A Gentle Introduction to Isabelle and Isabelle HOL

Gunnar Teege

February 12, 2023

Contents

Chapter 1

Isabelle System

Isabelle is a "proof assistant" for formal mathematical proofs. It supports a notation for propositions and their proofs, it can check whether a proof is correct, and it can even help to find a proof.

This introductory manual explains how to work with Isabelle to develop mathematical models. It does not presume prior knowledge about formal or informal proof techniques. It only assumes that the reader has a basic understanding of mathematical logics and the task of proving mathematical propositions.

1.1 Invoking Isabelle

After installation, Isabelle can be invoked interactively as an editor for entering propositions and proofs, or it can be invoked noninteractively to check a proof and generate a PDF document which displays the propositions and proofs.

1.1.1 Installation and Configuration

Isabelle is freely available from `https://isabelle.in.tum.de/` for Windows, Mac, and Linux. It is actively maintained, there is usually one release every year. Older releases are available in a distribution archive.

To install Isabelle, follow the instructions on

```
https://isabelle.in.tum.de/installation.html
```

Although there are many configuration options, there is no need for an initial configuration, interactive and noninteractive invocation is immediately possible.

1.1.2 Theories and Sessions

The propositions and proofs in Isabelle notation are usually collected in "theory files" with names of the form `name.thy`. A theory file must import at least one other theory file to build upon its content. For theories based on higher order logic ("HOL"), the usual starting point to import is the theory *Main*.

Several theory files can be grouped in a "session". A session is usually stored in a directory in the file system. It consists of a file named `ROOT` which contains a specification of the session, and the theory files which belong to the session.

When Isabelle loads a session it loads and checks all its theory files. Then it can generate a "heap file" for the session which contains the processed session content. The heap file can be reloaded by Isabelle to avoid the time and effort for processing and checking the theory files.

A session always has a single parent session, with the exception of the Isabelle builtin session `Pure`. Thus, every session depends on a linear sequence of ancestor sessions which begins at `Pure`. The ancestor sessions have separate heap files. A session is always loaded together with all ancestor sessions.

Every session has a name of the form `chap/sess` where `chap` is an arbitrary "chapter name", it defaults to `Unsorted`. The session name and the name of the parent session are specified in the `ROOT` file in the session directory. When a session is loaded by Isabelle, its directory and the directories of all ancestor sessions must be known by Isabelle.

The Isabelle distribution provides heap files for the session `HOL/HOL` and its parent session `Pure/Pure`, the session directories are automatically known.

Every session may be displayed in a "session document". This is a PDF document generated by translating the content of the session theory files to LaTeX. A frame LaTeX document must be provided which includes all content generated from the theory files. The path of the frame document, whether a session document shall be generated and which theories shall be included is specified in the `ROOT` file.

The command

```
isabelle mkroot [OPTIONS] [Directory]
```

can be used to initialize the given directory (default is the current directory) as session directory. It creates an initial `ROOT` file to be populated with theory file names and other specification for the session, and it creates a simple frame LaTeX document.

1.1.3 Invocation as Editor

Isabelle is invoked for editing using the command

```
isabelle jedit [OPTIONS] [Files ...]
```

It starts an interactive editor and opens the specified theory files. If no file is specified it opens the file `Scratch.thy` in the user's home directory. If that file does not exist, it is created as an empty file.

The editor also loads a session (together with its ancestors), the default session to load is HOL. If a heap file exists for the loaded session it is used, otherwise a heap file is created by processing all the session's theories.

The default session to load can be changed by the option

```
-l <session name>
```

Moreover the editor also loads (but does not open) theories which are transitively imported by the opened theory files. If these are Isabelle standard theories it finds them automatically. If they belong to the session in the current directory it also finds them. If they belong to other sessions, the option

```
-d <directory pathname>
```

must be used to make the session directory known to Isabelle. For every used session a separate option must be specified.

If an imported theory belongs to the loaded session or an ancestor, it is directly referenced there. Otherwise the theory file is loaded and processed.

1.1.4 Invocation for Batch Processing

Isabelle is invoked for batch processing of all theory files in one or more sessions using the command

```
isabelle build [OPTIONS] [Sessions ...]
```

It loads all theory files of the specified sessions and checks the contained proofs. It also loads all required ancestor sessions. If not known to Isabelle, the corresponding session directories must be specified using option `-d` as described in Section 1.1.3. Sessions required for other sessions are loaded from heap files if existent, otherwise the corresponding theories are loaded and a heap file is created.

If option `-b` is specified, heap files are also created for all sessions specified in the command. Option `-c` clears the specified sessions (removes their heap files) before processing them. Option `-n` omits the actual session processing, together with option `-c` it can be used to simply clear the heap files.

The specified sessions are only processed if at least one of their theory files has changed since the last processing or if the session is cleared using option

`-c`. If option `-v` is specified all loaded sessions and all processed theories are listed on standard output.

If specified for a session in its `ROOT` file (see Section 1.1.5), also the session document is generated when a session is processed.

1.1.5 Invocation for Document Creation

** todo **

1.2 Interactively Working with Isabelle

After invoking Isabelle as editor (see Section 1.1.3) it supports interactive work with theories.

The user interface consists of a text area which is surrounded by docking areas where additional panels can be displayed. Several panels can be displayed in the same docking area, using tabs to switch among them. Panels may also be displayed as separate undocked windows.

A panel can be displayed by selecting it in the `Plugins -> Isabelle` menu. Some of the panels are described in the following sections.

1.2.1 The Text Area

The text area displays the content of an open theory file and supports editing it. The font (size) used for display can be configured through the menu in `Utilities -> Global Options -> jEdit -> Text Area` together with many other options for display.

Moreover, in the default configuration, Isabelle automatically processes the theory text up to the current part visible in the text area window. This includes processing the content of all imported theory files, if the import statement is visible.

Whenever the window is moved forward the processing is continued if "Continuous checking" has not been disabled in the Theories panel. Whenever the content of the text area is modified the processing is set back and restarted at the modified position.

In the default configuration the progress of the processing is shown by shading the unprocessed text in red and by a bar on the right border of the text area which symbolizes the whole theory file and shows the unprocessed part by red shading as well.

1.2.2 The Sidekick Panel

todo

1.2.3 The Output Panel

The Output panel displays the result of the theory text processing when it reaches the cursor position in the text area.

The displayed information depends on the cursor position and may be an information about the current theorem or proof or it may be an error message.

1.2.4 The State Panel

The State panel displays a specific result of the theory text processing if the cursor position is in a proof. It is called the "goal state" (see Section 2.2.2), and describes what remains to be proven by the rest of the proof.

The Output panel can be configured to include the goal state in its display by checking the "Proof state" button.

1.2.5 The Symbols Panel

Isabelle uses a large set of mathematical symbols and other special symbols which are usually not on the keyboard. The Symbols panel can be used to input such symbols in the text area.

1.2.6 The Documentation Panel

A comprehensive set of documentation documents about Isabelle can be opened through the Documentation Panel. This manual refers to some of these documentations, if applicable.

For example, more information about the use of the interactive editor can be found in the Isabelle Reference Manual about jedit.

1.2.7 The Query Panel

** todo**

1.2.8 The Theories Panel

The Theories panel displays the loaded session and the opened or imported theories which do not belong to the loaded session or its ancestors. The (parts of) theories which have not been processed are shaded in red.

If the check button next to a theory is checked, the theory file is processed independently of the position of the text area window.

Chapter 2

Isabelle Basics

The basic mechanisms of Isabelle mainly support defining types, constants, and functions and specifying and proving statements about them.

2.1 Isabelle Theories

A theory is the content of an Isabelle theory file.

2.1.1 Theory Structure

The content of a theory file has the structure

theory *name*
imports $name_1$ ... $name_n$
begin
  ...
end

where *name* is the theory name and $name_1$ ... $name_n$ are the names of the imported theories. The theory name *name* must be the same which is used for the theory file, i.e., the file name must be `name.thy`.

The theory structure is a part of the Isabelle "outer syntax" which is mainly fixed and independent from the specific theories. Other kind of syntax is embedded into the outer syntax. The main embedded syntax is the "inner syntax" which is mainly used to denote types and terms. Content in inner syntax must always be surrounded by double quotes. The only exception is a single isolated identifier, for it the quotes may be omitted.

This introduction describes only a selected part of the outer syntax. The full outer syntax is described in the Isabelle/Isar Reference Manual.

Additionally, text written in LaTeX syntax can be embedded into the outer syntax using the form **text**‹ ... › and LaTeX sections can be created using

chapter‹ ... ›, **section**‹ ... ›, **subsection**‹ ... ›, **subsubsection**‹ ... ›, **paragraph**‹ ... ›, **subparagraph**‹ ... ›. Note that the delimiters used here are not the "lower" and "greater" symbols, but the "cartouche delimiters" available in the editor's Symbols panel in tab "Punctuation".

Embedded LaTeX text is intended for additional explanations of the formal theory content. It is displayed in the session document together with the formal theory content.

It is also possible to embed inner and outer syntax in the LaTeX syntax (see Chapter 4 in the Isabelle/Isar Reference Manual).

Moreover, comments of the form

```
(* ... *)
```

can be embedded into the outer syntax. They are only intended for the reader of the theory file and are not displayed in the session document.

2.1.2 Types

As usual in formal logics, the basic building blocks of propositions are terms. Terms denote arbitrary objects like numbers, sets, functions, or boolean values. Isabelle is strongly typed, so every term must have a type. However, in most situations Isabelle can derive the type of a term automatically, so that it needs not be specified explicitly. Terms and types are always denoted using the inner syntax.

Types are usually specified by type names. In Isabelle HOL (see Chapter 3) there are predefined types such as *nat* and *bool* for natural numbers and boolean values. New types can be defined in the form

typedecl *name*

which introduces the *name* for a new type for which the values are different from the values of all existing types and the set of values is not empty. No other information about the values is given, that must be done separately. See Chapter 3 for ways of defining types with specifying more information about their values.

Types can be parameterized, then the type arguments are denoted *before* the type name, such as in *nat set* which is the type of sets of natural numbers. A type name with *n* parameters is declared in the form

typedecl *('name$_1$,...,'name$_n$) name*

such as **typedecl** *('a) set*. The type parameters are denoted by "type variables" which always have the form *'name* with a leading single quote character. Every use where the parameters are replaced by actual types, such as in *nat set*, is called an "instance" of the parameterized type.

Alternatively a type name can be introduced as a synonym for an existing type in the form

type_synonym *name = type*

such as in **type_synonym** *natset = nat set*. Type synonyms can also be parameterized as in

type_synonym *('name$_1$,...,'name$_n$) name = type*

where the type variables occur in *type* in place of actual type specifications.

2.1.3 Terms

Constants and Variables

Terms are mainly built as syntactical structures based on constants and variables. Constants are usually denoted by names, using the same namespace as type names. Whether a name denotes a constant or a type depends on its position in a term. Predefined constant names of type *bool* are *True* and *False*.

Constants of number types, such as *nat*, may be denoted by number literals, such as *6* or *42*.

A constant can be defined by specifying its type. The definition

consts *name$_1$ :: type$_1$ ... name$_n$ :: type$_n$*

introduces *n* constants with their names and types. No information is specified about the constant's values, in this respect the constants are "underspecified". The information about the values must be specified separately.

If the constant's type contains type variables the constant is called "polymorphic". Thus the declaration

consts *myset :: "'a set"*

declares the polymorphic constant *myset* which may be a set of elements of arbitrary type.

A (term) variable has the same form as a constant name, but it has not been introduced as a constant. Whenever a variable is used in a term it has a specific type which is either derived from its context or is explicitly specified in the form *varname :: type*.

Nested terms are generally written by using parentheses *(...)*. There are many priority rules how to nest terms automatically, but if in doubt, it is always safe to use parentheses.

Functions

A constant name denotes an object, which, according to its type, may also be a function of arbitrary order. Functions basically have a single argument. The type of a function is written as *argtype* ⇒ *restype*.

Functions in Isabelle are always total, i.e., they map every value of type *argtype* to some value of type *restype*. However, a function may be "underspecified" so that no information is (yet) available about the result value for some or all argument values. A function defined by

consts *mystery :: nat ⇒ nat*

is completely underspecified: although it maps every natural number to a unique other natural number no information about these numbers is available. Functions may also be partially specified by describing the result value only for some argument values. This does not mean that the function is "partial" and has no value for the remaining arguments. The information about these values may always be provided later, this does not "modify" the function, it only adds information about it.

Functions with Multiple Arguments

The result type of a function may again be a function type, then it may be applied to another argument. This is used to represent functions with more than one argument. Function types are right associative, thus a type $argtype_1 \Rightarrow argtype_2 \Rightarrow \cdots \Rightarrow argtype_n \Rightarrow restype$ describes a function which can be applied to *n* arguments.

Function application terms for a function *f* and an argument a are denoted by *f* a, no parentheses are required around the argument. Function application terms are left associative, thus a function application to *n* arguments is written *f* a_1 ... a_n. Note that an application *f* a_1 ... a_m where *m* < *n* (a "partial application") is a correct term and denotes a function taking the remaining *n*-*m* arguments.

For every constant alternative syntax forms may be defined for application terms. This is often used for binary functions to represent application terms in infix notation with an operator symbol. As an example, the name for the addition function is *plus*, so an application term is denoted in the form *plus 3 5*. For *plus* an alternative syntax is defined as infix notation using the operator symbol +, therefore the application term can also be denoted by *3 + 5*. Similar infix notations are supported for many basic functions, using operator symbols such as -, **, =, ≠, ≤, or ∈.

Lambda-Terms

Functions can be denoted by lambda terms of the form $\lambda x.\ term$ where x is a variable which may occur in the *term*. A function to be applied to n arguments can be denoted by the lambda term $\lambda x_1\ \ldots\ x_n.\ term$ where $x_1\ \ldots\ x_n$ are distinct variables. As usual, types may be specified for (some of) the variables in the form $\lambda(x_1::t_1)\ \ldots\ (x_n::t_n).\ term$.

2.1.4 Definitions and Abbreviations

A constant name may be introduced together with information about its associated value by specifying a term for the value. There are two forms for introducing constant names in this way, definitions and abbreviations.

Definitions

A definition defines a new constant together with its type and value. It is denoted in the form

definition *name :: type*
where *"name ≡ term"*

Note that the "defining equation" *name* ≡ *term* is specified in inner syntax and, like *type* must be delimited by quotes. The *name* may not occur in the *term*, i.e., this form of definition do not support recursion.
If the type of the defined name is a function type, the *term* may be a lambda term. Alternatively, the definition for a function applicable to n arguments can be written in the form

definition *name :: type*
where *"name* $x_1\ \ldots\ x_n$ *≡ term"*

with variable names $x_1\ \ldots\ x_n$ which may occur in the *term*. This form is mainly equivalent to

definition *name :: type*
where *"name* $\equiv \lambda x_1\ \ldots\ x_n.$ *term"*

A short form of a definition is

definition *"name ≡ term"*

Here, the type of the new constant is derived as the type of the *term*.
Usually, a constant defined in this way is fully specified, i.e., all information about its value is available. However, if the term does not provide this information, the constant is still underspecified. Consider the definition

definition *"mystery2 ≡ mystery"*

where *mystery* is defined as above. Then it is only known that *mystery2* has type *nat ⇒ nat* and is the same total function as *mystery*, but nothing is known about its values.

Abbreviations

An abbreviation definition does not define a constant, it only introduces the name as a synonym for a term. Upon input the name is automatically expanded, and upon output it is used whenever a term matches its specification and the term is not too complex. An abbreviation definition is denoted in a similar form as a definition:

abbreviation *name :: type*
where *"name ≡ term"*

As for definitions, recursion is not supported, the *name* may not occur in the *term*. The short form is also available as for definitions.

The alternative form for functions is also available. The abbreviation definition

abbreviation *name :: type*
where *"name* x_1 ... x_n *≡ term"*

introduces a "parameterized" abbreviation. An application term *name* $term_1$ ... $term_n$ is replaced upon input by *term* where all occurrences of x_i have been substituted by $term_i$. Upon output terms are matched with the structure of *term* and if successful a corresponding application term is constructed and displayed.

2.1.5 Overloading

True Overloading

One way of providing information about the value of an underspecified constant is overloading. It provides the information with the help of another constant together with a definition for it.

Overloading depends on the type. Therefore, if a constant is polymorphic, different definitions can be associated for different type instances.

Overloading is only possible for constants which do not yet have a definition, i.e., they must have been defined by **consts** (see Section 2.1.3). Such a constant *name* is associated with *n* definitions by the following overloading specification:

overloading
  name$_1$ $\equiv$ *name*
    ...
  name$_n$ $\equiv$ *name*
begin
  definition *name*$_1$ *::* *type*$_1$ **where** ...
    ...
  definition *name*$_n$ *::* *type*$_n$ **where** ...
end

where all *type*$_i$ must be instances of the type declared for *name*.

The auxiliary constants *name*$_1$ ... *name*$_n$ are only introduced locally and cannot be used outside of the **overloading** specification.

Adhoc Overloading

There is also a form of overloading which achieves similar effects although it is implemented completely differently. It is only performed on the syntactic level, like abbreviations. To use it, the theory *HOL-Library.Adhoc_Overloading* must be imported by the surrounding theory:

imports *"HOL-Library.Adhoc_Overloading"*

(Here the theory name must be quoted because it contains a minus sign.)

Then a constant name *name* can be defined to be a "type dependent abbreviation" for *n* terms of different type instances by

adhoc_overloading *name* *term*$_1$ ... *term*$_n$

Upon input the type of *name* is determined from the context, then it is replaced by the corresponding *term*$_i$. Upon output terms are matched with the corresponding *term*$_i$ and if successful *name* is displayed instead.

Although *name* must be the name of an existing constant, only its type is used. The constant is not affected by the adhoc overloading, however, it becomes inaccessible because its name is now used as term abbreviation.

Several constant names can be overloaded in a common specification:

adhoc_overloading *name*$_1$ *term*$_{11}$ ... *term*$_{1n}$ **and** ... **and** *name*$_k$ ...

2.1.6 Propositions

A proposition denotes a statement, which can be valid or not. Valid statements are called "facts", they are the main content of a theory. Propositions are specific terms and are hence written in inner syntax and must be enclosed in quotes.

Formulas

In its simplest form a proposition is a single term of type `bool`, such as

```
6 * 7 = 42
```

Terms of type `bool` are also called "formulas".

A proposition may contain free variables as in

```
2 * x = x + x
```

A formula as proposition is valid if it evaluates to `True` for all possible values substituted for the free variables.

Derivation Rules

More complex propositions can express, "derivation rules" used to derive propositions from other propositions. Derivation rules are denoted using a "metalogic language". It is still written in inner syntax but uses a small set of "metalogic operators".

Derivation rules consist of assumptions and a conclusion. They can be written using the metalogic operator $\Longrightarrow$ in the form

$$A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow C$$

where the A_1 ... A_n are the assumptions and C is the conclusion. The conclusion must be a formula. The assumptions may be arbitrary propositions. If an assumption contains metalogic operators parentheses can be used to delimit them from the rest of the derivation rule.

A derivation rule states that if the assumptions are valid, the conclusion can be derived as also being valid. So it can be viewed as a "meta implication" with a similar meaning as a boolean implication, but with a different use.

An example for a rule with a single assumption is

$$(x::nat) < c \Longrightarrow n*x \leq n*c$$

Note that type `nat` is explicitly specified for variable `x`. This is necessary, because the constants `<`, `*`, and $\leq$ are overloaded and can be applied to other types than only natural numbers. Therefore the type of `x` cannot be derived automatically. However, when the type of `x` is known, the types of `c` and `n` can be derived to also be `nat`.

An example for a rule with two assumptions is

$$(x::nat) < c \Longrightarrow n > 0 \Longrightarrow n*x < n*c$$

In most cases the assumptions are also formulae, as in the example. However, they may also be again derivation rules. Then the rule is a "meta rule" which derives a proposition from other rules. This introductory manual usually does not take such meta rules into account.

Binding Free Variables

A proposition may contain universally bound variables, using the metalogic quantifier $\bigwedge$ in the form

$\bigwedge x_1 \ldots x_n.\ P$

where the $x_1 \ldots x_n$ may occur free in the proposition P. As usual, types may be specified for (some of) the variables in the form $\bigwedge (x_1::t_1) \ldots (x_n::t_n).\ P$. An example for a valid derivation rule with bound variables is

$\bigwedge (x::nat)\ c\ n\ .\ x < c \Longrightarrow n*x \leq n*c$

If a standalone proposition contains free variables they are implicitly universally bound. Thus the example derivation rule above is equivalent to the single-assumption example rule in the previous section. Explicit binding of variables is only required to avoid name clashes with constants of the same name. In the proposition

$\bigwedge (True::nat).\ True < c \Longrightarrow n*True \leq n*c$

the name `True` is used locally as a variable of type `nat` instead of the predefined constant of type `bool`. Of course, using well known constant names as variables is confusing and should be avoided.

Alternative Rule Syntax

An alternative, Isabelle specific syntax for derivation rules is

$\bigwedge x_1 \ldots x_m.\ [\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow C$

which is often considered as more readable, because it better separates the assumptions from the conclusion. In the interactive editor it may be necessary to switch to this form by setting `Print Mode` to `brackets` in `Plugin Options` for `Isabelle General`. The fat brackets are available for input in the editor's Symbols panel in tab "Punctuation".

Using this syntax the two-assumption example rule from the previous section is denoted by

$\bigwedge (x::nat)\ c\ n.\ [\![x < c;\ n > 0]\!] \Longrightarrow n*x < n*c$

or equivalently without quantifier by

$[\![(x::nat) < c;\ n > 0]\!] \Longrightarrow n*x < n*c$

Note that in the literature a derivation rule $[\![P;\ Q]\!] \Longrightarrow P \wedge Q$ is often denoted in the form

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

Another alternative, Isabelle specific syntax for a derivation rule $\bigwedge x_1 \ldots x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow C$ is the "structured" proposition

"C" **if** *"A₁"* ... *"Aₙ"* **for** $x_1 \ldots x_m$

The assumptions and the variables may be grouped or separated for better readability by the keyword **and**. For every group of variables a type may be specified in the usual form, it applies to all variables in the group. Note that the keywords **if**, **and**, **for** belong to the outer syntax. Thus, the original rule must be quoted as a whole, whereas in the structured proposition only the sub-propositions *C*, A_1, ..., A_n must be individually quoted. The x_1, ..., x_m need not be quoted, but if a type is specified for a variable the type must be quoted, if it is not a single type name.

If written in this form, the two-assumption example rule from the previous subsections becomes

*"n*x < n*c"* **if** *"x < c"* **and** *"n > 0"* **for** *x::nat* **and** *n c*

2.1.7 Theorems

A theorem specifies a proposition together with a proof, that the proposition is valid. Thus it adds a fact to the enclosing theory. A simple form of a theorem is

theorem *"prop"* ⟨*proof*⟩

where *prop* is a proposition in inner syntax and ⟨*proof*⟩ is a proof as described in Section 2.2. The keyword **theorem** can be replaced by one of the keywords **lemma**, **corollary**, **proposition** to give a hint about the use of the statement to the reader.

Unknowns

Whenever a theorem turns a proposition to a fact, the free (or universally bound) variables are replaced by "unknowns". For a variable *name* the corresponding unknown is *?name*. This is only a technical difference, it signals to Isabelle that the unknowns can be consistently substituted by arbitrary terms, as long as the types are preserved.

When turned to a fact, the example rule from the previous sections becomes

?x < ?c $\Longrightarrow$ *?n*?x* $\leq$ *?n*?c*

with type *nat* associated to all unknowns.

The result of such a substitution is always a special case of the fact and therefore also a fact. In this way a fact with unknowns gives rise to a (usually infinite) number of facts which are constructed by substituting unknowns by terms.

Isabelle can be configured to suppress the question mark when displaying unknowns, then this technical difference becomes invisible.

Named Facts

Facts are often used in proofs of other facts. For this purpose they can be named so that they can be referenced by name. A named fact is specified by a theorem of the form

theorem *name: "prop"* ⟨*proof*⟩

The example rule from the previous sections can be turned into a fact named *example1* by

theorem *example1: "(x::nat) < c ⟹ n*x ≤ n*c"* ⟨*proof*⟩

It is also possible to introduce named collections of facts. A simple way to introduce such a named collection is

lemmas *name = name*$_1$ ... *name*$_n$

where *name*$_1$ ... *name*$_n$ are names of existing facts or fact collections.

If there is a second rule stated as a named fact by

theorem *example2: "(x::nat) ≤ c ⟹ x + m ≤ c + m"* ⟨*proof*⟩

a named collection can be introduced by

lemmas *examples = example1 example2*

Alternatively a "dynamic fact set" can be declared by

named_theorems *name*

It can be used as a "bucket" where facts can be added afterwards by specifying the bucket name in the theorem:

theorem *[name]: "prop"* ⟨*proof*⟩

or together with specifying a fixed fact name *name*$_f$ by

theorem *name*$_f$ *[name]: "prop"* ⟨*proof*⟩

A named fact or fact set (but not a dynamic fact set) can be displayed using the command

thm *name*

which may be entered outside of theorems and at most positions in a proof. The facts are displayed in the Output panel (see Section 1.2.3).

Alternative Theorem Syntax

There is an alternative syntax for theorems which have a derivation rule as their proposition. A theorem **theorem** "$\bigwedge x_1 \ldots x_m. [\![A_1; \ldots; A_n]\!] \Longrightarrow C$" ⟨*proof*⟩ can also be specified in the form

theorem
  fixes x_1 ... x_m
  assumes "A_1" ... "A_n"
  shows "C"
  ⟨*proof*⟩

Similar to the structured proposition form, the variables and assumptions may be grouped by **and**, the keywords belong to the outer syntax and the C, A_1, ..., A_n must be individually quoted.
Using this syntax the two-assumption example rule from the previous sections can be written as

theorem
  fixes *x::nat* **and** *c n*
  assumes "*x* < *c*" **and** "*n* > *0*"
  shows "*n**x* < *n**c*"
  ⟨*proof*⟩

In contrast to the general theorem syntax this alternative syntax allows to specify names for some or all of the assumption groups as

assumes $name_1$: "A_{11}" ... "A_{1m1}" **and** ... **and** $name_n$: "A_{n1}" ... "A_{nmn}"

These names can (only) be used in the proof of the theorem. More consequences this syntax has for the proof are described in Section 2.2.
Note that a name specified for the conclusion as

shows *name*: "*C*"

becomes the name for the whole fact introduced by the theorem, not only for the conclusion. It is not available in the proof of the theorem. Alternatively the name for the fact can be specified after the **theorem** keyword:

```
theorem name:
  fixes x::nat and c and n
  assumes "x < c" and "n > 0"
  shows "n*x < n*c"
  ⟨proof⟩
```

Definitions as Facts

The definitions described in Section 2.1.4 also introduce facts in the enclosing theory. Every definition introduces a new constant and specifies a defining equation of the form *name* ≡ *term* for it. This equation is a proposition using the "meta equality" ≡ which is another metalogic operator. It is the initial information given for the new constant, thus it is valid "by definition" and is a fact in the theory.

These facts are automatically named. If *name* is the name of the defined constant, the defining equation is named *name_def*. Alternatively an explicit name can be specified in the form

```
definition name :: type
where fact_name: "name ≡ term"
```

Although the auxiliary constants used in an **overloading** specification (see Section 2.1.5) are not accessible outside the specification, their definitions are. So they can be referred by their names and used as information about the overloaded constant.

2.1.8 Locales

There are cases where theory content such as definitions and theorems occur which has similar structure but differs in some types or terms. Then it is useful to define a "template" and instantiate it several times. This can be done in Isabelle using a "locale".

Simple Locales

A locale can be seen as a parameterized theory fragment, where the parameters are terms. A locale with *n* parameters is defined by

```
locale name =
  fixes x1 ... xn
begin
  ...
end
```

where the variables x_1, ..., x_n are the parameters. Like the bound variables in a theorem they can be grouped by **and** and types can be specified for some or all groups. The content between **begin** and **end** may consist of definitions and theorems which may use the parameter names like constant names. Content may also be added to an existing locale in the form

context *name*
begin
  ...
end

Therefore the **begin** ... **end** block can also be omitted in the locale definition and the locale can be filled later.

An instance of the parameterized theory fragment is created by "interpreting" the locale in the form

interpretation *name* $term_1$ ... $term_n$.

where $term_1$ ... $term_n$ are the terms to be substituted for the locale parameters, their types must match the parameter types, i.e., must be instances of them. The final dot in the interpretation is a rudimentary proof. An actual proof is needed, if the locale definition specifies additional assumptions for the parameters.

Locales with Assumptions

Additional assumptions for locale parameters can be specified as propositions in the form

locale *name* =
  fixes x_1 ... x_n
  assumes A_1 ... A_m
begin
  ...
end

where the A_1, ..., A_m are propositions. Like in a theorem, they can be grouped by **and** and named. The names can be used to reference the assumptions as facts in proofs in the locale content. When the locale is interpreted, all the assumptions must be proved with the actual terms substituted for the parameters. Therefore the more general form of an interpretation is

interpretation *name* $term_1$ ... $term_n$ ⟨*proof*⟩

Extending Locales

A locale may extend one or more other locales using the form

locale *name* = $name_1$ + ⋯ + $name_n$ +
  fixes ...
  assumes ...
begin
  ...
end

where $name_1$ ... $name_n$ are the names of the extended locales. Their parameters become parameters of the defined locale, inserted before the parameters declared by the **fixes** ... clause.

2.2 Isabelle Proofs

Every proposition stated as a fact in an Isabelle theory must be proven immediately by specifying a proof for it. A proof may have a complex structure of several steps and nested sub-proofs, its structure is part of the outer syntax.

2.2.1 Proof Context

Every proof is performed in a temporary environment which collects facts and other proof elements. This environment is called the "proof context". At the end of the proof the proof context is disposed with all its content, only the proven fact remains in the enclosing entity.

The proof context may contain

- Facts: as usual, facts are valid propositions. However, they need not be globally valid, they can be assumed to be only valid locally during the proof.
- Goals: a goal is a proposition which has not yet been proven. Typically it is the duty of a proof to prove one or more goals in its proof context.
- Fixed variables: fixed variables are used to denote the "arbitrary but fixed" objects often used in a proof. They can be used in all facts and goals in the same proof context.
- Term abbreviations: these are names introduced locally for terms. Using such names for terms occurring in propositions it is often possible to denote propositions in a more concise form.

The initial proof context in a theorem of the form **theorem** *"prop"* ⟨*proof*⟩ has the proposition *prop* as the only goal and is otherwise empty.

2.2.2 Proof Procedure

Assume you want to prove a derivation rule $A \Longrightarrow C$ with a single assumption A and the conclusion C. The basic procedure to build a proof for it is to construct a sequence of the form $F_1 \Longrightarrow F_2$, $F_2 \Longrightarrow F_3$, $F_3 \Longrightarrow \cdots \Longrightarrow F_{n-1}$, $F_{n-1} \Longrightarrow F_n$ from rules $RA_i \Longrightarrow RC_i$ for $i=1 \ldots n-1$ which are already known to be valid (i.e., facts) where F_1 matches with A and RA_1, F_n matches with C and RC_{n-1}, and every other F_i matches with RA_i and RC_{i-1}.

The sequence can be constructed from left to right (called "forward reasoning") or from right to left (called "backward reasoning") or by a combination of both.

Consider the rule $(x::nat) < 5 \Longrightarrow 2*x+3 \leq 2*5+3$. A proof can be constructed from the two example rules *example1* and *example2* from the previous sections as the sequence $(x::nat) < 5 \Longrightarrow 2*x \leq 2*5$, $2*x \leq 2*5 \Longrightarrow 2*x+3 \leq 2*5+3$ consisting of three facts.

Forward reasoning starts by assuming A to be a local fact and incrementally constructs the sequence from it. An intermediate result is a part $F_1 \Longrightarrow \cdots \Longrightarrow F_i$ of the sequence, here F_i is the "current fact". A forward reasoning step consists of stating a proposition F_{i+1} and proving it to be a new local fact from the current fact F_i using a valid rule $RA_i \Longrightarrow RC_i$. The step results in the extended sequence $F_1 \Longrightarrow \cdots \Longrightarrow F_i$, $F_i \Longrightarrow F_{i+1}$ and the new current fact F_{i+1}. When a step successfully proves a current fact F_n which matches the conclusion C the proof is complete.

Backward reasoning starts at the conclusion C of the single goal $A \Longrightarrow C$ and incrementally constructs the sequence from it backwards. An intermediate result is a part $F_i \Longrightarrow \cdots \Longrightarrow F_n$ of the sequence where F_n matches C, here the remaining part of the sequence $F_1 \Longrightarrow F_i$ is the "current goal". A backward reasoning step consists of applying a proof method to F_i which constructs a new current goal $F_1 \Longrightarrow F_{i-1}$ and the extended sequence $F_{i-1} \Longrightarrow F_i$, $F_i \Longrightarrow \cdots \Longrightarrow F_n$. When a step produces the new current goal $F_1 \Longrightarrow F_1$, which is trivially valid, the proof is complete.

Note the slight difference in how the steps are specified: A forward step specifies the new current fact F_{i+1} and then proves it. A backward step specifies the proof method, the new current goal $F_1 \Longrightarrow F_{i-1}$ is constructed by the method and is not an explicit part of the proof text. For that reason a proof constructed by forward reasoning is usually easier to read and write than a proof constructed by backward reasoning, since in the former case the sequence of the facts F_i is explicitly specified in the proof text, whereas in the latter case the sequence of the facts F_i is implicitly constructed and the proof text specifies only the methods.

However, since every forward reasoning step again requires a proof as its part (a "subproof"), no proof can be written using only forward reasoning steps. The main idea of writing "good" proofs is to use nested forward rea-

soning until every subproof is simple enough to be done in a single backward reasoning step, i.e., the proof method directly goes from the conclusion to the assumption.

Unification

The matching at the beginning and end of the sequence and when joining the used rules is done by "unification". Two propositions `P` and `Q` are unified by substituting terms for unknowns in `P` and `Q` so that the results become syntactically equal.

Since only the $RA_i \Longrightarrow RC_i$ are facts containing unknowns, only they are modified by the unification, `A` and `C` remain unchanged.

Note that when an unknown is substituted by a term in RA_i, the same unknown must be substituted by the same term in RC_i and vice versa, to preserve the validness of the rule $RA_i \Longrightarrow RC_i$. In other words, the sequence is usually constructed from specializations of the facts $RA_i \Longrightarrow RC_i$ where every conclusion is syntactically equal to the assumption of the next rule.

In the example the assumption `?x < ?c` of rule `example1` is unified with `(x::nat) < 5` by substituting the term `5` for the unknown `?c`, and the variable `x` for the unknown `?x` resulting in the specialized rule `(x::nat) < 5` $\Longrightarrow$ `n*x` $\leq$ `n*5`. The conclusion `?x + ?m` $\leq$ `?c + ?m` of rule `example2` is unified with `2*x+3` $\leq$ `2*5+3` by substituting the term `2*x` for the unknown `?x`, the term `2*5` for the unknown `?c`, and the term `3` for the unknown `?m` resulting in the specialized rule `2*x` $\leq$ `2*5` $\Longrightarrow$ `2*x+3` $\leq$ `2*5+3`. Now the two specialized rules can be joined by substituting the term `2` for the unknown `?n` in the first, resulting in the sequence which constitutes the proof.

Multiple Assumptions

If the rule to be proven has more than one assumption `A` the sequence to be constructed becomes a tree where the branches start at (copies of) the assumptions $A_1, \ldots, A_n$ and merge to finally lead to the conclusion `C`. Two branches which end in facts F_{1n} and F_{2m} are joined by a step $[\![F_{1n}; F_{2m}]\!] \Longrightarrow F_1$ to a common branch which continues from fact F_1.

Now a forward reasoning step may use several current facts to prove a new current fact. Therefore all proven local facts are stored in the proof context for possible later use. Every forward reasoning step selects a subset of the stored local facts as the current facts and uses them to prove a new local fact from them.

A backward reasoning step may now produce several new current goals, which belong to different branches in the tree. A step always produces the goals for all branches, therefore the previous goal is never used again in a step and is removed from the proof context after the step. When a current

goal has the form `A` $\Longrightarrow$ `A` the proof method "`assumption`" removes it from the proof context without producing a new goal. Thus a proof ends when no goal remains in the proof context.

The set of current goals is called the "goal state" of the proof. Since it is not visible in the proof text, the interactive editor displays the current goal state in the separate State panel and optionally also in the Output panel (see Sections 1.2.3 and 1.2.4), according to the cursor position in the proof text.

Proving from External Facts

The branches in the fact tree need not always start at an assumption A_i, they may also start at an "external" fact which is not part of the local proof context. In such cases the used external facts are referenced by their names. In that way a proof can use facts from the enclosing theory and a subproof can use facts from the enclosing proof(s) and the enclosing toplevel theory.

In particular, if the proposition of a theorem has no assumptions, i.e., the proposition is a formula and consists only of the conclusion `C`, every proof must start at one or more external facts.

2.2.3 Basic Proof Structure

A proof is written in outer syntax and describes how the fact tree is constructed which leads from the assumptions or external facts to the conclusion.

Proof Modes

When writing a proof the "proof mode" determines the mode of operation: whether forward reasoning (mode: *proof(state)*) or backward reasoning (mode: *proof(prove)*) is used.

The mode names refer to what is done in the next steps: In mode *proof(state)* facts are "stated" which lead to the conclusion, whereas in mode *proof(prove)* the goals are "proven", leading to assumptions and external facts.

At the beginning of a proof the mode is always *proof(prove)*, i.e., backward reasoning. In the course of the proof it is possible to switch to forward reasoning mode *proof(state)*, but not back again. After switching to forward reasoning the proof must be completed in forward reasoning mode, only at the end a last backward reasoning step may be applied.

However, in forward reasoning for every stated fact a (sub-)proof must be specified, which again starts in backward reasoning mode. This way it is possible to freely switch between both modes in the course of a proof with nested subproofs.

Proof Syntax

If BS_i denote backward reasoning steps and FS_i denote forward reasoning steps, the general form of a proof is

BS_1 ... BS_n
proof BS_{n+1}
  FS_1 ... FS_m
qed BS_{n+2}

The last step BS_{n+2} can be omitted if it is not needed.

The part **proof** BS_{n+1} switches from backward reasoning mode *proof(prove)* to forward reasoning mode *proof(state)*.

The part **proof** ... **qed** can be replaced by **done**, then the proof only consists of backward reasoning steps and has the form BS_1 ... BS_n **done**. Such proofs are called "proof scripts".

If the backward reasoning steps BS_1 ... BS_n are omitted the proof only consists of the forward reasoning part and has the form

proof BS_1
  FS_1 ... FS_m
qed BS_2

where BS_2 can also be omitted. Such proofs are called "structured proofs".

A structured proof can be so simple, that it has no forward reasoning steps. For this case the syntax

by BS_1 BS_2

abbreviates the form **proof** BS_1 **qed** BS_2. Again, BS_2 can be omitted which leads to the form

by BS_1

In this form the proof consists of a single backward reasoning step which directly leads from the conclusion `C` to the assumptions and used external facts.

Fake Proofs

A proof can also be specified as

sorry

This is a "fake proof" which turns the proposition to a fact without actually proving it.

A fake proof can be specified at any point in backward reasoning mode, so it can be used to abort a proof script in the form *BS*$_1$ ... *BS*$_n$ **sorry**.

A structured proof in forward reasoning mode cannot be aborted in this way, however, subproofs can be specified as fake proofs. This makes it possible to interactively develop a structured proof in a top-down way, by first stating all required facts with fake proofs and then replacing the fake proofs by actual proofs.

Nested Proof Contexts

The proof contexts in a structured proof can be nested. In a nested context the content of the enclosing contexts is available together with the local content. When a nested context is ended, it is removed together with all its local content.

A nested proof context is created syntactically by enclosing forward reasoning steps in braces:

FS$_1$ ... *FS*$_m$ **{** *FS*$_{m+1}$ ... *FS*$_n$ **}** *FS*$_{n+1}$ ...

Note that according to the description until now the nested context is useless, because the facts introduced by its forward reasoning steps are removed at its end and cannot contribute to the proof. How the content of a nested context can be "exported" and preserved for later use will be explained further below.

For names, nested contexts behave like a usual block structure: A name can be redefined in a nested context, then the named object in the outer context becomes inaccessible ("shadowed") in the inner context, but becomes accessible again when the inner context ends.

When two nested contexts follow each other immediately, this has the effect of "clearing" the content of the inner contexts: the content of the first context is removed and the second context starts being empty. This can be denoted by the keyword

next

which can be thought of being equivalent to a pair `} {` of adjacent braces.

Moreover the syntax **proof** *method*$_1$ *FS*$_1$ ... *FS*$_n$ **qed** *method*$_2$ automatically wraps the forward steps *FS*$_1$ ... *FS*$_n$ in a nested context. Therefore it is possible to denote a structured proof which only consists of a sequence of nested contexts without braces as

proof *method*$_1$
  FS$_{11}$ ... *FS*$_{1m1}$ **next** *FS*$_{21}$ ... *FS*$_{2m2}$ **next** ... **next** *FS*$_{n1}$ ... *FS*$_{nmn}$
qed *method*$_{n+2}$

where each occurrence of **next** clears the content of the context. The goal state of the proof is maintained in the enclosing outermost context, thus it is preserved over the whole sequence of nested contexts.

2.2.4 Backward Reasoning Steps

A backward reasoning step consist of applying a proof method.

Proof Methods

Proof methods are basically denoted by method names, such as *standard*, *simp*, or *rule*. A proof method name can also be a symbol, such as -.

A method may have arguments, then it usually must be delimited by parentheses such as in *(rule example1)* or *(simp add: example2)*, where *example1* and *example2* are fact names.

Methods can be applied to the goal state, they modify the goal state by removing and adding goals.

The effect of applying a method is determined by its implementation and must be known to the proof writer. Isabelle supports a large number of proof methods. Methods used for the proofs described in this manual are described in Section 2.3.

Method Application

A standalone method application step is denoted as

apply *method*

where *method* denotes the proof method to be applied.

The backward reasoning steps which follow **proof** and **qed** in a structured proof are simply denoted by the applied method. Hence the general form of a proof where all backward reasoning steps are method applications is

apply $method_1$ ... **apply** $method_n$
proof $method_{n+1}$
  FS_1 ... FS_m
qed $method_{n+2}$

where FS_1 ... FS_m are forward reasoning steps.

2.2.5 Forward Reasoning Steps

A forward reasoning step consist of stating and proving a fact.

Stating a Fact

A fact is stated in the form

have *"prop"* ⟨*proof*⟩

where *prop* is a proposition in inner syntax and ⟨*proof*⟩ is a (sub-) proof for it. This form is similar to the specification of a theorem in a theory and has a similar effect in the local proof context.

As for a theorem the fact can be named:

have *name: "prop"* ⟨*proof*⟩

Note that the alternative form of a theorem using **fixes**, **assumes**, and **shows** (see Section 2.1.7) is not available for stating facts in a proof.

The subproof ⟨*proof*⟩ uses a nested context, therefore all content of the enclosing proof context is available there and can be referenced by name, as long as the name is not shadowed by a redefinition in the subproof. Note that the name given to the fact to be proven cannot be used to access it in the subproof, because it is only assigned after the proof has been finished.

Proving a Goal

A forward reasoning proof ends, if the last stated fact F_n unifies with the conclusion C. Therefore a special form of stating a fact exists, which, after proving the fact, replaces free variables by unknowns (which is called "exporting the fact") and tries to unify it with the conclusion of a goal in the goal state. If successful, it removes the goal from the goal state:

show *"prop"* ⟨*proof*⟩

The syntax is the same as for **have**. If the unification with some goal conclusion is not successful the step is erroneous and the proof cannot be continued, in the interactive editor an error message is displayed.

The **show** step tries to match and remove a goal from the innermost enclosing proof context which maintains a goal state. This is one way how a fact proven in a nested context can affect an enclosing context and thus contribute to the proof there. Since the forward reasoning steps in a structured proof are wrapped in a nested context, while the goal state is maintained in the enclosing outer context, the **show** step affects the goal state of the enclosing proof.

The **have** and **show** steps are called "goal statements", because they state the proposition *prop* as a goal which is then proven by the ⟨*proof*⟩.

Note that the proposition *prop* in a **show** statement often is the same proposition which has been specified as conclusion C in the proposition $[\![A_1; \ldots; A_n]\!]$

$\Longrightarrow$ `C` which should be proven by the proof. To avoid repeating it, Isabelle automatically provides the abbreviation `?thesis` for it. So in the simplest case the last step of a forward proof can be written as

show `?thesis` ⟨*proof*⟩

The abbreviation `?thesis` is a single identifier, therefore it needs not be quoted.

If, however, the application of `method` in a structured proof **proof** `method` ... modifies the original goal, this modification is not reflected in `?thesis`. So a statement **show** `?thesis` ⟨*proof*⟩ will usually not work, because `?thesis` no more unifies with the conclusion of the modified goal. Instead, the proof writer must know the modified goal and specify its conclusion explicitly as proposition in the **show** statement. If the `method` splits the goal into several new goals, several **show** statements may be needed to remove them.

To test whether a proposition unifies with the conclusion of a goal in the goal state, a **show** statement can be specified with a fake proof:

show `"prop"` **sorry**

If that statement is accepted, the proposition unifies with the conclusion of a goal and removes it.

2.2.6 Facts as Proof Input

If a linear fact sequence $F_1 \Longrightarrow \cdots \Longrightarrow F_n$ is constructed in forward reasoning mode in the form

have `"`F_1`"` ⟨*proof*⟩$_1$
...
have `"`F_n`"` ⟨*proof*⟩$_n$

every fact F_i needs to refer to the previous fact F_{i-1} in its proof $proof_i$. This can be done by naming all facts

have $name_1$`:` `"`F_1`"` ⟨*proof*⟩$_1$
...
have $name_n$`:` `"`F_n`"` ⟨*proof*⟩$_n$

and refer to F_{i-1} in $proof_i$ by $name_{i-1}$.

Isabelle supports an alternative way by passing facts as input to a proof.

Using Input Facts in a Proof

The input facts are passed as input to the first method applied in the proof. In a proof script it is the method applied in the first **apply** step, in a structured proof **proof** `method` ... it is `method`.

Every proof method accepts a set of facts as input. Whether it processes them and how it uses them depends on the kind of method. Therefore input facts for a proof only work in the desired way, if a corresponding method is used at the beginning of the proof. See Section 2.3 for descriptions how methods process input facts.

Inputting Facts into a Proof

In backward reasoning mode *proof(prove)* facts can be input to the remaining proof ⟨*proof*⟩ by

using *name*$_1$ ... *name*$_n$ ⟨*proof*⟩

where the *name*$_i$ are names of facts or fact sets. The union of all referred facts is input to the proof following the **using** specification. In a proof script it is input to the next **apply** step. If a structured proof follows, it is input to its initial method. Since in backward reasoning mode no local facts are stated by previous steps, only external facts can be input this way.

In forward reasoning mode *proof(state)* fact input is supported with the help of the special fact set name *this*. The statement

then

inputs the facts named *this* to the proof of the following fact statement.

The statement **then** must be immediately followed by a goal statement (**have** or **show**). This is enforced by a special third proof mode *proof(chain)*. In it only a goal statement is allowed, **then** switches to this mode, the goal statement switches back to mode *proof(state)* after its proof.

Note that **then** is allowed in forward reasoning mode, although it does not state a fact. There are several other such auxiliary statements allowed in mode *proof(state)* in addition to the goal statements **have** and **show**.

The fact set *this* can be set by the statement

note *name*$_1$ ... *name*$_n$

Therefore the statement sequence

note *name*$_1$ ... *name*$_n$
then have *"prop"* ⟨*proof*⟩

inputs the union of all facts referred by *name*$_1$ ... *name*$_n$ to the ⟨*proof*⟩, in the same way as **using** inputs them to the remaining proof following it.

The statement sequence

note *name*$_1$ ... *name*$_n$ **then**

can be abbreviated by the statement

from *name*$_1$ ... *name*$_n$

Like **then** it switches to mode *proof(chain)* and it inputs the union of the facts referred by *name*$_1$ ... *name*$_n$ to the proof of the following goal statement.

2.2.7 Fact Chaining

In both cases described for fact input until now, the facts still have been referred by names. This can be avoided by automatically using a stated fact as input to the proof of the next stated fact. That is called "fact chaining".

Automatic Update of the Current Facts

Fact chaining is achieved, because Isabelle automatically updates the fact set *this*. Whenever a new fact is added to the proof context, the set *this* is redefined to contain (only) this fact. In particular, after every goal statement *this* names the new proven fact. Therefore the fact set *this* is also called the "current facts".

Thus a linear sequence of facts can be constructed by

have *"F*$_1$*"* ⟨*proof*⟩$_1$
then have *"F*$_2$*"* ⟨*proof*⟩$_2$
...
then have *"F*$_n$*"* ⟨*proof*⟩$_n$

Now in every *proof*$_i$ the fact *F*$_{i-1}$ is available as input and can be used to prove *F*$_i$.

Chaining can be combined with explicit fact referral by a statement of the form

note *this name*$_1$ ... *name*$_n$

It sets *this* to the union of *this* and the *name*$_1$ ... *name*$_n$, i.e., it adds the *name*$_1$ ... *name*$_n$ to *this*. In this way the current facts can be extended with other facts and then chained to the proof of the next stated fact.

The statement sequence

note *this name*$_1$ ... *name*$_n$ **then**

can be abbreviated by the statement

with *name*$_1$ ... *name*$_n$

Like **then** it switches to mode *proof(chain)* and it inputs the union of the facts referred by $name_1$ ... $name_n$ together with the current facts to the proof of the following goal statement.

If a proof consists of a fact tree with several branches, every branch can be constructed this way. Before switching to the next branch the last fact must be named, so that it can later be used to prove the fact where the branches join. A corresponding proof pattern for two branches which join at fact `F` is

have "F_{11}" ⟨*proof*⟩$_{11}$
then have "F_{12}" ⟨*proof*⟩$_{12}$
...
then have $name_1$**:** "F_{1m}" ⟨*proof*⟩$_{1m}$
have "F_{21}" ⟨*proof*⟩$_{21}$
then have "F_{22}" ⟨*proof*⟩$_{22}$
...
then have "F_{2n}" ⟨*proof*⟩$_{2n}$
with $name_1$ **have** "F" ⟨*proof*⟩

Naming and Grouping Current Facts

Since the fact set built by a **note** statement is overwritten by the next stated fact, it is possible to give it an explicit name in addition to the name *this* in the form

note *name* = $name_1$ ... $name_n$

The *name* can be used later to refer to the same fact set again, when *this* has already been updated. Defining such names is only possible in the **note** statement, not in the abbreviated forms **from** and **with**.

The facts specified in **note**, **from**, **with**, and **using** can be grouped by separating them by **and**. Thus it is even possible to write

from $name_1$ **and** ... **and** $name_n$ **have** "prop" ⟨*proof*⟩

In the case of a **note** statement every group can be given an additional explicit name as in

note $name_1$ = $name_{11}$ ... $name_{1m1}$ **and** ... **and** $name_n$ = $name_{n1}$ ... $name_{nmn}$

Accessing Input Facts in a Structured Proof

At the beginning of a structured proof the set name *this* is undefined, the name cannot be used to refer to the input facts (which are the current facts in the enclosing proof). To access the input facts they must be named before they are chained to the goal statement, then they can be referenced in the subproof by that name. For example in

note *input = this*
then have *"prop"* ⟨*proof*⟩

the input facts can be referenced by the name *input* in ⟨*proof*⟩.

Exporting the Current Facts of a Nested Context

At the end of a nested context (see Section 2.2.3) the current facts are automatically exported to the enclosing context, i.e. they become available there as the fact set named *this*, replacing the current facts before the nested context. This is another way how facts from a nested context can contribute to the overall proof.

Basically, only the last fact is current at the end of a context. Arbitrary facts can be exported from the nested context by explicitly making them current at its end, typically using a **note** statement:

... **{**
  have f_1*:* *"*$prop_1$*"* ⟨*proof*⟩$_1$
  ...
  have f_n*:* *"*$prop_n$*"* ⟨*proof*⟩$_n$
  note f_1 ... f_n
  } ...

Here all facts are named and the **note** statement makes them current by referring them by their names. Note, that the names are only valid in the nested context and cannot be used to refer to the exported facts in the outer context.

The exported facts can be used in the outer context like all other current facts by directly chaining them to the next stated fact:

... **{** ... **}** **then have** *"prop"* ⟨*proof*⟩ ...

or by naming them for later use, with the help of a **note** statement:

... **{** ... **}** **note** *name = this* ...

2.2.8 Assuming Facts

The Assumptions $A_1, \ldots, A_n$ of the proposition to be proven are needed as facts in the proof context to start the branches of the fact tree. However, they are not automatically inserted, that must be done explicitly.

Introducing Assumed Facts

An assumption is inserted in the proof context by a statement of the form

assume *"prop"*

Several assumptions can be inserted in a single **assume** statement of the form

assume *"prop$_1$" ... "prop$_n$"*

As usual, the assumptions can be grouped by **and** and the groups can be named.
Like goal statements an **assume** statement makes the assumed facts current, i.e. it updates the set *this* to contain the specified propositions as facts, so that they can be chained to a following goal statement. This way the fact sequence $A \Longrightarrow F_1$, $F_1 \Longrightarrow \cdots$ of a proof using fact chaining can be started:

assume *"A"*
then have *"F_1"*
...

Alternatively, the assumed facts can be named:

assume *name: "prop$_1$" ... "prop$_n$"*

so that they can be referred by name in the rest of the proof. Then the fact chain can be started in the form

assume *a: "A"*
from *a* **have** *"F_1"*
...

This can be useful if the proof has several branches which all start at the same assumption.

Admissible Assumed Facts

In an **assume** statement no proof is needed, since these propositions are only "assumed" to be valid. Therefore, only the propositions occurring as assumptions in the goal $[\![A_1;\ldots;A_n]\!] \Longrightarrow C$ to be proven are allowed here.
Actually, this condition is not checked for the **assume** statement, an arbitrary proposition can be specified by it. The condition becomes only relevant in subsequent **show** statements. When the fact proven by a show statement is tried to be unified with the conclusion of a goal, additionally each proposition stated in an **assume** statement is tested for unifying with an assumption

of the same goal. If that is not satisfied, the **show** statement fails. Therefore, if a proposition is used in an **assume** statement which does not unify with an assumption in a goal, the proof cannot be completed, because all **show** statements will fail.

Exporting Facts with Assumptions

More generally, whenever a local fact *F* is exported from a proof context, it is combined with all locally assumed facts $AF_1, \ldots, AF_n$ to the derivation rule $[\![AF_1; \ldots; AF_n]\!] \Longrightarrow F$. This reflects the intention of the local assumptions: They may have been used locally to prove *F* without knowing whether they are valid. So outside the local context *F* is only known to be valid if all the assumptions are valid.

If the fact *F* is itself a derivation rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow C$ then the locally assumed facts are added, resulting in the exported rule $[\![AF_1; \ldots; AF_n; A_1; \ldots; A_n]\!] \Longrightarrow C$.

If the fact *F* has been proven in a **show** statement it is also exported in this way, resulting in a derivation rule. It matches with a goal if the conclusion of the exported rule unifies with the goal conclusion and if every assumption of the exported rule unifies with an assumption in the goal. This way of matching a rule with a goal is called "refinement". So the condition for a successful **show** statement can be stated as "the exported fact must refine a goal".

Avoiding Repeated Propositions for Assumed Facts

To make **show** statements succeed, an **assume** statement will usually repeat one or more assumptions from the proposition to be proven. This is similar to a **show** statement, which usually repeats the conclusion of that proposition. However, unlike *?thesis* for the conclusion, there is no abbreviation provided for the assumptions.

To avoid repeating propositions in **assume** statements, the proposition to be proven can be specified in the form (see Section 2.1.7)

theorem
  assumes *"*A_1*"* **and** ... **and** *"*A_n*"*
  shows *"C"*
  ⟨*proof*⟩

This form automatically inserts the assumptions as assumed facts in the proof context. No **assume** statements are needed, the assumptions need not be repeated, and the assumed facts are always safe for **show** statements.

The assumptions can still be named and referred by name in the proof. A proof can be started at assumption A_1 in the form

theorem
  assumes *name*$_1$*: "A*$_1$*"* **and** ... **and** *name*$_n$*: "A*$_n$*"*
  shows *"C"*
  proof *method*
    from *name*$_1$ **have** *"F*$_1$*"* ⟨*proof*⟩
    ...

Additionally, the set of all assumptions specified in this form of a theorem is automatically named *assms*. Since unneeded assumptions usually do not harm in a proof, each proof branch can be started in the form

from *assms* **have** *"F*$_1$*"*
...

but it is usually clearer for the reader to specify only the relevant assumption(s) by explicit names.

In subproofs **assume** statements cannot be avoided in this way, because propositions in goal statements cannot be specified using **assumes** and **shows**. However, goal statements usually specify only a fact as proposition without assumptions. Instead of assumed facts the subproof can either use facts provided as input, or use external facts from the enclosing proof context by referring to them by name.

Presuming Facts

It is also possible to use a proposition as assumed fact which does not unify with an assumption in a goal, but can be proven from them. In other words, the proof is started somewhere in the middle of the fact tree, works in forward reasoning mode, and when it reaches the conclusion the assumed fact remains to be proven. The statement

presume *"prop"*

inserts such a presumed fact into the proof context.

When a fact is exported from a context with presumed facts, they do not become a part of the exported rule. Instead, at the end of the context for each presumed fact F_p a new goal $[\![A_1; \ldots; A_n]\!] \Longrightarrow F_p$ is added to the enclosing goal state. So the proof has to continue after proving all original goals and is only finished when all such goals for presumed facts have been proven as well.

2.2.9 Fixing Variables

Variables occurring in the proposition of a theorem can be used in the proof as well, they are universally bound in the whole proof context, if they are

not explicitly bound in the proposition, which restricts their use to the proposition itself. Thus in

theorem `"⋀x::nat. x < 3 ⟹ x < 5"` ⟨*proof*⟩

the variable `x` is restricted to the proposition and is not accessible in ⟨*proof*⟩, whereas in

theorem `"(x::nat) < 3 ⟹ x < 5"` ⟨*proof*⟩

and

theorem `"x < 3 ⟹ x < 5"` **for** `x::nat` ⟨*proof*⟩

the variable `x` is accessible in ⟨*proof*⟩.

Local Variables

Additional local variables can be introduced ("fixed") in a proof context in mode `proof(state)` by the statement

fix x_1 ... x_n

As usual the variables can be grouped by **and** and types can be specified for (some of) the groups.

If a variable name is used in a proof context without explicitly fixing it, it either refers to a variable in an enclosing context or in the proposition to be proven, or it is free. If it is explicitly fixed it names a variable which is different from all variables with the same name in enclosing contexts and the proposition to be proven.

A fixed local variable is common to the whole local context. If it occurs in several local facts it always is the same variable, it is not automatically restricted to the fact, as for toplevel theorems. Hence in

fix `x::nat`
assume `a: "x < 3"`
have `"x < 5"` ⟨*proof*⟩

the ⟨*proof*⟩ may refer to fact `a` because the `x` is the same variable in both facts.

By convention variable names are often short consisting of one or two letters, whereas constants defined on toplevel in a theory have longer and more descriptive names. Therefore it is usually not necessary to explicitly fix the variables in the proposition of a theorem to prevent name clashes with constants. By contrast, in a nested proof context there may be other variables with the same name in enclosing contexts, therefore it is recommended to explicitly fix all local variables.

Exporting Facts with Local Variables

Explicitly fixing variables in a proof context is not only important for avoiding name clashes. If a fact is exported from a proof context, all fixed local variables are replaced by unknowns, other variables remain unchanged. Since unification only works for unknowns, it makes a difference whether a fact uses a local variable or a variable which origins from an enclosing context or is free.

The proposition `x < 3 ⟹ x < 5` can be proven by the statements

fix *y::nat*
assume *"y < 3"*
then show *"y < 5"* ⟨*proof*⟩

because when the fact `y < 5` is exported, the assumption is added (as described in Section 2.2.8) and then variable `y` is replaced by the unknown `?y` because `y` has been locally fixed. The result is the rule `?y<3 ⟹ ?y<5` which unifies with the proposition.

If, instead, `y` is not fixed, the sequence

assume *"(y::nat) < 3"*
then have *"y < 5"* ⟨*proof*⟩

still works and the local fact `y < 5` can be proven, but it cannot be used with the **show** statement to prove the proposition `x < 3 ⟹ x < 5`, because the exported rule is now `y<3 ⟹ y<5` which does not unify with the proposition, it contains a different variable instead of an unknown.

2.2.10 Obtaining Variables

Local variables may also be introduced together with a fact which allows to determine their values. This is done using a statement of the form

obtain x_1 ... x_m **where** *"prop"* ⟨*proof*⟩

where *prop* is a proposition in inner syntax which contains the variables x_1 ... x_m. Like for variables introduced by **fix** the variables can be grouped by **and** and types can be specified for (some of) the groups.

The proposition usually relates the values of the new variables to values of existing variables (which may be local or come from the environment). In the simplest case the proposition directly specifies terms for the new variables, such as in

fix *x::nat*
obtain *y z* **where** *"y = x + 3 ∧ z = x + 5"* ⟨*proof*⟩

But it is also possible to specify the values indirectly:

fix *x::nat*
obtain *y z* **where** *"x = y - 3 ∧ y + z = 2*x +8"* ⟨*proof*⟩

Here the proposition may be considered to be an additional assumption which is added to the proof context.

Proving *obtain* Statements

Actually, several propositions may be specified in an **obtain** statement:

obtain x_1 ... x_m **where** *"prop*$_1$*"* ... *"prop*$_n$*"* ⟨*proof*⟩

The propositions may be grouped by **and** and the groups can be named as usual. This **obtain** statement has a similar meaning as the statements

fix x_1 ... x_m
assume *"prop*$_1$*"* ... *"prop*$_n$*"*

but there is one important difference: the propositions in an **obtain** statement must be redundant in the local proof context.

That is the reason why an **obtain** statement is a goal statement and includes a proof. The proof must prove the redundancy of the propositions, which may be stated in the following way: if any other proposition can be derived from them in the local proof context it must be possible to also derive it without the propositions. This can be stated formally as

$$(\bigwedge x_1 \ldots x_m.\ [\![prop_1; \ldots; prop_n]\!] \Longrightarrow P) \Longrightarrow P$$

which is exactly the goal to be proven for the **obtain** statement.

Consider the statements

fix *x::nat*
obtain *y* **where** *"x = 2*y"* ⟨*proof*⟩

This proposition is not redundant, because it implies that *x* must be even. Therefore no proof exists.

Note that after a successful proof of an **obtain** statement the current facts are the propositions specified in the statement, not the proven redundancy statement. Input facts may be passed to **obtain** statements. Like for the other goal statements, they are input to the ⟨*proof*⟩.

Exporting Facts after Obtaining Variables

Unlike facts assumed by an **assume** statement (see Section 2.2.8) the propositions in an **obtain** statement are *not* added as assumptions when a fact `F` is exported from the local context. This is correct, since they have been proven to be redundant, therefore they can be omitted.

However, that implies that an exported fact `F` may not refer to variables introduced by an **obtain** statement, because the information provided by the propositions about them gets lost during the export.

2.2.11 Term Abbreviations

A term abbreviation is a name for a proposition or a term in it.

Defining Term Abbreviations

A term abbreviation can be defined by a statement of the form

let *?name = "term"*

Afterwards the name is "bound" to the term and can be used in place of the term in propositions and other terms, as in:

let *?lhs = "2*x+3"*
let *?rhs = "2*5+3"*
assume *"x < 5"*
have *"?lhs* $\leq$ *?rhs"* ⟨*proof*⟩

The name `?thesis` (see Section 2.2.5) is a term abbreviation of this kind.

A **let** statement can define several term abbreviations in the form

let *?name*$_1$ *= "term*$_1$*"* **and** ... **and** *?name*$_n$ *= "term*$_n$*"*

A **let** statement can occur everywhere in mode `proof(state)`. However, it does not preserve the current facts, the fact set `this` becomes undefined by it.

Pattern Matching

Note that term abbreviations have the form of "unknowns" (see Section 2.1.7), although they are defined ("bound"). The reason is that they are actually defined by unification.

The more general form of a **let** statement is

let *"pattern" = "term"*

where *pattern* is a term which may contain unbound unknowns. As usual, if the pattern consists of a single unknown, the quotes may be omitted. The **let** statement unifies *pattern* and *term*, i.e., it determines terms to substitute for the unknowns, so that the pattern becomes syntactically equal to *term*. If that is not possible, an error is signaled, otherwise the unknowns are bound to the substituting terms. Note that the equals sign belongs to the outer syntax, therefore both the pattern and the term must be quoted separately.

The **let** statement

let *"?lhs ≤ ?rhs" = "2*x+3 ≤ 2*5+3"*

binds the unknowns to the same terms as the two **let** statements above.

The term may contain unknowns which are already bound. They are substituted by their bound terms before the pattern matching is performed. Thus the term can be constructed with the help of abbreviation which have been defined previously. A useful example is matching a pattern with *?thesis*:

theorem *"x < 5 ⟹ 2*x+3 ≤ 2*5+3"*
proof *method*
  let *"?lhs ≤ ?rhs" = ?thesis*
  ...

to reuse parts of the conclusion in the proof.

Note that the unknowns are only bound at the end of the whole **let** statement. In the form

let *"pattern$_1$" = "term$_1$"* **and** ... **and** *"pattern$_n$" = "term$_n$"*

the unknowns in *pattern$_i$* cannot be used to build *term$_{i+1}$* because they are not yet bound. In contrast, in the sequence of **let** statements

let *"pattern$_1$" = "term$_1$"*
 ...
let *"pattern$_n$" = "term$_n$"*

the unknowns in *pattern$_i$* can be used to build *term$_{i+1}$* because they are already bound.

If a bound unknown occurs in the pattern its bound term is ignored and the unknown is rebound according to the pattern matching. In particular, it does not imply that the old and new bound terms must be equal, they are completely independent.

If a part of the term is irrelevant and need not be bound the dummy unknown "_" (underscore) can be used to match it in the pattern without binding an unknown to it:

let *"_ ≤ ?rhs" = "2*x+3 ≤ 2*5+3"*

will only bind *?rhs* to the term on the righthand side.
If the term internally binds variables which are used in a subterm, the subterm cannot be matched separately by an unknown because then the variable bindings would be lost. Thus the statement

let *"λx. ?t" = "λx. x+1"*

will fail to bind *?t* to *x+1* whereas

let *"λx. x+?t" = "λx. x+1"*

will successfully bind *?t* to *1* since the bound variable *x* does not occur in it.

2.2.12 Accumulating Facts

Instead of giving individual names to facts in the proof context, facts can be collected in named fact sets. Isabelle supports the specific fact set named *calculation* and provides statements for managing it.
The fact set *calculation* is intended to accumulate current facts for later use. Therefore it is typically initialized by the statement

note *calculation = this*

and afterwards it is extended by several statements

note *calculation = calculation this*

After the last extension the facts in the set are chained to the next proof:

note *calculation = calculation this* **then have** ...

Support for Fact Accumulation

Isabelle supports this management of *calculation* with two statements. The statement

moreover

is equivalent to **note** *calculation = this* when it occurs the first time in a context, afterwards it behaves like **note** *calculation = calculation this* but without making *calculation* current, instead, it leaves the current fact(s) unchanged. The statement

ultimately

is equivalent to **note** *calculation* = *calculation this* **then**, i.e., it adds the current facts to the set, makes the set current, and chains it to the next goal statement.

Due to the block structure of nested proofs, the *calculation* set can be reused in nested contexts without affecting the set in the enclosing context. The first occurrence of **moreover** in the nested context initializes a fresh local *calculation* set. Therefore fact accumulation is always local to the current proof context.

Accumulating Facts in a Nested Context

Fact accumulation can be used for collecting the facts in a nested context for export (see Section 2.2.7) without using explicit names for them:

```
... {
  have "prop_1" ⟨proof⟩_1
  moreover have "prop_2" ⟨proof⟩_2
  ...
  moreover have "prop_n" ⟨proof⟩_n
  moreover note calculation
  } ...
```

Accumulating Facts for Joining Branches

Fact accumulation can also be used for collecting the facts at the end of joined fact branches in a proof and inputting them to the joining step. A corresponding proof pattern for two branches which join at fact *F* is

```
have "F_11" ⟨proof⟩_11
then have "F_12" ⟨proof⟩_12
...
then have "F_1m" ⟨proof⟩_1m
moreover have "F_21" ⟨proof⟩_21
then have "F_22" ⟨proof⟩_22
...
then have "F_2n" ⟨proof⟩_2n
ultimately have "F" ⟨proof⟩
```

The **moreover** statement starts the second branch and saves the fact F_{1m} to *calculation*. The **ultimately** statement saves the fact F_{2n} to *calculation* and then inputs the set to the proof of *F*.

Note that **moreover** does not chain the current facts to the following goal statement.

Using nested contexts sub-branches can be constructed and joined in the same way.

2.2.13 Equational Reasoning

Often informal proofs on paper are written in the style

```
2*(x+3) = 2*x+6 ≤ 3*x+6 < 3*x+9 = 3*(x+3)
```

to derive the fact `2*(x+3) < 3*(x+3)`. Note that the formula shown above is not a wellformed proposition because of several occurrences of the toplevel relation symbols `=`, ≤ and `<`. Instead, the formula is meant as an abbreviated notation of the fact sequence

```
2*(x+3) = 2*x+6, 2*x+6 ≤ 3*x+6, 3*x+6 < 3*x+9, 3*x+9 = 3*(x+3)
```

which sketches a proof for `2*(x+3) < 3*(x+3)`. This way of constructing a proof is called "equational reasoning" which is a specific form of forward reasoning.

Transitivity Rules

The full proof needs additional facts which must be inserted into the sequence. From the first two facts the fact `2*(x+3)` ≤ `3*x+6` is derived, then with the third fact the fact `2*(x+3) < 3*x+9` is derived, and finally with the fourth fact the conclusion `2*(x+3) < 3*(x+3)` is reached. The general pattern of these additional derivations can be stated as the derivation rules ⟦`a = b; b ≤ c`⟧ ⟹ `a ≤ c`, ⟦`a ≤ b; b < c`⟧ ⟹ `a < c`, and ⟦`a < b; b = c`⟧ ⟹ `a < c`.

Rules of this form are called "transitivity rules". They are valid for relations such as equality, equivalence, orderings, and combinations thereof.

This leads to the general form of a proof constructed by equational reasoning: every forward reasoning step starts at a fact F_i of the form `a r b` where `r` is a relation symbol. It proves an intermediate fact H_i of the form `b r c` where `r` is the same or another relation symbol and uses a transitivity rule to prove the fact F_{i+1} which is `a r c`. In this way it constructs a linear sequence of facts which leads to the conclusion.

The intermediate facts H_i are usually proven from assumtions or external facts, or they may have a more complex proof which forms an own fact branch which ends at H_i and is joined with the main branch at F_{i+1} with the help of the transitivity rule.

Support for Equational Reasoning

Isabelle supports equational reasoning in the following form. It provides the statement

also

which expects that the set *calculation* contains the fact F_i and the current fact *this* is the fact H_i. It automatically selects an adequate transitivity rule, uses it to derive the fact F_{i+1} and replaces F_i in *calculation* by it. Upon its first use in a proof context **also** simply stores the current fact *this* in *calculation*. The statement

finally

behaves in the same way but additionally makes the resulting fact F_{i+1} current, i.e., puts it into the set *this*, and chains it into the next goal statement. In other words, **finally** is equivalent to **also from** *calculation*.

Note that **also** behaves like **moreover** and **finally** behaves like **ultimately**, both with additional application of the transitivity rule.

Additionally, Isabelle automatically maintains the term abbreviation ... (which is the three-dot-symbol available for input in the Symbols panel (see Section 1.2.5) of the interactive editor in tab "Punctuation") for the term on the right hand side of the current fact. Together, the example equational reasoning proof from above can be written

have *"2*(x+3) = 2*x+6"* ⟨*proof*⟩
also have *"... ≤ 3*x+6"* ⟨*proof*⟩
also have *"... < 3*x+9"* ⟨*proof*⟩
also have *"... = 3*(x+3)"* ⟨*proof*⟩
finally show *?thesis* ⟨*proof*⟩

where *?thesis* abbreviates the conclusion *2*(x+3) < 3*(x+3)*. This form is quite close to the informal paper style of the proof.

Determining Transitivity Rules

To automatically determine the transitivity rule used by **also** or **finally**, Isabelle maintains the dynamic fact set (see Section 2.1.7) named *trans*. It selects a rule from that set according to the relation symbols used in the facts in *calculation* and *this*.

A transitivity rule which is not in *trans* can be explicitly specified by name in the form

also *(name)*
finally *(name)*

2.3 Proof Methods

The basic building blocks of Isabelle proofs are the proof methods which modify the goal state. If there are several goals in the goal state it depends

on the specific method which goals are affected by it. In most cases only the first goal is affected.

2.3.1 The empty Method

The empty method is denoted by a single minus sign

```
-
```

If no input facts are passed to it, it does nothing, it does not alter the goal state.

Otherwise it affects all goals by inserting the input facts as assumptions after the existing assumptions. If the input facts are $F_1, \ldots, F_m$ a goal of the form $[\![A_1; \ldots; A_n]\!] \Longrightarrow C$ is replaced by the goal $[\![A_1; \ldots; A_n; F_1, \ldots, F_m]\!] \Longrightarrow C$. The reason for this behavior will be explained below.

The empty method is useful at the beginning of a structured proof of the form

proof *method* $FS_1 \ \ldots \ FS_n$ **qed**

If the forward reasoning steps $FS_1 \ \ldots \ FS_n$ shall process the unmodified original goal the empty method must be specified for *method*, thus the structured proof becomes

proof - $FS_1 \ \ldots \ FS_n$ **qed**

Note that it is possible to omit the *method* completely, but then it defaults to the method named *standard* which alters the goal state (see below).

2.3.2 Rule Application

As described in Section 2.2.2 the basic step in the construction of a proof is to establish the connection between a fact F_i and a fact F_{i+1} in the fact sequence. Assume that there is already a valid derivation rule $RA_i \Longrightarrow RC_i$ named r_i where RA_i unifies with F_i and RC_i unifies with F_{i+1}. Then the connection can be established by applying that rule.

The *rule* Method

A rule is applied by the method

```
rule name
```

where *name* is the name of a valid rule. The method only affects the first goal. If that goal has the form $[\![A_1; \ldots; A_n]\!] \Longrightarrow C$ and the rule referred by

name has the form $[\![RA_1;\ldots;RA_m]\!] \Longrightarrow RC$ the method first unifies *RC* with the goal conclusion *C*. That yields the specialized rule $[\![RA_1';\ldots;RA_m']\!] \Longrightarrow RC'$ where *RC'* is syntactically equal to *C* and every RA_j' results from RA_j by substituting unknowns by the same terms as in *RC'*. Note that the goal normally does not contain unknowns, therefore *C* is not modified by the unification. If the unification fails the method cannot be executed on the goal state and an error is signaled. Otherwise the method replaces the goal by the *m* new goals $[\![A_1;\ldots;A_n]\!] \Longrightarrow RA_j'$.

If the rule has the form $RA \Longrightarrow RC$ with only one assumption the method replaces the goal by the single new goal $[\![A_1;\ldots;A_n]\!] \Longrightarrow RA'$. If the rule is a formula *RC* without any assumptions the method removes the goal without introducing a new goal.

Using the *rule* Method for Backward Reasoning Steps

Assume that during a proof as described in Section 2.2.2 the intermediate fact sequence $F_{i+1} \Longrightarrow \cdots \Longrightarrow F_n$ has already been constructed by backward reasoning and the current goal is $F_1 \Longrightarrow F_{i+1}$. The backward reasoning step

apply *(rule* r_i*)*

will replace that goal by $F_1 \Longrightarrow F_i$ and thus extend the fact sequence to $F_i \Longrightarrow \cdots \Longrightarrow F_n$. The fact F_i is the specialized assumption RA_i' constructed by the method from the assumption RA_i of rule r_i.

Therefore the fact sequence $F_1 \Longrightarrow \cdots \Longrightarrow F_n$ of the complete proof can be constructed by the proof script consisting of the backward reasoning steps

apply *(rule* r_{n-1}*)*
...
apply *(rule* r_1*)*

Note however, that this proof script does not complete the proof, since it results in the goal $F_1 \Longrightarrow F_1$. The proof method

assumption

must be used to process it. The method only affects the first goal. If that goal has the form $[\![A_1;\ldots;A_n]\!] \Longrightarrow C$ and one assumption A_i is syntactically equal to *C* the method removes the goal, otherwise an error is signaled.

Together, the full proof script has the form

apply *(rule* r_{n-1}*)*
...
apply *(rule* r_1*)*
apply *(assumption)*
done

If the example from Section 2.2.2 is proven this way the theorem is written together with its proof as

```
theorem "x < 5 ⟹ 2*x+3 ≤ 2*5 + 3" for x :: nat
  apply(rule example2)
  apply(rule example1)
  apply(assumption)
  done
```

Note that the assumption `A` of the initial goal must be reached exactly by the sequence of rule applications. If it is replaced in the example by the stronger assumption `x < 3` the rule applications will lead to the goal `x < 3` $\Longrightarrow$ `x < 5` which is trivial for the human reader but not applicable to the `assumption` method.

Using the `rule` Method for Forward Reasoning Steps

Assume that during a proof as described in Section 2.2.2 the intermediate fact sequence $F_1 \Longrightarrow \cdots \Longrightarrow F_i$ has already been constructed by forward reasoning, so that the next step is to state fact F_{i+1} and prove it. Using method `rule` the step can be started by

have "F_{i+1}" **proof** (rule r_i)

The goal of this subproof is simply F_{i+1}, so applying the `rule` method with r_i will result in the new goal RA_i' which is F_i, as above. The subproof is not finished, since its goal state is not empty. But the goal is an already known fact. The proof method

```
fact name
```

can be used to remove that goal. The method only affects the first goal. If the fact referred by `name` unifies with it, the goal is removed, otherwise an error is signaled.
Using this method the forward reasoning step can be completed as

have "F_{i+1}" **proof** (rule r_i) **qed** (fact f_i)

if F_i has been named f_i. This can be abbreviated (see Section 2.2.3) to

have "F_{i+1}" **by** (rule r_i) (fact f_i)

Therefore the fact sequence $F_1 \Longrightarrow \cdots \Longrightarrow F_n$ of the complete proof can be constructed by the structured proof of the form

```
proof -
assume f1: "F1"
have f2: "F2" by (rule r1) (fact f1)
...
have fn-1: "Fn-1" by (rule rn-2) (fact fn-2)
show "Fn" by (rule rn-1) (fact fn-1)
qed
```

where the last fact F_n is usually the conclusion C and can be specified as *?thesis*.

If the example from Section 2.2.2 is proven this way the theorem is written together with its proof as

```
theorem "x < 5 ⟹ 2*x+3 ≤ 2*5 + 3" for x :: nat
proof -
  assume f1: "x < 5"
  have f2: "2*x ≤ 2*5" by (rule example1) (fact f1)
  show ?thesis by (rule example2) (fact f2)
qed
```

The *fact* method can be specified in the form

```
fact
```

without naming the fact to be used. Then it selects a fact automatically. It uses the first fact from the proof context which unifies with the goal. If there is no such fact in the proof context an error is signaled.

Thus the example can be written without naming the facts as

```
theorem "x < 5 ⟹ 2*x+3 ≤ 2*5 + 3" for x :: nat
proof -
  assume "x < 5"
  have "2*x ≤ 2*5" by (rule example1) fact
  show ?thesis by (rule example2) fact
qed
```

Input Facts for the *rule* Method

If input facts F_1, ..., F_n are passed to the *rule* method, they are used to remove assumptions from the rule applied by the method. If the rule has the form $[\![RA_1;\ldots;RA_{n+m}]\!] \Longrightarrow RC$ and every fact F_i unifies with assumption RA_i the first n assumptions are removed and the rule becomes $[\![RA_{n+1};\ldots;RA_{n+m}]\!] \Longrightarrow RC$. Then it is applied to the first goal in the usual way. If there are more facts than assumptions or if a fact does not unify, an error is signaled.

This allows to establish the connection from a fact F_i to F_{i+1} in a fact chain by a forward reasoning step of the form

from f_i **have** "F_{i+1}" **by** (*rule* r_i)

where f_i names the fact F_i. When it is input to the goal statement it is passed to the *rule* method and removes the assumption from the applied rule $RA_i \Longrightarrow RC_i$, resulting in the assumption-less "rule" RC_i. When it is applied to the goal F_{i+1} it unifies and removes the goal, thus the subproof is complete.

For the fact sequence chaining can be used to write a structured proof without naming the facts:

proof -
assume "F_1"
then have "F_2" **by** (*rule* r_1)
...
then have "F_{n-1}" **by** (*rule* r_{n-2})
then show "F_n" **by** (*rule* r_{n-1})
qed

If the example from Section 2.2.2 is proven this way the theorem is written together with its proof as

theorem "x < 5 $\Longrightarrow$ 2*x+3 $\leq$ 2*5 + 3" **for** x :: nat
proof -
  assume "x < 5"
  then have "2*x $\leq$ 2*5" **by** (*rule example1*)
  then show ?thesis **by** (*rule example2*)
qed

The Method *this*

Rule application can also be done by the method

this

Instead of applying a named method, it applies the input fact as rule to the first goal.

If several input facts are given, the method applies them exactly in the given order. Therefore the fact sequence can also be constructed by a structured proof of the form:

proof -
assume "F_1"
with r_1 **have** "F_2" **by** *this*
...
with r_{n-2} **have** "F_{n-1}" **by** *this*
with r_{n-1} **show** "F_n" **by** *this*
qed

The **with** statement inserts the explicitly named facts *before* the current facts. Therefore every goal statement for F_i gets as input the rule r_{i-1} followed by the chained fact F_{i-1}. The method *this* first applies the rule which replaces the goal by F_{i-1}. Then it applies the fact F_{i-1} as rule to this goal which removes it and finishes the subproof.

The proof

```
by this
```

can be abbreviated by . (a single dot).

Therefore the example from Section 2.2.2 can also be proven in the form

```
theorem "x < 5 ⟹ 2*x+3 ≤ 2*5 + 3" for x :: nat
proof -
  assume "x < 5"
  with example1 have "2*x ≤ 2*5" .
  with example2 show ?thesis .
qed
```

Adding Input Facts to the Goal

Another way of using the previous fact F_i in the proof of F_{i+1} is to add it as assumption to the goal. That is what the empty method does (see Section 2.3.1) if F_i is passed as input.

Thus the forward reasoning step can be written in the form

```
from f_i have "F_{i+1}"
proof -
  assume "F_i"
  then show ?thesis by (rule r_i)
qed
```

The empty method - replaces the goal F_{i+1} by the goal $F_i \Longrightarrow F_{i+1}$ which is then proven by a forward reasoning step.

Of course this is much longer than the proof using the *rule* method immediately. But it shows generally how the step from F_i to F_{i+1} can be established using an arbitrary structured proof. That can be useful if no direct derivation rule r_i is available.

Automatic Rule Selection

The *rule* method can be specified in the form

```
rule
```

without naming the rule to be applied. Then it selects a rule automatically. It uses the first rule from the dynamic fact set *intro* for which the conclusion unifies with the goal conclusion. If there is no such rule in the set an error is signaled.

If the rules *example1* and *example2* would be in the *intro* set, the example proof could be written as

```
theorem "x < 5 ⟹ 2*x+3 ≤ 2*5 + 3" for x :: nat
proof -
  assume "x < 5"
  then have "2*x ≤ 2*5" by rule
  then show ?thesis by rule
qed
```

However, the set *intro* is intended by Isabelle for a specific kind of rules called "introduction rules". In such a rule the toplevel operator of the conclusion does not occur in any assumption, hence it is "introduced" by the rule. When an introduction rule is applied backwards, the operator is removed from the goal. This can be iterated to "deconstruct" the goal, some proofs can be written using this technique, however, the content of the set must be designed very carefully to not run into cycles.

Since in the rule *example2* the toplevel operator $\leq$ occurs in the assumption it is not an introduction rule and should not be added to *intro*. Rule *example1* is an introduction rule but would interfere with predefined rules in the set.

The *standard* Method

The method

```
standard
```

is a method alias which can be varied for different Isabelle applications. Usually it is an alias for the *rule* method.

The *standard* method is the default, if no method is specified as the initial step in a structured proof. Thus

proof FS_1 ... FS_n **qed**

is an abbreviation for

proof *standard* FS_1 ... FS_n **qed**

Note that the *standard* method will usually affect the goal by applying an introduction rule to it. That may be useful in some cases, but it has to be taken into account when writing the forward reasoning steps of the proof.

For example, in Isabelle HOL there is an introduction rule in *intro* which splits a logical conjunction into two subgoals (see Section **??**). Therefore the proof of a conjunction `P ∧ Q` can be written

proof
show `P` ⟨*proof*⟩
show `Q` ⟨*proof*⟩
qed

because the goal is split by the *standard* method. If the empty method is used instead, the proof has to be of the form

proof -
show `"P ∧ Q"` ⟨*proof*⟩
qed

because the goal is not modified.

In the abbreviated form **by** *method* of a structured proof the method cannot be omitted, but the proof **by** *standard* can be abbreviated to

..

(two dots). It can be used as complete proof for a proposition which can be proven by a single automatic rule application. Hence, if the rules *example1* and *example2* would be in the *intro* set, the example proof could be further abbreviated as

```
theorem "x < 5 ⟹ 2*x+3 ≤ 2*5 + 3" for x :: nat
proof -
  assume "x < 5"
  then have "2*x ≤ 2*5" ..
  then show ?thesis ..
qed
```

2.3.3 Composed Proof Methods

Proof methods can be composed from simpler methods with the help of "method expressions". A method expression has one of the following forms:

- m_1`, ...,` m_n : a sequence of methods which are applied in their order,
- m_1`; ...;` m_n : a sequence of methods where each is applied to the goals created by the previous method,
- m_1`| ...|` m_n : a sequence of methods where only the first applicable method is applied,

- *m[n]* : the method *m* is applied to the first *n* goals,
- *m?* : the method *m* is applied if it is applicable,
- *m+* : the method *m* is applied once and then repeated as long as it is applicable.

Parentheses are used to structure and nest composed methods.

Composed methods can be used to combine backward reasoning steps to a single step. Using composed methods the example backward reasoning proof from Section 2.3.2 can be written as

```
theorem "x < 5 ⟹ 2*x+3 ≤ 2*5 + 3" for x :: nat
  apply(rule example2,rule example1,assumption)
  done
```

In particular, it is possible to apply an arbitrarily complex backward reasoning step as the first method in a structured proof. Using composed methods the first example forward reasoning proof can be written

```
theorem "x < 5 ⟹ 2*x+3 ≤ 2*5 + 3" for x :: nat
proof -
  assume "x < 5"
  have "2*x ≤ 2*5" by (rule example1,fact)
  show ?thesis by (rule example2,fact)
qed
```

2.3.4 The Simplifier

A common proof technique is "rewriting". If it is known that a term `a` is equal to a term `b`, some occurrences of `a` in a proposition can be replaced by `b` without changing the validity of the proposition.

Equality of two terms `a` and `b` can be expressed by the proposition `a = b`. If that proposition has been proven to be valid, i.e., is a fact, `a` can be substituted by `b` and vice versa in goals during a proof.

The *subst* Method

Rewriting is performed by the method

```
subst name
```

where *name* references an equality fact. The method only affects the first goal. If the referenced fact has the form `a = b` the method replaces the first occurrence of `a` in the goal conclusion by `b`. The order of the terms in the equality fact matters, the method always substitutes the term on the left by that on the right.

If the equality contains unknowns unification is used: `a` is unified with every sub-term of the goal conclusion, the first match is replaced by `b'` which is `b` after substituting unknowns in the same way as in `a`. If there is no match of `a` in the goal conclusion an error is signaled.

For a goal $[\![A_1; \ldots; A_n]\!] \Longrightarrow C$ the method only rewrites in the conclusion `C`. The first match in the assumptions $A_1 \ldots A_n$ can be substituted by the form

```
subst (asm) name
```

If not only the first match shall be substituted, a number of the match or a range of numbers may be specified in both forms as in

```
subst (asm) (i..j) name
```

The equality fact can also be a meta equality of the form `a` $\equiv$ `b`. Therefore the method can be used to expand constant definitions. After the definition

definition `"inc x ≡ x + 1"`

the method *subst inc_def* will rewrite the first occurrence of a function application *(inc t)* in the goal conclusion to *(t + 1)*. Remember from Section 2.1.4 that the defining equation is automatically named *inc_def*. Note the use of unification to handle the actual argument term *t*.

The equality fact may be conditional, i.e., it may be a derivation rule with assumptions of the form $[\![RA_1; \ldots; RA_m]\!] \Longrightarrow$ `a = b`. When the *subst* method applies a conditional equation of this form to a goal $[\![A_1; \ldots; A_n]\!] \Longrightarrow C$, it adds the goals $[\![A_1; \ldots; A_n]\!] \Longrightarrow RA_i'$ to the goal state after rewriting, where RA_i' result from RA_i by the unification of `a` in `C`. These goals are inserted before the original goal, so the next method application will usually process the goal $[\![A_1; \ldots; A_n]\!] \Longrightarrow RA_1'$.

As an example if there are theorems

theorem *eq1*: `"n = 10 ⟹ n+3 = 13"` **for** `n::nat` ⟨*proof*⟩
theorem *eq2*: `"n = 5 ⟹ 2*n = 10"` **for** `n::nat` ⟨*proof*⟩

the method *subst (2) eq2* replaces the goal *(x::nat) < 5* $\Longrightarrow$ *2*x+3* $\leq$ *2*5 + 3* by the goals

```
x < 5 ⟹ 5 = 5
x < 5 ⟹ 2 * x + 3 ≤ 10 + 3
```

where the first is trivial (but still must be removed by applying a rule). The second goal is replaced by the method *subst (2) eq1* by

```
x < 5 ⟹ 10 = 10
x < 5 ⟹ 2 * x + 3 ≤ 13
```

Note that the method *subst eq2* would unify *2*n* with the first match *2*x* in the original goal and replace it by

```
x < 5 ⟹ x = 5
x < 5 ⟹ 10 + 3 ≤ 2 * 5 + 3
```

where the first goal cannot be proven because it is invalid.

Simplification

If the term *b* in an equation `a = b` is in some sense "simpler" than `a`, the goal will also become simpler by successful rewriting with the equation. If there are several such equations a goal can be replaced by successively simpler goals by rewriting with these equations. This technique can contribute to the goal's proof and is called "simplification".

Basically, simplification uses a set of equations and searches an equation in the set where the left hand side unifies with a sub-term in the goal, then substitutes it. This step is repeated until no sub-term in the goal unifies with a left hand side in an equation in the set.

It is apparent that great care must be taken when populating the set of equations, otherwise simplification may not terminate. If two equations `a = b` and `b = a` are in the set simplification will exchange matching terms forever. If an equation `a = a+0` is in the set, a term matching `a` will be replaced by an ever growing sum with zeroes.

Simplification with a set of definitional equations from constant definitions (see Section 2.1.4) always terminates. Since constant definitions cannot be recursive, every substitution removes one occurrence of a defined constant from the goal. Simplification terminates if no defined constant from the set remains in the goal. Although the resulting goal usually is larger than the original goal, it is simpler in the sense that it uses fewer defined constants.

If the set contains conditional equations, simplification may produce additional goals. Then simplification is applied to these goals as well. Together, simplification may turn a single complex goal into a large number of simple goals, but it cannot reduce the number of goals. Therefore simplification is usually complemented by methods which remove trivial goals like *x = x*, *A ⟹ A*, and *True*. Such an extended simplification may completely solve and remove the goal to which it is applied.

The *simp* Method

Isabelle supports simplification by the method

simp

which is also called "the simplifier". It uses the dynamic fact set *simp* as the set of equations, which is also called "the simpset". The method only affects the first goal. If no equation in *simp* is applicable to it or it is not modified by the applicable equations an error is signaled.

The *simp* method simplifies the whole goal, i.e., it applies rewriting to the conclusion and to all assumptions.

The simpset may contain facts which are not directly equations, but can be converted to an equation. In particular, an arbitrary derivation rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow C$ can always be converted to the equation $[\![A_1; \ldots; A_n]\!] \Longrightarrow C = True$. The simplifier performs this conversion if no other conversion technique applies, therefore the simpset may actually contain arbitrary facts.

The *simp* method also detects several forms of trivial goals and removes them. Thus a complete proof may be performed by a single application of the simplifier in the form

by *simp*

In Isabelle HOL (see Section **??**) the simpset is populated with a large number of facts which make the simplifier a very useful proof tool. Actually all examples of facts used in the previous sections can be proven by the simplifier:

theorem *example1:* *"(x::nat) < c ⟹ n*x ≤ n*c"* **by** *simp*
theorem *example2:* *"(x::nat) ≤ c ⟹ x + m ≤ c + m"* **by** *simp*
theorem *"(x::nat) < 5 ⟹ 2*x+3 ≤ 2*5 + 3"* **by** *simp*
theorem *eq1:* *"n = 10 ⟹ n+3 = 13"* **for** *n::nat* **by** *simp*
theorem *eq2:* *"n = 5 ⟹ 2*n = 10"* **for** *n::nat* **by** *simp*

Configuring the Simplifier

The simplifier can be configured by modifying the equations it uses. The form

simp add: name$_1$ *... name*$_n$

uses the facts *name*$_1$*, ..., name*$_n$ in addition to the facts in the simpset for its rewriting steps. The form

simp del: name$_1$ *... name*$_n$

uses only the facts from the simpset without the facts *name*$_1$ *...* *name*$_n$, and the form

simp only: name$_1$ *... name*$_n$

uses only the facts *name*$_1$ *...* *name*$_n$. The three forms can be arbitrarily combined.

As usual, a theorem may be added permanently to the simpset as described in Section 2.1.7 by specifying it as

theorem *[simp]: "prop"* ⟨*proof*⟩

and the defining equation of a definition can be added by

definition *name::type* **where** *[simp]: "name* ≡ *term"*

Adding own constant definitions to the simplifier is a common technique to expand the definition during simplification. However, this may also have a negative effect: If an equation has been specified using the defined constant, it is no more applicable for rewriting after expanding the definition. Note that the facts in the simpset and the facts provided by *add:*, *del:*, and *only:* are not simplified themselves, the defined constant will not be expanded there.

Therefore it is usually not recommended to add defining equations to the simpset permanently. Instead, they can be specified by *add:* when they really shall be expanded during simplification.

Splitting Terms

There are certain terms in which the simplifier will not apply its simpset rules. A typical example are terms with an internal case distinction (see Section **??**). To process such terms in a goal conclusion the terms must be split. Splitting a term usually results in several new goals with simpler terms which are then further processed by the simplifier.

Term splitting is done by applying specific rules to the goal. These rules are called "split rules". Usually split rules are not automatically determined and applied by the simplifier, this must be configured explicitly in the form

simp split: name$_1$ *... name*$_n$

where the *name*$_i$ are the names of the split rules to use. This configuration can be arbitrarily combined with the other simplifier configuration options.

Input Facts for the *simp* Method

As usual, facts may be input to the *simp* method. Like the empty method (see Section 2.3.1) it inserts these facts as assumptions into the goal, before it starts simplification. Since simplification is also applied to the assumptions, the input facts will be simplified as well.

As a possible effect of this behavior, after simplifying an input fact and the goal conclusion the results may unify, leading to the situation where the goal is removed by the *assumption* method (see Section 2.3.2). This is also done by the simplifier, hence in this way the input fact may contribute to prove the goal.

The *simp_all* Method

The method

simp_all

behaves like the *simp* method but processes all goals. It inserts input facts to all goals in the goal state and simplifies them. If it fails for all goals an error is signaled. Otherwise it simplifies only the goals for which it does not fail. If it achieves to remove all goals the proof is finished.

The *simp_all* method can be configured by *add:*, *del:*, and *only:* in the same way as the *simp* method.

The *simp_all* method is useful, if first a method *method* is applied to the goal which splits it into several subgoals which all can be solved by simplification. Then the complete proof can be written as

by *method simp_all*

Debugging the Simplifier

If the simplifier fails, it may be difficult to find out the reason. There are several debugging techniques which may help.

The content of the simpset can be displayed by the command

print_simpset

which may be written in the proof text in modes *proof(prove)* and *proof(state)* and outside of proofs. In the interactive editor the result is displayed in the Output panel (see Section 1.2.3).

There is also a simplifier trace which displays the successful rewrite steps. It is activated by the command

declare *[[simp_trace_new depth=n]]*

outside a theorem or definition. The number *n* should be atleast *2*. When the cursor is positioned on an application of the *simp* method the button "Show trace" can be used in the Simplifier Trace Panel to display the trace in a separate window. See the documentation for more information about how to use the trace.

Another technique is to replace the *simp* method by a sequence of *subst* method applications and explicitly specify the equations which should have been used. To do this for a structured proof, replace it by a proof script for the *subst* methods.

2.3.5 Other Automatic Proof Methods

Isabelle provides several other proof methods which internally perform several steps, like the simplifier.

Automatic Methods

The following list contains automatic methods other than *simp*:

- *blast* mainly applies logical rules and can be used to solve complex logical formulas.
- *clarify* is similar but does not split goals and does not follow unsafe paths. It can be used to show the problem if *blast* fails.
- *auto* combines logical rule application with simplification. It processes all goals and leaves those it cannot solve.
- *clarsimp* combines *clarify* with simplification. It processes only the first goal and usually does not split goals.
- *fastforce* uses more techniques than *auto*, but processes only the first goal.
- *force* uses even more techniques and tries to solve the first goal.

  The methods which do simplification can be configured like the simplifier by adding specifications *simp add:*, *simp del:*, *simp only:*, and *split:*. For example, additional simplification rules can be specified for the *auto* method in the form

  auto simp add: $name_1$ ... $name_n$

  For more information about these methods see the Isabelle documentation.

Trying Methods

Instead of manually trying several automatic methods it is possible to specify the command

try

anywhere in mode *proof(prove)*, i.e. at the beginning of a proof or in a proof script. It will try many automatic proof methods and describe the result in the Output window. It may take some time until results are displayed, in particular, if the goal is invalid and cannot be proven.

If **try** finds a proof for one or more goals it displays them as single proof methods, which, by clicking on them can be copied to the cursor position in the text area. The **try** command must be removed manually.

If **try** tells you that the goal can be "directly solved" by some fact, you can prove it by the *fact* method, but that also means that there is already a fact of the same form and your theorem is redundant.

It may also be the case that **try** finds a counterexample, meaning that the goal is invalid and cannot be proven.

2.4 Case Based Proofs

If a proof method splits a goal at the beginning of a structured proof the resulting subgoals must be proven separately, each with an own **show** statement. Some proof methods provide support for this by initializing a proof context for each goal resulting from the method application.

2.4.1 Goal Cases

If a method supports this, it creates a "case" for each corresponding goal. The cases are named. Using these names a proof context can be populated with elements prepared by the method.

Named Contexts

Every case name is associated with proof context elements, thus it can be seen as a named context which has been prepared by the method for later use. Such named contexts may contain facts, local variables, and term abbreviations. The actual content depends on the proof method and the goals for which the cases are created.

When the named context is used in a proof, its content is "injected" into the current proof context. Usually a named context is used in this way to initialize a new nested context immediately after its beginning.

The case Command

A case may be injected into the current proof context by the command

case *name*

where *name* is the case name. It mainly has the effect of the sequence

fix x_1 ... x_k
let ?a_1 = t_1 **and** ... ?a_m = t_m
assume *name:* "A_1" ... "A_n"

where x_1 ... x_k are the local variables, ?a_1, ..., ?a_m are the term abbreviations, and A_1, ..., A_n are the facts in the named context of the case. The facts are injected as assumptions and the set of these assumptions is named using the case name. Moreover, like the **assume** statement the **case** command makes the assumed facts current.

Instead of using the case name for naming the assumptions an explicit assumption name *aname* may be specified:

case *aname: name*

The local variables x_1 ... x_k are fixed by the **case** command but are hidden, they cannot be used in the subsequent proof text. If they should be used, explicit names must be specified for them in the form

case *(name* y_1 ... y_j*)*

Then the names y_1 ... y_j can be used to reference the fixed variables in the current proof context. If fewer names are specified only the first variables are named, if more names are specified than there are local variables in the case an error is signaled.

When methods create named contexts they usually only define the term abbreviation *?case* for the conclusion of the corresponding goal.

Proof Structure with Cases

The usual proof structure using cases consists of a sequence of nested contexts. At its beginning each context is initialized by a **case** command, at its end it uses a **show** statement to remove the corresponding goal:

proof *method*
case $name_1$
...
show *?case* ⟨*proof*⟩
next

case *name*$_2$
...
show *?case* ⟨*proof*⟩
next
...
next
case *name*$_n$
...
show *?case* ⟨*proof*⟩
qed

Every **show** command uses the local term abbreviation *?case* to refer to the conclusion of the corresponding goal.

In the interactive editor, when the cursor is positioned on **proof** *method* where the method supports cases, a skeleton of such a proof using the specific cases provided by the method is displayed in the Output panel. By clicking on it it may be copied into the text area immediately after the method specification.

2.4.2 The *goal_cases* Method

The simplest method with support for cases is

goal_cases

Without modifying the goal state it creates a named case for every existing goal. Input facts are ignored.

For a goal $\bigwedge x_1 \ldots x_m. [\![A_1; \ldots; A_n]\!] \Longrightarrow C$ the created named context contains the local variables $x_1 \ldots x_m$, the facts $A_1, \ldots, A_n$, and the term abbreviation *?case* bound to C. If the goal contains variables which are not explicitly bound by $\bigwedge$ these variables are not added to the context.

The effect is that if no other variables are fixed and no other facts are assumed a statement **show** *?case* after the corresponding **case** command will match the goal.

The cases are named by numbers starting with *1*. If other names should be used they can be specified as arguments to the method:

goal_cases name$_1$ ... *name*$_n$

If fewer names are specified than goals are present, only for the first *n* goals cases are created. If more names are specified an error is signaled.

When *goal_cases* is used in a composed proof method it can provide cases for the goals produced by arbitrary other methods:

proof *(method, goal_cases)*

provides cases for all goals existing after *method* has been applied. If *method* does not split the goal there will be only one case. This can be useful to work with a goal produced by *method*. In particular, the conclusion of that goal is available as *?case*.

Note that the proof state(s) resulting from *goal_cases* are not visible for the reader of the proof. Therefore it should only be applied if the goals produced by *method* are apparent. This can be supported by defining an abbreviated form of the conclusion by

let *"pattern" = ?case*

2.4.3 Case Based Reasoning

In "case based reasoning" a goal is proven by processing "all possible cases" for an additional assumption separately. If the conclusion can be proven for all these cases and the cases cover all possibilities, the conclusion holds generally.

In the simplest form a single proposition is used as additional assumption. Then there are only two cases: if the proposition is *True* or if it is *False*.

Consider the derivation rule *(x::nat) < c* $\Longrightarrow$ *n*x* $\leq$ *n*c* from Section 2.1.6. As additional assumption the proposition whether *n* is zero can be used. Then there are the two cases *n = 0* and *n* $\neq$ *0* and clearly these cover all possibilities. Using the first case as assumption implies that *n*x* and *n*c* are both zero and thus *n*x = n*c*. Using the second case as assumption together with the original assumption implies that *n*x < n*c*. Together the conclusion *n*x* $\leq$ *n*c* follows.

Case Rules

To prove a goal in this way it must be split into a separate goal for each case. All these goals must have the same conclusion but differ in the additional assumptions. This splitting can be done by applying a meta-rule of the form

$$[\![Q_1 \Longrightarrow \mathit{?P};\ \ldots;\ Q_n \Longrightarrow \mathit{?P}]\!] \Longrightarrow \mathit{?P}$$

Such rules are called "case rules".

When this case rule is applied to a goal $\bigwedge x_1 \ldots x_m.\ [\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow C$ as described in Section 2.3.2, it unifies *?P* with the conclusion *C* and replaces the goal by the *n* goals

$$\bigwedge x_1 \ldots x_m.\ [\![A_1;\ \ldots;\ A_n;\ Q_1]\!] \Longrightarrow C$$
$$\ldots$$
$$\bigwedge x_1 \ldots x_m.\ [\![A_1;\ \ldots;\ A_n;\ Q_n]\!] \Longrightarrow C$$

where every goal has one of the propositions Q_i as additional assumption.

The case rule is only valid, if the Q_i together cover all possibilities, i.e., if $Q_1 \vee \ldots \vee Q_n$ holds. Then it has been proven and is available as a fact which can be applied. Since the whole conclusion is the single unknown *?P* it unifies with every proposition used as conclusion in a goal, hence a case rule can always be applied to arbitrary goals. It depends on the Q_i whether splitting a specific goal with the case rule is useful for proving the goal.

A case rule for testing a natural number for being zero would be

$$[\![?n = 0 \Longrightarrow ?P;\ ?n \neq 0 \Longrightarrow ?P]\!] \Longrightarrow ?P$$

It contains the number to be tested as the unknown *?n*, so that an arbitrary term can be substituted for it. This is not automatically done by unifying *?P* with the goal's conclusion, thus the rule must be "prepared" for application to a specific goal. To apply it to the goal $(x::nat) < c \Longrightarrow n*x \leq n*c$ in the intended way the unknown *?n* must be substituted by the variable *n* from the goal conclusion. If the prepared rule is then applied to the goal it splits it into the goals

$$[\![(x::nat) < c;\ n = 0]\!] \Longrightarrow n*x \leq n*c$$
$$[\![(x::nat) < c;\ n \neq 0]\!] \Longrightarrow n*x \leq n*c$$

which can now be proven separately.

Actually, the much more general case rule

$$[\![?Q \Longrightarrow ?P;\ \neg\ ?Q \Longrightarrow ?P]\!] \Longrightarrow ?P$$

is used for this purpose. Here the unknown *?Q* represents the complete proposition to be used as additional assumption, therefore the rule can be used for arbitrary propositions. By substituting the term *n = 0* for *?Q* the rule is prepared to be applied in the same way as above.

Case rules may even be more general than shown above. Instead of a single proposition Q_i every case may have locally bound variables and an arbitrary number of assumptions, resulting in the most general form

$$\begin{array}{l}[\![\bigwedge x_{11} \ldots x_{1k1}.\ [\![Q_{11}; \ldots; Q_{1m1}]\!] \Longrightarrow ?P; \\ \quad \ldots; \\ \bigwedge x_{n1} \ldots x_{nkn}.\ [\![Q_{n1}; \ldots; Q_{nmn}]\!] \Longrightarrow ?P]\!] \Longrightarrow ?P\end{array}$$

of a case rule. When applied it splits the goal and adds the variables x_{i1} ... x_{iki} and the assumptions Q_{i1} ... Q_{imi} to the ith case.

The *cases* Method

Case based reasoning can be performed in a structured proof using the method *cases* in the form

```
cases "term" rule: name
```

where *name* is the name of a valid case rule. The method prepares the rule by substituting the specified *term* for the unknown in the assumptions, and applies the rule to the first goal in the goal state.

Additionally, the method creates a named context for every goal resulting from the rule application. The context contains the variables and assumptions specified in the corresponding case in the case rule. For the most general form depicted above the context for the ith case contains the variables x_{i1} ... x_{iki} and the assumptions Q_{i1} ... Q_{imi}. No term abbreviation *?case* is defined, because the conclusion of every new goal is the same as that of the original goal, thus the existing abbreviation *?thesis* can be used instead.

Often a case rule has only one unknown in the case assumptions. If there are more, several terms may be specified in the *cases* method for preparing the rule.

The *cases* method treats input facts like the empty method (see Section 2.3.1) by inserting them into the original goal before splitting it.

Like the *rule* method (see Section 2.3.2) the *cases* method supports automatic rule selection for the case rule and may be specified in the form

```
cases "term"
```

Normally the rule is selected according to the type of the specified *term*. In Isabelle HOL (see Section **??**) most types have an associated case rule.

The rule $[\![?Q \Longrightarrow ?P;\ \neg\ ?Q \Longrightarrow ?P]\!] \Longrightarrow ?P$ depicted above is associated with type *bool*. Therefore the case splitting for comparing *n* with zero can be done by the method

```
cases "n = 0"
```

The names used for the contexts created by the *cases* method can be specified by attributing the case rule. The case rule for *bool* is attributed to use the case names *True* and *False*. Note that these are names, not the constants for the values of type *bool*.

The proof writer may not know the case names specified by the automatically selected case rule. However, they can be determined from the proof skeleton which is displayed in the interactive editor when the cursor is positioned on the *cases* method (see Section 2.4.1).

Together, a structured proof for the goal *(x::nat) < c* $\Longrightarrow$ *n*x* $\leq$ *n*c* with case splitting may have the form

```
proof (cases "n = 0")
case True
```

...
show `?thesis` ⟨*proof*⟩
next
case `False`
...
show `?thesis` ⟨*proof*⟩
qed

The `cases` method adds the assumptions `n=0` and `n≠0` to the goals of the cases, the **case** commands add them as assumed facts to the local context, so that they are part of the rule exported by the **show** statement and match the assumption in the corresponding goal.

Note that the **case** command adds only the assumptions originating from the case rule. The other assumptions in the original goal (here `x < c`) must be added to the context in the usual ways (see Section 2.2.8) if needed for the proof.

2.4.4 Induction

todo

Induction Rules

todo

The `induct` Method

todo

Chapter 3

Isabelle HOL Basics

The basic mechanisms described in Chapter 2 can be used for working with different "object logics". An object logic defines the types of objects available, constants of these types, and facts about them. An object logic may also extend the syntax, both inner and outer syntax.

The standard object logic for Isabelle is the "Higher Order Logic" HOL, it covers a large part of standards mathematics and is flexibly extensible. This chapter introduces some basic mechanisms defined by HOL which are used to populate HOL with many of its mathematical objects and functions and which can also be used to extend HOL to additional kinds of objects. Basically these mechanisms support the definition of new types.

HOL extends the basic type introduction mechanisms of Isabelle (see Section 2.1.2) by several ways of specifying the set of values of a new type. This section introduces four of them: algebraic types, records, subtypes, and quotient types. Additionally it introduces "let"-terms which can be used with arbitrary types.

3.1 Algebraic Types

Roughly an algebraic type is equivalent to a union of cartesian products with support for recursion. In this way most data types used in programming languages can be covered, such as records, unions, enumerations, and pointer structures. Therefore Isabelle also uses the notion "datatype" for algebraic types.

3.1.1 Definition of Algebraic Types

Basically, an algebraic type is defined in the form

datatype *name* = alt_1 | ... | alt_n

where *name* is the name of the new algebraic type and every alternative alt_i is a "constructor specification" of the form

$cname_i$ "$type_{i1}$" ... "$type_{iki}$"

The $cname_i$ are names and the $type_{ij}$ are types. The types are specified in inner syntax and must be quoted, if they are not a single type name. All other parts belong to the outer syntax.

Recursion is supported for the types, i.e., the name *name* of the defined type may occur in the type specifications $type_{ij}$. However, there must be atleast one constructor specification which is not recursive, otherwise the definition does not "terminate". Isabelle checks this condition and signals an error if it is not satisfied.

As a convention, capitalized names are used in Isabelle HOL for the $cname_i$.

An example for a datatype definition with two constructor specifications is

```
datatype coord =
  Dim2 nat nat
| Dim3 nat nat nat
```

Its value set is equivalent to the union of pairs and triples of natural numbers.

3.1.2 Constructors

Every $cname_i$ is used by the definition to introduce a "(value) constructor function", i.e., a constant

$cname_i$ :: "$type_{i1}$ ⇒ ... ⇒ $type_{iki}$ ⇒ *name*"

which is a function with *ki* arguments mapping their arguments to values of the new type *name*.

Every datatype definition constitutes a separate namespace for the functions it introduces. Therefore the same names may be used in constructor specifications of different datatype definitions. If used directly, a name refers to the constructor function of the nearest preceding datatype definition. To refer to constructor functions with the same name of other datatypes the name may be qualified by prefixing it with the type name in the form $name.cname_i$.

The definition of type *coord* above introduces the two constructor functions *Dim2* :: *nat* ⇒ *nat* ⇒ *coord* and *Dim3* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *coord*. Their qualified names are *coord.Dim2* and *coord.Dim3*.

Constructing Values

These constructor functions are assumed to be injective, thus their result values differ if atleast one argument value differs. This implies that the set

of all values of the constructor function *cname*$_i$ is equivalent to the cartesian product of the value sets of *type*$_{i1}$ ... *type*$_{iki}$: for every tuple of arguments there is a constructed value and vice versa. Note, however, that as usual the values of the new type are distinct from the values of all other types, in particular, they are distinct from the argument tuples.

Moreover the result values of different constructor functions are also assumed to be different. Together the set of all values of the defined type is equivalent to the (disjoint) union of the cartesian products of all constructor argument types. Moreover, every value of the type may be denoted by a term

cname$_i$ *term*$_1$ ... *term*$_{ki}$

where each *term*$_j$ is of type *type*$_{ij}$ and specifies an argument for the constructor function application.

Values of type *coord* as defined above are denoted by terms such as *Dim2 0 1* and *Dim3 10 5 21*.

Constant Constructors and Enumeration Types

A constructor specification may consist of a single constructor name *cname*$_i$, then the constructor functions has no arguments and always constructs the same single value. The constructor is equivalent to a constant of type *name*. As a consequence an "enumeration type" can be defined in the form

datatype *three = Zero | One | Two*

This type *three* has three values denoted by *Zero*, *One*, and *Two*.

Types with a Single Constructor

If a datatype definition consists of a single constructor specification its value set is equivalent to the corresponding cartesian product. The corresponding tuples have a separate component for every constructor argument type. As a consequence a "record type" can be defined in the form

datatype *recrd = MkRecrd nat "nat set" bool*

Its values are equivalent to triples where the first component is a natural number, the second component is a set of natural numbers, and the third component is a boolean value. An example value is denoted by *MkRecrd 5 {1,2,3} True*.

Since there must be atleast one nonrecursive constructor specification, definitions with a single constructor specification cannot be recursive.

3.1.3 Destructors

Since constructor functions are injective it is possible to determine for every value of the defined type the value of each constructor argument used to construct it. Corresponding mechanisms are called "destructors", there are three different types of them.

Selectors

The most immediate form of a destructor is a selector function. For the constructor argument specified by $type_{ij}$ the selector function is a function of type $name \Rightarrow type_{ij}$. For every value constructed by $cname_i\ term_1\ \ldots\ term_{ki}$ it returns the value denoted by $term_j$.

The names of selector functions must be specified explicitly. This is done using the extended form of a constructor specification

$cname_i$ `(`$sname_{i1}$ `: "`$type_{i1}$`") ... (`$sname_{iki}$ `: "`$type_{iki}$`")`

where the $sname_{ij}$ are the names used for the corresponding selector functions. Selector names may be specified for all or only for some constructor arguments. As for constructors, selector names belong to the namespace of the defined type and may be qualified by prefixing the type name.

An example datatype definition with selectors is

```
datatype recrd = MkRecrd (n:nat) (s:"nat set") (b:bool)
```

It shows that the selector functions correspond to the field names used in programming languages in record types to access the components. For every term `r` of type `recrd` the selector term `s r` denotes the set component of `r`.

An example for a datatype with multiple constructor specifications is

```
datatype coord =
  Dim2 (x:nat) (y:nat)
| Dim3 (x:nat) (y:nat) (z:nat)
```

Note that the selectors `x` and `y` are specified in both alternatives. Therefore a single selector function `x :: coord ⇒ nat` is defined which yields the first component both for a two-dimensional and a three-dimensional coordinate and analogously for `y`. If instead the definition is specified as

```
datatype coord =
  Dim2 (x2:nat) (y:nat)
| Dim3 (x3:nat) (y:nat) (z:nat)
```

two separate selector functions `x2` and `x3` are defined where the first one is only applicable to two-dimensional coordinates and the second one only to three-dimensional coordinates.

If a selector name does not occur in all constructor specifications, the selector function is still total, like all functions in Isabelle, but it is underspecified (see Section 2.1.3). It maps values constructed by other constructors to a unique value of its result type, even if that other constructor has no argument of this type. However, no information is available about that value.

For the type *coord* the selector function *z :: coord ⇒ nat* is also applicable to two-dimensional coordinates, however, the values it returns for them is not specified.

Such selector values are called "default selector values". They may be specified in the extended form of a datatype definition

datatype *name = alt*$_1$ *| ... | alt*$_n$
where *"prop*$_1$*" | ... | "prop*$_m$*"*

where every *prop*$_p$ is a proposition of the form

sname$_{ij}$ *(cname*$_q$ *var*$_1$ *... var*$_{kq}$*) = term*$_p$

and specifies *term*$_p$ as the default value of selector *sname*$_{ij}$ for values constructed by *cname*$_q$.

The definition

```
datatype coord =
  Dim2 (x:nat) (y:nat)
| Dim3 (x:nat) (y:nat) (z:nat)
where "z (Dim2 a b) = 0"
```

specifies *0* as default value for selector *z* if applied to a two-dimensional coordinate.

Discriminators

If an underspecified selector is applied to a datatype value it may be useful to determine which constructor has been used to construct the value. This is supported by discriminator functions. For every constructor specification for *cname*$_i$ the discriminator function has type *name ⇒ bool* and returns true for all values constructed by *cname*$_i$. Like selector names, discriminator names must be explicitly specified using the extended form of a datatype definition

datatype *name = dname*$_1$*: alt*$_1$ *| ... | dname*$_n$*: alt*$_n$

Discriminator names may be specified for all alternatives or only for some of them. Note that for a datatype with a single constructor the discriminator returns always *True* and for a datatype with two constructors one discriminator is the negation of the other.

An example datatype definition with discriminators is

```
datatype coord =
  is_2dim: Dim2 nat nat
| is_3dim: Dim3 nat nat nat
```

In a datatype definition both discriminators and selectors may be specified.

The *case* Term

Additionally to using discriminators and selectors Isabelle HOL supports `case` terms. A `case` term specifies depending on a datatype value a separate term variant for every constructor of the datatype. In these variants the constructor arguments are available as bound variables.

A `case` term for a datatype `name` defined as in Section 3.1.1 has the form

```
case term of
  cname_1 var_11 ... var_1k1 ⇒ term_1
| ...
| cname_n var_n1 ... var_nkn ⇒ term_n
```

where `term` is of type `name` and the $term_i$ have an arbitrary but common type which is also the type of the `case` term. In the alternative for constructor $cname_i$ the var_{11} ... var_{1k1} must be distinct variables, they are bound to the constructor arguments and may be used in $term_i$ to access them. The value of var_{ij} is the same as the value selected by $sname_{ij}$ `term`.

If `cv` is a variable or constant of type `coord` an example `case` term for it is

```
case cv of
  Dim2 a b ⇒ a + b
| Dim3 a b c ⇒ a + b + c
```

It denotes the sum of the coordinates of `cv`, irrespective whether `cv` is two-dimensional or three-dimensional.

A `case` term is useful even for a datatype with a single constructor. If `rv` is of type `recrd` as defined in Section 3.1.3 the `case` term

```
case rv of MkRecrd nv sv bv ⇒ term
```

makes the components of `rv` locally available in `term` as `nv`, `sv`, `bv`. It is equivalent to `term` where `nv`, `sv`, and `bv` have been substituted by the selector applications `(n rv)`, `(s rv)`, and `(b rv)`.

The variant terms in a `case` term cannot be matched directly by a **let** statement in a proof (see Section 2.2.11). The statement

```
let "case rv of MkRecrd nv sv bv ⇒ ?t"
   = "case rv of MkRecrd nv sv bv ⇒ term"
```

will fail to bind *?t* to *term* because then the variables *nv*, *sv*, and *bv* would occur free in it and the relation to the constructor arguments would be lost. Instead, the statement

```
let "case rv of MkRecrd nv sv bv ⇒ ?t nv sv bv"
   = "case rv of MkRecrd nv sv bv ⇒ term"
```

successfully binds *?t* to the lambda term *λnv sv bv. term* which denotes the function which results in *term* when applied to the constructor arguments.

3.1.4 Rules

A datatype definition also introduces a large number of named facts about the constructors and destructors of the defined type. All fact names belong to the namespace of the datatype definition. Since the fact names cannot be specified explicitly, all datatype definitions use the same fact names, therefore the fact names must always be qualified by prefixing the type name.

Several rules are configured for automatic application, e.g., they are added to the simpset for automatic application by the simplifier (see Section 2.3.4). Other rules must be explicitly used by referring them by their name.

Only some basic rules are described here, for more information refer to the Isabelle documentation about datatypes.

Simplifier Rules

The rules added by a datatype definition to the simpset (see Section 2.3.4) support many ways for the simplifier to process terms with constructors and destructors. An example are rules of the form

$$sname_{ij}\ (cname_i\ x_1\ \ldots\ x_{ki}) = x_j$$

The set of all rules added to the simpset is named *name.simps*. By displaying it using the **thm** command (see Section 2.1.7) it can be inspected to get an idea how the simplifier processes terms for datatypes.

Case Rule

Every datatype definition introduces a case rule (see Section 2.4.3) of the form

lemma *name.exhaust:*

$$\begin{aligned}&"[\![\bigwedge x_1\ \ldots\ x_{k1}.\ y = cname_1\ x_1\ \ldots\ x_{k1} \Longrightarrow P;\\ &\quad \ldots\ ;\\ &\quad \bigwedge x_1\ \ldots\ x_{kn}.\ y = cname_n\ x_1\ \ldots\ x_{kn} \Longrightarrow P]\!] \Longrightarrow P"\end{aligned}$$

It is valid because the constructor applications cover all possibilities of constructing a value `y` of the datatype.

This rule is associated with the datatype for use by the `cases` method (see Section 2.4.3). Therefore the application of the method

```
cases "term"
```

where `term` is of type `name` splits an arbitrary goal into `n` subgoals where every subgoal uses a different constructor to construct the `term`.

The names for the named contexts created by the `cases` method are simply the constructor names $cname_i$. Therefore a structured proof using case based reasoning for a `term` of datatype `name` has the form

proof *(cases "term")*
  case *(*$cname_1$ x_1 ... x_{k1}*)* ... **show** *?thesis* ⟨*proof*⟩
next
...
next
  case *(*$cname_n$ x_1 ... x_{kn}*)* ... **show** *?thesis* ⟨*proof*⟩
qed

The names x_i of the locally fixed variables can be freely selected, they denote the constructor arguments of the corresponding constructor. Therefore the case specification *(*$cname_i$ x_1 ... x_{ki}*)* looks like a constructor application to variable arguments, although it is actually a context name together with locally fixed variables.

Split Rule

todo

Induction Rule

todo

3.2 Record Types

todo

3.3 Subtypes

A subtype specifies the values of a type simply as a set of values of an existing type. However, since the values of different types are always disjoint,

the values in the set are not directly the values of the new type, instead, there is a 1-1 relation between them, they are isomorphic. The values in the set are called "representations", the values in the new type are called "abstractions".

3.3.1 Subtype Definitions

A subtype is defined in the form

typedef *name* `= "term"` ⟨*proof*⟩

where *name* is the name of the new type and *term* is a term for the representing set. See Section 4.5 for how to denote such terms. The ⟨*proof*⟩ must prove that the representing set is not empty.

A simple example is the type

typedef *three* `= "{1::nat,2,3}"` **by** `auto`

which has three values. The representations are natural numbers, as usual, the type *nat* must be specified because the constants *1, 2, 3* may also denote values of other types. However, they do not denote the values of the new type *three*, the type definition does not introduce constants for them.

Instead, a subtype definition **typedef** `t = "term"` ⟨*proof*⟩ introduces two functions *Abs_t* and *Rep_t*. These are morphisms between the set and the new type, *Abs_t* maps from the set to type *t*, *Rep_t* is its inverse. Both functions are injective, together they provide the 1-1 mapping between the subtype and the representing set. The function *Abs_t* can be used to denote the values of the subtype.

In the example the morphisms are *Abs_three :: nat* ⇒ *three* and *Rep_three :: three* ⇒ *nat*. The values of type *three* may be denoted as *(Abs_three 1)*, *(Abs_three 2)*, and *(Abs_three 3)*.

Alternative names may be specified for the morphisms in the form

typedef `t = "term"` **morphisms** *rname aname* ⟨*proof*⟩

where *rname* replaces *Rep_t* and *aname* replaces *Abs_t*.

Like declared types subtypes may be parameterized (see Section 2.1.2):

typedef *('name$_1$,...,'name$_n$) name* `= "term"` ⟨*proof*⟩

where the *'name$_i$* are the type parameters. They may occur in the type of the *term*, i.e., the *term* may be polymorphic (see Section 2.1.3).

3.3.2 Subtype Rules

todo

3.4 Quotient Types

todo

3.5 Type Independent Mechanisms

todo

3.5.1 Undefined Value

todo

3.5.2 Let Terms

todo

Chapter 4

Isabelle HOL Types

This chapter introduces a small basic part of the types available in HOL. The selected types are considered useful for some forms of program verification. Most of the types are algebraic types (see Section 3.1). Although some of them are defined differently for technical reasons, they are configured afterwards to behave as if they have been defined as algebraic types. Therefore they are described here using the corresponding datatype definition.

4.1 Boolean Values

todo

- $\wedge$, $\vee$, $\neg$, $\longrightarrow$
- =, $\neq$, $\longleftrightarrow$
- $\forall$, $\exists$
- intro

4.1.1 Conditional Terms

todo

4.1.2 Logic Rules

todo

- conjI, conjE, disjI1, disjI2, disjE, impI, mp

- contrapos_*
- iffI, iffE, iffD1, iffD2
- allI, allE, exI, exE

4.2 Natural Numbers

todo

4.3 Tuple Types

Tuple types are constructed by cartesian product of existing types. Tuple types can be directly denoted by type expressions of the form

$t_1 \times \ldots \times t_n$

in inner syntax. It is not necessary to introduce a name to use a tuple type, however, this is possible by defining a type synonym (see Section 2.1.2):

type_synonym $t = t_1 \times \ldots \times t_n$

4.3.1 Tuple Values

todo

4.4 Optional Values

todo

4.5 Sets

todo

4.6 Lists

todo

4.7 Fixed-Size Binary Numbers

todo