2

Programming and Proving

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

2.1 Basics

2.1.1 Types, Terms and Formulas

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

base types, in particular *bool*, the type of truth values, *nat*, the type of natural numbers ($\mathbb{N}$), and *int*, the type of mathematical integers ($\mathbb{Z}$).

type constructors, in particular *list*, the type of lists, and *set*, the type of sets. Type constructors are written postfix, i.e., after their arguments. For example, *nat list* is the type of lists whose elements are natural numbers.

function types, denoted by $\Rightarrow$.

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

Note that *'a* $\Rightarrow$ *'b list* means *"'a* $\Rightarrow$ (*'b list*)*"*, not (*'a* $\Rightarrow$ *'b*) *list*: postfix type constructors have precedence over $\Rightarrow$.

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

There are many predefined infix symbols like $+$ and $\leqslant$. The name of the corresponding binary function is *op* $+$, not just $+$. That is, $x + y$ is nice surface syntax ("syntactic sugar") for *op* $+\ x\ y$.

HOL also supports some basic constructs from functional programming:

T. Nipkow and G. Klein, *Concrete Semantics*,
DOI 10.1007/978-3-319-10542-0_2

$(\mathit{if}\ b\ \mathit{then}\ t_1\ \mathit{else}\ t_2)$
$(\mathit{let}\ x = t\ \mathit{in}\ u)$
$(\mathit{case}\ t\ \mathit{of}\ \mathit{pat}_1 \Rightarrow t_1 \mid \ldots \mid \mathit{pat}_n \Rightarrow t_n)$


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

Terms may also contain λ-abstractions. For example, $\lambda x.\ x$ is the identity function.

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

Equality is available in the form of the infix function $=$ of type $'a \Rightarrow 'a \Rightarrow \mathit{bool}$. It also works for formulas, where it means "if and only if".

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

Isabelle automatically computes the type of each variable in a term. This is called **type inference**. Despite type inference, it is sometimes necessary to attach an explicit **type constraint** (or **type annotation**) to a variable or term. The syntax is $t :: \tau$ as in $m + (n::\mathit{nat})$. Type constraints may be needed to disambiguate terms involving overloaded functions such as $+$.

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


Right-arrows of all kinds always associate to the right. In particular, the formula $A_1 \Longrightarrow A_2 \Longrightarrow A_3$ means $A_1 \Longrightarrow (A_2 \Longrightarrow A_3)$. The (Isabelle-specific[1]) notation $[\![\ A_1; \ldots; A_n\ ]\!] \Longrightarrow A$ is short for the iterated implication $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A$. Sometimes we also employ inference rule notation: $\dfrac{A_1 \quad \ldots \quad A_n}{A}$

2.1.2 Theories

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

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

[1] To display implications in this style in Isabelle/jedit you need to set Plugins > Plugin Options > Isabelle/General > Print Mode to "`brackets`" and restart.

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


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

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

2.1.3 Quotation Marks

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

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

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

2.2.1 Type *bool*

The type of boolean values is a predefined datatype

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

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

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

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

2.2.2 Type *nat*

Natural numbers are another predefined datatype:

datatype $nat = 0 \mid Suc\ nat$

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

fun $add :: "nat \Rightarrow nat \Rightarrow nat"$ **where**
$"add\ 0\ n = n" \mid$
$"add\ (Suc\ m)\ n = Suc(add\ m\ n)"$

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

lemma add_02: $"add\ m\ 0 = m"$
apply$(induction\ m)$
apply$(auto)$
done

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

1. $add\ 0\ 0 = 0$
2. $\bigwedge m.\ add\ m\ 0 = m \Longrightarrow add\ (Suc\ m)\ 0 = Suc\ m$

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

thm add_02

which displays

$add\ ?m\ 0 = ?m$

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

Note that there is also a proof method *induct*, which behaves almost like *induction*; the difference is explained in Chapter 5.

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

Numerals (0, 1, 2, ...) and most of the standard arithmetic operations ($+$, $-$, $*$, $\leqslant$, $<$, etc.) are overloaded: they are available not just for natural numbers but for other types as well. For example, given the goal $x + 0 = x$, there is nothing to indicate that you are talking about natural numbers. Hence Isabelle can only infer that x is of some arbitrary type where 0 and $+$ exist. As a consequence, you will be unable to prove the goal. In this particular example, you need to include an explicit type constraint, for example $x{+}0 = (x{::}nat)$. If there is enough contextual information this may not be necessary: $Suc\ x = x$ automatically implies $x{::}nat$ because *Suc* is not overloaded.

An Informal Proof

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

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

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

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

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

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

2.2.3 Type *list*

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

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

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

We also define two standard functions, append and reverse:

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

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

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

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

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

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

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

yields *Cons b (Cons a Nil)*.

Figure 2.1 shows the theory created so far. Because *list*, *Nil*, *Cons*, etc. are already predefined, Isabelle prints qualified (long) names when executing this theory, for example, *MyList.Nil* instead of *Nil*. To suppress the qualified names you can insert the command `declare [[names_short]]`. This is not recommended in general but is convenient for this unusual example.

Structural Induction for Lists

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

```
theory MyList
imports Main
begin

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

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

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

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

(* a comment *)

end
```

Fig. 2.1. A theory of lists

though the length remains implicit. To prove that some property P holds for all lists xs, i.e., $P\ xs$, you need to prove

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

This is often called **structural induction** for lists.

2.2.4 The Proof Process

We will now demonstrate the typical proof process, which involves the formulation and proof of auxiliary lemmas. Our goal is to show that reversing a list twice produces the original list.

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

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

apply(*induction xs*)

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

1. $rev\ (rev\ Nil) = Nil$
2. $\bigwedge x1\ xs.$
   $rev\ (rev\ xs) = xs \Longrightarrow rev\ (rev\ (Cons\ x1\ xs)) = Cons\ x1\ xs$

Let us try to solve both goals automatically:

apply$(auto)$

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

1. $\bigwedge x1\ xs.$
   $rev\ (rev\ xs) = xs \Longrightarrow$
   $rev\ (app\ (rev\ xs)\ (Cons\ x1\ Nil)) = Cons\ x1\ xs$

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

A First Lemma

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

lemma $rev\_app\ [simp]$: $"rev(app\ xs\ ys) = app\ (rev\ ys)\ (rev\ xs)"$

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

apply$(induction\ xs)$

This time not even the base case is solved automatically:

apply$(auto)$
1. $rev\ ys = app\ (rev\ ys)\ Nil$
A total of 2 subgoals...

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

A Second Lemma

We again try the canonical proof procedure:

lemma $app\_Nil2\ [simp]$: $"app\ xs\ Nil = xs"$
apply$(induction\ xs)$
apply$(auto)$
done

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

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

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

1. $\bigwedge x1\ xs.$
   $rev\ (app\ xs\ ys) = app\ (rev\ ys)\ (rev\ xs) \Longrightarrow$
   $app\ (app\ (rev\ ys)\ (rev\ xs))\ (Cons\ x1\ Nil) =$
   $app\ (rev\ ys)\ (app\ (rev\ xs)\ (Cons\ x1\ Nil))$

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

Associativity of *app*

The canonical proof procedure succeeds without further ado:

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

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

Another Informal Proof

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

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

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

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

  and we need to show

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

The proof is as follows:

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

Didn't we say earlier that all proofs are by simplification? But in both cases, going from left to right, the last equality step is not a simplification at all! In the base case it is $app\ ys\ zs = app\ Nil\ (app\ ys\ zs)$. It appears almost mysterious because we suddenly complicate the term by appending Nil on the left. What is really going on is this: when proving some equality $s = t$, both s and t are simplified until they "meet in the middle". This heuristic for equality proofs works well for a functional programming context like ours. In the base case both $app\ (app\ Nil\ ys)\ zs$ and $app\ Nil\ (app\ ys\ zs)$ are simplified to $app\ ys\ zs$, the term in the middle.

2.2.5 Predefined Lists

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

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

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

```
fun map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list"
"map_list f Nil = Nil" |
"map_list f (Cons x21.0 x22.0) = Cons (f x21.0) (map_list f x22.0)"
```

Also useful are the **head** of a list, its first element, and the **tail**, the rest of the list:

```
fun hd :: 'a list ⇒ 'a
hd (x # xs) = x

fun tl :: 'a list ⇒ 'a list
tl [] = [] |
tl (x # xs) = xs
```

Note that since HOL is a logic of total functions, $hd\ []$ is defined, but we do now know what the result is. That is, $hd\ []$ is not undefined but underdefined.

From now on lists are always the predefined lists.

Exercises

Exercise 2.1. Use the **value** command to evaluate the following expressions: $"1 + (2::nat)"$, $"1 + (2::int)"$, $"1 - (2::nat)"$ and $"1 - (2::int)"$.

Exercise 2.2. Start from the definition of *add* given above. Prove that *add* is associative and commutative. Define a recursive function $double :: nat \Rightarrow nat$ and prove $double\ m = add\ m\ m$.

Exercise 2.3. Define a function $count :: 'a \Rightarrow 'a\ list \Rightarrow nat$ that counts the number of occurrences of an element in a list. Prove $count\ x\ xs \leqslant length\ xs$.

Exercise 2.4. Define a recursive function $snoc :: 'a\ list \Rightarrow 'a \Rightarrow 'a\ list$ that appends an element to the end of a list. With the help of *snoc* define a recursive function $reverse :: 'a\ list \Rightarrow 'a\ list$ that reverses a list. Prove $reverse\ (reverse\ xs) = xs$.

Exercise 2.5. Define a recursive function $sum :: nat \Rightarrow nat$ such that $sum\ n = 0 + ... + n$ and prove $sum\ n = n * (n + 1)\ div\ 2$.

2.3 Type and Function Definitions

Type synonyms are abbreviations for existing types, for example

type_synonym *string* = *"char list"*

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

2.3.1 Datatypes

The general form of a datatype definition looks like this:

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

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

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

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

Like in functional programming languages, datatype values can be taken apart with case expressions, for example

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

Case expressions must be enclosed in parentheses.

As an example of a datatype beyond *nat* and *list*, consider binary trees:

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

with a mirror function:

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

The following lemma illustrates induction:

lemma *"mirror(mirror t) = t"*
apply(*induction t*)

yields

1. *mirror (mirror Tip) = Tip*
2. $\bigwedge$*t1 x2 t2.*
   ⟦*mirror (mirror t1) = t1*; *mirror (mirror t2) = t2*⟧
   ⟹ *mirror (mirror (Node t1 x2 t2)) = Node t1 x2 t2*

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

A very simple but also very useful datatype is the predefined

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

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

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

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

Note that $\tau_1 * \tau_2$ is the type of pairs, also written $\tau_1 \times \tau_2$. Pairs can be taken apart either by pattern matching (as above) or with the projection functions *fst* and *snd*: *fst* $(x, y) = x$ and *snd* $(x, y) = y$. Tuples are simulated by pairs nested to the right: (a, b, c) is short for $(a, (b, c))$ and $\tau_1 \times \tau_2 \times \tau_3$ is short for $\tau_1 \times (\tau_2 \times \tau_3)$.

2.3.2 Definitions

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

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

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

2.3.3 Abbreviations

Abbreviations are similar to definitions:

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

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

The ASCII representation of ≡ is == or `\<equiv>`.

2.3.4 Recursive Functions

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

Isabelle's automatic termination checker requires that the arguments of recursive calls on the right-hand side must be strictly smaller than the arguments on the left-hand side. In the simplest case, this means that one fixed argument position decreases in size with each recursive call. The size is measured as the number of constructors (excluding 0-ary ones, e.g., *Nil*). Lexicographic combinations are also recognized. In more complicated situations, the user may have to prove termination by hand. For details see [49].

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

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

does not just define *div2* but also proves a customized induction rule:

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

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

lemma *"div2(n) = n div 2"*
apply(*induction n rule*: *div2.induct*)

(where the infix *div* is the predefined division operation) yields the subgoals

1. *div2 0 = 0 div 2*
2. *div2 (Suc 0) = Suc 0 div 2*
3. $\bigwedge n.$ *div2 n = n div 2 ⟹*
   div2 (Suc (Suc n)) = Suc (Suc n) div 2

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

This example leads to the following induction heuristic:

> *Let f be a recursive function. If the definition of f is more complicated than having one equation for each constructor of some datatype, then properties of f are best proved via f.induct.*

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

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

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

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

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

apply(*induction* x_1 ... x_n *rule*: $f.induct$)

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

Exercises

Exercise 2.6. Starting from the type *'a tree* defined in the text, define a function *contents* :: *'a tree* $\Rightarrow$ *'a list* that collects all values in a tree in a list, in any order, without removing duplicates. Then define a function *treesum* :: *nat tree* $\Rightarrow$ *nat* that sums up all values in a tree of natural numbers and prove *treesum t* = *listsum* (*contents t*).

Exercise 2.7. Define a new type *'a tree2* of binary trees where values are also stored in the leaves of the tree. Also reformulate the *mirror* function accordingly. Define two functions *pre_order* and *post_order* of type *'a tree2* $\Rightarrow$ *'a list* that traverse a tree and collect all stored values in the respective order in a list. Prove *pre_order* (*mirror t*) = *rev* (*post_order t*).

Exercise 2.8. Define a function *intersperse* :: *'a* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* such that *intersperse a* [x_1, ..., x_n] = [x_1, *a*, x_2, *a*, ..., *a*, x_n]. Now prove that *map f* (*intersperse a xs*) = *intersperse* (*f a*) (*map f xs*).

2.4 Induction Heuristics

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

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

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

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

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

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

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

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

There is no choice as to the induction variable:

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

Unfortunately, this attempt does not prove the induction step:

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

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

Generalize goals for induction by replacing constants by variables.

Of course one cannot do this naively: *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*, as required. In this instance it was easy to guess the right generalization. Other situations can require a good deal of creativity.

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

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

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

but the induction hypothesis needs to be applied with $a \# ys$ instead of ys. Hence we prove the theorem for all ys instead of a fixed one. We can instruct induction to perform this generalization for us by adding *arbitrary*: ys.

apply(*induction xs arbitrary*: *ys*)

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

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

Thus the proof succeeds:

apply *auto*
done

This leads to another heuristic for generalization:

> *Generalize induction by generalizing all free variables*
> (except the induction variable itself).

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

Exercises

Exercise 2.9. Write a tail-recursive variant of the *add* function on *nat*: *itadd*. Tail-recursive means that in the recursive case, *itadd* needs to call itself directly: $itadd\ (Suc\ m)\ n = itadd\ \ldots$. Prove $itadd\ m\ n = add\ m\ n$.

2.5 Simplification

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

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

To emphasize the directionality, equations that have been given the *simp* attribute are called **simplification rules**. Logically, they are still symmetric,

but proofs by simplification use them only in the left-to-right direction. The proof tool that performs simplifications is called the **simplifier**. It is the basis of *auto* and other related proof methods.

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

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

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

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

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

2.5.1 Simplification Rules

The attribute *simp* declares theorems to be simplification rules, which the simplifier will use automatically. In addition, **datatype** and **fun** commands implicitly declare some simplification rules: **datatype** the distinctness and injectivity rules, **fun** the defining equations. Definitions are not declared as simplification rules automatically! Nearly any theorem can become a simplification rule. The simplifier will try to transform it into an equation. For example, the theorem $\neg\ P$ is turned into $P = False$.

Only equations that really simplify, like $rev\ (rev\ xs) = xs$ and $xs\ @\ [] = xs$, should be declared as simplification rules. Equations that may be counterproductive as simplification rules should only be used in specific proof steps (see Section 2.5.4 below). Distributivity laws, for example, alter the structure of terms and can produce an exponential blow-up.

2.5.2 Conditional Simplification Rules

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

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

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

2.5.3 Termination

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

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

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

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

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

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

2.5.4 The *simp* Proof Method

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

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

the command

apply$(\mathit{simp}\ \mathit{add}\text{: } \mathit{th}_1\ \ldots\ \mathit{th}_n)$

simplifies the assumptions P_i and the conclusion C using

- all simplification rules, including the ones coming from **datatype** and **fun**,
- the additional lemmas $\mathit{th}_1\ \ldots\ \mathit{th}_n$, and

- the assumptions.

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

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

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

Note that *simp* acts only on subgoal 1, *auto* acts on all subgoals. There is also *simp_all*, which applies *simp* to all subgoals.

2.5.5 Rewriting with Definitions

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

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

apply(*simp add*: *f_def*)

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

2.5.6 Case Splitting With *simp*

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

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

For example, *simp* can prove

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

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

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

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

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

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

splits all case-expressions over natural numbers. For an arbitrary datatype *t* it is *t.split* instead of *nat.split*. Method *auto* can be modified in exactly the same way. The modifier *split*: can be followed by multiple names. Splitting if or case-expressions in the assumptions requires *split*: *if_splits* or *split*: *t.splits*.

Exercises

Exercise 2.10. Define a datatype *tree0* of binary tree skeletons which do not store any information, neither in the inner nodes nor in the leaves. Define a function *nodes* :: *tree0* ⇒ *nat* that counts the number of all nodes (inner nodes and leaves) in such a tree. Consider the following recursive function:

fun *explode* :: *"nat* ⇒ *tree0* ⇒ *tree0"* **where**
"explode 0 t = t" |
"explode (Suc n) t = explode n (Node t t)"

Find an equation expressing the size of a tree after exploding it (*nodes* (*explode n t*)) as a function of *nodes t* and *n*. Prove your equation. You may use the usual arithmetic operators, including the exponentiation operator " ^". For example, 2 ^ 2 = 4.

Hint: simplifying with the list of theorems *algebra_simps* takes care of common algebraic properties of the arithmetic operators.

Exercise 2.11. Define arithmetic expressions in one variable over integers (type *int*) as a data type:

datatype *exp* = *Var* | *Const int* | *Add exp exp* | *Mult exp exp*

Define a function *eval* :: *exp* ⇒ *int* ⇒ *int* such that *eval e x* evaluates *e* at the value *x*.

A polynomial can be represented as a list of coefficients, starting with the constant. For example, [4, 2, − 1, 3] represents the polynomial $4+2x-x^2+3x^3$. Define a function *evalp* :: *int list* ⇒ *int* ⇒ *int* that evaluates a polynomial at the given value. Define a function *coeffs* :: *exp* ⇒ *int list* that transforms an expression into a polynomial. This may require auxiliary functions. Prove that *coeffs* preserves the value of the expression: *evalp* (*coeffs e*) *x* = *eval e x*. Hint: consider the hint in Exercise 2.10.

http://www.springer.com/978-3-319-10541-3

Concrete Semantics
With Isabelle/HOL
Nipkow, T.; Klein, G.
2014, XIII, 298 p. 87 illus., 1 illus. in color., Hardcover
ISBN: 978-3-319-10541-3