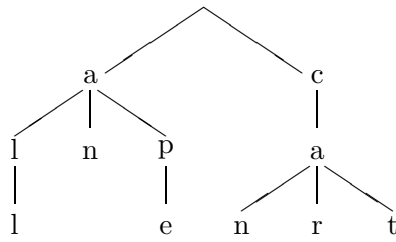


Figure 3.1: A sample trie

subtrie to search next. In this case study we model tries as a datatype, define a lookup and an update function, and prove that they behave as expected.

Proper tries associate some value with each string. Since the information is stored only in the final node associated with the string, many nodes do not carry any value. This distinction is captured by the following predefined datatype (from theory *Option*, which is a parent of *Main*):

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

To minimize running time, each node of a trie should contain an array that maps letters to subtries. We have chosen a (sometimes) more space efficient representation where the subtries are held in an association list, i.e. a list of (letter,trie) pairs. Abstracting over the alphabet *'a* and the values *'v* we define a trie as follows:

```
datatype ('a,'v)trie = Trie "'v option" "('a * ('a,'v)trie)list"
```

The first component is the optional value, the second component the association list of subtries. This is an example of nested recursion involving products, which is fine because products are datatypes as well. We define two selector functions:

```
consts value :: "('a,'v)trie ⇒ 'v option"
        alist :: "('a,'v)trie ⇒ ('a * ('a,'v)trie)list"
primrec "value(Trie ov al) = ov"
primrec "alist(Trie ov al) = al"
```

Association lists come with a generic lookup function:

```
consts  assoc :: "('key * 'val)list ⇒ 'key ⇒ 'val option"
primrec "assoc [] x = None"
        "assoc (p#ps) x =
          (let (a,b) = p in if a=x then Some b else assoc ps x)"
```

Now we can define the lookup function for tries. It descends into the trie examining the letters of the search string one by one. As recursion on lists is simpler than on tries, let us express this as primitive recursion on the search string argument:

```

consts lookup :: "('a,'v)trie ⇒ 'a list ⇒ 'v option"
primrec "lookup t [] = value t"
         "lookup t (a#as) = (case assoc (alist t) a of
                               None ⇒ None
                               | Some at ⇒ lookup at as)"

```

As a first simple property we prove that looking up a string in the empty trie `Trie None []` always returns `None`. The proof merely distinguishes the two cases whether the search string is empty or not:

```

lemma [simp]: "lookup (Trie None []) as = None"
by(case_tac as, simp_all)

```

Things begin to get interesting with the definition of an update function that adds a new (string,value) pair to a trie, overwriting the old value associated with that string:

```

consts update :: "('a,'v)trie ⇒ 'a list ⇒ 'v ⇒ ('a,'v)trie"
primrec
  "update t [] v = Trie (Some v) (alist t)"
  "update t (a#as) v =
    (let tt = (case assoc (alist t) a of
                None ⇒ Trie None [] | Some at ⇒ at)
     in Trie (value t) ((a,update tt as v)#alist t))"

```

The base case is obvious. In the recursive case the subtrie `tt` associated with the first letter `a` is extracted, recursively updated, and then placed in front of the association list. The old subtrie associated with `a` is still in the association list but no longer accessible via `assoc`. Clearly, there is room here for optimizations!

Before we start on any proofs about `update` we tell the simplifier to expand all `lets` and to split all `case`-constructs over options:

```

declare Let_def[simp] option.split[split]

```

The reason becomes clear when looking (probably after a failed proof attempt) at the body of `update`: it contains both `let` and a case distinction over type `option`.

Our main goal is to prove the correct interaction of `update` and `lookup`:

```

theorem "∀ t v bs. lookup (update t as v) bs =
          (if as=bs then Some v else lookup t bs)"

```

Our plan is to induct on `as`; hence the remaining variables are quantified. From the definitions it is clear that induction on either `as` or `bs` is required. The choice of `as` is merely guided by the intuition that simplification of `lookup` might be easier if `update` has already been simplified, which can only happen if `as` is instantiated. The start of the proof is completely conventional:

```
apply(induct_tac as, auto)
```

Unfortunately, this time we are left with three intimidating looking subgoals:

1. ... \implies `lookup ... bs = lookup t bs`
2. ... \implies `lookup ... bs = lookup t bs`
3. ... \implies `lookup ... bs = lookup t bs`

Clearly, if we want to make headway we have to instantiate `bs` as well now. It turns out that instead of induction, case distinction suffices:

```
by(case_tac[!] bs, auto)
```

All methods ending in `tac` take an optional first argument that specifies the range of subgoals they are applied to, where `[!]` means all subgoals, i.e. `[1-3]` in our case. Individual subgoal numbers, e.g. `[2]` are also allowed.

This proof may look surprisingly straightforward. However, note that this comes at a cost: the proof script is unreadable because the intermediate proof states are invisible, and we rely on the (possibly brittle) magic of `auto` (`simp_all` will not do—try it) to split the subgoals of the induction up in such a way that case distinction on `bs` makes sense and solves the proof. Part ?? shows you how to write readable and stable proofs.

Exercise 3.4.4 Write an improved version of `update` that does not suffer from the space leak in the version above. Prove the main theorem for your improved `update`.

Exercise 3.4.5 Write a function to `delete` entries from a trie. An easy solution is a clever modification of `update` which allows both insertion and deletion with a single function. Prove (a modified version of) the main theorem above. Optimize you function such that it shrinks tries after deletion, if possible.

3.5 Total recursive functions

Although many total functions have a natural primitive recursive definition, this is not always the case. Arbitrary total recursive functions can be defined by means of **recdef**: you can use full pattern-matching, recursion need not involve datatypes, and termination is proved by showing that the arguments of all recursive calls are smaller in a suitable (user supplied) sense.

3.5.1 Defining recursive functions

Here is a simple example, the Fibonacci function:

```
consts fib :: "nat  $\Rightarrow$  nat"
recdef fib "measure( $\lambda$ n. n)"
```

```

"fib 0 = 0"
"fib 1 = 1"
"fib (Suc(Suc x)) = fib x + fib (Suc x)"

```

The definition of *fib* is accompanied by a **measure function** $\lambda n. n$ which maps the argument of *fib* to a natural number. The requirement is that in each equation the measure of the argument on the left-hand side is strictly greater than the measure of the argument of each recursive call. In the case of *fib* this is obviously true because the measure function is the identity and *Suc (Suc x)* is strictly greater than both *x* and *Suc x*.

Slightly more interesting is the insertion of a fixed element between any two elements of a list:

```

consts sep :: "'a × 'a list ⇒ 'a list"
recdef sep "measure (λ(a,xs). length xs)"
  "sep(a, []) = []"
  "sep(a, [x]) = [x]"
  "sep(a, x#y#zs) = x # a # sep(a,y#zs)"

```

This time the measure is the length of the list, which decreases with the recursive call; the first component of the argument tuple is irrelevant.

Pattern matching need not be exhaustive:

```

consts last :: "'a list ⇒ 'a"
recdef last "measure (λxs. length xs)"
  "last [x] = x"
  "last (x#y#zs) = last (y#zs)"

```

Overlapping patterns are disambiguated by taking the order of equations into account, just as in functional programming:

```

consts sep1 :: "'a × 'a list ⇒ 'a list"
recdef sep1 "measure (λ(a,xs). length xs)"
  "sep1(a, x#y#zs) = x # a # sep1(a,y#zs)"
  "sep1(a, xs) = xs"

```

This defines exactly the same function as *sep* above, i.e. *sep1* = *sep*.

! **recdef** only takes the first argument of a (curried) recursive function into account. This means both the termination measure and pattern matching can only use that first argument. In general, you will therefore have to combine several arguments into a tuple. In case only one argument is relevant for termination, you can also rearrange the order of arguments as in the following definition:

```

consts sep2 :: "'a list ⇒ 'a ⇒ 'a list"
recdef sep2 "measure length"
  "sep2 (x#y#zs) = (λa. x # a # sep2 zs a)"
  "sep2 xs = (λa. xs)"

```

Because of its pattern-matching syntax, **recdef** is also useful for the definition of non-recursive functions:

```
consts swap12 :: "'a list ⇒ 'a list"
recdef swap12 "{}"
  "swap12 (x#y#zs) = y#x#zs"
  "swap12 zs      = zs"
```

For non-recursive functions the termination measure degenerates to the empty set $\{\}$.

3.5.2 Proving termination

When a function is defined via **recdef**, Isabelle tries to prove its termination with the help of the user-supplied measure. All of the above examples are simple enough that Isabelle can prove automatically that the measure of the argument goes down in each recursive call. As a result, $f.simps$ will contain the defining equations (or variants derived from them) as theorems. For example, look (via **thm**) at $sep.simps$ and $sep1.simps$ to see that they define the same function. What is more, those equations are automatically declared as simplification rules.

In general, Isabelle may not be able to prove all termination condition (there is one for each recursive call) automatically. For example, termination of the following artificial function

```
consts f :: "nat × nat ⇒ nat"
recdef f "measure(λ(x,y). x-y)"
  "f(x,y) = (if x ≤ y then x else f(x,y+1))"
```

is not proved automatically (although maybe it should be). Isabelle prints a kind of error message showing you what it was unable to prove. You will then have to prove it as a separate lemma before you attempt the definition of your function once more. In our case the required lemma is the obvious one:

```
lemma termi_lem: "¬ x ≤ y ⇒ x - Suc y < x - y"
```

It was not proved automatically because of the special nature of $-$ on nat . This requires more arithmetic than is tried by default:

```
by(arith)
```

Because **recdef**'s termination prover involves simplification, we include with our second attempt the hint to use $termi_lem$ as a simplification rule:

```
consts g :: "nat × nat ⇒ nat"
recdef g "measure(λ(x,y). x-y)"
  "g(x,y) = (if x ≤ y then x else g(x,y+1))"
(hints recdef_simp: termi_lem)
```

This time everything works fine. Now `g.simps` contains precisely the stated recursion equation for `g` and they are simplification rules. Thus we can automatically prove

```
theorem "g(1,0) = g(1,1)"
by(simp)
```

More exciting theorems require induction, which is discussed below.

If the termination proof requires a new lemma that is of general use, you can turn it permanently into a simplification rule, in which case the above **hint** is not necessary. But our `termi_lem` is not sufficiently general to warrant this distinction.

The attentive reader may wonder why we chose to call our function `g` rather than `f` the second time around. The reason is that, despite the failed termination proof, the definition of `f` did not fail, and thus we could not define it a second time. However, all theorems about `f`, for example `f.simps`, carry as a precondition the unproved termination condition. Moreover, the theorems `f.simps` are not simplification rules. However, this mechanism allows a delayed proof of termination: instead of proving `termi_lem` up front, we could prove it later on and then use it to remove the preconditions from the theorems about `f`. In most cases this is more cumbersome than proving things up front.

Although all the above examples employ measure functions, **recdef** allows arbitrary wellfounded relations. For example, termination of Ackermann's function requires the lexicographic product `<*lex*>`:

```
consts ack :: "nat × nat ⇒ nat"
recdef ack "measure(λm. m) <*lex*> measure(λn. n)"
  "ack(0,n)          = Suc n"
  "ack(Suc m,0)     = ack(m, 1)"
  "ack(Suc m,Suc n) = ack(m,ack(Suc m,n))"
```

For details see the manual [6] and the examples in the library.

3.5.3 Simplification with **recdef**

Once we have succeeded in proving all termination conditions, the recursion equations become simplification rules, just as with **primrec**. In most cases this works fine, but there is a subtle problem that must be mentioned: simplification may not terminate because of automatic splitting of `if`. Let us look at an example:

```
consts gcd :: "nat × nat ⇒ nat"
recdef gcd "measure (λ(m,n).n)"
  "gcd (m, n) = (if n=0 then m else gcd(n, m mod n))"
```

According to the measure function, the second argument should decrease with each recursive call. The resulting termination condition

$$n \neq 0 \implies m \bmod n < n$$

is provided automatically because it is already present as a lemma in the arithmetic library. Thus the recursion equation becomes a simplification rule. Of course the equation is nonterminating if we are allowed to unfold the recursive call inside the *if* branch, which is why programming languages and our simplifier don't do that. Unfortunately the simplifier does something else which leads to the same problem: it splits *ifs* if the condition simplifies to neither *True* nor *False*. For example, simplification reduces

$$\text{gcd } (m, n) = k$$

in one step to

$$(\text{if } n = 0 \text{ then } m \text{ else } \text{gcd } (n, m \bmod n)) = k$$

where the condition cannot be reduced further, and splitting leads to

$$(n = 0 \longrightarrow m = k) \wedge (n \neq 0 \longrightarrow \text{gcd } (n, m \bmod n) = k)$$

Since the recursive call $\text{gcd } (n, m \bmod n)$ is no longer protected by an *if*, it is unfolded again, which leads to an infinite chain of simplification steps. Fortunately, this problem can be avoided in many different ways.

The most radical solution is to disable the offending *split_if* as shown in the section on case splits in §3.1. However, we do not recommend this because it means you will often have to invoke the rule explicitly when *if* is involved.

If possible, the definition should be given by pattern matching on the left rather than *if* on the right. In the case of *gcd* the following alternative definition suggests itself:

```
consts gcd1 :: "nat × nat ⇒ nat"
redef gcd1 "measure (λ(m,n).n)"
  "gcd1 (m, 0) = m"
  "gcd1 (m, n) = gcd1(n, m mod n)"
```

Note that the order of equations is important and hides the side condition $n \neq 0$. Unfortunately, in general the case distinction may not be expressible by pattern matching.

A very simple alternative is to replace *if* by *case*, which is also available for *bool* but is not split automatically:

```
consts gcd2 :: "nat × nat ⇒ nat"
redef gcd2 "measure (λ(m,n).n)"
  "gcd2(m,n) = (case n=0 of True ⇒ m | False ⇒ gcd2(n,m mod n))"
```

In fact, this is probably the neatest solution next to pattern matching.

A final alternative is to replace the offending simplification rules by derived conditional ones. For *gcd* it means we have to prove

```
lemma [simp]: "gcd (m, 0) = m"
by(simp)
```

```
lemma [simp]: "n ≠ 0 ⇒ gcd(m, n) = gcd(n, m mod n)"
by (simp)
```

after which we can disable the original simplification rule:

```
declare gcd.simps [simp del]
```

3.5.4 Induction

Assuming we have defined our function such that Isabelle could prove termination and that the recursion equations (or some suitable derived equations) are simplification rules, we might like to prove something about our function. Since the function is recursive, the natural proof principle is again induction. But this time the structural form of induction that comes with datatypes is unlikely to work well—otherwise we could have defined the function by **primrec**. Therefore **recdef** automatically proves a suitable induction rule *f.induct* that follows the recursion pattern of the particular function *f*. We call this **recursion induction**. Roughly speaking, it requires you to prove for each **recdef** equation that the property you are trying to establish holds for the left-hand side provided it holds for all recursive calls on the right-hand side. Here is a simple example

```
lemma "map f (sep(x,xs)) = sep(f x, map f xs)"
```

involving the predefined *map* functional on lists: *map f xs* is the result of applying *f* to all elements of *xs*. We prove this lemma by recursion induction w.r.t. *sep*:

```
apply (induct_tac x xs rule: sep.induct)
```

The resulting proof state has three subgoals corresponding to the three clauses for *sep*:

1. $\bigwedge a. \text{map } f (\text{sep } (a, [])) = \text{sep } (f a, \text{map } f [])$
2. $\bigwedge a x. \text{map } f (\text{sep } (a, [x])) = \text{sep } (f a, \text{map } f [x])$
3. $\bigwedge a x y zs. \text{map } f (\text{sep } (a, y \# zs)) = \text{sep } (f a, \text{map } f (y \# zs)) \implies \text{map } f (\text{sep } (a, x \# y \# zs)) = \text{sep } (f a, \text{map } f (x \# y \# zs))$

The rest is pure simplification:

```
by simp_all
```

Try proving the above lemma by structural induction, and you find that you need an additional case distinction. What is worse, the names of variables are invented by Isabelle and have nothing to do with the names in the definition of *sep*.

In general, the format of invoking recursion induction is

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

where $x_1 \dots x_n$ is a list of free variables in the subgoal and f the name of a function that takes an n -tuple. Usually the subgoal will contain the term $f x_1 \dots x_n$ but this need not be the case. The induction rules do not mention f at all. For example *sep.induct*

$$\begin{aligned} & \llbracket \bigwedge a. P a []; \\ & \quad \bigwedge a x. P a [x]; \\ & \quad \bigwedge a x y zs. P a (y \# zs) \implies P a (x \# y \# zs) \rrbracket \\ & \implies P u v \end{aligned}$$

merely says that in order to prove a property P of u and v you need to prove it for the three cases where v is the empty list, the singleton list, and the list with at least two elements (in which case you may assume it holds for the tail of that list).

Advanced Simplification, Recursion, and Induction

Although we have already learned a lot about simplification, recursion and induction, there are some advanced proof techniques that we have not covered yet and which are worth knowing about if you intend to become a serious (human) theorem prover. The three sections of this chapter are almost independent of each other and can be read in any order. Only the notion of *congruence rules*, introduced in the section on simplification, is required for parts of the section on recursion.

4.1 Simplification

This section discusses some additional nifty features not covered so far and gives a short introduction to the simplification process itself. The latter is helpful to understand why a particular rule does or does not apply in some situation.

4.1.1 Advanced features

Congruence rules

It is hardwired into the simplifier that while simplifying the conclusion Q of $P \implies Q$ it is legal to make use of the assumptions P . This kind of contextual information can also be made available for other operators. For example, $xs = [] \longrightarrow xs @ xs = xs$ simplifies to True because we may use $xs = []$ when simplifying $xs @ xs = xs$. The generation of contextual information during simplification is controlled by so-called **congruence rules**. This is the one for \longrightarrow :

$$\llbracket P = P'; P' \implies Q = Q' \rrbracket \implies (P \longrightarrow Q) = (P' \longrightarrow Q')$$

It should be read as follows: In order to simplify $P \longrightarrow Q$ to $P' \longrightarrow Q'$, simplify P to P' and assume P' when simplifying Q to Q' .

Here are some more examples. The congruence rules for bounded quantifiers supply contextual information about the bound variable:

$$\begin{aligned} & \llbracket A = B; \bigwedge x. x \in B \implies P x = Q x \rrbracket \\ & \implies (\forall x \in A. P x) = (\forall x \in B. Q x) \end{aligned}$$

The congruence rule for conditional expressions supply contextual information for simplifying the arms:

$$\begin{aligned} & \llbracket b = c; c \implies x = u; \neg c \implies y = v \rrbracket \\ & \implies (\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v) \end{aligned}$$

A congruence rule can also *prevent* simplification of some arguments. Here is an alternative congruence rule for conditional expressions:

$$b = c \implies (\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } x \text{ else } y)$$

Only the first argument is simplified; the others remain unchanged. This makes simplification much faster and is faithful to the evaluation strategy in programming languages, which is why this is the default congruence rule for *if*. Analogous rules control the evaluation of *case* expressions.

You can declare your own congruence rules with the attribute *cong*, either globally, in the usual manner,

declare *theorem-name* [*cong*]

or locally in a *simp* call by adding the modifier

cong: *list of theorem names*

The effect is reversed by *cong del* instead of *cong*.

! The congruence rule *conj_cong*

$$\llbracket P = P'; P' \implies Q = Q' \rrbracket \implies (P \wedge Q) = (P' \wedge Q')$$

is occasionally useful but not a default rule; you have to use it explicitly.

Permutative rewrite rules

An equation is a **permutative rewrite rule** if the left-hand side and right-hand side are the same up to renaming of variables. The most common permutative rule is commutativity: $x + y = y + x$. Other examples include $x - y - z = x - z - y$ in arithmetic and $\text{insert } x (\text{insert } y A) = \text{insert } y (\text{insert } x A)$ for sets. Such rules are problematic because once they apply, they can be used forever. The simplifier is aware of this danger and treats permutative rules by means of a special strategy, called **ordered rewriting**: a permutative rewrite rule is only applied if the term becomes “smaller” (w.r.t. some fixed lexicographic ordering on terms). For example, commutativity rewrites $b + a$ to $a + b$, but then stops because $a + b$ is strictly smaller than $b + a$. Permutative rewrite rules can be turned into

simplification rules in the usual manner via the *simp* attribute; the simplifier recognizes their special status automatically.

Permutative rewrite rules are most effective in the case of associative-commutative operators. (Associativity by itself is not permutative.) When dealing with an AC-operator f , keep the following points in mind:

- The associative law must always be oriented from left to right, namely $f(f(x, y), z) = f(x, f(y, z))$. The opposite orientation, if used with commutativity, can lead to nontermination.
- To complete your set of rewrite rules, you must add not just associativity (A) and commutativity (C) but also a derived rule, **left-commutativity** (LC): $f(x, f(y, z)) = f(y, f(x, z))$.

Ordered rewriting with the combination of A, C, and LC sorts a term lexicographically:

$$f(f(b, c), a) \xrightarrow{A} f(b, f(c, a)) \xrightarrow{C} f(b, f(a, c)) \xrightarrow{LC} f(a, f(b, c))$$

Note that ordered rewriting for $+$ and $*$ on numbers is rarely necessary because the builtin arithmetic capabilities often take care of this.

4.1.2 How it works

Roughly speaking, the simplifier proceeds bottom-up (subterms are simplified first) and a conditional equation is only applied if its condition could be proved (again by simplification). Below we explain some special

Higher-order patterns

Local assumptions

The preprocessor

4.2 Advanced forms of recursion

In §3.4.2 we defined the datatype of terms

```
datatype ('a, 'b)term = Var 'a | App 'b "('a, 'b)term list"
```

and closed with the observation that the associated schema for the definition of primitive recursive functions leads to overly verbose definitions. Moreover, if you have worked exercise 3.4.3 you will have noticed that you needed to reprove many lemmas reminiscent of similar lemmas about *rev*. We will now show you how **recdef** can simplify definitions and proofs about nested recursive datatypes. As an example we choose exercise 3.4.3:

```
consts trev :: "('a,'b)term ⇒ ('a,'b)term"
```

Although the definition of `trev` is quite natural, we will have overcome a minor difficulty in convincing Isabelle of its termination. It is precisely this difficulty that is the *raison d'être* of this subsection.

Defining `trev` by **recdef** rather than **primrec** simplifies matters because we are now free to use the recursion equation suggested at the end of §3.4.2:

```
recdef trev "measure size"
  "trev (Var x)      = Var x"
  "trev (App f ts) = App f (rev(map trev ts))"
```

Remember that function `size` is defined for each **datatype**. However, the definition does not succeed. Isabelle complains about an unproved termination condition

$$t \in \text{set } ts \longrightarrow \text{size } t < \text{Suc } (\text{term_list_size } ts)$$

where `set` returns the set of elements of a list and `term_list_size :: term list ⇒ nat` is an auxiliary function automatically defined by Isabelle (when `term` was defined). First we have to understand why the recursive call of `trev` underneath `map` leads to the above condition. The reason is that **recdef** “knows” that `map` will apply `trev` only to elements of `ts`. Thus the above condition expresses that the size of the argument $t \in \text{set } ts$ of any recursive call of `trev` is strictly less than $\text{size } (\text{App } f \text{ } ts) = \text{Suc } (\text{term_list_size } ts)$. We will now prove the termination condition and continue with our definition. Below we return to the question of how **recdef** “knows” about `map`.

The termination condition is easily proved by induction:

```
lemma [simp]: "t ∈ set ts ⟶ size t < Suc(term_list_size ts)"
  by(induct_tac ts, auto)
```

By making this theorem a simplification rule, **recdef** applies it automatically and the above definition of `trev` succeeds now. As a reward for our effort, we can now prove the desired lemma directly. The key is the fact that we no longer need the verbose induction schema for type `term` but the simpler one arising from `trev`:

```
lemma "trev(trev t) = t"
  apply(induct_tac t rule:trev.induct)
```

This leaves us with a trivial base case $\text{trev } (\text{trev } (\text{Var } x)) = \text{Var } x$ and the step case

$$\forall t. t \in \text{set } ts \longrightarrow \text{trev } (\text{trev } t) = t \implies \\ \text{trev } (\text{trev } (\text{App } f \text{ } ts)) = \text{App } f \text{ } ts$$

both of which are solved by simplification:

```
by(simp_all add:rev_map sym[OF map_compose] cong:map_cong)
```

If the proof of the induction step mystifies you, we recommend to go through the chain of simplification steps in detail; you will probably need the help of `trace_simp`. Theorem `map_cong` is discussed below.

The above definition of `trev` is superior to the one in §3.4.2 because it brings `rev` into play, about which already know a lot, in particular `rev (rev xs) = xs`. Thus this proof is a good example of an important principle:

*Chose your definitions carefully
because they determine the complexity of your proofs.*

Let us now return to the question of how `recdef` can come up with sensible termination conditions in the presence of higher-order functions like `map`. For a start, if nothing were known about `map`, `map trev ts` might apply `trev` to arbitrary terms, and thus `recdef` would try to prove the unprovable `size t < Suc (term_list_size ts)`, without any assumption about `t`. Therefore `recdef` has been supplied with the congruence theorem `map_cong`:

$$\begin{aligned} & \llbracket xs = ys; \bigwedge x. x \in \text{set } ys \implies f\ x = g\ x \rrbracket \\ & \implies \text{map } f\ xs = \text{map } g\ ys \end{aligned}$$

Its second premise expresses (indirectly) that the second argument of `map` is only applied to elements of its third argument. Congruence rules for other higher-order functions on lists would look very similar but have not been proved yet because they were never needed. If you get into a situation where you need to supply `recdef` with new congruence rules, you can either append a hint locally to the specific occurrence of `recdef`

```
(hints cong: map_cong)
```

or declare them globally by giving them the `recdef_cong` attribute as in

```
declare map_cong[recdef_cong]
```

Note that the global `cong` and `recdef_cong` attributes are intentionally kept apart because they control different activities, namely simplification and making recursive definitions. The local `cong` in the hints section of `recdef` is merely short for `recdef_cong`.

4.3 Advanced induction techniques

Now that we have learned about rules and logic, we take another look at the finer points of induction. The two questions we answer are: what to do if the proposition to be proved is not directly amenable to induction, and how to utilize and even derive new induction schemas.

4.3.1 Massaging the proposition

So far we have assumed that the theorem we want to prove is already in a form that is amenable to induction, but this is not always the case:

```
lemma "xs ≠ [] ⇒ hd(rev xs) = last xs"
apply(induct_tac xs)
```

(where *hd* and *last* return the first and last element of a non-empty list) produces the warning

Induction variable occurs also among premises!

and leads to the base case

```
1. xs ≠ [] ⇒ hd (rev []) = last []
```

which, after simplification, becomes

```
1. xs ≠ [] ⇒ hd [] = last []
```

We cannot prove this equality because we do not know what *hd* and *last* return when applied to `[]`.

The point is that we have violated the above warning. Because the induction formula is only the conclusion, the occurrence of *xs* in the premises is not modified by induction. Thus the case that should have been trivial becomes unprovable. Fortunately, the solution is easy:

Pull all occurrences of the induction variable into the conclusion using `→`.

This means we should prove

```
lemma hd_rev: "xs ≠ [] → hd(rev xs) = last xs"
```

This time, induction leaves us with the following base case

```
1. [] ≠ [] → hd (rev []) = last []
```

which is trivial, and `auto` finishes the whole proof.

If *hd_rev* is meant to be a simplification rule, you are done. But if you really need the \Rightarrow -version of *hd_rev*, for example because you want to apply it as an introduction rule, you need to derive it separately, by combining it with modus ponens:

```
lemmas hd_revI = hd_rev[THEN mp]
```

which yields the lemma we originally set out to prove.

In case there are multiple premises A_1, \dots, A_n containing the induction variable, you should turn the conclusion *C* into

$$A_1 \longrightarrow \dots A_n \longrightarrow C$$

(see the remark?? in §??). Additionally, you may also have to universally quantify some other variables, which can yield a fairly complex conclusion. Here is a simple example (which is proved by *blast*):

lemma simple: " $\forall y. A y \longrightarrow B y \longrightarrow B y \ \& \ A y$ "

You can get the desired lemma by explicit application of modus ponens and *spec*:

lemmas myrule = simple[*THEN spec, THEN mp, THEN mp*]

or the wholesale stripping of \forall and \longrightarrow in the conclusion via *rule_format*

lemmas myrule = simple[*rule_format*]

yielding $[A y; B y] \Longrightarrow B y \wedge A y$. You can go one step further and include these derivations already in the statement of your original lemma, thus avoiding the intermediate step:

lemma myrule[*rule_format*]: " $\forall y. A y \longrightarrow B y \longrightarrow B y \ \& \ A y$ "

A second reason why your proposition may not be amenable to induction is that you want to induct on a whole term, rather than an individual variable. In general, when inducting on some term t you must rephrase the conclusion as

$$\forall y_1 \dots y_n. x = t \longrightarrow C$$

where $y_1 \dots y_n$ are the free variables in t and x is new, and perform induction on x afterwards. An example appears below.

4.3.2 Beyond structural and recursion induction

So far, inductive proofs were by structural induction for primitive recursive functions and recursion induction for total recursive functions. But sometimes structural induction is awkward and there is no recursive function in sight either that could furnish a more appropriate induction schema. In such cases some existing standard induction schema can be helpful. We show how to apply such induction schemas by an example.

Structural induction on *nat* is usually known as “mathematical induction”. There is also “complete induction”, where you must prove $P(n)$ under the assumption that $P(m)$ holds for all $m < n$. In Isabelle, this is the theorem *nat_less_induct*:

$$(\bigwedge n. \forall m. m < n \longrightarrow P m \Longrightarrow P n) \Longrightarrow P n$$

Here is an example of its application.

consts f :: "nat => nat"

axioms f_ax: " $f(f(n)) < f(\text{Suc}(n))$ "

From the above axiom¹ for f it follows that $n \leq f\ n$, which can be proved by induction on $f\ n$. Following the recipe outlined above, we have to phrase the proposition as follows to allow induction:

lemma *f_incr_lem*: " $\forall i. k = f\ i \longrightarrow i \leq f\ i$ "

To perform induction on k using *nat_less_induct*, we use the same general induction method as for recursion induction (see §3.5.4):

apply(*induct_tac* *k* *rule*: *nat_less_induct*)

which leaves us with the following proof state:

1. $\bigwedge n. \forall m. m < n \longrightarrow (\forall i. m = f\ i \longrightarrow i \leq f\ i)$
 $\implies \forall i. n = f\ i \longrightarrow i \leq f\ i$

After stripping the $\forall i$, the proof continues with a case distinction on i . The case $i = 0$ is trivial and we focus on the other case:

1. $\bigwedge n\ i\ \text{nat.}$
 $\llbracket \forall m. m < n \longrightarrow (\forall i. m = f\ i \longrightarrow i \leq f\ i); i = \text{Suc}\ \text{nat} \rrbracket$
 $\implies n = f\ i \longrightarrow i \leq f\ i$

by(*blast* *intro!*: *f_ax* *Suc_leI* *intro*: *le_less_trans*)

It is not surprising if you find the last step puzzling. The proof goes like this (writing j instead of *nat*). Since $i = \text{Suc}\ j$ it suffices to show $j < f\ (\text{Suc}\ j)$ (by *Suc_leI*: $m < n \implies \text{Suc}\ m \leq n$). This is proved as follows. From *f_ax* we have $f\ (f\ j) < f\ (\text{Suc}\ j)$ (1) which implies $f\ j \leq f\ (f\ j)$ (by the induction hypothesis). Using (1) once more we obtain $f\ j < f\ (\text{Suc}\ j)$ (2) by transitivity (*le_less_trans*: $\llbracket i \leq j; j < k \rrbracket \implies i < k$). Using the induction hypothesis once more we obtain $j \leq f\ j$ which, together with (2) yields $j < f\ (\text{Suc}\ j)$ (again by *le_less_trans*).

This last step shows both the power and the danger of automatic proofs: they will usually not tell you how the proof goes, because it can be very hard to translate the internal proof into a human-readable format. Therefore §?? introduces a language for writing readable yet concise proofs.

We can now derive the desired $i \leq f\ i$ from *f_incr*:

lemmas *f_incr* = *f_incr_lem*[*rule_format*, *OF refl*]

The final *refl* gets rid of the premise $?k = f\ ?i$. Again, we could have included this derivation in the original statement of the lemma:

lemma *f_incr*[*rule_format*, *OF refl*]: " $\forall i. k = f\ i \longrightarrow i \leq f\ i$ "

Exercise 4.3.1 From the above axiom and lemma for f show that f is the identity.

¹In general, the use of axioms is strongly discouraged, because of the danger of inconsistencies. The above axiom does not introduce an inconsistency because, for example, the identity function satisfies it.

In general, `induct_tac` can be applied with any rule r whose conclusion is of the form $?P ?x_1 \dots ?x_n$, in which case the format is

```
apply(induct_tac  $y_1 \dots y_n$  rule:  $r$ )
```

where y_1, \dots, y_n are variables in the first subgoal. In fact, `induct_tac` even allows the conclusion of r to be an (iterated) conjunction of formulae of the above form, in which case the application is

```
apply(induct_tac  $y_1 \dots y_n$  and ... and  $z_1 \dots z_m$  rule:  $r$ )
```

4.3.3 Derivation of new induction schemas

Induction schemas are ordinary theorems and you can derive new ones whenever you wish. This section shows you how to, using the example of `nat_less_induct`. Assume we only have structural induction available for `nat` and want to derive complete induction. This requires us to generalize the statement first:

```
lemma induct_lem: " $(\bigwedge n::nat. \forall m < n. P\ m \implies P\ n) \implies \forall m < n. P\ m$ "  
apply(induct_tac  $n$ )
```

The base case is trivially true. For the induction step ($m < \text{Suc } n$) we distinguish two cases: case $m < n$ is true by induction hypothesis and case $m = n$ follows from the assumption, again using the induction hypothesis:

```
apply(blast)  
by(blast elim:less_SucE)
```

The elimination rule `less_SucE` expresses the case distinction:

$$\llbracket m < \text{Suc } n; m < n \implies P; m = n \implies P \rrbracket \implies P$$

Now it is straightforward to derive the original version of `nat_less_induct` by manipulating the conclusion of the above lemma: instantiate n by `Suc n` and m by n and remove the trivial condition $n < \text{Sc } n$. Fortunately, this happens automatically when we add the lemma as a new premise to the desired goal:

```
theorem nat_less_induct: " $(\bigwedge n::nat. \forall m < n. P\ m \implies P\ n) \implies P\ n$ "  
by(insert induct_lem, blast)
```

Finally we should mention that HOL already provides the mother of all inductions, *wellfounded induction* (`wf_induct`):

$$\llbracket \text{wf } r; \bigwedge x. \forall y. (y, x) \in r \longrightarrow P\ y \implies P\ x \rrbracket \implies P\ a$$

where `wf r` means that the relation r is wellfounded. For example, theorem `nat_less_induct` can be viewed (and derived) as a special case of `wf_induct` where r is `<` on `nat`. For details see the library.

Appendix

\wedge	\vee	\longrightarrow	\neg	\neq	\forall	\forall	\exists	\exists	$\exists!$	$\exists!$
&		-->	~	~=	ALL	!	EX	?	EX!	?!
\llbracket	\rrbracket	\implies	\wedge	\equiv	λ	\Rightarrow				
$\llbracket $	$ \rrbracket$	\implies	!!	==	%	\Rightarrow				
\circ	\leq	\times								
\circ	\leq	*								

Figure A.1: Mathematical symbols and their ASCII-equivalents

ALL	case	div	dvd	else
EX	if	in	INT	Int
LEAST	let	mod	0	o
of	op	PROP	SIGMA	then
Times	UN	Un		

Figure A.2: Reserved words in HOL terms

Bibliography

- [1] David Aspinall. Proof General: A generic tool for proof development. In *ETAPS / TACAS*, 2000.
- [2] David Aspinall et al. Proof General — Organize your proofs! <http://www.proofgeneral.org>.
- [3] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.
- [4] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1975.
- [5] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *J. Func. Prog.*, 1999.
- [6] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle's Logics: HOL*. <http://isabelle.in.tum.de/doc/logics-HOL.pdf>.
- [7] Lawrence C. Paulson. *The Isabelle Reference Manual*. <http://isabelle.in.tum.de/doc/ref.pdf>.
- [8] Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.
- [9] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.
- [10] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.
- [11] Markus Wenzel. *The Isabelle/Isar Reference Manual*. <http://isabelle.in.tum.de/doc/isar-ref.pdf>.

Index

\forall , **4, 59**
!, **59**
 \exists , **4, 59**
?, **5, 59**
 $\exists!$, **4, 59**
?!, **59**
 \wedge , **3, 59**
&, **59**
=, **3**
 \longrightarrow , **3, 59**
 \dashrightarrow , **59**
 \neg , **3, 59**
 \sim , **59**
 \neq , **59**
 \approx , **59**
 \vee , **3, 59**
|, **59**
o, **59**
*, **21, 59**
+, **21**
-, **21, 22**
 \leq , **21, 59**
 \leq , **59**
<, **21**
[], **8**
#, **8**
@, **8**
 \wedge , **10, 59**
!!, **59**
 \equiv , **23, 59**
 \equiv , **59**
 \Rightarrow , **3, 59**
 \times , **22, 59**
 \Rightarrow , **59**
 \implies , **59**
 \implies , **59**
 \implies , **59**
[[, **11, 59**

[|, **59**
]], **11, 59**
|], **59**
 λ , **3, 59**
%, **59**
::, **4**
,, **28**
., **12**
;, **5**
'a, **3**

0, **20, 21**

abandon proof, **11**
abandon theory, **15**
ALL, **59**
apply, **14**
arith, **21**
arithmetic, **20–22, 30**
ASCII symbols, **59**
associative-commutative operators,
52
auto, **35**

bool, **2, 3**
by, **12**

case, **3, 4, 16, 28, 29**
case distinction, **17**
case splits, **28**
case_tac, **17**
congruence rules, **50**
Cons, **8**
constdefs, **23**

datatype, **8, 35–40**
defer, **14**
definition, **23**
 unfolding, **27**

- defs, **23**
- div, **21**
- EX, **59**
- EX!, **59**
- False, **3**
- flag, **3, 4, 30**
 - (re)setting, **3**
- formula, **3**
- hd, **15**
- identifier, **4**
 - qualified, **2**
- if, **3, 4**
- induct_tac, **10, 17, 49, 58**
- induction, **54–58**
 - recursion, **48–49**
 - structural, **17**
- infixr, **8**
- inner syntax, **9**
- kill, **15**
- LEAST, **21**
- lemma, **11**
- lemma, **11**
- let, **3, 4, 27**
- list, **2, 8, 15**
- Main, **2**
- max, **21, 22**
- measure function, **44**
- method, **14**
- min, **21, 22**
- mod, **21**
- nat, **2, 20**
- Nil, **8**
- no_asm, **26**
- no_asm_simp, **26**
- no_asm_use, **27**
- None, **41**
- o, **38, 59**
- oops, **11**
- option, **41**
- ordered rewriting, **51**
- outer syntax, **9**
- parent theory, **2**
- permutative rewrite rule, **51**
- pr, **14**
- prefer, **14**
- primitive recursion, **16**
- primrec, **8, 16, 35–40**
- proof
 - abandon, **11**
- Proof General, **5**
- recdef, **43–49, 52–54**
- recursion induction, **48–49**
- redo, **14**
- rev, **8**
- rewrite rule, **25**
 - permutative, **51**
- rewriting, **25**
 - ordered, **51**
- schematic variable, **5**
- set, **2**
- show_brackets, **4**
- show_types, **3, 14**
- simp (attribute), **9, 25**
- simp (method), **25**
- simp_all, **26**
- simplification, **24–30, 50–52**
 - of let-expressions, **27**
 - ordered, **51**
 - with definitions, **27**
 - with/of assumptions, **26**
- simplification rule, **25**
- simplifier, **24**
- Some, **41**
- split, **28–29**
- split rule, **29**
- split_if, **28**
- structural induction, **17**
- Suc, **20**

- term, **3**
- term, **14**
- term rewriting, **25**
- theorem, **11**
- theorem, **9, 11**
- theory, **1**
 - abandon, **15**
- theory file, **2**
- thm, **14**
- t1, **15**
- total, **9**
- trace_simp, **30**
- tracing the simplifier, **30**
- True, **3**
- typ, **14**
- type, **2**
- type constraint, **4**
- type inference, **3**
- type synonym, **23**
- type variable, **3**
- types, **23**

- undo, **14**
- unfold, **27**
- unknown, **5**
- use_thy, **15**

- variable, **5**
 - schematic, **5**
 - type, **3**