The Tutorial
— DRAFT —

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

31 October 1999

Contents

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.

Chapter 1

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 [1] or Paulson [8]. 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.

A tutorial is by definition incomplete. To fully exploit the power of the system you need to consult the Isabelle Reference Manual [6] for details about Isabelle and the Isabelle/HOL manual [5] for details relating to HOL. Both manuals have a comprehensive index.

1.2 Theories, proofs and interaction

Working with Isabelle means creating two different kinds of documents: theories and proof scripts. Roughly speaking, a **theory** is a named collection of types and functions, much like a module in a programming language or a specification in a specification language. In fact, theories in HOL can be either. Theories must reside in files with the suffix `.thy`. The general format of a theory file `T.thy` is

```
T = B1 + ··· + Bn +
⟨declarations⟩
end
```

where $\mathtt{B}_1, \ldots, \mathtt{B}_n$ are the names of existing theories that `T` is based on and ⟨*declarations*⟩ stands for the newly introduced concepts (types, functions etc). The $\mathtt{B}_i$ 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`. HOL's theory library is available online at

```
http://www.cl.cam.ac.uk/Research/HVG/Isabelle/library/
http://isabelle.in.tum.de/library/
```

and is recommended browsing.

! 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.

This tutorial is concerned with introducing you to the different linguistic constructs that can fill ⟨*declarations*⟩ in the above theory template. A complete grammar of the basic constructs is found in Appendix A of [6], for reference in times of doubt.

The tutorial is also concerned with showing you how to prove theorems about the concepts in a theory. This involves invoking predefined theorem proving commands. Because Isabelle is written in the programming language ML,[1] interacting with Isabelle means calling ML functions. Hence **proof scripts** are sequences of calls to ML functions that perform specific theorem proving tasks. Nevertheless, familiarity with ML is absolutely not required. All proof scripts for theory `T` (defined in file `T.thy`) should be contained in file `T.ML`. Theory and proof scripts are loaded (and checked!) by calling the ML function `use_thy`:

```
use_thy "T";
```

There are more advanced interfaces for Isabelle that hide the ML level from you and replace function calls by menu selection. There is even a special font with mathematical symbols. For details see the Isabelle home page. This tutorial concentrates on the bare essentials and ignores such niceties.

1.3 Types, terms and formulae

Embedded in the declarations of 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.

[1]Many concepts in HOL and ML are similar. Make sure you do not confuse the two levels.

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)foo`).

function types, denoted by `=>`. In HOL `=>` represents *total* functions only. As is customary, τ_1 `=>` τ_2 `=>` τ_3 means τ_1 `=>` `(`τ_2 `=>` τ_3`)`. Isabelle also supports the notation `[`$\tau_1, \ldots, \tau_n$`]` `=>` τ which abbreviates τ_1 `=>` $\cdots$ `=>` τ_n `=>` τ.

type variables, denoted by `'a`, `'b` etc, just like in ML. They give rise to polymorphic types like `'a => '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

```
set show_types;
```

at the ML-level. This can be reversed by `reset show_types;`.

Terms are formed as in functional programming by applying functions to arguments. If `f` is a function of type τ_1 `=>` τ_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` t_1 `else` t_2 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` means `0+0`. Multiple bindings are separated by semicolons: `let` x_1 `=` t_1`;` `...;` x_n `=` t_n `in` u.

`case` e `of` c_1 `=>` e_1 `| ...|` c_n `=>` e_n evaluates to e_i if e is of the form c_i. See §2.4.4 for details.

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

`~` ('not'), `&` ('and'), `|` ('or') and `-->` ('implies'), all of which (except the unary `~`) associate to the right. In particular `A --> B --> C` means `A --> (B --> C)` and is thus logically equivalent with `A & B --> C` (which is `(A & B) --> C`).

Equality is available in the form of the infix function `=` of type `'a => 'a => 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 syntax for quantifiers is `!` x`.` P ('for all x') and `?` x`.` P ('exists x'). There is even `?!` x`.` P, which means that there exists exactly one x that satisfies P. Instead of `!` and `?` you may also write `ALL` and `EX`. Nested quantifications can be abbreviated: `!`x y z`.` P means `!`x`. !`y`. !`z`.` P.

Despite type inference, it is sometimes necessary to attach explicit **type constraints** to a term. The syntax is t`::`τ as in `x < (y::nat)`. Note that `::` binds weakly and should therefore be enclosed in parentheses: `x < y::nat` is ill-typed because it is interpreted as `(x < y)::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`:

```
set show_brackets; ...; 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, `~ ~ P = P` means `~ ~(P = P)` and not `(~ ~P) = P`. When using `=` to mean logical equivalence, enclose both operands in parentheses, as in `(A & B) = (B & 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 = (%x. x)`. This includes `if`, `let`, `case`, `%` and quantifiers.

- Never write `%x.x` or `!x.x=x` because `x.x` is always read as a single qualified identifier that refers to an item `x` in theory `x`. Write `%x. x` and `!x. x=x` instead.

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 (i.e. `qed` at the end of a proof) 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.

! The existential quantifier `?` needs to be followed by a space. Otherwise `?x` is interpreted as a schematic variable.

1.5 Getting started

Assuming you have installed Isabelle, you start it by typing `isabelle 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 theories (`.thy` files) and proof scripts (`.ML` files). While you are developing a proof, we recommend to type each proof command into the ML-file first and then enter it into Isabelle by copy-and-paste, thus ensuring that you have a complete record of your proof.

[2] Simply executing `isabelle` without an argument 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.

Chapter 2

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 what could be called *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 Fig. 2.1.

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 `Datatype`, a theory that contains everything required for datatype definitions but does not have `List` as a parent, thus avoiding ambiguities caused by defining lists twice.

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.


```
ToyList = Datatype +

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 # [])"

end
```

Figure 2.1: A theory of lists

! 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, the functions `app` and `rev` are declared. 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 `app` 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. The equations for `app` and `rev` hardly need comments: `app` 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 (see §2.4). 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 could 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. A.1 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 should be enclosed in "...". The same holds for identifiers that happen to be keywords, as in

```
consts "end" :: 'a list => 'a
```

To lessen this burden, quotation marks around types can be dropped, provided their syntax does not go beyond what is described in §1.3. Types containing further operators, e.g. `*` for Cartesian products, need quotation marks.

When Isabelle prints a syntax error message, it refers to the HOL syntax as the **inner syntax**.

2.2 An introductory proof

Having defined `ToyList`, we load it with the ML command

```
use_thy "ToyList";
```

and 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. Typing

```
Goal "rev(rev xs) = xs";
```

establishes a new goal to be proved in the context of the current theory, which is the one we just loaded. Isabelle's response is to print the current proof state:

```
Level 0
rev (rev xs) = xs
 1. rev (rev xs) = xs
```

Until we have finished a proof, the proof state always looks like this:

```
Level i
G
 1. G1
 ⋮
 n. Gn
```

where `Level` i indicates that we are i steps into the proof, G is the overall goal that we are trying to prove, and the numbered lines contain the subgoals

$G_1, \ldots, G_n$ that we need to prove to establish G. At `Level 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`:

```
by(induct_tac "xs" 1);
```

This tells Isabelle to perform induction on variable `xs` in subgoal 1. The new proof state contains two subgoals, namely the base case (`Nil`) and the induction step (`Cons`):

```
1. rev (rev []) = []
2. !!a list. rev (rev list) = list ==> rev (rev (a # list)) = a # list
```

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

```
i. !!x1 ... xn. assumptions ==> conclusion
```

The prefix of bound variables `!!`$x_1 \ldots x_n$ 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:

```
by(Auto_tac);
```

This command tells Isabelle to apply a proof strategy called `Auto_tac` to all subgoals. Essentially, `Auto_tac` 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. !!a list. rev(rev list) = list ==> 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)`

We start the proof as usual:

```
Goal "rev(xs @ ys) = (rev ys) @ (rev xs)";
```

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:

```
by(induct_tac "xs" 1);
```

This time not even the base case is solved automatically:

```
by(Auto_tac);
  1. rev ys = rev ys @ []
  2. ...
```

We need another lemma.

Second lemma: `xs @ [] = xs`

This time the canonical proof procedure

```
Goal "xs @ [] = xs";
by(induct_tac "xs" 1);
by(Auto_tac);
```

leads to the desired message `No subgoals!`:

```
  Level 2
  xs @ [] = xs
  No subgoals!
```

Now we can give the lemma just proved a suitable name

```
qed "app_Nil2";
```

and tell Isabelle to use this lemma in all future proofs by simplification:

```
Addsimps [app_Nil2];
```

Note that in the theorem `app_Nil2` 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

```
Goal "rev(xs @ ys) = (rev ys) @ (rev xs)";
by(induct_tac "xs" 1);
by(Auto_tac);
```

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

```
  1. !!a list.
        rev (list @ ys) = rev ys @ rev list
        ==> (rev ys @ rev list) @ a # [] = 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 the definition of `ToyList`). Thus the conclusion really is

```
==> (rev ys @ rev list) @ (a # []) = rev ys @ (rev list @ (a # []))
```

and the missing lemma is associativity of `@`.

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

This time the canonical proof procedure

```
Goal "(xs @ ys) @ zs = xs @ (ys @ zs)";
by(induct_tac "xs" 1);
by(Auto_tac);
```

succeeds without further ado. Again we name the lemma and add it to the set of lemmas used during simplification:

```
qed "app_assoc";
Addsimps [app_assoc];
```

Now we can go back and prove the first lemma

```
Goal "rev(xs @ ys) = (rev ys) @ (rev xs)";
by(induct_tac "xs" 1);
by(Auto_tac);
```

add it to the simplification lemmas

```
qed "rev_app";
Addsimps [rev_app];
```

and then solve our main theorem:

```
Goal "rev(rev xs) = xs";
by(induct_tac "xs" 1);
by(Auto_tac);
```

Review

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

- the standard theorem proving procedure: state a goal; proceed with proof until a new 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_tac`.
- 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. Proof commands are always of the form `by(`*tactic*`);`

where **tactic** is a synonym for "theorem proving function". Typical examples are `induct_tac` and `Auto_tac` — the suffix `_tac` is merely a mnemonic. Further tactics are introduced throughout the tutorial.

The most useful auxiliary commands are:

Printing the current state Type `pr();` to redisplay the current proof state, for example when it has disappeared off the screen.

Limiting the number of subgoals Typing `prlim` k`;` tells Isabelle to print only the first k subgoals from now on and redisplays the current proof state. This is helpful when there are many subgoals.

Undoing Typing `undo();` undoes the effect of the last tactic.

Context switch Every proof happens in the context of a **current theory**. By default, this is the last theory loaded. If you want to prove a theorem in the context of a different theory `T`, you need to type `context T.thy;` first. Of course you need to change the context again if you want to go back to your original theory.

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`.

(Re)loading theories Initially you load theory `T` by typing `use_thy "T";`, which loads all parent theories of `T` automatically, if they are not loaded already. If you modify `T.thy` or `T.ML`, you can reload it by typing `use_thy "T";` again. This time, however, only `T` is reloaded. If some of `T`'s parents have changed as well, type `update_thy "T";` to reload `T` and all of its parents that have changed (or have changed parents).

Further commands are found in the Reference Manual.

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: `[`x_1`,`$\dots$`,`x_n`]` abbreviates x_1`#`$\dots$`#`x_n`#[]`. 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

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

where α_i are type variables (the parameters), C_i are distinct constructor names and τ_{ij} are types; it is customary to capitalize the first letter in constructor names. There are a number of restrictions (such as the type should not be empty) detailed elsewhere [5]. Isabelle notifies you if you violate them.

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

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 explained in §3.5.

[1] `http://isabelle.in.tum.de/library/HOL/List.html`

Exercise 2.4.1 Given the datatype of binary trees

```
datatype 'a tree = Tip | Node ('a tree) 'a ('a tree)
```

define a function `mirror` that mirrors the structure of a binary tree by swapping subtrees (recursively). Prove `mirror(mirror(t)) = t`.

2.4.4 `case`-expressions

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

```
case xs of [] => 0 | y#ys => y
```

evaluates to `0` 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::nat` and hence `xs::(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

$$\begin{array}{rlll} \texttt{case}\ e\ \texttt{of} & C_1\ x_{11}\ \dots\ x_{1k_1} & \Rightarrow & e_1 \\ \vdots & & & \\ \mid & C_m\ x_{m1}\ \dots\ x_{mk_m} & \Rightarrow & e_m \end{array}$$

! *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 [] => 0 | [x] => x | x#(y#zs) => y
```

write

```
case xs of [] => 0 | x#ys => (case ys of [] => x | y#zs => y)
```

Note that `case`-expressions should be enclosed in parentheses to indicate their scope.

2.4.5 Structural induction

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 `exhaust_tac`. A trivial example:

```
Goal "(case xs of [] => [] | y#ys => xs) = xs";
by(exhaust_tac "xs" 1);
 1. xs = [] ==> (case xs of [] => [] | y # ys => xs) = xs
 2. !!a list. xs = a # list ==> (case xs of [] => [] | y # ys => xs) = xs
by(Auto_tac);
```

Note that this particular case distinction could have been automated completely. See §3.1.1.

! Induction is only allowed on a free variable that should not occur among the assumptions of the subgoal. Exhaustion works for arbitrary terms.

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 the terms `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 boolean expressions?

The value of a boolean expressions depends on the value of its variables. Hence the function `value` takes an additional parameter, an *environment* of type `nat => bool`, which maps variables to their values:

```
consts value :: boolex => (nat => bool) => bool
primrec
"value (Const b) env = b"
"value (Var x)   env = env x"
"value (Neg b)   env = (~ value b env)"
"value (And b c) env = (value b env & 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 => (nat => bool) => 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 `boolex` is close to the customary representation of logical formulae, whereas `ifex` is designed for efficiency. Thus we need to translate from `boolex` into `ifex`:

```
consts bool2if :: boolex => 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.

```
Goal "valif (bool2if b) env = value b env";
```

The proof is canonical:

```
by(induct_tac "b" 1);
by(Auto_tac);
```

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 => ifex => ifex => 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 => 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:

```
Goal "valif (norm b) env = valif b env";
```

The proof is canonical, provided we first show the following lemma (which also helps to understand what `normif` does) and make it available for simplification via `Addsimps`:


```
Goal "!t e. valif (normif b t e) env = valif (IF b t e) env";
```

But how can we be sure that `norm` really produces a normal form in the above sense? We have to prove

```
Goal "normal(norm b)";
```

where `normal` expresses that an If-expression is in normal form:

```
consts normal :: ifex => bool
primrec
"normal(CIF b) = True"
"normal(VIF x) = True"
"normal(IF b t e) = (normal t & normal e &
      (case b of CIF b => True | VIF x => True | IF x y z => False))"
```

Of course, this requires a lemma about normality of `normif`

```
Goal "!t e. normal(normif b t e) = (normal t & normal e)";
```

that has to be made available for simplification via `Addsimps`.

How does one come up with the required lemmas? Try to prove the main theorems without them and study carefully what `Auto_tac` leaves unproved. This has to provide the clue. The necessity of universal quantification (`!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 `IF`s must be a variable. Adapt the above development to this changed requirement. (Hint: you may need to formulate some of the goals as implications (`-->`) 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

```
Goal "sum n + sum n = n*(Suc n)";
by(induct_tac "n" 1);
by(Auto_tac);
  sum n + sum n = n * Suc n
  No subgoals!
```

The usual arithmetic operations `+`, `-`, `*`, `div`, `mod`, `min` and `max` are predefined, as are the relations `<=` and `<`. There is even a least number operation `LEAST`. For example, `(LEAST n. 1 < n) = 2` (HOL does not prove this completely automatically).

! The operations `+`, `-`, `*`, `min`, `max`, `<=` 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+y = y+x`, there is nothing to indicate that you are talking about natural numbers. Hence Isabelle can only infer that `x` and `y` are of some arbitrary type where `+` is 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+y = y+(x::nat)`. If there is enough contextual information this may not be necessary: `x+0 = x` automatically implies `x::nat`.

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

```
Goal "[| ~ 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 tactic `arith_tac`. It proves arithmetic goals involving the usual logical connectives (`~`, `&`, `|`, `-->`), the relations `<=` and `<`, and the operations `+`, `-`, `min` and `max`. For example, it can prove

```
Goal "min i (max j (k*k)) = max (min (k*k) i) (min i (j::nat))";
```

because `k*k` can be treated as atomic. In contrast, $n*n = n \Longrightarrow n = 0 \vee n = 1$ is not even proved by `arith_tac` because the proof relies essentially on properties of multiplication.

! The running time of `arith_tac` is exponential in the number of occurrences of `-`, `min` and `max` because they are first eliminated by case distinctions.
`arith_tac` 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 τ_1 `*` τ_2 provided each a_i is of type τ_i. The components of a pair are extracted by `fst` and `snd`: `fst(`x`,`y`) =` x and `snd(`x`,`y`) =` y. Tuples are simulated by pairs nested to the right: `(`a_1`,`a_2`,`a_3`)` and τ_1 `*` τ_2 `*` τ_3 stand for `(`a_1`,(`a_2`,`a_3`))` and τ_1 `* (`τ_2 `*` τ_3`)`. Therefore `fst(snd(`a_1`,`a_2`,`a_3`)) =` a_2.

It is possible to use (nested) tuples as patterns in abstractions, for example `%(x,y,z).x+y+z` and `%((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.

! 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 `%p. fst p + snd p` instead of `%(x,y). x + y`. See §?? for theorem proving with tuple patterns.

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"
```

The synonym `alist` shows that in general the type on the right-hand side needs to be enclosed in double quotation marks (see the end of §2.1).

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

```
consts nand, 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 arbitrary user-supplied names. The symbol `==` is a special form of equality that should only be used in constant definitions. Declarations and definitions can also be merged

```
constdefs nand :: gate
          "nand A B == ~(A & B)"
           exor :: gate
          "exor 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 $f\ x_1\ \ldots\ x_n$ `==` 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:

```
defs  prime_def "prime(p) == (m divides p) --> (m=1 | m=p)"
```

Isabelle rejects this 'definition' because of the extra `m` on the right-hand side, which would introduce an inconsistency. What you should have written is

```
defs  prime_def "prime(p) == !m. (m divides p) --> (m=1 | m=p)"
```

Chapter 3

More Functional Programming

The purpose of this chapter is to deepen the reader's understanding of the concepts encountered so far and to introduce an advanced method for defining 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). Finally we present a very general method for defining recursive functions that goes well beyond what `primrec` allows (§3.5).

3.1 Simplification

So far we have proved our theorems by `Auto_tac`, which 'simplifies' all subgoals. In fact, `Auto_tac` 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_tac`. This section covers the tactic that `Auto_tac` 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 twofold: to introduce the many features of the simplifier (§3.1.1) and to explain a little bit how the simplifier works (§3.1.2). Anybody intending to use HOL should read §3.1.1, and the serious student should read §3.1.2 as well in order to understand what happened in case things do not simplify as expected.

3.1.1 Using the simplifier

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#[]) @ []` $\leadsto$ `0#((1#[]) @ [])` $\leadsto$ `0#(1#([] @ []))` $\leadsto$ `0#1#[]`

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

Simpsets

To facilitate simplification, each theory has an associated set of simplification rules, known as a **simpset**. Within a theory, proofs by simplification refer to the associated simpset by default. The simpset of a theory is built up as follows: starting with the union of the simpsets of the parent theories, each occurrence of a `datatype` or `primrec` construct augments the simpset. Explicit definitions are not added automatically. Users can add new theorems via `Addsimps` and delete them again later by `Delsimps`.

You may augment a simpset not just by equations but by pretty much any theorem. The simplifier will try to make sense of it. For example, a theorem ~P is automatically turned into P `= False`. The details are explained in §3.1.2.

As a rule of thumb, rewrite rules that really simplify a term (like `xs @ [] = xs` and `rev(rev xs) = xs`) should be added to the current simpset right after they have been proved. Those of a more specific nature (e.g. distributivity laws, which alter the structure of terms considerably) should only be added for specific proofs and deleted again afterwards. Conversely, it may also happen that a generally useful rule needs to be removed for a certain proof and is added again afterwards. The need of frequent temporary additions or deletions may indicate a badly designed simpset.

! Simplification may not terminate, for example if both $f(x) = g(x)$ and $g(x) = f(x)$ are in the simpset. It is the user's responsibility not to include rules that can lead to nontermination, either on their own or in combination with other rules.

Simplification tactics

There are four main simplification tactics:

`Simp_tac` i simplifies the conclusion of subgoal i using the theory's simpset. It may solve the subgoal completely if it has become trivial. For example:

```
 1. [] @ [] = []
by(Simp_tac 1);
 No subgoals!
```

`Asm_simp_tac` is like `Simp_tac`, but extracts additional rewrite rules from the assumptions of the subgoal. For example, it solves

```
1. xs = [] ==> xs @ ys = ys @ xs
```

which `Simp_tac` does not do.

`Full_simp_tac` is like `Simp_tac`, but also simplifies the assumptions (without using the assumptions to simplify each other or the actual goal).

`Asm_full_simp_tac` is like `Asm_simp_tac`, but also simplifies the assumptions. In particular, assumptions can simplify each other. For example:

```
 1. [| xs @ zs = ys @ xs; [] @ xs = [] @ [] |] ==> ys = zs
by(Asm_full_simp_tac 1);
 No subgoals!
```

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`. (See also the paragraph on tracing below.)

`Asm_full_simp_tac` is the most powerful of this quartet of tactics. In fact, `Auto_tac` starts by applying `Asm_full_simp_tac` to all subgoals. The only reason for the existence of the other three tactics is that sometimes one wants to limit the amount of simplification, for example to avoid nontermination:

```
1. ! x. f x = g (f (g x)) ==> f [] = f [] @ []
```

is solved by `Simp_tac`, but `Asm_simp_tac` and `Asm_full_simp_tac` loop because the rewrite 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.

Modifying simpsets locally

If a certain theorem is merely needed in one proof by simplification, the pattern

```
Addsimps [rare_theorem];
by(Simp_tac 1);
Delsimps [rare_theorem];
```

is awkward. Therefore there are lower-case versions of the simplification tactics (`simp_tac`, `asm_simp_tac`, `full_simp_tac`, `asm_full_simp_tac`) and of the simpset modifiers (`addsimps`, `delsimps`) that do not access or modify the implicit simpset but explicitly take a simpset as an argument. For example, the above three lines become

```
by(simp_tac (simpset() addsimps [rare_theorem]) 1);
```

where the result of the function call `simpset()` is the simpset of the current theory and `addsimps` is an infix function. The implicit simpset is read once but not modified. This is far preferable to pairs of `Addsimps` and `Delsimps`. Local modifications can be stacked as in

```
by(simp_tac (simpset() addsimps [rare_theorem] delsimps [some_thm]) 1);
```

Rewriting with definitions

Constant definitions (§2.6.2) are not automatically included in the simpset of a theory. Hence such definitions are not expanded automatically either, 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 completely. For example, given the theory

```
Exor = Main +
constdefs
 exor :: bool => bool => bool
"exor A B == (A & ~B) | (~A & B)"
end
```

we may want to prove `exor A (~A)`. Instead of `Goal` we use

```
Goalw [exor_def] "exor A (~A)";
```

which tells Isabelle to expand the definition of `exor`—the first argument of `Goalw` can be a list of definitions—in the initial goal:

```
exor A (~ A)
 1. A & ~ ~ A | ~ A & ~ A
```

In this simple example, the goal is proved by `Simp_tac`. Of course the resulting theorem is insufficient to characterize `exor` completely.

In case we want to expand a definition in the middle of a proof, we can simply add the definition locally to the simpset:

```
by(simp_tac (simpset() addsimps [exor_def]) 1);
```

You should normally not add the definition permanently to the simpset using `Addsimps` because this defeats the whole purpose of an abbreviation.

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

Simplifying `let`-expressions

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

```
   1. (let xs = [] in xs @ xs) = ys
by(simp_tac (simpset() addsimps [Let_def]) 1);
   1. [] = ys
```

If, in a particular context, there is no danger of a combinatorial explosion of nested `let`s one could even add `Let_def` permanently via `Addsimps`.

Conditional equations

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

```
xs ~= []  ==>  hd xs # tl xs = xs                                  (*)
```

(which is proved by `exhaust_tac` on `xs` followed by `Asm_full_simp_tac` twice). Assuming that this theorem together with `(rev xs = []) = (xs = [])` are part of the simpset, the subgoal

```
1. xs ~= [] ==> hd(rev xs) # tl(rev xs) = rev xs
```

is proved by simplification: the conditional equation (∗) above can simplify `hd(rev xs) # tl(rev xs)` to `rev xs` because the corresponding precondition `rev xs ~= []` simplifies to `xs ~= []`, 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

```
1. ! xs. if xs = [] then rev xs = [] else rev xs ~= []
```

can be split into

```
1. ! xs. (xs = [] --> rev xs = []) & (xs ~= [] --> rev xs ~= [])
```

by typing

```
by(split_tac [split_if] 1);
```

Because this is almost always the right proof strategy, the simplifier performs case-splitting on `if`s automatically. Try `Simp_tac` on the initial goal above.

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

```
1. (case xs of [] => zs | y#ys => y#(ys@zs)) = xs@zs
```

becomes

```
1. (xs = [] --> zs = xs @ zs) &
   (! a list. xs = a # list --> a # list @ zs = xs @ zs)
```

by typing

```
by(split_tac [list.split] 1);
```

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. Nevertheless the simplifier can be instructed to perform `case`-splits by adding the appropriate rule to the simpset:

```
by(simp_tac (simpset() addsplits [split_list_case]) 1);
```

solves the initial goal outright, which `Simp_tac` alone will not do.

In general, every datatype t comes with a rule t`.split` that can be added to the simpset either locally via `addsplits` (see above), or permanently via

```
Addsplits [t.split];
```

Split-rules can be removed globally via `Delsplits` and locally via `delsplits` as, for example, in

```
by(simp_tac (simpset() addsimps [...] delsplits [split_if]) 1);
```

Arithmetic

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

```
Goal "[| ~ m < n; m < n+1 |] ==> m = n";
```

is proved by simplification, whereas the only slightly more complex

```
Goal "~ m < n & m < n+1 ==> m = n";
```

is not proved by simplification and requires `arith_tac`.

Permutative rewrite rules

A rewrite rule is **permutative** if the left-hand side and right-hand side are the same up to renaming of variables. The most common permutative rule is commutativity: $x + y = y + x$. Another example is $(x - y) - z = (x - z) - y$. Such rules are problematic because once they apply, they can be used forever. The simplifier is aware of this danger and treats permutative rules separately. For details see [6].

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:

```
   1. rev [x] = []
set trace_simp;
by(Simp_tac 1);
  Applying instance of rewrite rule:
  rev (?x # ?xs) == rev ?xs @ [?x]
  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:
  ?x # ?t = ?t == False
  Rewriting:
  [x] = [] == False
  Level 1
  rev [x] = []
   1. False
```

In more complicated cases, the trace can be enormous, especially since invocations of the simplifier are often nested (e.g. when solving conditions of rewrite rules).

3.1.2 How it works

Higher-order patterns

Local assumptions

The preprocessor

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:

> *1. Theorems about recursive functions are proved by induction.*

In case the function has more than one argument

> *2. 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.

We define a tail-recursive version of list-reversal, i.e. one that can be compiled into a loop:

```
Itrev = Main +
consts itrev :: 'a list => 'a list => 'a list
primrec
"itrev []     ys = ys"
"itrev (x#xs) ys = itrev xs (x#ys)"
end
```

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. We need to show that `itrev` does indeed reverse its first argument provided the second one is empty:

```
Goal "itrev xs [] = rev xs";
```

There is no choice as to the induction variable, and we immediately simplify:

```
by(induct_tac "xs" 1);
by(Auto_tac);
 1. !!a list. 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:

> *3. 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

```
Goal "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 is 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. !!a list.
       itrev list ys = rev list @ ys
       ==> 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:


```
Goal "!ys. itrev xs ys = rev xs @ ys";
```

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

> *4. 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.

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 `rev` on the right-hand side. This constitutes another, albeit weak heuristic that is not restricted to induction:

> *5. The right-hand side of an equation should (in some sense) be simpler than the left-hand side.*

The 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 the symmetric equation!

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 combination of two subexpressions with a binary operation.

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

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

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

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

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 => (('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:

```
exec s [] (comp e) = [value s e]
```

This is generalized to

```
Goal "!vs. exec s vs (comp e) = (value s e) # vs";
```

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

```
Goal "!vs. exec s vs (xs@ys) = exec s (exec s vs xs) ys";
```

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(asm_simp_tac (simpset() addsplits [instr.split]) 1);
```

We could now go back and prove `exec s [] (comp e) = [value s e]` 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 `datatype`s and (in the near future!) records.

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 `Sum` and `Diff`. Boolean expressions can be arithmetic comparisons, conjunctions and negations. The semantics is fixed via two evaluation functions

```
consts  evala :: ('a => nat) => 'a aexp => nat
        evalb :: ('a => nat) => 'a bexp => bool
```

that take an environment (a mapping from variables `'a` to values `nat`) and an expression 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 env (IF b a1 a2) =
     (if evalb env b then evala env a1 else evala env a2)"
  "evala env (Sum a1 a2) = evala env a1 + evala env a2"
  "evala env (Diff a1 a2) = evala env a1 - evala env a2"
  "evala env (Var v) = env v"
  "evala env (Num n) = n"

  "evalb env (Less a1 a2) = (evala env a1 < evala env a2)"
  "evalb env (And b1 b2) = (evalb env b1 & evalb env b2)"
  "evalb env (Neg b) = (~ evalb env b)"
```

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

```
consts substa :: ('a => 'b aexp) => 'a aexp => 'b aexp
       substb :: ('a => 'b aexp) => 'a bexp => '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
  "substa s (IF b a1 a2) =
     IF (substb s b) (substa s a1) (substa s a2)"
  "substa s (Sum a1 a2) = Sum (substa s a1) (substa s a2)"
  "substa s (Diff a1 a2) = Diff (substa s a1) (substa s a2)"
  "substa s (Var v) = s v"
  "substa s (Num n) = Num n"

  "substb s (Less a1 a2) = Less (substa s a1) (substa s a2)"
  "substb s (And b1 b2) = And (substb s b1) (substb s b2)"
  "substb s (Neg b) = Neg (substb 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:

```
Goal
 "evala env (substa s a) = evala (%x. evala env (s x)) a & \
\ evalb env (substb s b) = evalb (%x. evala env (s x)) b";
by(induct_tac "a b" 1);
```

The resulting 8 goals (one for each constructor) are proved in one fell swoop `by Auto_tac;`.

In general, given n mutually recursive datatypes $\tau_1, \ldots, \tau_n$, Isabelle expects an inductive proof to start with a goal of the form

$$P_1(x_1) \ \& \ \ldots \ \& \ P_n(x_n)$$

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

```
by(induct_tac "x1 ... xn" k);
```

Exercise 3.4.1 Define a function `norma` of type `'a aexp => 'a aexp` that replaces `IF`s with complex boolean conditions by nested `IF`s 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)
```

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
   substs :: ('a => ('a,'b)term) => ('a,'b)term list => ('a,'b)term list

primrec
  "subst s (Var x) = s x"
  "subst s (App f ts) = App f (substs s ts)"

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

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:

```
Goal "subst  Var t  = (t ::('a,'b)term)  & \
\     substs Var ts = (ts::('a,'b)term list)";
by(induct_tac "t ts" 1);
by(Auto_tac);
```

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:

```
subst (f o g) t = subst f (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 the theorem `o_def`).

Returning to the definition of `subst`, we have to admit that it does not really need the auxiliary `substs` function. The `App`-case can directly be expressed as

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

Although Isabelle insists on the more verbose version, we can easily *prove* that the `map`-equation holds: simply by induction on `ts` followed by `Auto_tac`.

Exercise 3.4.3 Define a function `rev_term` of type `term -> term` that recursively reverses the order of arguments of all function symbols in a term. Prove that `rev_term(rev_term t) = t`.

Of course, you may also combine mutual and nested recursion as in the following example

```
datatype expr = Var string | Lam string expr | App expr expr
              | Data data
and      data = Bool bool | Num nat
              | Closure string expr "(string * data)list"
```

taken from an operational semantics of applied lambda terms. Note that double quotes are required around the type involving `*`, as explained on page 8.

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. The `data` example above involves products, which is fine because products are also datatypes. However, functions are most emphatically not allowed:

```
datatype t = C (t => 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 => bool` have the same cardinality—an impossibility. In theory, we could allow limited forms of recursion involving function spaces. For example

```
datatype t = C (nat => t) | D
```

is unproblematic. However, Isabelle does not support recursion involving `=>` at all at the moment.

For a theoretical analysis of what kinds of datatypes are feasible in HOL see, for example, [2]. There are alternatives to pure HOL: LCF [7] is a logic where types like

```
datatype t = C (t -> t)
```

do indeed make sense (note the intentionally different arrow `->`!). There is even a version of LCF on top of HOL, called HOLCF [4].

3.4.4 Case study: Tries

Tries are a classic search tree data structure [3] 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 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.

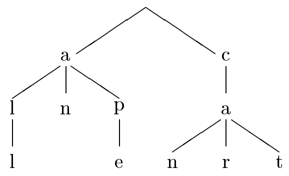

Figure 3.1: A sample trie

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:

```
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 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. We define two corresponding 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:

```
Goal "lookup (Trie None []) as = None";
by(exhaust_tac "as" 1);
by(Auto_tac);
```

This lemma is added to the simpset as usual.

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!

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

```
Goal "!t v bs. lookup (update t as v) bs = \
\              (if as=bs then Some v else lookup t bs)";
```

Our plan will be 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:

```
by(induct_tac "as" 1);
by(Auto_tac);
```

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

```
1. ... ==> ... lookup (...) bs = lookup t bs
2. ... ==> ... lookup (...) bs = lookup t bs
3. ... ==> ... 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(ALLGOALS (exhaust_tac "bs"));
```

The *tactical* `ALLGOALS` merely applies the tactic following it to all subgoals, which results in a total of six subgoals; `Auto_tac` solves them all.

This proof may look surprisingly straightforward. The reason is that we have told the simplifier (without telling the reader) to expand all `let`s and to split all `case`-constructs over options before we started on the main goal:

```
Addsimps [Let_def];
Addsplits [option.split];
```

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 Modify `update` so that it can also *delete* entries from a trie. It is up to you if you want to shrink tries if possible. Prove (a modified version of) the main theorem above.

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 => nat
recdef fib "measure(%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 `%n. n`** that 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 (%(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 (%xs. 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:

```
recdef sep "measure (%(a,xs). length xs)"
  "sep(a, x#y#zs) = x # a # sep(a,y#zs)"
  "sep(a, xs)     = xs"
```

This defines exactly the same function `sep` as further above.

! Currently `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

```
consts sep :: 'a list => 'a => 'a list
recdef sep "measure length"
  "sep (x#y#zs) = (%a. x # a # sep zs a)"
  "sep xs       = (%a. xs)"
```

When loading a theory containing a `recdef` of a function f, Isabelle proves the recursion equations and stores the result as a list of theorems f`.rules`. It can be viewed by typing

```
prths f.rules;
```

All of the above examples are simple enough that Isabelle can determine automatically that the measure of the argument goes down in each recursive call. In that case f`.rules` contains precisely the defining equations.

In general, Isabelle may not be able to prove all termination conditions automatically. For example, termination of

```
consts gcd :: "nat*nat => nat"
recdef gcd "measure ((%(m,n).n))"
  "gcd (m, n) = (if n=0 then m else gcd(n, m mod n))"
```

relies on the lemma `mod_less_divisor`

```
0 < n ==> m mod n < n
```

that is not part of the default simpset. As a result, Isabelle prints a warning and `gcd.rules` contains a precondition:

```
(! m n. 0 < n --> m mod n < n) ==> gcd (m, n) = (if n=0 ...)
```

We need to instruct `recdef` to use an extended simpset to prove the termination condition:

```
recdef gcd "measure ((%(m,n).n))"
  simpset "simpset() addsimps [mod_less_divisor]"
  "gcd (m, n) = (if n=0 then m else gcd(n, m mod n))"
```

This time everything works fine and `gcd.rules` contains precisely the stated recursion equation for `gcd`.

When defining some nontrivial total recursive function, the first attempt will usually generate a number of termination conditions, some of which may require new lemmas to be proved in some of the parent theories. Those lemmas can then be added to the simpset used by `recdef` for its proofs, as shown for `gcd`.

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

```
consts ack :: "nat*nat => nat"
recdef ack "measure(%m. m) ** measure(%n. 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 [5] and the examples in the library.

3.5.2 Deriving simplification rules

Once we have succeeded in proving all termination conditions, we can start to derive some theorems. In contrast to `primrec` definitions, which are automatically added to the simpset, `recdef` rules must be included explicitly, for example as in

```
Addsimps fib.rules;
```

However, some care is necessary now, in contrast to `primrec`. Although `gcd` is a total function, its defining equation leads to nontermination of the simplifier, because the subterm `gcd(n, m mod n)` on the right-hand side can again be simplified by the same equation, and so on. The reason: the simplifier rewrites the `then` and `else` branches of a conditional if the condition simplifies to neither `True` nor `False`. Therefore it is recommended to derive an alternative formulation that replaces case distinctions on the right-hand side by conditional equations. For `gcd` it means we have to prove


```
           gcd (m, 0) = m
n ~= 0 ==> gcd (m, n) = gcd(n, m mod n)
```

To avoid nontermination during those proofs, we have to resort to some low level tactics:

```
Goal "gcd(m,0) = m";
by(resolve_tac [trans] 1);
by(resolve_tac gcd.rules 1);
by(Simp_tac 1);
```

At this point it is not necessary to understand what exactly `resolve_tac` is doing. The main point is that the above proof works not just for this one example but in general (except that we have to use `Asm_simp_tac` and *f*`.rules` in general). Try the second `gcd`-equation above!

3.5.3 Induction

Assuming we have added the recursion equations (or some suitable derived equations) to the simpset, 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*. 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. Applying *f*`.induct` requires its explicit instantiation. See §**??** for details.

Appendix

and	arities	assumes	axclass	binder
classes	constdefs	consts	default	defines
defs	end	fixes	global	inductive
infixl	infixr	instance	local	locale
mixfix	ML	MLtext	nonterminals	oracle
output	path	primrec	rules	setup
syntax	translations	typedef	types	

Figure A.1: Keywords in theory files

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

Figure A.2: Reserved words in HOL

Bibliography

[1] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[2] Elsa L. Gunter. Why we can't have SML style datatype declarations in HOL. In L.J.M. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications: Proc. IFIP TC10/WG10.2 Intl. Workshop, 1992*, pages 561–568. North-Holland, 1993.

[3] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1975.

[4] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *J. Func. Prog.*, 1999.

[5] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle's Logics: HOL*.

[6] Lawrence C. Paulson. *The Isabelle Reference Manual*.

[7] Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[8] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

Index