

A Gentle Introduction to Isabelle and Isabelle/HOL

Gunnar Teege
Universität der Bundeswehr München
`gunnar.teege@unibw.de`

March 27, 2025
Compatible with Isabelle Release 2025

Contents

1	Isabelle System	8
1.1	Invoking Isabelle	8
1.1.1	Installation and Configuration	8
1.1.2	Theories and Sessions	9
1.1.3	Invocation as Editor	9
1.1.4	Invocation for Batch Processing	10
1.1.5	Invocation for Document Creation	11
1.2	Interactively Working with Isabelle	11
1.2.1	The Text Area	11
1.2.2	The Sidekick Panel	12
1.2.3	The Output Panel	12
1.2.4	The State Panel	12
1.2.5	The Symbols Panel	12
1.2.6	The Documentation Panel	13
1.2.7	The Query Panel	13
1.2.8	The Theories Panel	13
2	Isabelle Theories	14
2.1	Theory Notation	14
2.1.1	Languages for Theory Content	14
2.1.2	Meta Level and Object Level	15
2.1.3	Theory Structure	16
2.2	Terms and Types	16
2.2.1	Types	17
2.2.2	Constants and Variables	18
2.2.3	Functions	19
2.2.4	Functions with Multiple Arguments	19
2.2.5	Lambda-Terms	20
2.2.6	Searching Constants	21
2.3	Definitions and Abbreviations	21
2.3.1	Definitions	22
2.3.2	Abbreviations	23
2.4	Overloading	23

2.4.1	True Overloading	23
2.4.2	Adhoc Overloading	24
2.5	Propositions	24
2.5.1	Formulas	24
2.5.2	Derivation Rules	25
2.5.3	Binding Free Variables	26
2.5.4	Rules with Multiple Conclusions	26
2.5.5	Alternative Rule Syntax	26
2.5.6	Structured Rule Syntax	27
2.5.7	Conditional Definitions	28
2.6	Theorems	28
2.6.1	Specifying Theorems	28
2.6.2	Unknowns	30
2.6.3	Named Facts	31
2.6.4	Alternative Theorem Syntax	32
2.6.5	Definitions as Facts	33
2.6.6	Displaying and Searching Named Facts	33
3	Isabelle Proofs	35
3.1	Maintaining Proof State	35
3.1.1	Proof Context	35
3.1.2	Goal State	36
3.1.3	Initial Proof State	36
3.2	Proof Procedure	37
3.2.1	Constructing a Proof	38
3.2.2	Unification	38
3.2.3	Storing Facts During a Proof	39
3.2.4	Multiple Assumptions	40
3.2.5	Proving from External Facts	41
3.3	Basic Proof Structure	41
3.3.1	Statements and Methods	41
3.3.2	Proof Syntax	42
3.3.3	Bogus Proofs	44
3.3.4	Nested Proof Contexts	44
3.3.5	Subproofs	46
3.4	Method Application	47
3.4.1	Proof Methods	47
3.4.2	Method Application	47
3.5	Stating Facts	47
3.5.1	Adding a Fact to the Proof Context	48
3.5.2	Proving a Goal	48
3.6	Facts as Proof Input	50
3.6.1	Using Input Facts in a Proof	50
3.6.2	Inputting Facts into a Proof	51

3.7	Fact Chaining	52
3.7.1	Automatic Update of the Current Facts	52
3.7.2	Naming and Grouping Current Facts	54
3.7.3	Accessing Input Facts in a Proof	54
3.7.4	Exporting the Current Facts of a Nested Context	54
3.8	Assuming Facts	55
3.8.1	Introducing Assumed Facts	55
3.8.2	Exporting Facts with Assumptions	56
3.8.3	Presuming Facts	57
3.9	Fixing Variables	57
3.9.1	Local Variables	58
3.9.2	Exporting Facts with Local Variables	58
3.10	Defining Variables	59
3.10.1	Defining Equations Viewed as Local Facts	60
3.10.2	Exporting Facts after Defining Variables	60
3.11	Obtaining Variables	60
3.11.1	Proving <i>obtain</i> Statements	61
3.11.2	Exporting Facts after Obtaining Variables	62
3.12	Term Abbreviations	62
3.12.1	Defining Term Abbreviations	62
3.12.2	Pattern Matching	62
3.12.3	Casual Term Abbreviations	64
3.13	Accumulating Facts	65
3.13.1	Support for Fact Accumulation	66
3.13.2	Accumulating Facts in a Nested Context	66
3.13.3	Accumulating Facts for Joining Branches	66
3.14	Equational Reasoning	67
3.14.1	Transitivity Rules	67
3.14.2	Support for Equational Reasoning	68
3.14.3	Determining Transitivity Rules	69
4	Proof Methods	70
4.1	The empty Method	70
4.2	Terminating Proof Scripts	71
4.2.1	The <i>assumption</i> Method	71
4.3	Basic Rule Application	72
4.3.1	The <i>rule</i> Method	72
4.3.2	Using the <i>rule</i> Method for Backward Reasoning Steps	73
4.3.3	Automatic Rule Selection	74
4.3.4	Introduction Rules	74
4.3.5	The <i>standard</i> Method	75
4.4	Rule Application in Forward Reasoning	76
4.4.1	Using the <i>rule</i> Method for Forward Reasoning Steps	77
4.4.2	Input Facts for the <i>rule</i> Method	78

4.4.3	The Method <i>this</i>	79
4.4.4	The Methods <i>frule</i> and <i>drule</i>	80
4.4.5	Destruction Rules	82
4.5	Composed Proof Methods	83
4.6	The Simplifier	84
4.6.1	The <i>subst</i> Method	84
4.6.2	Simplification	86
4.6.3	The <i>simp</i> Method	87
4.6.4	Configuring the Simplifier	88
4.6.5	Splitting Terms	89
4.6.6	Searching Simplifier Equations	90
4.6.7	Input Facts for the <i>simp</i> Method	90
4.6.8	The <i>simp_all</i> Method	90
4.6.9	Debugging the Simplifier	91
4.7	Other Automatic Proof Methods	91
4.7.1	Automatic Methods	91
4.7.2	Trying Methods	92
5	Case Based Proofs	93
5.1	Named Proof Contexts	93
5.1.1	The <i>case</i> Statement	93
5.1.2	Proof Structure with Cases	94
5.2	The <i>goal_cases</i> Method	95
5.3	Case Based Reasoning	97
5.3.1	Case Rules	97
5.3.2	The <i>cases</i> Method	99
5.3.3	Alternative Syntax for Case Rules	102
5.4	Elimination	104
5.4.1	The Method <i>erule</i>	104
5.4.2	Elimination Rules	105
5.4.3	Alternative Syntax for Elimination Rules	107
5.4.4	Elimination in Structured Proofs	107
5.5	Induction	108
5.5.1	Induction Rules	109
5.5.2	The <i>induction</i> Method	111
5.5.3	Case Assumption Naming and the <i>induct</i> Method	113
5.5.4	Goals with Assumptions	113
5.5.5	Induction with Elimination	115
5.5.6	Arbitrary Variables	115
6	Isabelle/HOL Basics	117
6.1	Predicates and Relations	117
6.1.1	Predicates	117
6.1.2	Unary Predicates and Sets	118

6.1.3	Relations	118
6.2	Equality, Orderings, and Lattices	120
6.2.1	The Equality Relation	120
6.2.2	The Ordering Relations	120
6.2.3	Lattice Operations	121
6.3	Description Operators	122
6.3.1	The Choice Operator	122
6.3.2	The Definite Description Operator	122
6.3.3	The Least and Greatest Value Operators	123
6.4	Undefined Value	123
6.5	Let Terms	124
6.6	Tuples	125
6.6.1	Function Argument Tuples	125
6.6.2	Relations as Tuple Sets	125
6.7	Inductive Definitions	126
6.7.1	The Defining Rules	126
6.7.2	Fixed Arguments	129
6.7.3	The <i>cases</i> Rule	129
6.7.4	The Induction Rule	132
6.7.5	Single-Step Inductive Definitions	133
6.7.6	Mutually Inductive Definitions	134
6.8	Well-Founded Relations	135
6.8.1	Induction	135
6.8.2	The Accessible Part of a Relation	136
6.8.3	Measure Functions	136
6.8.4	The <i>size</i> Function	137
6.9	The Proof Method <i>atomize_elim</i>	137
6.10	Recursive Functions	138
6.10.1	The Defining Equations	139
6.10.2	Covering All Arguments	141
6.10.3	Uniqueness	143
6.10.4	The Domain Predicate	144
6.10.5	Rules Provided by Recursive Definitions	145
6.10.6	Termination	146
6.10.7	Rules Provided by Termination Proofs	148
6.10.8	Mutual Recursion	149
6.11	Functors	150
6.11.1	Mappers	150
6.11.2	Multivariate Functors	152
6.11.3	Registering Functors	153
6.12	Bounded Natural Functors	154
6.12.1	Natural Functors	154
6.12.2	Predicators and Relators	157
6.12.3	Bounded Natural Functors	159

6.12.4	Registering Bounded Natural Functors	161
6.13	Transfer	163
6.13.1	Transfer Rules	164
6.13.2	The <i>transfer</i> Method	168
6.13.3	Polymorphic Types	170
6.14	Quotients and Lifting	172
6.14.1	Abstract and Raw Types	172
6.14.2	Quotient Theorems	173
6.14.3	Polymorphic Quotients	175
6.14.4	Registering Quotients	176
6.14.5	Lifting	180
6.14.6	Quotients as Bounded Natural Functors	184
7	Isabelle/HOL Type Definitions	188
7.1	Algebraic Types	188
7.1.1	Definition of Algebraic Types	188
7.1.2	Constructors	190
7.1.3	Destructors	192
7.1.4	Parameterized Algebraic Types as Bounded Natural Functors	196
7.1.5	Rules	197
7.1.6	Recursive Functions on Algebraic Types	202
7.2	Subtypes	205
7.2.1	Subtypes as Quotients	205
7.2.2	Parameterized Subtypes	206
7.2.3	Type Copies	207
7.2.4	Subtype Rules	207
7.2.5	Registering Subtypes as Quotients	210
7.2.6	Lifting Functions to Subtypes	211
7.2.7	Subtypes of Bounded Natural Functors	212
7.3	Quotient Types	213
7.3.1	Quotient Types as Quotients	214
7.3.2	Parameterized Quotient Types	215
7.3.3	Quotient Type Rules	216
7.3.4	Lifting Functions to Quotient Types	218
7.3.5	Quotient Types for Bounded Natural Functors	219
7.4	Record Types	220
7.4.1	Record Definitions	220
7.4.2	Record Constructors	223
7.4.3	Record Destructors	225
7.4.4	Record Updates	225
7.4.5	Record Rules	226
7.4.6	Records as Bounded Natural Functors	229

8 Isabelle/HOL Types	232
8.1 Boolean Values	232
8.1.1 Values	233
8.1.2 Destructors	233
8.1.3 Functions	233
8.1.4 Rules	235
8.2 The Unit Type	237
8.2.1 Values	237
8.2.2 Destructors	238
8.2.3 Rules	238
8.3 Natural Numbers	238
8.3.1 Values	238
8.3.2 Destructors	239
8.3.3 Functions	239
8.3.4 Rules	240
8.4 Sets	242
8.4.1 Values	242
8.4.2 Destructors	243
8.4.3 Functions	244
8.4.4 Rules	247
8.5 Optional Values	249
8.5.1 Values	249
8.5.2 Destructors	250
8.5.3 Functions	250
8.5.4 Rules	251
8.6 Tuples	252
8.6.1 Values	253
8.6.2 Destructors	253
8.6.3 Functions	254
8.6.4 Rules	256
8.7 Functions	257
8.7.1 Values	258
8.7.2 Functions on Functions	258
8.7.3 Rules	262
8.8 Relations	263
8.8.1 Functions	263
8.9 The Sum Type	263
Bibliography	264
Index	265

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/> and other mirror sites 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.

Processing the Text Area Content

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.

Displaying Definitions of Identifiers

Most identifiers used in Isabelle content have been defined in some theory file, this also holds for commands and other elements of the Isabelle syntax. The definition can be accessed by holding down CTRL (CMD on Macs) and clicking on an identifier. This also works for identifiers displayed in most other panels.

The definition is displayed by opening the corresponding theory file in the text area and positioning the window on the definition text.

If an identifier's definition has been loaded from a heap file (see Section 1.1.2) it is still displayed by opening the theory file, however, its content is not processed in the way described above. In particular, this is usually the case for all identifiers defined in the sessions `HOL/HOL` and `Pure/Pure`.

1.2.2 The Sidekick Panel

The Sidekick panel displays a structured view of the content of the text area and supports interactively expanding and collapsing substructures. It can be used for navigation in the text area by clicking on an item displayed in the Sidekick panel.

The structure view is not updated automatically upon changes in the text area, to update it the text area content must be saved to its theory file.

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 3.1.2), and describes what remains to be proved 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. It comprises several tabs for selecting different symbol sets.

Alternatively, symbols not available on the keyboard may be entered by a specific sequence of keys, called an abbreviation. As an example, the sequence `==>` is an abbreviation for the symbol \Rightarrow consisting of three separate keys. In the interactive editor Isabelle replaces some abbreviations upon entering automatically by their symbol, others are left as they are.

If the mouse is positioned on a symbol in the Symbols panel available abbreviations are displayed as `abbrev:...` in the popup message.

1.2.6 The Documentation Panel

A comprehensive set of documentation about Isabelle can be opened through the Documentation panel. This introduction refers to some of these documentations, if applicable, through the bibliography.

For example, more information about the use of the interactive editor can be found in the Isabelle Manual about jEdit [8].

1.2.7 The Query Panel

The Query panel supports active searching for items in the content of all loaded sessions. There are several tabs for searching different kinds of items. Depending on the kind, a search specification must be entered in the tab, the results are displayed in the window below.

Note that a simple full text search is usually not supported. More information about supported search specifications for different kinds of items can be found in Chapter 2.

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 Theories

A theory is the content of an Isabelle theory file.

2.1 Theory Notation

Theories are written using a notation which follows strict syntax rules, similar to a programming language. The interactive editor helps using correct notation by immediately signaling syntax errors upon input. However, it is crucial to also understand the *meaning* of the written text. This introduction describes both in a systematic stepwise manner.

2.1.1 Languages for Theory Content

Isabelle aims at two main goals: Supporting interactive work with mathematical formulas and proofs, and supporting their presentation together with explanations as nicely typeset text in the form of an article or a book. For this purpose Isabelle combines three different languages for writing theory content:

- a language for mathematics, called the “inner syntax”,
- a language for textual content, which is an extended form of \LaTeX source code,
- a language for organizing fragments of the first two languages in a theory, called the “outer syntax”.

Inner Syntax

For the inner syntax Isabelle tries to be as close as possible to the form how mathematical content is traditionally presented in books. It supports

formulas such as $\forall x \leq 100 . x \in \{y^2 \mid y. y \leq 10\}$ both in the interactive editor and the textual presentation. It does not support larger two-dimensional constructs like matrices or fractions.

Outer Syntax

The outer syntax resembles a programming language. It uses keywords to construct larger entities like definitions and proofs. These entities usually contain mathematical formulas written in inner syntax. To clearly separate both languages, content in inner syntax must always be surrounded by double quotes `"..."` or by the “cartouche delimiters” `<...>` available in the editor’s Symbols panel in tab “Punctuation”. The only exception is a single isolated identifier, for it the quotes or delimiters may be omitted.

This introduction describes only a selected part of the outer and inner syntax. The full notation used by Isabelle is described in the Isabelle/Isar reference manual [7] with some more advanced parts in other documentation [1, 2, 4].

Embedded L^AT_EX Code

Additionally, text written in L^AT_EX syntax can be embedded into the outer syntax using the form `text< ... >` and L^AT_EX sections can be created using `chapter< ... >`, `section< ... >`, `subsection< ... >`, `subsubsection< ... >`, `paragraph< ... >`, `subparagraph< ... >`.

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

Comments

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.

Line breaks are ignored as part of the outer and inner syntax and have the same effect as a space.

2.1.2 Meta Level and Object Level

Isabelle consists of a small fixed kernel which is called “meta-logic”. It is implemented in the session `Pure/Pure` (see Section 1.1.2) which is the ancestor of all other sessions. To be useful the meta-logic must be extended by an “object logic”. It may consist of one or more sessions, all sessions other than

`Pure`/`Pure` are parts of an object logic. There are many object logics available for Isabelle, the most versatile is HOL. Although called a “logic” it is a full extensible representation of mathematics.

This introduction first describes the Isabelle kernel features, followed by a description of the object logic HOL in Chapters 6 and subsequent chapters.

Both outer and inner syntax consist of a part for the kernel (the “meta-level” of the languages) and a part introduced by and specific for the object logic (the “object-level” of the languages). While the meta-level of the inner syntax is extremely small, mainly consisting of only four logical operators, the meta-level of the outer syntax supports a large set of constructs for specifying entities like axioms, definitions, proofs, and, finally, whole theories.

This introduction describes the meta-level of the inner syntax mainly in Sections 2.2 and 2.5. The rest of the sections about the Isabelle kernel describe the meta-level of the outer syntax. Chapter 6 describes basic parts of the object level of the inner and outer syntax for HOL. Chapter 7 describes a major part of the outer syntax for HOL, whereas Chapter 8 describes an important part of the inner syntax for HOL.

2.1.3 Theory Structure

The content of a theory file has the outer syntax structure

```
theory name
imports name1 ... namen
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.

2.2 Terms and Types

The two main constituents of the inner syntax are terms and 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 which names the type of values denoted by the term. However, in most situations Isabelle can derive the type of a term automatically, so that it needs not be specified explicitly.

2.2.1 Types

Types are basically specified by type names. In Isabelle HOL (see Chapter 6) there are predefined types such as `nat` and `bool` for natural numbers and boolean values. With the exception of function types, types like these with a mathematical meaning always belong to an object logic. Chapter 8 gives a detailed description of several important types of HOL. Due to the lack of adequate types in the meta-logic this introduction uses a small set of HOL types for examples to illustrate concepts on the meta-level, assuming an intuitive understanding of the associated operations and terms.

Type Declarations

New type names can be introduced using the outer syntax construct

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 may be done separately. See Chapter 7 for ways of defining types with specifying more information about their values.

Parameterized Types and Polymorphic Types

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

typedecl (*'name₁, ..., 'name_n*) *name*

where the parentheses may be omitted if *n* = 1, such as in **typedecl** *'a set*. The type parameters are denoted by “type variables” which always have the form *'name* with a leading single quote character.

A type name with parameters is called a “type constructor” because it is not a type on its own. Every use where the parameters are replaced by actual types, such as in `nat set`, is called an “instance” of the parameterized type. Parameters of a type constructor may again be replaced by parameterized types, such as in `('a set) set`. In this way arbitrary complex “type expressions” can be built, consisting of type constructors, type names, type variables, and parentheses. Generally, types in Isabelle may be specified by arbitrary type expressions.

If a type expression contains type variables, such as in *'a set* or if it consists of a single type variable such as *'a* the denoted type is called “polymorphic”.

A polymorphic type can be used as a type specification, its meaning is that an arbitrary instance can be used where the type variables are replaced by actual types.

Type Synonyms

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 ('name1, ..., 'namen) name = type
```

where *type* may be a polymorphic type which contains atmost the type variables *'name*₁, ..., *'name*_{*n*}.

2.2.2 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. In HOL 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*. 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.

Constants Declarations

A new constant name can be introduced by specifying its type. The outer syntax construct

```
consts name1 :: type1 ... namen :: typen
```

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

If the constant's type is polymorphic (see Section 2.2.1) the constant is also called polymorphic. Thus the declaration

```
consts myset :: "'a set"
```

declares the polymorphic constant *myset* which may be a set of elements of arbitrary type. Note the use of quotes because the type is specified in inner syntax and is not a single type name.

Variables

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 inner syntax in the form *varname* :: *type*.

2.2.3 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 in inner syntax as *argtype* \Rightarrow *restype* or equivalently as (*argtype*, *restype*) *fun* (thus *fun* is a type constructor with two arguments). These ways of denoting function types belongs to the meta-level of the inner syntax and is thus available in all object logics.

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  $\Rightarrow$  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.

2.2.4 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 functions which can be applied to *n* arguments.

Function Application

Function application terms for a function f and an argument a are denoted in inner syntax 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 $f\ a_1 \dots a_n$. Note that an application $f\ a_1 \dots a_m$ where $m < n$ (a “partial application”) is a correct term and denotes a function taking the remaining $n-m$ arguments.

Infix Function Application

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 in HOL is *plus*, so an application term is denoted in the form *plus* 3 5. For *plus* the alternative name $(+)$ is defined (the parentheses are part of the name). For functions with such “operator names” an application term $(+)\ 3\ 5$ can also be denoted in infix form $3 + 5$. Infix notation is supported for many basic functions and predicates in HOL, having operator names such as $(-)$, $(**)$, $(=)$, (\neq) , (\leq) , or (\in) .

2.2.5 Lambda-Terms

Functions can be denoted in inner syntax by lambda terms of the form $\lambda x. \text{term}$ where x is a variable which may occur in the *term*. The space between the dot and the *term* is often required to separate both. A function to be applied to n arguments can be denoted by the lambda term $\lambda x_1 \dots x_n. \text{term}$ where x_1, \dots, x_n are distinct variables. As usual, types may be specified for (some of) the variables in the form $\lambda(x_1::t_1) \dots (x_n::t_n). \text{term}$. The parentheses may be omitted if there is only one argument variable.

A constant function has a value which does not depend on the argument, thus the variable x does not occur in the *term*. Then its name is irrelevant and it may be replaced by the “wildcard” $_$ (an underscore) as in $\lambda_. \text{term}$.

Binding Variables

If a variable from the x_1, \dots, x_n occurs in the *term* of $\lambda x_1 \dots x_n. \text{term}$ it is called a “bound” occurrence and denotes the corresponding function argument. If an occurrence of a variable x is not a part of a lambda term $\lambda \dots x \dots . \text{term}$ the occurrence is called “free”.

A lambda term is a case of “binder syntax”. It consists of a “binder” (here λ) followed by one or more variables with optional type specifications, followed by a dot and a term (called “body”). Terms of the inner syntax nearly always

have either the form of a function application, possibly in infix notation, or the form of a binder syntax.

2.2.6 Searching Constants

Constants may be searched using the command

find_consts *criterion*₁ ... *criterion*_{*n*}

or using the Query panel (see Section 1.2.7) in the “Find Constants” tab using the sequence *criterion*₁ ... *criterion*_{*n*} as search specification in the “Find:” input field. The *criterion*_{*i*} are combined by conjunction.

The command **find_consts** may be entered in the text area between other theory content such as type or constant declarations. It finds all named constants which have been introduced before the command position. Searches using the Query panel find all named constants which have been introduced before the cursor position in the text area.

Searching by Type

A *criterion*_{*i*} may be a type, specified in inner syntax and quoted if not a single type name. Then the search finds all constants where the type occurs as a part of the constant’s type. For example, it finds all functions which have the specified type as argument or result type.

A *criterion*_{*i*} may also have the form *strict*: “*type*”, then the search only finds constants which have that type. In both cases the specified type may be a function type, then the search finds corresponding named functions.

If the specified type is polymorphic the search will also find constants which have an instance of it as their type or as a part of the type, respectively.

Searching by Name

A *criterion*_{*i*} may also have the form *name*: *strpat* where *strpat* is a string pattern which may use “*” as wildcard (then the pattern must be enclosed in double quotes). Then all constants are found where the *strpat* matches a substring of their name.

2.3 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. Both are constructs of the outer syntax.

2.3.1 Definitions

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

definition *name* **where** "*name* \equiv *term*"

Note that the “defining equation” *name* \equiv *term* is specified in inner syntax and must be delimited by quotes. The operator \equiv is the equality operator of the meta-logic (see Section 2.1.2). The *name* may not occur in the *term*, i.e., this form of definition does not support recursion. Also, no free variables may occur in the *term*. In the object logic HOL (see Chapter 6) also the normal equality operator = may be used instead of \equiv .

The type of the defined constant is the same as that of the *term*. If that type is polymorphic (see Section 2.2.1) a more specific type may be specified explicitly in the form

definition *name* **::** *type* **where** "*name* \equiv *term*"

As usual the type is specified in inner syntax and must be quoted if it is not a single type name.

A short form of a definition is

definition "*name* \equiv *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* **where** "*mystery2* \equiv *mystery*"

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

Defining Functions

If the type of the defined constant 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* **where** "*name* $x_1 \dots x_n \equiv$ *term*"

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

definition *name* **where** "*name* $\equiv \lambda x_1 \dots x_n. \textit{term}$ "

2.3.2 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* **where** "*name* \equiv *term*"

As for definitions, recursion is not supported, the *name* may not occur in the *term* and also no free variables. An explicit type may be specified for *name* and the short form is also available as for definitions.

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

abbreviation *name* **where** "*name* $x_1 \dots x_n \equiv$ *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.

2.4 Overloading

2.4.1 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 2.2.2). Such a constant *name* is associated with *n* definitions by the following overloading specification:

```
overloading
  name1  $\equiv$  name
  ...
  namen  $\equiv$  name
begin
  definition name1 :: type1 where ...
  ...
  definition namen :: typen where ...
end
```


where all type_i must be instances of the type declared for name .

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

2.4.2 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.

A constant name name can be defined to be a “type dependent abbreviation” for n terms of different type instances by

adhoc_overloading $\text{name} \rightleftharpoons \text{term}_1 \dots \text{term}_n$

The syntactic operator \rightleftharpoons is available for input in the editor’s Symbols panel in tab “Arrow”. It means that translation between both sides occurs upon input and output. 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 $\text{name}_1 \rightleftharpoons \text{term}_{1,1} \dots \text{term}_{1,n}$ **and** \dots **and** $\text{name}_k \rightleftharpoons \dots$

2.5 Propositions

A proposition denotes an assertion, which can be valid or not. Valid propositions 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.

2.5.1 Formulas

A simple form of a proposition is a single term of type **bool**, such as

$6 * 7 = 42$

The $*$ is the infix operator for multiplication, it may not be omitted in arithmetic terms.

Terms of type `bool` are also called “formulas”. Since `bool` belongs to the object logic HOL, formulas are also specific for HOL or another object logic, there are no formulas in the meta-logic. The simplest form of a proposition on meta-level is a single variable.

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.

2.5.2 Derivation Rules

More complex propositions on the meta-level can express “derivation rules” used to derive propositions from other propositions. Derivation rules are denoted using the meta-logic operator \implies and can thus be expressed independent of an object logic.

Derivation rules consist of assumptions and a conclusion. They are written in the form

$$A_1 \implies \dots \implies A_n \implies C$$

where the $A_1 \dots A_n$ are the assumptions and C is the conclusion. Since \implies is right-associative the conclusion can be assumed to be a single variable or a formula. The assumptions may be arbitrary propositions. If an assumption contains meta-logic 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 \implies 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

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

Note that type `nat` is explicitly specified for variable `x`. This is necessary, because the constants `<`, `*`, and `≤` 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

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

In most cases the assumptions are also formulas, 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.

2.5.3 Binding Free Variables

A proposition may contain universally bound variables, using the meta-logic quantifier \bigwedge in the form

$$\bigwedge x_1 \dots x_n. P$$

where the $x_1 \dots x_n$ may occur free in the proposition P . This is another case of binder syntax (see Section 2.2.5). As usual, types may be specified for (some of) the variables in the form $\bigwedge (x_1::\tau_1) \dots (x_n::\tau_n). P$. An example for a valid derivation rule with bound variables is

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

2.5.4 Rules with Multiple Conclusions

A derivation rule may specify several propositions to be derivable from the same assumptions using the meta-logic operator $\&\&\&$ in the form

$$A_1 \implies \dots \implies A_n \implies C_1 \ \&\&\& \dots \ \&\&\& C_h$$

Here the C_i may be arbitrary propositions, like the assumptions. If they contain meta-logic operators they must be enclosed in parentheses because $\&\&\&$ binds stronger than the other meta-logic operators.

The rule is equivalent to the set of rules deriving each C_i separately:

$$\begin{array}{l} A_1 \implies \dots \implies A_n \implies C_1 \\ \dots \\ A_1 \implies \dots \implies A_n \implies C_h \end{array}$$

The rule states that if the assumptions are valid then all conclusions are valid. So $\&\&\&$ can be viewed as a “meta conjunction” with a similar meaning as a boolean conjunction, but with a different use.

An example for a rule with two conclusions is

$$(x::nat) < c \implies n*x \leq n*c \ \&\&\& \ n+x < n+c$$

2.5.5 Alternative Rule Syntax

An alternative, Isabelle specific syntax for derivation rules with possibly multiple conclusions is

$$\bigwedge x_1 \dots x_m. [A_1; \dots; A_n] \implies C_1 \ \&\&\& \dots \ \&\&\& C_h$$

which is often considered as more readable, because it better separates the assumptions from the conclusions. In the interactive editor to switch to this form it may be necessary to set `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

$$\bigwedge (x::nat) \ c \ n. \ \llbracket x < c; \ n > 0 \rrbracket \Longrightarrow n*x < n*c$$

or equivalently without quantifier by

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

Note that in the literature a derivation rule $\llbracket P; Q \rrbracket \Longrightarrow P \wedge Q$ is often denoted in the form

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

2.5.6 Structured Rule Syntax

Isabelle supports another alternative syntax for derivation rules with possibly multiple conclusions. It is called "structured" form, since the rule is not specified by a single proposition but by several separate propositions for the parts of the rule:

$$"C_1" \ \dots \ "C_h" \ \text{if} \ "A_1" \ \dots \ "A_n" \ \text{for} \ x_1 \ \dots \ x_m$$

Here the conclusions, assumptions and the variables may be grouped or separated for better readability by the keyword **and**. For every group of variables (but not for single variables in a group) a type may be specified in the form $x_1 \ \dots \ x_m :: \text{"type"}$, it applies to all variables in the group.

The keywords **if**, **and**, **for** belong to the outer syntax. Thus, a rule in structured form cannot occur nested in another proposition, such as an assumption in another rule. Moreover, the original rule must be quoted as a whole, whereas in the structured form only the sub-propositions $C_1 \ \dots \ C_h$, A_1 , \dots , A_n must be individually quoted. The x_1 , \dots , x_m need not be quoted, but if a type is specified for a variable group 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 section may become

$$"n*x < n*c" \ \text{if} \ "x < c" \ \text{and} \ "n > 0" \ \text{for} \ x::nat \ \text{and} \ n \ c$$

and the rule with two conclusions depicted earlier may become

" $n * x \leq n * c$ " and " $n + x < n + c$ " if " $x < c$ " for $x :: \text{nat}$ and $n \ c$

The assumptions and the conclusion in a rule in structured form may be arbitrary propositions, in particular, they may be derivation rules (in unstructured form). If a conclusion is a derivation rule the assumptions A_1, \dots, A_n are added to the assumptions present in the conclusion.

2.5.7 Conditional Definitions

A definition, as described in Section 2.3.1 may be conditional, then its defining equation has the form of a derivation rule

definition *name* **where** " $\llbracket A_1; \dots; A_n \rrbracket \implies \text{name} \equiv \text{term}$ "

or in structured form:

definition *name* **where** " $\text{name} \equiv \text{term}$ " **if** " A_1 " ... " A_n "

Since no free variables are allowed in *term* it is not possible to bind variables using \wedge . The meaning of a conditional definition is that the value of *name* is only defined by the *term* if all assumptions are valid. Otherwise it is underspecified.

For the rule conclusion also the form $\text{name } x_1 \dots x_n \equiv \text{term}$ can be used if *name* has a function type. Then the $x_1 \dots x_n$ may occur in the A_1, \dots, A_n and restrict the specification of function values to specific function arguments.

As for normal definitions a type may be specified for *name* and the short form may be used where only the defining rule is given. For the abbreviations described in Section 2.3.2 a conditional form is not available.

2.6 Theorems

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

2.6.1 Specifying Theorems

A simple form of a theorem is

theorem "*prop*" $\langle \text{proof} \rangle$

where *prop* is a proposition in inner syntax and $\langle \text{proof} \rangle$ is a proof as described in Chapter 3. The keyword **theorem** can be replaced by one of the keywords **lemma**, **corollary**, **proposition** to give a hint about the use of the theorem to the reader.

The example rule from the previous sections can be stated as a fact by the theorem

theorem " $\wedge (x::nat) \ c \ n \ . \ x < c \implies n*x \leq n*c$ " $\langle proof \rangle$

Implicit Variable Bindings

If the proposition in a theorem contains free variables they are implicitly universally bound. Thus the previous example theorem is equivalent to the theorem

theorem " $(x::nat) < c \implies n*x \leq n*c$ " $\langle proof \rangle$

Explicit binding of variables is only required to avoid name clashes with constants of the same name. In the theorem

theorem " $\wedge (True::nat) . \ True < c \implies n*True \leq n*c$ " $\langle proof \rangle$

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.

Explicit binding of variables can also be useful to specify the variable types at a common place in the proposition instead of scattering them over it. Finally, explicit variable binding makes a difference for the $\langle proof \rangle$: it restricts the variables to the proposition so that they are not available in the $\langle proof \rangle$ (see Sections 3.1.3 and 3.9).

Structured Form

If the proposition in a theorem is a derivation rule with possibly multiple conclusions it may also be specified in structured form (see Section 2.5.6):

theorem " $C_1 \ \dots \ C_h$ " **if** " $A_1 \ \dots \ A_n$ " **for** $x_1 \ \dots \ x_m$ $\langle proof \rangle$

with optional grouping of all components by **and**. Remember that the C_i may be arbitrary propositions, therefore a theorem in this form may specify several derivation rules with additional common assumptions and common bound variables. As for the unstructured form the explicit binding of variables in the **for** part is optional. If a theorem has free variables in a C_i or in a A_i a **for** part is automatically added for all such variables.

Multiple Conclusions

A theorem with multiple conclusions adds a separate fact for every conclusion to the enclosing theory (copying all assumptions to every such fact), as if there had been a separate theorem for every conclusion.

In particular, if no variables are explicitly bound and no common assumptions are present this form allows to specify several propositions in a common theorem of the form

theorem " C_1 " ... " C_h " $\langle proof \rangle$

where all propositions can be proved together in a single proof.

The example rules from the previous sections can be stated as facts by a common theorem in the form

theorem " $(x : \text{nat}) < c \implies n * x \leq n * c$ "
and " $(x : \text{nat}) \leq c \implies x + m \leq c + m$ "
 $\langle proof \rangle$

Although the resulting facts are completely independent of each other, the variables are common to both propositions. This means that it suffices to specify the type `nat` for `x` in one of them. If different types are specified for `x` in the two propositions an error is signaled.

2.6.2 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.

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.

When turned to a fact, the rule used in the example theorems becomes

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

with type `nat` associated to all unknowns.

Propositions specified in a theorem may not contain unknowns, they are only introduced by Isabelle after proving the proposition.

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

2.6.3 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*" \langle *proof* \rangle

The names used for facts have the same form as names for constants and variables (see Section 2.2.2). The same name can be used for a variable and a fact, they can always be distinguished by the usage context.

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

theorem *example1*: " $(x::nat) < c \implies n*x \leq n*c$ " \langle *proof* \rangle

Named Fact Collections

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*: " $(x::nat) \leq c \implies x + m \leq c + m$ " \langle *proof* \rangle

a named collection can be introduced by

lemmas *examples* = *example1 example2*

If a theorem with multiple conclusions is named in the form

theorem *name*: "*C*₁" ... "*C*_h" **if** "*A*₁" ... "*A*_n" **for** *x*₁ ... *x*_m \langle *proof* \rangle

it introduces the name for the collection of all resulting facts. Moreover, if the conclusions are grouped by **and**, (some of) the groups may be named separately in the form

theorem *name*₁: "*C*_{1,1}" ... "*C*_{1,g₁}" **and** ... **and** *name*_h: "*C*_{h,1}" ... "*C*_{h,g_h}" **if** "*A*₁" ... "*A*_n" **for** *x*₁ ... *x*_m \langle *proof* \rangle

which introduces the names for the corresponding collections of facts according to the groups.

In this way the two example facts may be specified and named by the common theorem

theorem *example1*: " $(x::nat) < c \implies n*x \leq n*c$ "
and *example2*: " $(x::nat) \leq c \implies x + m \leq c + m$ "
 \langle *proof* \rangle

Dynamic Fact Sets

As an alternative to introducing fact names in theorems 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 namef [name]: "prop" <proof>
```

There are also some predefined “internal fact sets”. For them the name can only be used to add facts as described above, the set cannot be used or displayed by referring it by name. Examples are the internal fact sets *intro* (see Section 4.3.4) and *simp* (see Section 4.6.3).

2.6.4 Alternative Theorem Syntax

If the proposition of a theorem is a derivation rule with possibly multiple conclusions Isabelle supports an alternative structured form for it:

```
theorem  
  fixes x1 ... xm  
  assumes "A1" ... "An"  
  shows "C1" ... "Ch"  
  <proof>
```

As for the general structured form (see Section 2.5.6) the variables, assumptions, and conclusions may be grouped by **and**, a type may be specified for each variable group, the keywords belong to the outer syntax and the *C_i* and *A_i* must be individually quoted. Moreover, the explicit binding of variables in the **fixes** part is optional, as in a **for** part.

Note that this structured form may only be used if a derivation rule is specified in a theorem.

Using this syntax the two-assumption example rule from Section 2.5 can be written as

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

As for the general structured form of a theorem (some of) the conclusion groups may be named individually which introduces the names for the corresponding fact collections. A possibly additional name specified after the **theorem** keyword names the collection of the resulting facts from all groups together:

```
theorem name:
  fixes  $x_1 \dots x_m$ 
  assumes " $A_1$ " ... " $A_n$ "
  shows  $name_1$ : " $C_{1,1}$ " ... " $C_{1,g_1}$ " and ... and  $name_h$ : " $C_{h,1}$ " ... " $C_{h,g_h}$ "
  <proof>
```

2.6.5 Definitions as Facts

The definitions described in Section 2.3.1 also introduce facts in the enclosing theory. Every definition introduces a new constant and specifies a defining equation of the form $name \equiv term$ for it. This equation is a proposition. 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: " $name \equiv term$ "
```

Although the auxiliary constants used in an **overloading** specification (see Section 2.4.1) 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.

2.6.6 Displaying and Searching Named Facts

Fact Display

A named fact or fact set (but not a dynamic fact set) can be displayed in its standard form as proposition using the command

```
thm name
```

and it can be displayed in its structured form with **fixes**, **assumes**, and **shows** using the command

```
print_statement name
```

In the interactive editor a list of the named facts introduced by an outer syntax construct such as **theorem** can be obtained by positioning the cursor after the construct and using the Query panel (see Section 1.2.7) in tab “Print Context” by checking “theorems”. For theorems the introduced facts are usually apparent, but there are constructs in HOL, such as inductive or recursive definitions (see Sections 6.7 and 6.10), where it may be interesting to look at the introduced facts.

Fact Search

Named facts may be searched using the command

find_theorems *criterion*₁ ... *criterion*_{*n*}

or using the Query panel (see Section 1.2.7) in the “Find Theorems” tab using the sequence *criterion*₁ ... *criterion*_{*n*} as search specification in the “Find:” input field. The *criterion*_{*i*} are combined by conjunction.

A *criterion*_{*i*} may be a term containing unknowns as subterms (called a “term pattern”). Then all facts are found which contain a matching subterm in their proposition. A term pattern matches a subterm if the unknowns in the pattern can be consistently replaced by terms so that the result is syntactically equal to the subterm. The term pattern is specified in inner syntax and must be quoted. Only named facts can be found in this way.

The example theorems *example1* and *example2* can be found using the term pattern “*?t1 ≤ ?t2*”, whereas the pattern “*?t1 + ?c ≤ ?t2 + ?c*” will only find *example2*.

A *criterion*_{*i*} may also have the form *name: strpat* where *strpat* is a string pattern which may use “*” as wildcard (then the pattern must be enclosed in double quotes). Then all facts are found where the *strpat* matches a substring of the fact name. After naming the example theorems as above the criterion *name: example* will display the theorems *example1* and *example2* with their names and propositions.

The commands for display and search may be entered in the text area outside of theorems and at most positions in a proof. The found facts are displayed with their names in the Output panel (see Section 1.2.3).

Chapter 3

Isabelle Proofs

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

3.1 Maintaining Proof State

Usually it is necessary during a proof to collect information for later use in the proof. For every proof such state is maintained in two structures: the “proof context” and the “goal state”. At the end of a proof all proof state is disposed, only the proved fact remains in the enclosing environment.

3.1.1 Proof Context

The proof context is similar to a temporary theory which collects facts and other proof elements. It 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. As in a theory facts and fact sets may be named in a 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 in the same proof context. They can be roughly compared to the constants in a theory.
- Term abbreviations: these are names introduced locally for terms. They can be roughly compared to abbreviations defined in a theory. Using such names for terms occurring in propositions it is often possible to denote propositions in a more concise form.

As in a theory the names for facts and fixed variables have the same form, they can always be distinguished by their usage context. The names for term abbreviations have the form of unknowns (see Section 2.6.2) and are thus always different from variable names.

Since the proof context is usually populated by explicitly specifying its elements it is visible in the proof text and also in the session document. In the interactive editor a list of all elements of the proof context at the cursor position can be obtained in the Query panel (see Section 1.2.7) in tab “Print Context” by checking “terms” (for term abbreviations) and/or “theorems” (for facts).

3.1.2 Goal State

The goal state is used to collect propositions which have not yet been proved. It is used in the form of a “to-do list”. It is the duty of a proof to prove all goals in its goal state. During the proof goals may be removed from the goal state or may be added. A proof may be terminated when its goal state is empty.

The content of the goal state is not maintained by explicit specifications of the proof writer, it is updated implicitly by the Isabelle proof mechanism. As a consequence it is usually not visible in the session document. In the interactive editor it is displayed in the State panel (see Section 1.2.4) or in the Output panel (see Section 1.2.3).

If a subproof is nested in another proof the goal state of the inner proof hides the goal state of the outer proof until the inner proof is complete.

3.1.3 Initial Proof State

The initial proof state in a theorem of the form

theorem " $\bigwedge x_1 \dots x_m. [A_1; \dots; A_n] \implies C_1 \ \&\&\& \dots \ \&\&\& C_h$ " *<proof>*

has the proposition $\bigwedge x_1 \dots x_m. [A_1; \dots; A_n] \implies C_1 \ \&\&\& \dots \ \&\&\& C_h$ as the only goal in the goal state and an empty proof context. In particular, the explicitly bound variables $x_1 \dots x_m$ are not added as fixed variables to the proof state.

If the proposition of a theorem is specified in structured form

theorem "*C*" **if** "*A*₁" **...** "*A*_{*n*}" **for** $x_1 \dots x_m$ *<proof>*

or

theorem

fixes $x_1 \dots x_m$ **assumes** "*A*₁" **...** "*A*_{*n*}" **shows** "*C*" *<proof>*

the initial goal state only contains the conclusion C , whereas the initial proof context contains the assumptions A_1, \dots, A_n as (assumed) facts and the variables $x_1 \dots x_m$ as fixed variables.

Since variables occurring free in an A_i or in C (i.e., not being explicitly bound by \wedge) are automatically by a **for** part (see Section 2.6.1), they will also be added to the initial proof state as fixed variables.

If the theorem has multiple conclusions such as

theorem " C_1 " ... " C_h " **if** " A_1 " ... " A_n " **for** $x_1 \dots x_m$ $\langle proof \rangle$

the initial goal state contains the single conclusion $C_1 \&\&\& \dots \&\&\& C_h$, i.e., the “meta conjunction” of the separate conclusions. This will be split into separate goals for the individual conclusions upon the first application of a proof method (see Chapter 4).

Assumption Naming

Both structured forms support naming the assumptions in the proof context. Every assumption group separated by **and** may be given a name, i.e., the assumptions may be specified in the form

if $name_1$: " $A_{1,1}$ " ... " A_{1,m_1} " **and** ... **and** $name_n$: " $A_{n,1}$ " ... " A_{n,m_n} "

or

assumes $name_1$: " $A_{1,1}$ " ... " A_{1,m_1} " **and** ... **and** $name_n$: " $A_{n,1}$ " ... " A_{n,m_n} "

respectively, in the same way as the conclusion groups may be named. However, the assumption names are only valid in the proof context, whereas the conclusion names are only valid outside of the proof context after the proof is complete.

Additionally, Isabelle always automatically names the assumptions in all groups together. For the structured form beginning with **if** it uses the name *that*, for the structured form beginning with **assumes** it uses the name *assms*. Since the assumption names are only defined in the proof context they can only be used locally in the proof of the theorem. Therefore, if the general structured form of a proposition beginning with **if** is used in a context where no proof is required, such as in an **assume** statement (see Section 3.8.1), it is not possible to specify names for the assumption groups.

3.2 Proof Procedure

Assume you want to prove a derivation rule $A \implies C$ with a single assumption A and a single conclusion C . There are several different ways how to proceed.

3.2.1 Constructing a Proof

The basic procedure to build a proof for it is to construct a sequence of the form $F_1 \implies F_2, F_2 \implies F_3, F_3 \implies \dots \implies F_{n-1}, F_{n-1} \implies F_n$ where F_1 matches with A and F_n matches with C (for the exact meaning of “matches” see Section 3.2.2 below). The sequence is constructed from derivation rules R_i for $i=1..n-1$ which are already known to be valid (i.e., facts). Every rule R_i has the form $RA_i \implies RC_i$ where RA_i denotes the assumption of the rule and RC_i denotes its conclusion. Every assumption RA_i matches with F_i in the sequence and every conclusion RC_i matches with F_{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 $(x::nat) < 5 \implies 2*x+3 \leq 2*5+3$. A proof can be constructed from the two example rules *example1* and *example2* from the previous sections as the sequence $(x::nat) < 5 \implies 2*x \leq 2*5, 2*x \leq 2*5 \implies 2*x+3 \leq 2*5+3$ consisting of three facts.

Forward Reasoning

Forward reasoning starts by assuming A to be the local fact F_1 and incrementally constructs the sequence from it. An intermediate result is a part A, F_2, \dots, 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 $RA_i \implies RC_i$. The step results in the extended sequence $A, F_2, \dots, F_i, F_{i+1}$ with 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

Backward reasoning starts at the conclusion C and incrementally constructs the sequence from it backwards. An intermediate result is a part F_i, \dots, F_{n-1}, C of the sequence, here F_i is the “current goal”. A backward reasoning step consists of constructing a new current goal F_{i-1} and the extended sequence $F_{i-1}, F_i, \dots, F_{n-1}, C$ using a valid rule $RA_{i-1} \implies RC_{i-1}$. When a step produces a new current goal F_1 , which matches the assumption A , the proof is complete.

3.2.2 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 usually only the $RA_i \implies RC_i$ 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 $RA_i \implies RC_i$. In other words, the sequence is usually constructed from specializations of the facts $RA_i \implies RC_i$ 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 $(x::nat) < 5 \implies n*x \leq n*5$. 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 $2*x \leq 2*5 \implies 2*x+3 \leq 2*5+3$. 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.

3.2.3 Storing Facts During a Proof

In a proof for a derivation rule $A \implies C$ the assumption A , the conclusion C and the intermediate facts F_1, F_2, \dots, F_n constructed by the proof steps must be stored. There are mainly two ways how this can be done in an Isabelle proof.

Storing Facts in the Context

The first way is to store the facts at the beginning of the sequence in the proof context and the facts at the end of the sequence in the goal state. Initially, A is the only fact in the proof context and C is the only goal in the goal state. A forward reasoning step consists of adding fact F_{i+1} to the proof context and proving its validness using rule $RA_i \implies RC_i$. The goal state remains unchanged. A backward reasoning step consists of replacing the current goal F_i by the new current goal F_{i-1} in the goal state and proving that the new goal implies the previous one using rule $RA_{i-1} \implies RC_{i-1}$. The proof context remains unchanged. The proof is complete when the current goal matches (unifies with) a fact in the proof context.

Note that the facts in the proof context and in the goal state are treated differently. A backward step replaces the goal since the old goal needs not be handled again in the proof. Whenever the new goal has been proved the old goal is known to be valid as well. Since the goal state is used to determine when the proof is complete, it is crucial to remove all unnecessary

goals from it. A forward step, instead, adds a fact to the proof context. It could remove the previous facts, since they are not needed in the special case described here, however, there are proofs where facts are used more than once, therefore it is usually useful to keep them in the proof context, and it is irrelevant for detecting whether a proof is complete.

Storing Facts as Assumptions in the Goal State

The second way is to store all facts in the goal state by using a current goal of the form $\llbracket F_1; \dots; F_i \rrbracket \Rightarrow F_{i+j}$, i.e., a derivation rule. The proof context is not used at all. Initially, the goal state contains the goal $A \Rightarrow C$. A forward reasoning step consists of adding fact F_{i+1} as assumption to the current goal and proving its validness as above. A backward reasoning step consists of replacing the conclusion F_{i+j} of the current goal by F_{i+j-1} and proving that it implies F_{i+j} as above. The proof is complete when the conclusion of the current goal unifies with one of its assumptions.

Note that these two ways correspond to the initial proof states prepared by the different forms of theorems. The basic form **theorem** " $A \Rightarrow C$ " puts $A \Rightarrow C$ into the goal state and leaves the proof context empty, as required for the second way. The structured forms, such as **theorem** " C " if " A " put A into the proof context and C into the goal state, as required for the first way.

3.2.4 Multiple Assumptions

If the rule to be proved 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, \dots, A_n and merge to finally lead to the conclusion C . Two branches which end in facts $F_{1,n}$ and $F_{2,m}$ are joined by a step $\llbracket F_{1,n}; F_{2,m} \rrbracket \Rightarrow F_1$ to a common branch which continues from fact F_1 .

A proof which constructs this tree may again do this by forward reasoning (beginning at the branches), by backward reasoning (beginning at the common conclusion) or a mixture of both. It may use the proof context to store facts or it may use rules in the goal state, as described in the previous sections.

A forward reasoning step at a position where several branches join uses several current facts to prove a new current fact. 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 which join at the current position in the tree. In this situation a single goal in the goal state is replaced by several goals. If rules are used as goals the assumptions from the old goal must be copied to all

new goals. Facts stored in the proof context need not be copied since they are available for all goals. A proof is complete when it is complete for all goals in the goal state.

3.2.5 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 (if C is no tautology which is valid by itself).

3.3 Basic Proof Structure

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

3.3.1 Statements and Methods

There are two possible operations for modifying the proof state: statements and method applications.

A statement adds one or more elements to the proof context. In particular, a statement may “state a fact”, i.e., add a fact to the proof context, this is the reason for its name. A statement normally does not modify the goal state, there is one specific statement which may remove a goal from the goal state.

A method application modifies the goal state but normally leaves the proof context unchanged. The goal state is always modified so that, if all goals in the new state can be proved, then also all goals in the old state can be proved. This kind of goal state modification is also called a “refinement step”.

Proof Mode

When writing a proof the “proof mode” determines the kind of operation which may be written next: whether a statement (mode: `proof(state)`) or a method application (mode: `proof(prove)`) is admissible.

At the beginning of a proof the mode is always `proof(prove)`, i.e., a method application is expected. In the course of the proof it is possible to switch

to mode *proof(state)* for entering statements, but not back again. After switching to statement mode the proof must be completed without further modifications to the goal state other than removing goals, only at the end a last method may be applied.

However, for every statement that states a fact a (sub-)proof must be specified, which again starts in mode *proof(prove)*. This way it is possible to freely switch between both modes in the course of a proof with nested sub-proofs.

Choosing Between Statements and Methods

A backward reasoning step always modifies the goal state, therefore it must be expressed by a method application. A forward reasoning step may be expressed by a statement, if intermediate facts are stored in the proof context. If intermediate facts are stored as assumptions in rules in the goal state, forward reasoning steps must also be expressed by method applications.

This implies that a sequence of statements can only represent a proof by forward reasoning where intermediate facts are stored in the proof context, whereas a sequence of method applications can represent an arbitrary proof where all facts are stored using rules in the goal state.

As described in Section 3.1.1 statements have the advantage that the facts added to the proof context are explicitly specified by the proof writer and are visible in the session document. That makes it easier to write and read a proof which consists only of statements. Method applications specify an operation on the goal state by name, the resulting new goal state is determined by Isabelle. It is visible for the proof writer in the interactive editor, but it is not visible in the session document for a reader of the proof. Therefore proofs consisting of method applications are difficult to understand and the proof writer must anticipate the effect of a method on the goal state when writing a proof step.

3.3.2 Proof Syntax

If MA_i denote method applications and ST_i denote statements, the general form of a proof is

```
MA1 ... MAn
proof MAn+1
    ST1 ... STm
qed MAn+2
```

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

The part **proof** MA_{n+1} switches from method application mode *proof(prove)* to statement mode *proof(state)*.

Proof Scripts

The part **proof** ... **qed** can be omitted and replaced by **done**, then the proof only consists of method applications and has the form $MA_1 \dots MA_n$ **done**. Such proofs are called “proof scripts”.

Since a proof script does not contain statements it cannot use the proof context to store facts. Proof scripts are intended to store facts as assumptions in the goal state or to apply only backward reasoning, where no intermediate facts need to be stored in addition to the goals (see Section 3.2.3).

Structured Proofs

If the method applications $MA_1 \dots MA_n$ are omitted the proof only consists of the statements part and has the form

```
proof  $MA_1$   
       $ST_1 \dots ST_m$   
qed  $MA_2$ 
```

where MA_2 can also be omitted. Such proofs are called “structured proofs” and the syntactic elements used to write them are denoted as “Isar sublanguage” of the Isabelle outer syntax.

Since structured proofs consist nearly completely of statements, they are intended to use forward reasoning and store all assumptions and intermediate facts in the proof context.

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

```
by  $MA_1$   $MA_2$ 
```

abbreviates the form **proof** MA_1 **qed** MA_2 . Again, MA_2 can be omitted which leads to the form

```
by  $MA_1$ 
```

In this form the proof consists of a single method application which directly leads from the assumptions and used external facts to the conclusion C .

Use Proof Scripts or Structured Proofs?

As described in the previous section, a structured proof is usually easier to read and write than a proof script, 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 statement for a forward reasoning step again requires a proof as its part (a “subproof” for the stated fact), no proof can be written using statements alone. The main idea of writing “good” proofs is to use nested structured proofs until every subproof is simple enough to be done in a single method application, i.e., the applied method directly goes from the assumptions to the conclusion of the subproof. Such a simple proof can always be written in the form **by** *MA*.

3.3.3 Bogus Proofs

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 method application mode, so it can be used to abort a proof script in the form *MA*₁ ... *MA*_{*n*} **sorry**.

A structured proof in statement 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 for the sequence from the assumptions to the goal with fake subproofs and then replacing the fake proofs by actual subproofs.

Fake proofs are dangerous. Isabelle blindly registers the proposition as valid, so that it can be used for other proofs. If it is not valid, everything can be proved from it. That sounds nicely but is not what you really want.

Aborted Proofs

There is a second way to abort a proof script by specifying a proof as

oops

Other than by using **sorry** Isabelle will *not* turn the proposition to a fact, instead, it ignores it. This can be used to document in a theory that you have tried to prove a proposition but you did not succeed.

3.3.4 Nested Proof Contexts

Instead of a single proof context a proof may use a set of nested proof contexts, starting with an outermost proof context. 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 statements in braces:

$$ST_1 \dots ST_m \{ ST_{m+1} \dots ST_n \} ST_{n+1} \dots$$

Note that according to the description until now the nested context is useless, because the facts introduced by its statements 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 (of fixed variables, facts and term abbreviations), 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.

Sequencing Nested Contexts

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.

Automatic Nesting of Statements

Moreover the syntax **proof** *method*₁ $ST_1 \dots ST_n$ **qed** *method*₂ automatically wraps the statements $ST_1 \dots ST_n$ 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 method1
   $ST_{1,1} \dots ST_{1,m_1}$  next  $ST_{2,1} \dots ST_{2,m_2}$  next ... next  $ST_{n,1} \dots ST_{n,m_n}$ 
qed method $n+2$ 
```

where each occurrence of **next** clears the content of the inner context.

Another consequence of this wrapping is that no statement can add elements directly to the outermost proof context. The outermost proof context can only be filled by the initializations done by the structured theorem forms as described in Section 3.1.3. The resulting content of the context is not

affected by clearing nested contexts and remains present until the end of the proof.

Also the goal state of a proof is not affected by the begin or end of a nested context. The goal state can be considered to be in the scope of the outermost context, it may use fixed variables from it. However, it is outside of all nested contexts and cannot contain elements from them.

3.3.5 Subproofs

A subproof takes a single goal and solves it as part of another proof. It has its own goal state which hides the goal state of the enclosing proof until the subproof is complete.

The outermost proof context used by the subproof is nested in the context of the enclosing proof, 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.

Kinds of Subproofs

There are mainly two kinds of subproofs: specifying a new goal which becomes a fact in the proof context after proving it, or proving a goal which is already in the goal state of the enclosing proof.

The first kind of subproof occurs as part of some statements, as described in Section 3.5.

The second kind of subproof has the form

subgoal $\langle proof \rangle$

and may be specified in method application mode *proof*(*prove*) in place of a single method application. Its initial goal state contains a copy of the first goal from the goal state of the enclosing proof. If the subproof is successfully terminated it removes that goal from the goal state of the enclosing proof.

Structured Subproofs

There is also the “structured form”

subgoal *premises name* $\langle proof \rangle$

If the goal processed by the subproof is a derivation rule $\llbracket A_1; \dots; A_n \rrbracket \Rightarrow C$ it takes the assumptions A_1, \dots, A_n , and adds them as assumed facts into the outermost proof context of the subproof, like the structured forms of theorems do for their assumptions (see Section 3.1.3). If *name* is specified it

is used for naming the set of assumptions, if it is omitted the default name *prems* is used.

Using this form a subproof can be written as a structured proof which stores its facts in its proof context, although the enclosing proof is a proof script and stores its facts as rule assumptions in the goal state.

3.4 Method Application

A method application mainly specifies a proof method to be applied.

3.4.1 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.

Isabelle supports a large number of proof methods. A selection of proof methods used in this manual is described in Chapter 4.

3.4.2 Method Application

A standalone method application step is denoted as

apply *method*

where *method* denotes the proof method to be applied, together with its arguments.

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

```
apply method1 ... apply methodn
proof methodn+1
  ST1 ... STm
qed methodn+2
```

where *ST*₁ ... *ST*_{*m*} are statements. The method *method*_{*n*+1} is called the “initial method” of the structured proof part.

3.5 Stating Facts

The most basic kind of statements is used to add a fact to the proof context.

3.5.1 Adding a Fact to the Proof Context

A fact is added to the (innermost enclosing) proof context by a statement of the form

have "*prop*" $\langle proof \rangle$

where *prop* is a proposition in inner syntax and $\langle proof \rangle$ 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*" $\langle proof \rangle$

The scope of the name is the innermost proof context enclosing the statement. In their scope named facts can be displayed and searched as described for theorems in Section 2.6.6.

As for a theorem, if the fact is a derivation rule with possibly multiple conclusions it may also be specified in structured form:

have "*C*₁" ... "*C*_{*h*}" **if** "*A*₁" ... "*A*_{*n*}" **for** *x*₁ ... *x*_{*m*} $\langle proof \rangle$

where the conclusions, assumptions and variables may be grouped by **and**, conclusion and assumption groups may be named, and a type may be specified for each variable group.

Note, however, that the structured form using **fixes**, **assumes**, and **shows** (see Section 2.6.4) is not available for stating facts in a proof.

The **have** statement is called a “goal statement”, because it states the proposition *prop* as a (local) goal which is then proved by the subproof $\langle proof \rangle$.

Note that the names given to facts by naming conclusion groups cannot be used to access them in the subproof, because they are only assigned after the proof has been finished, whereas names given to assumption groups can only be used in the subproof because their scope is the proof context of the subproof.

3.5.2 Proving a Goal

A proof using only forward reasoning ends, if the last stated fact F_n unifies with the conclusion *C*. If the facts are stored in the proof context, F_n will be added by a statement. Therefore a special form of stating a fact exists, which, after proving the fact, tries to unify it with a goal in the goal state of the enclosing proof, and, if successful, removes the goal from that goal state. This is done by the statement

show "*prop*" $\langle proof \rangle$

which is the only statement which may affect the goal state of the enclosing proof. Its syntax is the same as for **have**, including naming and structured form. Like **have** it is also called a “goal statement”.

Exporting the Fact

As described in Section 3.3.4, statements are always wrapped by a nested proof context. When the **show** statement tries to unify its fact with a goal from the goal state it replaces all variables fixed in an enclosing nested proof context by unknowns (which is called “exporting the fact” from the proof context) so that they can match arbitrary subterms (of the correct type) in the goal.

If the unification of the exported fact with some goal is not successful the step is erroneous and the proof cannot be continued, in the interactive editor an error message is displayed.

If a goal has the form of a derivation rule, the exported fact is only unified with the conclusion of the goal. If also the exported fact is a derivation rule, additionally each of its assumptions must unify with an assumption of the goal.

This is a special case of a refinement step in the sense of Section 3.3.1. Whenever the exported fact can be proved, also the matching goal can be proved. Since the exported fact has just been proved by the subproof, the matching goal has been proved as well and may be removed from the enclosing goal state. So the condition for a successful **show** statement can be stated as “the exported fact must refine a goal” (this term is used in error messages).

To test whether a proposition refines a goal in the enclosing goal state, a **show** statement can be specified with a fake proof (see Section 3.3.3):

```
show "prop" sorry
```

If that statement is accepted, the proposition refines a goal and removes it. Do not forget to replace the fake proof by a genuine proof to make sure that the proposition is actually valid.

The term abbreviation *?thesis*

Note that in a proof using only forward reasoning the proposition *prop* in a **show** statement is the same proposition which has been specified as conclusion *C* in the proposition $\llbracket A_1; \dots; A_n \rrbracket \implies C$ which shall be proved by the proof. To avoid repeating it, Isabelle automatically provides the term abbreviation *?thesis* for it in the outermost proof context. So in the simplest case the last step of a structured proof by forward reasoning can be written as

show *?thesis* $\langle proof \rangle$

The abbreviation *?thesis* is a single identifier, therefore it needs not be quoted. If the proposition has multiple conclusions the abbreviation *?thesis* is not introduced.

If, however, the application of the initial method *method* in a structured proof **proof** *method* ... modifies the original goal, this modification is not reflected in *?thesis*. So a statement **show** *?thesis* $\langle proof \rangle$ will usually not work, because *?thesis* no more refines the modified goal. Instead, the proof writer must know the modified goal and specify it 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.

3.6 Facts as Proof Input

If a linear fact sequence F_1, \dots, F_n where every fact implies the next one is constructed in statement mode in the form

```
have "F1"  $\langle proof \rangle_1$ 
...
have "Fn"  $\langle proof \rangle_n$ 
```

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

```
have name1: "F1"  $\langle proof \rangle_1$ 
...
have namen: "Fn"  $\langle proof \rangle_n$ 
```

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.

3.6.1 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 the initial method *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 Chapter 4 for descriptions how methods process input facts.

3.6.2 Inputting Facts into a Proof

Method Application Mode

In method application mode *proof(prove)* facts can be input to the remaining proof $\langle proof \rangle$ by

using *name*₁ ... *name*_{*n*} $\langle proof \rangle$

where the *name*_{*i*} are names of facts or fact sets. The sequence 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 method application mode no local facts are stated by previous steps, the facts can only be initial facts in the outermost proof context (see Section 3.1.3), or they may be external facts from the enclosing theory, or, if in a subproof, they may be facts from contexts of enclosing proofs.

Statement Mode

In statement 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 goal 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 statement 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*" $\langle proof \rangle$

inputs the sequence of all facts referred by *name*₁ ... *name*_{*n*} to the $\langle proof \rangle$, 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 $name_1 \dots name_n$

Like **then** it switches to mode *proof(chain)* and it inputs the sequence of the facts referred by $name_1 \dots name_n$ to the proof of the following goal statement.

3.7 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”.

3.7.1 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 proved fact. Therefore the fact set *this* is also called the “current facts”.

Thus a linear sequence of facts where each fact implies the next one can be constructed by

```
have "F1" <proof>1
then have "F2" <proof>2
...
then have "Fn" <proof>n
```

Now in every *proof_i* the fact F_{i-1} is available as input and can be used to prove F_i .

In this way a structured proof for a rule $A \implies C$ can be written which uses a fact sequence $A, F_2, \dots, F_{n-1}, C$. If the theorem is specified in the structured form **theorem "C" if "A"** which adds the assumption to the proof context and names it *that* (see Section 3.1.3) the proof consists of the statement sequence

```
from that have "F2" <proof>1
then have "F3" <proof>2
...
then have "Fn-1" <proof>2
then show ?thesis <proof>n
```

For the structured form **theorem assumes "A" shows "C"** the assumption name *assms* must be used instead of *that*.

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 sequence 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 sequence 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 "F1,1" <proof>1,1
then have "F1,2" <proof>1,2
...
then have name1: "F1,m" <proof>1,m
have "F2,1" <proof>2,1
then have "F2,2" <proof>2,2
...
then have "F2,n" <proof>2,n
with name1 have "F" <proof>

```

If the theorem has been specified in structured form **theorem** "*C*" **if** "*A*₁" ... "*A*_{*n*}" every branch can be started in the form

```

from that have "F1,1"
...

```

which will input all assumptions to every branch. This works since unneeded assumptions usually do not harm in a proof, but it is often clearer for the reader to explicitly name the assumptions

theorem "*C*" **if** *a*₁: "*A*₁" **and** ... **and** *a*_{*n*}: "*A*_{*n*}"

and specify only the relevant assumption by name in the proof:

```

from a1 have "F1,1"
...

```

3.7.2 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 = name1 ... namen
```

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 possible to write

```
from name1 and ... and namen have "prop" <proof>
```

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

```
note name1 = name1,1 ... name1,m1 and ... and namen = namen,1 ... namen,mn
```

3.7.3 Accessing Input Facts in a Proof

At the beginning of a 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 other than by the first proof method 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*>.

3.7.4 Exporting the Current Facts of a Nested Context

At the end of a nested context (see Section 3.3.4) 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 existing 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  $f_1$ : "prop1" <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 ...

```

3.8 Assuming Facts

Facts can be added to the proof context without proving them, then they are only assumed.

3.8.1 Introducing Assumed Facts

A proposition is inserted as assumption in the proof context by a statement of the form

```

assume "prop"

```

Assumed facts may be derivation rules with possibly multiple conclusions, then they may be specified directly as proposition or in structured form

```

assume " $C_1$ " ... " $C_h$ " if " $A_1$ " ... " $A_n$ " for  $x_1$  ...  $x_m$ 

```

The conclusions C_1, \dots, C_h and the rule assumptions A_1, \dots, A_n may be grouped by **and**, however, names may only be specified for the conclusion groups in the usual way to name the assumed facts resulting from the statement. The rule assumptions cannot be named since there is no subproof where the names could be used. Note that variables occurring in the propositions $C_1, \dots, C_h, A_1, \dots, A_n$ are only turned to unknowns if they are explicitly bound in the **for** part, otherwise they refer to variables bound in an

enclosing proof context or remain free in the assumed rule (which is usually an error).

In their scope named facts introduced by an **assume** statement can be displayed and searched like other named facts (see Section 2.6.6).

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:

```
assume "C"
then have "prop" <proof>
...
```

3.8.2 Exporting Facts with Assumptions

Assumed facts may be used to prove other local facts, so that arbitrary local facts may depend on the validness of the assumed facts. This is taken into account when local facts are exported from a proof context (see Section 3.5.2). Whenever a local fact F is exported it is combined with copies of all locally assumed facts AF_1, \dots, AF_n to the derivation rule $\llbracket AF_1; \dots; AF_n \rrbracket \implies F$, so that F still depends on the assumptions after leaving the context.

If the fact F is itself a derivation rule $\llbracket A_1; \dots; A_n \rrbracket \implies C$ then the locally assumed facts are prepended, resulting in the exported rule $\llbracket AF_1; \dots; AF_n; A_1; \dots; A_n \rrbracket \implies C$.

If the fact F has been proved in a **show** statement it is also exported in this way, resulting in a derivation rule with all local assumptions added. Therefore it will only refine a goal if every local assumption unifies with an assumption present in the goal, (see Section 3.5.2).

If in a previous part of a proof facts have been stored as rule assumptions in the goal state (see Section 3.2.3), they can be “copied” to the proof context with the help of **assume** statements and will be “matched back” by the **show** statements used to remove the goals.

In particular, if a theorem specifies a rule $A \implies C$ directly as proposition it will become the initial goal, as described in Section 3.1.3. Then a structured proof using the fact sequence $A, F_2, \dots, F_{n-1}, C$ can be written

```
assume "A"
then have "F2" <proof>
...
then have "Fn-1" <proof>
then show ?thesis <proof>
```

The **show** statement will export the rule $A \implies C$ which matches and removes the goal.

3.8.3 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 proved from them. In other words, the proof is started somewhere in the middle of the fact tree, works by forward reasoning, and when it reaches the conclusion the assumed fact remains to be proved. The statement

presume "*prop*"

inserts such a presumed fact into the proof context. As for **assume** the structured form with **if** and **for** is supported.

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 $\llbracket A_1; \dots; A_n \rrbracket \implies F_p$ is added to the goal state of the enclosing proof where A_1, \dots, A_n are the facts assumed in the ended context. So the proof has to continue after proving all original goals and is only finished when all such goals for presumed facts have been proved as well.

3.9 Fixing Variables

As described in Section 3.1.3 variables occurring free in the assumptions or conclusions of a theorem are automatically added as fixed variables to the outermost proof context. Thus they can be used everywhere in the proof where they are not shadowed. If, instead, they are explicitly bound in the proposition (see Section 2.5.3), their use is restricted to the proposition itself. Thus in

theorem " $\bigwedge x::nat. x < 3 \implies x < 5$ " $\langle proof \rangle$

the variable x is restricted to the proposition and is not accessible in $\langle proof \rangle$, whereas in

theorem " $(x::nat) < 3 \implies x < 5$ " $\langle proof \rangle$

the variable x is accessible in $\langle proof \rangle$. If the theorem is specified in structured form, variables may be explicitly specified to be fixed in the outermost proof context using **for** or **fixes**, respectively (see Section 3.1.3). Therefore the forms

theorem " $x < 3 \implies x < 5$ " **for** $x::nat$ $\langle proof \rangle$

and

theorem **fixes** $x::nat$ **shows** " $x < 3 \implies x < 5$ " $\langle proof \rangle$

are completely equivalent to the previous form.

3.9.1 Local Variables

Additional variables can be added as fixed to the innermost enclosing proof context by the statement

```
fix  $x_1 \dots x_n$ 
```

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

A fixed variable in a proof context denotes a specific value of its type. However, if the variable has been introduced by **fix**, **for** or **fixes** it is underspecified in the same way as a constant introduced by **consts** in a theory (see Section 2.2.2): no information is given about its value. In this sense it denotes an “arbitrary but fixed value”.

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 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.

If a variable is free in the proof for a theorem its value cannot be proved to be related in any way to values used in the theorem. Therefore it is useless for the proof and its occurrence should be considered to be an error. In the interactive editor Isabelle marks free variables by a light red background as an alert for the proof writer.

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

```
fix  $x :: \text{nat}$   
assume  $a$ : " $x < 3$ "  
have " $x < 5$ "  $\langle proof \rangle$ 
```

the $\langle proof \rangle$ may refer to fact a because the x is the same variable in both propositions.

3.9.2 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 originates from an enclosing context or is free.

The proposition $x < 3 \implies x < 5$ can be proved 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 3.8.2) and then variable y is replaced by the unknown $?y$ because y has been locally fixed. The result is the rule $?y < 3 \implies ?y < 5$ 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 proved, but it cannot be used with the **show** statement to prove the proposition $x < 3 \implies x < 5$, because the exported rule is now $y < 3 \implies y < 5$ which does not unify with the proposition, it contains a different variable instead of an unknown.

If variables are explicitly bound in the proposition of a theorem they are not accessible in the proof. Instead, corresponding new local variables (which may have the same names) must be fixed in the proof context and used for the proof. Upon export by a **show** statement these variables will be replaced by unknowns which then are unified with the variables in the goal. A corresponding proof for the proposition $\bigwedge x :: \text{nat}. x < 3 \implies x < 5$ is

```

fix y :: nat
assume "y < 3"
then show "y < 5" <proof>

```

3.10 Defining Variables

Local variables may also be introduced together with specifying a value for them. This is done using a statement of the form

```

define x1 ... xm where "x1 ≡ term1" ... "xm ≡ termm"

```

which specifies a defining equation for every defined variable. As usual, the variables and equations may be grouped by **and**, equation groups may be named, and types may be specified for (some of) the variable groups. In the object logic HOL (see Chapter 6) also the normal equality operator $=$ may be used instead of \equiv .

The variable x_i may not occur free in term_i , otherwise an error is signaled. However, the other variables are allowed in term_i . This may lead to cyclic definitions which is not checked by Isabelle, but then the definition cannot be used to determine the defined value for the corresponding variables, it is underspecified.

3.10.1 Defining Equations Viewed as Local Facts

If there is already a previous definition for an x_i in the same proof context the former x_i is shadowed and becomes inaccessible and a new local variable x_i is introduced.

Thus a defining equation can never contradict any existing fact in the proof context and Isabelle safely adds all defining equations as facts to the proof context enclosing the **define** statement. The collection of all these facts is automatically named $x_1 \dots x_m_def$, replacing this name if it already exists in the local proof context. Explicit names specified for equation groups are used to name the corresponding facts.

3.10.2 Exporting Facts after Defining Variables

Unlike facts assumed by an **assume** statement (see Section 3.8.1) the facts created by a **define** statement are *not* added as assumptions when a fact F is exported from the local context. Instead, if a locally defined variable x_i occurs in F it is replaced by the $term_i$ according to its defining equation.

If the definition of x_i is cyclic the occurrences of x_i in F are not replaced and become an unknown during export, however, then the fact F usually is invalid and cannot be proved.

3.11 Obtaining Variables

Local variables may also be introduced together with an arbitrary proposition which allows to specify additional information about their values. This is done using a statement of the form

obtain $x_1 \dots x_m$ **where** "*prop*" $\langle proof \rangle$

where *prop* is a proposition in inner syntax which contains the variables $x_1 \dots x_m$. As for variables introduced by **fix** or **define** the variables may be grouped by **and** and types may be specified for (some of) the groups.

If the proposition *prop* is a derivation rule with possibly multiple conclusions it may be specified in structured form (see Section 2.5.6) using several separate propositions:

obtain $x_1 \dots x_m$ **where** " C_1 " \dots " C_h " **if** " A_1 " \dots " A_n " **for** $y_1 \dots y_p$ $\langle proof \rangle$

As usual, the conclusions, assumptions, and bound variables may be grouped by **and**, the assumption and conclusion groups may be named and types may be specified for (some of) the groups of bound variables.

The proposition(s) usually relate the values of the new variables to values of existing variables (which may be local or come from enclosing contexts).

In the simplest case the proposition(s) directly specify terms for the new variables, like a **define** statement (see Section 3.10) such as in

```
fix x::nat
obtain y z where "y = x + 3" and "z = x + 5" <proof>
```

But it is also possible to specify the values indirectly:

```
fix x::nat
obtain y z where "x = y - 3" and "y + z = 2*x + 8" <proof>
```

Here the propositions may be considered to be additional facts which are added to the proof context.

3.11.1 Proving *obtain* Statements

An **obtain** statement has a similar meaning as the statements

```
fix x1 ... xm
assume "prop"
```

but there is one important difference: the proposition 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 proposition, which may be stated in the following way: if any other proposition can be derived from it in the local proof context it must be possible to also derive it without the proposition. This can be stated formally as

$$(\bigwedge x_1 \dots x_m. \text{prop} \implies P) \implies P$$

which is exactly the goal to be proved 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 fact is the proposition specified in the statement, not the proved redundancy goal. If the proposition is specified in structured form with multiple conclusions the current facts are the collection of facts corresponding to the conclusions and if names are specified for conclusion groups they are used to name the resulting facts.

Input facts may be passed to **obtain** statements. As for the other goal statements, they are input to the *<proof>*.

3.11.2 Exporting Facts after Obtaining Variables

Unlike facts assumed by an **assume** statement (see Section 3.8.1) 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 proved to be redundant, therefore they can be omitted. Also, unlike variables introduced by a **define** statement (see Section 3.10) occurrences of obtained variables in F are not replaced by defining terms, since such terms are not available in the general case.

That implies that an exported fact F may not refer to variables introduced by an **obtain** statement, because the information provided by the proposition about them gets lost during the export. Otherwise an error is signaled.

3.12 Term Abbreviations

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

3.12.1 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 ≤ ?rhs" <proof>
```

The name `?thesis` (see Section 3.5.2) is a term abbreviation of this kind. It is defined automatically for every proof in the outermost proof context.

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

```
let ?name1 = "term1" and ... and ?namen = "termn"
```

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.

3.12.2 Pattern Matching

Note that term abbreviations have the form of “unknowns” (see Section 2.6.2), 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.

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.

Parameterized Term Abbreviations

Pattern matching can be used to define parameterized abbreviations. If the pattern has the form of a function application where the unknown is the function, it will be bound to a function which, after substituting the arguments, will be equal to the *term*. So it can be applied to other arguments to yield terms where the corresponding parts have been replaced. This kind of pattern matching is called “higher order unification” and only succeeds if the pattern and *term* are not too complex.

The **let** statement

```
let "?lhs x ≤ ?rhs 5" = "2*x+3 ≤ 2*5+3"
```

binds both unknowns to the lambda term $\lambda a. 2 * a + 3$. Thus afterwards the use *?lhs 7* results in the term $2 * 7 + 3$.

Already Bound Unknowns

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  $\implies$  2*x+3  $\leq$  2*5+3"
proof method
  let "?lhs  $\leq$  ?rhs" = ?thesis
  ...

```

to reuse parts of the conclusion in the proof without specifying them explicitly.

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

```

let "pattern1" = "term1" and ... and "patternn" = "termn"

```

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 "pattern1" = "term1"
...
let "patternn" = "termn"

```

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.

Bound Variables in the Matched Term

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.

3.12.3 Casual Term Abbreviations

Term abbreviations may also be defined in a “casual way” by appending a pattern to a proposition which is used for some other purpose in the form

```

"prop" (is "pattern")

```

The pattern is matched with the proposition *prop* and if successful the unknowns in the pattern are bound as term abbreviations.

This is possible for all propositions used in a theorem (see Section 2.6), such as in

```
theorem "prop" (is ?p) <proof>
theorem fixes x::nat and c and n
  assumes asm1: "x < c" (is ?a1) and "n > 0" (is ?a2)
  shows "n*x < n*c" (is "?lhs < ?rhs")
  <proof>
```

and also in the structured form using **if** (see Section 2.5.6). The unknowns will be bound as term abbreviations in the outermost proof context in the proof of the theorem.

Note the difference between the fact name *asm1* and the term abbreviation name *?a1* in the example. The fact name refers to the proposition $x < c$ as a valid fact, it can only be used to work with the proposition as a logical entity. The abbreviation name *?a1*, instead, refers to the proposition as a syntactic term of type *bool*, it can be used to construct larger terms such as in $?a1 \wedge ?a2$ which is equivalent to the term $x < c \wedge n > 0$.

Casual term abbreviations may also be defined for propositions used in goal statements (see Sections 3.5 and 3.11) and in **assume**, **presume**, and **define** statements (see Sections 3.8 and 3.10). Then the unknowns will be bound as term abbreviations in the *enclosing* proof context, so that they are available after the statement (and also in the nested subproof of the goal statements).

3.13 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 ...
```

3.13.1 Support for Fact Accumulation

Isabelle supports this management of *calculation* with two specific 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 proof contexts, 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.

3.13.2 Accumulating Facts in a Nested Context

Fact accumulation can be used for collecting the facts in a nested context for export (see Section 3.7.4) without using explicit names for them:

```
... {  
  have "prop1" <proof>1  
  moreover have "prop2" <proof>2  
  ...  
  moreover have "propn" <proof>n  
  moreover note calculation  
}
```

3.13.3 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 "F1,1" <proof>1,1  
then have "F1,2" <proof>1,2  
...  
then have "F1,m" <proof>1,m
```

moreover have " $F_{2,1}$ " $\langle proof \rangle_{2,1}$
then have " $F_{2,2}$ " $\langle proof \rangle_{2,2}$
 ...
then have " $F_{2,n}$ " $\langle proof \rangle_{2,n}$
ultimately have " F " $\langle proof \rangle$

The **moreover** statement starts the second branch and saves the fact $F_{1,m}$ to *calculation*. The **ultimately** statement adds the fact $F_{2,n}$ 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.

3.14 Equational Reasoning

Often informal proofs on paper are written in the style

$$2*(x+3) = 2*x+6 \leq 3*x+6 < 3*x+9 = 3*(x+3)$$

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 $=$, \leq and $<$. Instead, the formula is meant as an abbreviated notation of the fact sequence

$$2*(x+3) = 2*x+6, 2*x+6 \leq 3*x+6, 3*x+6 < 3*x+9, 3*x+9 = 3*(x+3)$$

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.

3.14.1 Transitivity Rules

The full example proof needs additional facts which must be inserted into the sequence. From the first two facts the fact $2*(x+3) \leq 3*x+6$ 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 $\llbracket a = b; b \leq c \rrbracket \implies a \leq c$, $\llbracket a \leq b; b < c \rrbracket \implies a < c$, and $\llbracket a < b; b = c \rrbracket \implies a < c$.

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 proved from assumptions 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.

3.14.2 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 \dots (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.

3.14.3 Determining Transitivity Rules

To automatically determine the transitivity rule used by **also** or **finally**, Isabelle maintains the dynamic fact set (see Section 2.6.3) 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)
```

Chapter 4

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.

4.1 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. An exception is a goal of the form $C_1 \ \&\&\& \ \dots \ \&\&\& \ C_h$ which is always split to separate goals C_i whenever a method is applied (see section 3.1.3).

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

proof method $ST_1 \ \dots \ ST_n$ **qed**

If the statements $ST_1 \ \dots \ ST_n$ shall process the unmodified original goal the empty method must be specified for *method*, thus the structured proof becomes

proof - $ST_1 \ \dots \ ST_n$ **qed**

Note that it is possible to syntactically omit the *method* completely, but then it defaults to the method named *standard* which alters the goal state (see Section 4.3.5).

If input facts are passed to the empty method, it affects all goals by inserting the input facts as assumptions after the existing assumptions. If the input facts are F_1, \dots, F_m a goal of the form $\llbracket A_1; \dots; A_n \rrbracket \implies C$ is replaced by the goal $\llbracket A_1; \dots; A_n; F_1, \dots, F_m \rrbracket \implies C$.

This makes it possible to transfer facts stored in the proof context to the goal state where they are stored as rule assumptions (see Section 3.2.3). Since this way of storing facts is not useful for structured proofs it is normally useless to input facts into a structured proof with the empty method as initial method.

If a goal statement instead uses a proof script as subproof the script can be started by applying the empty method to transfer input facts into the goal state for use by further method applications:

... then have "C" apply - $MA_1 \dots MA_n$ done

If the current facts are F_1, \dots, F_m before the **then** statement the goal state in the subproof contains the goal $\llbracket F_1, \dots, F_m \rrbracket \implies C$ before method application MA_1 , so the facts are available for use in the proof script.

4.2 Terminating Proof Scripts

As described in Section 3.1.2 a proof is complete when its goal state is empty. In a structured proof goals are removed from the goal state by successful **show** statements. Therefore a structured proof is usually terminated by a **show** statement which removes the last goal in the goal state.

Proof scripts, instead, are intended to store facts as rule assumptions in the goal state (see Section 3.3.2). Then the proof of a goal is complete when the conclusion of the current goal unifies with one of its assumptions (see Section 3.2.1).

4.2.1 The *assumption* Method

Such goals are removed from the goal state by the method

assumption

The method only affects the first goal. If that goal has the form $\llbracket A_1; \dots; A_n \rrbracket \implies C$ and one assumption A_i unifies with C the method removes the goal, otherwise an error is signaled.

The *assumption* method is automatically applied by a proof part of the form

qed *method*

after applying the specified *method*. The application of the *assumption* method is repeated as long as it is applicable, thus **qed** removes all goals from the goal state where the conclusion matches an assumption. The same holds for the abbreviated forms **by *method*** and **by *method*₁ *method*₂** (see Section 3.3.2).

Therefore a proof script consisting of method applications $MA_1 \dots MA_n$ can be terminated by **by** - if the method applications refine all goals to the form where the conclusion unifies with an assumption. Note that **done** does not remove such goals, when it is used to terminate a proof it expects that the goal state is already empty.

4.3 Basic Rule Application

As described in Section 3.2.1 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 $RA_i \implies RC_i$ 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.

4.3.1 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 $\llbracket A_1; \dots; A_n \rrbracket \implies C$ and the rule referred by *name* has the form $\llbracket RA_1; \dots; RA_m \rrbracket \implies RC$ the method first unifies RC with the goal conclusion C . That yields the specialized rule $\llbracket RA_1'; \dots; RA_m' \rrbracket \implies RC'$ 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' . If the goal does not contain unknowns (which is the normal case) 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 $\llbracket A_1; \dots; A_n \rrbracket \implies RA_j'$.

If the rule has the form $RA \implies RC$ with only one assumption the method replaces the goal by the single new goal $\llbracket A_1; \dots; A_n \rrbracket \implies RA'$. If the rule is a formula RC without any assumptions the method removes the goal without introducing a new goal.

Note that if facts are stored as rule assumptions in the goal state (see Section 3.2.3) an application of method *rule* preserves these facts and copies them to every new goal.

If an assumption RA_j is again a rule (i.e., the applied rule is a meta rule) and has the form $\llbracket B_1; \dots; B_k \rrbracket \implies B$ the j th new goal becomes $\llbracket A_1; \dots; A_n \rrbracket \implies (\llbracket B_1'; \dots; B_k' \rrbracket \implies B')$ which by definition of the $\llbracket \dots; \dots \rrbracket$ syntax (see Section 2.5.5) is equivalent to $\llbracket A_1; \dots; A_n; B_1'; \dots; B_k' \rrbracket \implies B'$. In this way the rule can introduce additional assumptions in the resulting goals, which are inserted after the existing assumptions.

4.3.2 Using the *rule* Method for Backward Reasoning Steps

Assume that during a proof for $A \implies C$ as described in Section 3.2.1 the intermediate fact sequence $F_{i+1}, \dots, F_{n-1}, C$ has already been constructed by backward reasoning, i.e., the current goal is F_{i+1} . If r_i names a rule $RA_i \implies RC_i$, the successful method application

```
apply (rule  $r_i$ )
```

will replace the current goal by F_i and thus extend the fact sequence to F_i, \dots, F_{n-1}, C . The fact F_i is the specialized assumption RA_i constructed by the method from the assumption RA_i of rule r_i . Together, this application of the *rule* method implements the backwards reasoning step from F_{i+1} to F_i .

Therefore the fact sequence F_1, \dots, F_{n-1}, C of the complete proof can be constructed by the proof script consisting of the backward reasoning steps

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

Note that we used the *assumption* method to remove the goal $A \implies F_1$ by unifying F_1 with A . Alternatively we can use the form

```
apply (rule  $r_{n-1}$ )
...
apply(rule  $r_1$ )
by -
```

where the *assumption* method is applied implicitly, as described in Section 4.2.1, or even shorter

```
apply (rule  $r_{n-1}$ )
...
by (rule  $r_1$ )
```

If the example from Section 3.2.1 is proved this way the theorem is written together with its proof as

```
theorem "( $x::nat$ ) < 5  $\implies$  2*x+3  $\leq$  2*5 + 3"
  apply (rule example2)
  by (rule example1)
```

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 $x < 3 \implies x < 5$ which is trivial for the human reader but not applicable to the *assumption* method.

4.3.3 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 internal 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:nat) < 5 ==> 2*x+3 ≤ 2*5 + 3"
  apply rule
  by rule
```

4.3.4 Introduction Rules

The set *intro* is intended by Isabelle for a specific kind of rules called “introduction rules”. In such a rule a specific function *f* (perhaps written using an operator name) occurs only in the conclusion, but not in any assumption, hence it is “introduced” by the rule.

When an introduction rule for *f* is applied to a goal by the *rule* method, it works “backwards” and removes *f* from the goal. If the set *intro* only contains introduction rules and no rule adds a function which has been removed by another rule in the set, an iterated application of the *rule* method with automatic rule selection will “deconstruct” the goal: every step removes a function from the goal. The iteration stops when no rule in *intro* is applicable. In some sense the resulting goals are simpler because the set of functions used in them has been reduced. Some proofs can be written using this technique, however, they depend on the careful selection of the introduction rule in *intro*.

A rule is also called an introduction rule for a function *f* if *f* occurs in some assumption(s) but in a different form (usually applied to other arguments). Then an application by the *rule* method will replace a term containing *f* by a different term containing *f*. The idea is to replace terms in several steps using one or more introduction rules until finally removing *f* completely.

The *intro* Method

This proof technique can also be applied by the method

intro name₁ ... name_n

where $name_1 \dots name_n$ refer to valid rules. It iterates applying rules from the given set to a goal in the goal state as long as this is possible. It is intended to be used with introduction rules. Then it automatically deconstructs the goals as much as possible with the given rules. Note that the method does not use the *intro* set.

Examples

The rule *example1* from Section 2.6.3 is an introduction rule for both functions (\leq) and ($*$), but it is only applicable to special uses of them and it replaces them by the function ($<$) which also is only useful for some specific proofs. In the rule *example2* the function (\leq) also occurs in the assumption, however applied to other arguments, therefore it can also be considered as an introduction rule for (\leq).

Using the *intro* method the example proof can be written as

```
theorem "(x::nat) < 5  $\implies$  2*x+3  $\leq$  2*5 + 3"
  by (intro example1 example2)
```

The rules usually present in *intro* in Isabelle are carefully selected to be as generally applicable and useful for a large number of proofs as possible. Besides by the *rule* method the *intro* set is also used internally by some of the automatic methods described in Section 4.7.1.

Searching Introduction Rules

If the cursor in the text area is positioned in a proof, introduction rules applicable to the first goal in the goal state of the enclosing proof can be searched using the keyword *intro* as criterion for a search by **find_theorems** or in the Query panel, as described in Section 2.6.6. It finds all named facts which can be applied by the *rule* method to the first goal, i.e., the conclusions can be unified.

4.3.5 The *standard* Method

The method

```
standard
```

is a method alias which can be varied for different Isabelle applications. Usually it is mainly 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 ST1 ... STn qed
```

is an abbreviation for

```
proof standard  $ST_1 \dots ST_n$  qed
```

Note that the *standard* method as initial method in a structured proof will usually affect the goal by applying a rule from the set *intro* to it. That may be useful in some cases, but it has to be taken into account when writing the statements of the proof. If the rules *example1* and *example2* are again considered to be in the set *intro*, the example proof could be written as

```
theorem "(x:nat) < 5  $\implies$  2*x+3  $\leq$  2*5 + 3"
proof
  show "x < 5  $\implies$  2*x  $\leq$  2*5" by (rule example1)
qed
```

because *proof* (without the empty method -) already applies the rule *example2* by applying the *standard* method. It replaces the original goal by $x < 5 \implies 2*x \leq 2*5$, so only this goal remains to be proved. However, this goal replacement is often not apparent to the reader of the proof. Therefore this form of structured proof should be used with care, it is only intended for some standard cases where the goal replacement is clearly expected by the proof reader and writer. Also note that *?thesis* still abbreviates the original goal conclusion and thus cannot be used in the proof anymore.

The .. Proof

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 proved by a single automatic rule application. Since in the example proof also rule *example1* could be automatically selected by the *standard* method, it could be further abbreviated as

```
theorem "(x:nat) < 5  $\implies$  2*x+3  $\leq$  2*5 + 3"
proof
  show "x < 5  $\implies$  2*x  $\leq$  2*5" ..
qed
```

4.4 Rule Application in Forward Reasoning

Assume that during a proof for $A \implies C$ as described in Section 3.2.1 the intermediate fact sequence A, F_2, \dots, F_i has already been constructed by forward reasoning and has been stored in the proof context. Then the next step is to state fact F_{i+1} and prove it using a rule $RA_i \implies RC_i$ named r_i .

4.4.1 Using the *rule* Method for Forward Reasoning Steps

Using method *rule* the step can be started by the statement

```
have "Fi+1" proof (rule ri)
```

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 unifies with F_i . The subproof is not finished, since its goal state is not empty. But the goal unifies with an already known fact. Solving a goal which resulted from an assumption of an applied rule is also called “discharging an assumption”.

Discharging Assumptions Using the *fact* Method

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 "Fi+1" proof (rule ri) qed (fact fi)
```

if F_i has been named f_i . This can be abbreviated (see Section 3.3.2) to

```
have "Fi+1" by (rule ri) (fact fi)
```

Therefore the fact sequence $A, F_2, \dots, F_{n-1}, C$ of the complete proof for the goal $A \implies C$ can be constructed by the structured proof of the form

```
proof -  
  assume a: "A"  
  have f2: "F2" by (rule r1) (fact a)  
  ...  
  have fn-1: "Fn-1" by (rule rn-2) (fact fn-2)  
  show "?thesis" by (rule rn-1) (fact fn-1)  
qed
```

where *?thesis* abbreviates the conclusion C , as usual.

If the example from Section 3.2.1 is proved this way the theorem is written together with its proof as

```
theorem "(x::nat) < 5  $\implies$  2*x+3  $\leq$  2*5 + 3"  
proof -  
  assume a: "x < 5"  
  have f2: "2*x  $\leq$  2*5" by (rule example1) (fact a)  
  show ?thesis by (rule example2) (fact f2)  
qed
```

Automatic Fact Selection

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

4.4.2 Input Facts for the *rule* Method

If input facts F_1, \dots, 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 $\llbracket RA_1; \dots; RA_{n+m} \rrbracket \implies RC$ and every fact F_i unifies with assumption RA_i the first n assumptions are removed and the rule becomes $\llbracket RA_{n+1}'; \dots; RA_{n+m}' \rrbracket \implies RC'$, specialized according to the term substitutions performed by the unifications. Then it is applied to the first goal in the usual way. If there are more input facts than assumptions or if a fact does not unify, an error is signaled.

This behavior of the *rule* method can be explained as follows: as described above, for every assumption in the applied rule the method creates a goal which has the assumption as conclusion. As usual, the goal is considered solved, if the conclusion unifies with a fact in the proof context. By unifying the input facts with the rule assumptions the method determines the goals which would immediately be solved and thus can be omitted, then it removes the assumptions from the rule so that the corresponding goals are never created.

Discharging Assumptions Using Explicit Fact Input

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 $RA_i \implies RC_i$, 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.

Discharging Assumptions Using Chaining

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

```
proof -
  assume "F1"
  then have "F2" by (rule r1)
  ...
  then have "Fn-1" by (rule rn-2)
  then show "Fn" by (rule rn-1)
qed
```

As described in Section 4.3.5 the subproof by `(rule ri)` can be abbreviated as `..` if the rule r_i is in the `intro` rule set.

If the example from Section 3.2.1 is proved this way the theorem is written together with its proof as

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

4.4.3 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 "F1"
  with r1 have "F2" 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 specified 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

The proof

by *this*

can be abbreviated by . (a single dot).

Therefore the example from Section 3.2.1 can also be proved in the form

```

theorem " $(x::nat) < 5 \implies 2*x+3 \leq 2*5 + 3$ "
proof -
  assume " $x < 5$ "
  with example1 have " $2*x \leq 2*5$ " .
  with example2 show ?thesis .
qed

```

4.4.4 The Methods *frule* and *drule*

Instead of storing facts in the proof context and using a structured proof for a forward reasoning proof the facts may be stored as rule assumptions in the goal state and the forward reasoning proof may be written as a proof script (see Section 3.3.2).

To construct a proof for the goal $A \implies C$ as fact sequence $A, F_2 \dots F_n$ by forward reasoning as described in Section 3.2.1 in this way, the intermediate fact sequence A, F_2, \dots, F_i is stored in the extended goal $\llbracket A; F_2; \dots; F_i \rrbracket \implies C$ where the last assumption is the current fact F_i . Then the next forward reasoning step consists of adding the fact F_{i+1} as new assumption to the goal and prove it using a rule $RA_i \implies RC_i$ named r_i . When the fact sequence is complete the goal is $\llbracket A; F_2; \dots; F_n \rrbracket \implies C$ and F_n unifies with C , thus an application of method *assumption* will remove the goal and terminate the proof (see Section 4.2.1).

The *frule* Method

A rule is applied for forward reasoning by the method

frule name

where *name* is the name of a valid rule. The method only affects the first goal. If that goal has the form $\llbracket A_1; \dots; A_n \rrbracket \implies C$ and the rule referred by *name* has the form $\llbracket RA_1; \dots; RA_m \rrbracket \implies RC$ the method first unifies the first assumption RA_1 in the rule with the first assumption A_k of the $A_1 \dots A_n$ which can be unified with RA_1 . That yields the specialized rule $\llbracket RA_1'; \dots; RA_m' \rrbracket \implies RC'$ where RA_1' is syntactically equal to A_k and every RA_j' with $j > 1$ and RC' results from RA_j or RC , respectively, by substituting unknowns by the same terms as in RA_1' . If the goal does not contain unknowns (which is the normal case) A_k is not modified by the unification. If no A_i unifies with RA_1 the method cannot be executed on the goal state and an error is signaled. Otherwise the method replaces the goal by the $m-1$ new goals $\llbracket A_1; \dots; A_n \rrbracket \implies RA_j'$ for $j > 1$ and the goal $\llbracket A_1; \dots; A_n; RC' \rrbracket \implies C$.

If the rule has the form $RA \implies RC$ with only one assumption the method replaces the goal by the single new goal $\llbracket A_1; \dots; A_n; RC' \rrbracket \implies C$, i.e., it adds the conclusion of the rule as assumption in the goal. If the rule is a formula RC without any assumptions the method is not applicable and signals an error.

Since the first assumption RA_1 in the rule plays a special role in this context, it is also called the “major premise” of the rule.

Together, a forward reasoning step as described above can be implemented by the method application

apply (*frule* r_i)

and the full proof script for a forward reasoning proof has the form

```
apply(frule  $r_1$ )
...
apply (frule  $r_{n-1}$ )
apply(assumption)
done
```

or shorter

```
apply (frule  $r_1$ )
...
by (frule  $r_{n-1}$ )
```

If the example from Section 3.2.1 is proved this way the theorem is written together with its proof as

```

theorem "(x:nat) < 5  $\implies$  2*x+3  $\leq$  2*5 + 3"
  apply (frule example1)
  by (frule example2)

```

However, this form of forward reasoning proof has several drawbacks. First, as always in proof scripts, the facts F_i are not specified explicitly, they are constructed implicitly by the *frule* method and can only be seen by interactively inspecting the goal state. Next, since the current fact is the last assumption in the goal, it is not guaranteed that the rule r_i is applied to it. If a previous assumption also unifies with the major premise of r_i the rule is not applied in the intended way. Finally, it is not possible to generalize this approach to proofs with several branches. The branches cannot be joined, because *frule* always takes only one assumption into account.

The *drule* Method

The second drawback can be compensated for by using another method for applying the rule. This is done by the method

```
drule name
```

where *name* is the name of a valid rule. The method works like *frule*, but instead of adding RC' to the assumptions it replaces A_k by it. Thus the method replaces the goal by the $m-1$ new goals $\llbracket A_1; \dots; A_{k-1}; A_{k+1}; \dots; A_n \rrbracket \implies RA_j'$ for $j > 1$ and the goal $\llbracket A_1; \dots; A_{k-1}; A_{k+1}; \dots; A_n; RC' \rrbracket \implies C$.

When using *drule* for constructing a proof in the way described above, it always replaces the current fact by the next one in the fact sequence. The intermediate fact sequence is represented by the goal $F_i \implies C$. Since the current fact is the only assumption present in the goal, the step **apply** (*drule* r_i) is always applied to it and replaces the goal by $F_{i+1} \implies C$, as intended. The methods *frule* and *drule* do not support input facts.

4.4.5 Destruction Rules

Not all rules can always usefully be applied by *frule* and *drule*. Since both methods only unify their first assumption (the major premise) of the rule with a term in the goal and then replace it by the conclusion, the first assumption should have some effect on the conclusion. In particular, the conclusion should not be a single unknown which does not occur in the first assumption.

If additionally a specific function f (perhaps written using an operator name) occurs only in the first assumption and neither in the conclusion, nor in other assumptions, the rule is called a “destruction rule” for f . If it is applied in forward direction, such as with *frule* and *drule*, f will be removed from the goal, it will be “destroyed”.

Similar to introduction rules (see Section 4.3.4) *f* may occur in the conclusion if it has a different form, so that it may be removed by several steps through intermediate forms.

Analogous to the *intro* set for introduction rules there is an internal fact set *dest* for destruction rules. It is used by some automatic methods, however, it is not used for automatically selecting rules for *frule* and *drule*.

Searching Destruction Rules

If the cursor in the text area is positioned in a proof, destruction rules applicable to the first goal in the goal state of the enclosing proof can be searched using the keyword *dest* as criterion for a search by **find_theorems** or in the Query panel, as described in Section 2.6.6. It finds all named facts which can be applied by the *frule* or *drule* method to the first goal, i.e., the major premise unifies with a goal assumption.

Examples

The rule *example1* from Section 2.6.3 is a destruction rule for function (*<*), but it is also only applicable to special uses of it and it replaces it by the functions (*≤*), (***), and (*+*) which does not help for most proofs. In the rule *example2* the operator (*≤*) also occurs in the conclusion, but in different form. Therefore it can be considered as destruction rule for (*≤*), although the form in the conclusion is more complex which also does not help for most proofs.

4.5 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, \dots, m_n : a sequence of methods which are applied in their order,
- $m_1; \dots; m_n$: a sequence of methods where each is applied to the goals created by the previous method,
- $m_1 / \dots / m_n$: 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 method applications to a single step. Using composed methods the example backward reasoning proof script from Section 4.3.2 can be written as

```
theorem "(x::nat) < 5  $\implies$  2*x+3  $\leq$  2*5 + 3"
  apply(rule example2,rule example1,assumption)
  done
```

In particular, it is also possible to apply an arbitrarily complex composed method as initial method in a structured proof. Using composed methods the example forward reasoning proof in Section 4.4.1 can be written as

```
theorem "(x::nat) < 5  $\implies$  2*x+3  $\leq$  2*5 + 3"
proof -
  assume a: "x < 5"
  have f2: "2*x  $\leq$  2*5" by (rule example1,fact a)
  show ?thesis by (rule example2,fact f2)
qed
```

4.6 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 proved to be valid, i.e., is a fact, **a** can be substituted by **b** and vice versa in goals during a proof.

4.6.1 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.

Rewriting in Goal Assumptions

For a goal $\llbracket A_1; \dots; A_n \rrbracket \Rightarrow C$ the method only rewrites in the conclusion C . The first match in the assumptions $A_1 \dots A_n$ can be substituted by the form

```
subst (asm) name
```

Rewriting Specific Sub-Term Occurrences

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
```

Rewriting With Defining Equations

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 $x \equiv x + 1$ "

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 2.6.5 that the defining equation is automatically named `inc_def`. Note the use of unification to handle the actual argument term t .

Rewriting With Conditional Equations

The equality fact may be conditional, i.e., it may be a derivation rule with assumptions of the form $\llbracket RA_1; \dots; RA_m \rrbracket \Rightarrow a = b$. When the `subst` method applies a conditional equation of this form to a goal $\llbracket A_1; \dots; A_n \rrbracket \Rightarrow C$, it adds the goals $\llbracket A_1; \dots; A_n \rrbracket \Rightarrow RA_i'$ 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 $\llbracket A_1; \dots; A_n \rrbracket \Rightarrow RA_1'$.

Examples

As an example if there are theorems

```
theorem eq1: "n = 10  $\Rightarrow$  n+3 = 13" for n::nat <proof>  
theorem eq2: "n = 5  $\Rightarrow$  2*n = 10" for n::nat <proof>
```

the method `subst (2) eq2` replaces the goal $(x:\text{nat}) < 5 \implies 2*x+3 \leq 2*5 + 3$ by the goals

$$\begin{aligned} x < 5 &\implies 5 = 5 \\ x < 5 &\implies 2 * x + 3 \leq 10 + 3 \end{aligned}$$

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

$$\begin{aligned} x < 5 &\implies 10 = 10 \\ x < 5 &\implies 2 * x + 3 \leq 13 \end{aligned}$$

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

$$\begin{aligned} x < 5 &\implies x = 5 \\ x < 5 &\implies 10 + 3 \leq 2 * 5 + 3 \end{aligned}$$

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

4.6.2 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.

Non-Terminating Simplification

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 Definitional Equations

Simplification with a set of definitional equations from constant definitions (see Section 2.3.1) 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.

Simplification and Trivial Goals

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 $x = x$, $A \implies A$, and True . Such an extended simplification may completely solve and remove the goal to which it is applied.

4.6.3 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 the simpset 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.

Simplification Using Arbitrary Facts

The simpset may contain facts which are not directly equations, but can be converted to an equation. In particular, an arbitrary derivation rule $\llbracket A_1; \dots; A_n \rrbracket \implies C$ can always be converted to the conditional equation $\llbracket A_1; \dots; A_n \rrbracket \implies C = \text{True}$. The simplifier (and also the *subst* method) 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*

Simplification in HOL

In Isabelle HOL (see Section 6) 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 proved by the simplifier:

```
theorem example1: "(x::nat) < c  $\implies$  n*x  $\leq$  n*c" by simp
theorem example2: "(x::nat)  $\leq$  c  $\implies$  x + m  $\leq$  c + m" by simp
theorem "(x::nat) < 5  $\implies$  2*x+3  $\leq$  2*5 + 3" by simp
theorem eq1: "n = 10  $\implies$  n+3 = 13" for n::nat by simp
theorem eq2: "n = 5  $\implies$  2*n = 10" for n::nat by simp
```

4.6.4 Configuring the Simplifier

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

```
simp add: name1 ... namen
```

uses the facts *name*₁, ..., *name*_n in addition to the facts in the simpset for its rewriting steps. The form

```
simp del: name1 ... namen
```

uses only the facts from the simpset without the facts *name*₁, ..., *name*_n, and the form

```
simp only: name1 ... namen
```

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 2.6.3 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  $\equiv$  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.

4.6.5 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 7.1.3). 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.

Split Rules

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: name1 ... namen`

where the `namei` are the names of the split rules to use. This configuration can be arbitrarily combined with the other simplifier configuration options. The usual form of a split rule is a proposition

`"?P(term) = ((Q1 → ?P(term1)) ∧ ... ∧ (Qn → ?P(termn)))"`

where the `termi` are subterms of `term` and every `Qi` represents a condition for which `term` can be reduced to `termi`. The simplifier applies such a split rule to a goal $\llbracket A_1; \dots; A_m \rrbracket \Longrightarrow C$ by first unifying the left hand side with the conclusion `C` (which succeeds if `term` occurs in `C`), then replacing it by the conjunction on the right, then splitting the goal into a separate goal for every conjunct, and finally moving every `Qi` to the assumptions of their goal. Thus the resulting goals have the form

$$\begin{aligned} \llbracket A_1; \dots; A_m; Q_1 \rrbracket &\Longrightarrow C_1 \\ \dots \\ \llbracket A_1; \dots; A_m; Q_n \rrbracket &\Longrightarrow C_n \end{aligned}$$

where `Ci` is constructed from `C` by mainly replacing `term` by `termi`.

Note that this form of a split rule can only be applied for splitting terms in the conclusion of a goal. See the Isabelle/Isar reference manual [7] for other forms which split terms in assumptions of a goal.

4.6.6 Searching Simplifier Equations

Named facts applicable for simplification may be searched using the command `find_theorems` or in the Query panel, as described in Section 2.6.6, using a *criterion_i* of the form `simp: termpat` where *term_{pat}* is a term pattern (a term which may contain unknowns) specified in inner syntax.

Such a search finds named facts where the conclusion is an equation (using either the operator `=` or the meta equality `≡`) and the left side unifies with the specified term pattern. It also finds facts where the conclusion unifies with the term pattern, if the term pattern has type `bool`, because such facts are equivalent to an equation with `True` (see above). Facts are found independent of whether they are in the simpset or not.

A found fact may be used by specifying it for the `subst` method or, if not yet in the simpset, by configuring the `simp` method to use it with the help of `add:` or `only:`.

4.6.7 Input Facts for the *simp* Method

As usual, facts may be input to the `simp` method. Like the empty method (see Section 4.1) 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 4.2.1). This is also done by the simplifier, hence in this way the input fact may contribute to prove the goal.

4.6.8 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:`, `only:`, and `split:` in the same way as the `simp` method.

The `simp_all` method is useful, if first a method *m* 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 *m simp_all*

4.6.9 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 specified 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 at least 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 Isabelle/Isar reference manual [7] 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* method applications.

4.7 Other Automatic Proof Methods

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

4.7.1 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.
- *linarith* solves linear arithmetic problems (on integer and real numbers) for the first goal. It is automatically included by the simplifier.
- *arith* uses more techniques than *linarith* but may be slower.
- *metis* is a versatile method using other techniques.

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: name1 ... namen
```

For more information about these methods see the Isabelle/Isar reference manual [7].

4.7.2 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 proved.

If **try** finds a proof for one or more goals it displays it as a single (composed) proof method, which, by clicking on it 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 usually 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 proved.

Chapter 5

Case Based Proofs

If during a proof the goal state contains several goals they are often proved sequentially. Although there are proof methods which process several goals at once (see examples in Section 4.7.1) most methods only process the first goal. In a proof script, when a method has solved and removed the first goal, the next goal will become the first and will be processed by the next method application steps. In a structured proof (see Section 3.3.2) it is not so simple. To prove a goal which is in the goal state its bound variables and assumptions have to be inserted into the local proof context (by **fix** and **assume** statements) and its conclusion must be stated by a **show** statement and must be proved. Isabelle provides some support for simplifying these tasks for working on a sequence of goals.

5.1 Named Proof Contexts

Isabelle supports some methods which are able to create “cases” for goals. A case contains bound variables and assumptions from a goal and it may contain other context elements, such as names for assumptions or assumption groups and an abbreviation for the conclusion of a goal. The cases are named. Using these names a proof context can be populated with all content of a case in a single step.

Since a case contains context elements it can be seen as a named context which has been prepared by a method for later use, but has not been “activated” yet. Usually a named context is used to initialize a new nested proof context immediately after its beginning by inserting the content of the named context into the new context.

5.1.1 The *case* Statement

The content of a case may be inserted into the current proof context by the statement

case *name*

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

```
fix  $x_1 \dots x_k$   
let  $?a_1 = t_1$  and  $\dots ?a_m = t_m$   
assume name: " $A_1$ "  $\dots$  " $A_n$ "
```

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

Explicit Assumption Naming

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

case *aname*: *name*

If defined in the case, other names for single assumptions or assumption groups may be automatically introduced.

Naming Local Variables

The local variables $x_1 \dots x_k$ are fixed by the **case** statement 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* $y_1 \dots y_j$)

Then the names $y_1 \dots y_j$ 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 a named context for a goal they usually only define the term abbreviation *?case* for the conclusion of the goal.

5.1.2 Proof Structure with Cases

The usual proof structure using cases consists of an initial method which creates cases and of a sequence of nested contexts (see Section 3.3.4). At its beginning each context is initialized by a **case** statement naming one of the created cases, at its end it uses a **show** statement to remove the corresponding goal:

```

proof method
case name1
...
show ?case ⟨proof⟩
next
case name2
...
show ?case ⟨proof⟩
next
...
next
case namen
...
show ?case ⟨proof⟩
qed

```

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

Determining the Cases Introduced by a Method Application

To find out which cases have been introduced by a method application, the command

```
print_cases
```

can be used at arbitrary places in the following proof to display the cases in the Output panel.

In the interactive editor also the Query panel (see Section 1.2.7) can be used to display the cases available at the cursor position by selecting the tab “Print Context” and checking “cases”.

Also in the interactive editor, when the cursor is positioned on **proof *method*** where the method supports cases, a skeleton of 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.

5.2 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 $\bigwedge x_1 \dots x_m. [A_1; \dots; A_n] \implies C$ the created named context contains the local variables $x_1 \dots x_m$, the facts A_1, \dots, 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** statement will refine the goal and remove it from the goal state.

Case Naming

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

```
goal_cases name1 ... namen
```

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.

Composed Methods

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`.

Inspecting the Cases

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. In a case the goal conclusion can be shown partially or fully by defining a possibly abbreviated form of it by

```
let "pattern" = ?case
```

where the *pattern* may contain unknowns which abbreviate sub terms of the conclusion.

A better way to use cases is together with a method which combines both: creating new goals in a simple and expected way, and immediately creating cases only for these goals.

5.3 Case Based Reasoning

Case based proofs are especially convenient for implementing case based reasoning. The technique of “case based reasoning” uses a specific kind of forward reasoning steps (see Section 3.2.1). It adds a new fact F_{i+1} to the proof context “out of the blue” without proving it from the existing facts. For the proof to stay correct, this must be done for “all possible cases” of F_{i+1} , and the proof must be completed separately for each of them.

In its simplest form this can be done by adding an arbitrary fact F_{i+1} and its negation $\neg F_{i+1}$, this covers all possibilities, since F_{i+1} may be either *True* or *False*.

Consider the derivation rule $(x::nat) < c \implies n*x \leq n*c$ named *example1* in Section 2.6.3. To prove it using case based reasoning the additional assumption that 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.

Since the proof must be completed separately for every such case, a separate goal is required for each of them. This cannot be achieved by statements, the old goal must be replaced by several new goals which is only possible by applying a proof method.

More specific, if $Q_1 \dots Q_k$ are propositions for the different cases which together cover all possibilities, a goal $\bigwedge x_1 \dots x_m. [A_1; \dots; A_n] \implies C$ must be replaced by the k goals

$$\begin{aligned} &\bigwedge x_1 \dots x_m. [A_1; \dots; A_n; Q_1] \implies C \\ &\dots \\ &\bigwedge x_1 \dots x_m. [A_1; \dots; A_n; Q_k] \implies C \end{aligned}$$

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

Note that this technique extends the proof procedure described in Section 3.2.1. There a proof consisted of a single tree of facts which started at the assumptions and all branches joined to finally reach the conclusion. With case based reasoning at any position the remaining tree may be split into several “copies” with additional assumptions which then must all be completed separately.

5.3.1 Case Rules

This splitting of a goal into goals for different cases can be done by applying a single meta rule of the specific form

$$[Q_1 \implies ?P; \dots; Q_k \implies ?P] \implies ?P$$

where $Q_1 \dots Q_k$ are all cases of the additional assumption. Such rules are called “case rules”.

When this case rule is applied to the goal $\bigwedge x_1 \dots x_m. \llbracket A_1; \dots; A_n \rrbracket \implies C$ as described in Section 4.3.1, it unifies $?P$ with the conclusion C and replaces the goal in the way described above.

Validity of Case Rules

A case rule is only valid, if the Q_i together cover all possibilities, i.e., if $Q_1 \vee \dots \vee Q_k$ holds. If this has been proved the case rule 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.

Examples

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

$$\llbracket ?n = 0 \implies ?P; ?n \neq 0 \implies ?P \rrbracket \implies ?P$$

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

$$\begin{aligned} \llbracket (x::nat) < c; n = 0 \rrbracket &\implies n*x \leq n*c \\ \llbracket (x::nat) < c; n \neq 0 \rrbracket &\implies n*x \leq n*c \end{aligned}$$

which can now be proved separately.

Actually, the much more general case rule

$$\llbracket ?Q \implies ?P; \neg ?Q \implies ?P \rrbracket \implies ?P$$

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.

General Form of Case Rules

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 a meta rule of the form

$$\llbracket P_1; \dots; P_k \rrbracket \implies ?P$$

where every P_i is a rule of the form

$$\bigwedge x_{i,1} \dots x_{i,p_i} \cdot \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \implies ?P$$

That means, the P_i may be arbitrary rules but they must all have as conclusion the unknown $?P$ which is also the conclusion of the overall case rule. When such a case rule is applied to a goal it splits the goal into k cases and adds the variables $x_{i,1} \dots x_{i,p_i}$ and the assumptions $Q_{i,1} \dots Q_{i,q_i}$ to the i th case.

Case Rules and obtain Statements

Note that the goal which must be proved for an **obtain** statement (see Section 3.11) has the form of a case rule with only one case of the form $\bigwedge x_1 \dots x_m \cdot \text{prop} \implies P$. Thus a proof for this goal shows that **prop** covers all cases, i.e., it is redundant.

Specifying Case Rules

Remember that to write your own case rule you have to specify a theorem which uses variables in place of the unknowns, such as

```
theorem mycaserule: "[n = 0  $\implies$  P; n  $\neq$  0  $\implies$  P]  $\implies$  P" <proof>
```

which will be converted to unknowns upon turning the proposition to a fact after the proof.

5.3.2 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 first 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 (only) the variables and assumptions specified in the corresponding case in the case rule. For the most general form depicted above the context for the i th case contains the variables $x_{i,1} \dots x_{i,p_i}$ and the assumptions $Q_{i,1} \dots Q_{i,q_i}$. 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.

The names used for the contexts created by the `cases` method can be specified by attributing the case rule. Therefore, predefined case rules often create cases with names which are easy to understand by a proof writer.

Examples

In Isabelle HOL (see Section 8) the rule $\llbracket ?Q \implies ?P; \neg ?Q \implies ?P \rrbracket \implies ?P$ is available under the name `case_split`. It is attributed to use the case names `True` and `False`. Note that these are names, not the constants for the values of type `bool`.

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

```
proof (cases "n = 0" rule: case_split)
case True
...
show ?thesis <proof>
next
case False
...
show ?thesis <proof>
qed
```

The `cases` method adds the assumptions `n=0` and `n≠0` respectively to the goals of the cases, the `case` statements 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` statement 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 3.8) if needed for the proof.

Case Rules with Several Unknowns

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. If less terms are specified than there are unknowns in the case

assumptions the resulting goals will contain unbound unknowns which must be instantiated in the rest of the proof (e.g., by method *drule*). If more terms are specified an error is signaled.

Input Facts for the *cases* Method

The *cases* method treats input facts like the empty method (see Section 4.1) by inserting them as assumptions into the original goal before splitting it. Therefore it can be used both in proof scripts, where facts are stored as rule assumptions in the goal state, and in structured proofs where facts are stored in the proof context and are input to the initial methods of subproofs.

However, if the *cases* method is specified in the form

```
cases "term"
```

without explicitly naming a case rule and the first input fact has the form of a case rule, it is used as case rule to split the goal and create the named cases. Therefore in a proof the example goal can be proved as local fact by inputting (see Section 3.6.2) the case rule in the form

```
from case_split
have "(x::nat) < c  $\implies$  n*x  $\leq$  n*c"
proof (cases "n = 0")
...
```

Automatic Case Rule Selection

Like the *rule* method (see Section 4.3.1) the *cases* method supports automatic rule selection for the case rule, if no case rule is specified or input to the method. Normally the rule is selected according to the type of the specified *term*. In Isabelle HOL (see Section 6) most types have an associated case rule. Only case rules with a single unknown in the case assumptions can be automatically selected in this way.

The rule *case_split* is associated with type *bool*. Therefore the example sub proof shown above also works without inputting the method to the **have** statement, because then it is selected automatically because the term *n* = 0 has type *bool*.

The proof writer may not know the case names specified by an automatically selected case rule. However, they can be determined using the command **print_cases** or from the proof skeleton which is displayed in the interactive editor when the cursor is positioned on the *cases* method (see Section 5.1.2).

5.3.3 Alternative Syntax for Case Rules

A case rule as described above always uses an unknown $?P$ (or with any other name) as conclusion and as conclusion in all assumptions. It is used technically to preserve the original conclusion when a goal is split by applying the rule. Therefore Isabelle supports an alternative syntax for specifying case rules as theorems which omits the variable for this unknown and specifies only the information which becomes the content of the named cases.

In a theorem specified in structured form using **shows** (see Section 2.6.4) the part of the form **shows "C"** may alternatively be specified in the form

obtains $C_1 / \dots / C_k$

where every case C_i has the form

$x_{i,1} \dots x_{i,p_i}$ **where** " $Q_{i,1}$ " \dots " Q_{i,q_i} "

As usual, the variables $x_{i,1} \dots x_{i,p_i}$ and the propositions $Q_{i,1} \dots Q_{i,q_i}$ may be grouped by **and**, for every variable group a type may be specified and the proposition groups may be named. The keywords and the $/$ separators belong to the outer syntax, the propositions must be individually quoted.

This form is mainly equivalent to

shows " $\llbracket P_1; \dots; P_k \rrbracket \implies thesis$ "

where every P_i is a rule

$\bigwedge x_{i,1} \dots x_{i,p_i} \cdot \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \implies thesis$

which is exactly the general form of a case rule stated by the **shows** clause, using, after proof, the unknown $?thesis$ for all conclusions.

For its own proof the **obtains** form creates the same goal as the **shows** form, but additionally it adds all P_i as assumed facts to the outermost proof context and names this set **that**.

Specifying Explicit Case Names

When the theorem is applied as case rule by the **cases** method the named context created for case C_i is simply named i . An explicit name may be specified for it by using the extended form

($name_i$) $x_{i,1} \dots x_{i,p_i}$ **where** " $Q_{i,1}$ " \dots " Q_{i,q_i} "

For its own proof the **obtains** form uses $name_i$, if present, as additional name for P_i in its proof context.

If a case C_i has no bound variables it has simply the form

" $Q_{i,1}$ " ... " Q_{i,q_i} "

which omits the keyword **where**, also if an explicit name is specified.

As an example, the rule `case_split` may be defined and proved as

```
theorem case_split:
  obtains (True) "Q" / (False) " $\neg$  Q"
  by blast
```

using the case names *True* and *False*, as described above, and using the *blast* method (see Section 4.7.1) for the proof.

The consider Statement

There is also a statement for stating a case rule on the fly in a structured proof. It has the form

```
consider  $C_1$  / ... /  $C_k$   $\langle$ proof $\rangle$ 
```

where the cases C_i are as above. It is mainly equivalent to the statement

```
have " $[P_1; \dots; P_k] \implies thesis$ "  $\langle$ proof $\rangle$ 
```

with P_i as above and is thus also a goal statement.

However, it is not possible to introduce additional unknowns in the P_i in a **consider** statement. All variables occurring free in the P_i are assumed to be bound in the context and are not converted to unknowns at the end of the statement. Therefore the statement cannot be used to define general case rules like `case_split` which contains the additional unknown $?Q$. It can only be used to state that specific propositions cover all (remaining) possibilities in the local proof context. They need not cover all globally possible cases, if some cases have already been excluded using locally assumed or proved facts, only the remaining possibilities need to be covered.

Chaining a consider Statement

Since case rules can be input as fact to a proof by case based reasoning, fact chaining can be used to immediately apply a locally defined case rule in a subsequent subproof. This yields the pattern

```
consider  $C_1$  / ... /  $C_k$   $\langle$ proof $\rangle$ 
then have "prop"
proof cases
  case 1 ... show ?thesis  $\langle$ proof $\rangle$ 
next
...
  case k ... show ?thesis  $\langle$ proof $\rangle$ 
qed
```


using the default case names. The first $\langle proof \rangle$ proves that the cases cover all local possibilities, the other $\langle proof \rangle$ s prove the stated proposition, each using one of the cases C_i as additional assumption. The `cases` method is always applied without arguments, since there are no additional unknowns in the C_i which can be instantiated.

Example

If the example goal $(x::nat) < c \implies n*x \leq n*c$ should be stated for locally fixed variables, the cases $n=0$ and $n \neq 0$ can be proved to cover all possibilities and then used as cases by the statements

```
fix x c n::nat
consider (Zero) "n = 0" | (Notzero) "n ≠ 0" by blast
then have "(x::nat) < c ⟹ n*x ≤ n*c"
proof cases
  case Zero ... show ?thesis ⟨proof⟩
next
  case Notzero ... show ?thesis ⟨proof⟩
qed
```

5.4 Elimination

The proof technique of “(generalized) elimination” can be seen as a combination of applying a destruction rule (see Section 4.4.5) and an optional case split. It removes an assumption with a function application from a goal and splits the rest into cases with additional assumptions.

5.4.1 The Method *erule*

Like destruction rule application and case splitting such a step can be implemented by applying a rule in a specific way to a goal. This is done by the method

`erule name`

where *name* is the name of a valid rule. The method only affects the first goal. If that goal has the form $\llbracket A_1; \dots; A_n \rrbracket \implies C$ and the rule referred by *name* has the form $\llbracket RA_1; \dots; RA_m \rrbracket \implies RC$ the method first unifies the first assumption RA_1 in the rule with the first assumption A_k of the $A_1 \dots A_n$ which can be unified with RA_1 and it unifies the rule conclusion RC with the goal conclusion C . That yields the specialized rule $\llbracket RA_1'; \dots; RA_m' \rrbracket \implies RC'$ where RA_1' is syntactically equal to A_k , RC' is syntactically equal to C and every RA_j' with $j > 1$ results from RA_j by substituting unknowns by

the same terms as in RA_1' and RC' . If the goal does not contain unknowns (which is the normal case), A_k and C are not modified by the unifications. If no A_i unifies with RA_1 or C does not unify with RC the method cannot be executed on the goal state and an error is signaled. Otherwise the method replaces the goal by the $m-1$ new goals $\llbracket A_1; \dots; A_n \rrbracket \Rightarrow RA_j'$ for $j > 1$.

If the rule has the form $RA \Rightarrow RC$ with only one assumption a successful application with `erule` removes the goal from the goal state. If the rule is a formula RC without any assumptions the method cannot be applied to the goal state and an error is signaled. If an assumption RA_j is a rule with own assumptions, these assumptions are appended to the assumptions in the resulting goal, as described for the `rule` method in Section 4.3.1.

As for rules applied by `frule` or `drule` (see Section 4.4.4) the first assumption RA_1 in the rule is called the “major premise” in this context.

The method `erule` does not support input facts.

5.4.2 Elimination Rules

An elimination rule is a generalized combination of a destruction rule and a case rule. It has the specific form

$$\llbracket RA_1; \dots; RA_n \rrbracket \Rightarrow ?P$$

where the first assumption contains the application of a specific function f (perhaps written using an operator name) which may only occur in different form in the other assumptions. The conclusion is a single unknown which must not occur in the first assumption and may only occur as conclusion in other assumption (as in an assumption in a case rule).

When an elimination rule is applied to a goal by the `erule` method, it removes (“eliminates”) the function application from an assumption in the goal or it replaces it by a different form, so that it may be removed in several steps. Depending on the form of the other assumptions the resulting goals have either the same conclusion as the original goal or are unrelated to it. Therefore such an application of an elimination rule can be seen as a forward reasoning step, possibly with case splitting.

Since the order of the assumptions after the major premise is irrelevant for the rule application the general form of an elimination rule can be considered to be

$$\textbf{theorem } \llbracket RA_1; \dots; RA_m; P_1; \dots; P_k \rrbracket \Rightarrow P'' \langle proof \rangle$$

where every P_i is a rule of the form

$$\bigwedge x_{i,1} \dots x_{i,p_i} . \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \Rightarrow P$$

and the variable P does not occur in the $RA_1 \dots RA_m$.

Analogous to the *intro* set for introduction rules there is an internal fact set *elim* for elimination rules. It is used by some automatic methods, however, it is not used for automatically selecting rules for *erule*.

Example

As an example consider the elimination rule specified as

theorem elimexp: " $\llbracket (x::nat) \leq c; x < c \implies P; x = c \implies P \rrbracket \implies P$ "

It replaces an assumption $x \leq c$ by two cases with assumptions $x < c$ and $x = c$. If applied to the goal $(x::nat) \leq 5 \implies C$ by

apply (*erule elimexp*)

it replaces the goal by the two goals

$x < 5 \implies C$
 $x = 5 \implies C$

Searching Elimination Rules

If the cursor in the text area is positioned in a proof, elimination rules applicable to the first goal in the goal state of the enclosing proof can be searched using the keyword *elim* as criterion for a search by **find_theorems** or in the Query panel, as described in Section 2.6.6. It finds all named facts which can be applied by the *erule* method to the first goal, i.e., the major premise unifies with a goal assumption and the conclusions unify as well.

The *elim* Method

Elimination rule application can be iterated by the method

elim *name*₁ ... *name*_{*n*}

where *name*₁ ... *name*_{*n*} refer to valid rules. It iterates applying rules by *erule* from the given set to a goal in the goal state as long as this is possible. It is intended to be used with elimination rules. Then it automatically eliminates functions from assumptions in the goals as much as possible with the given rules. Note that the method does not use the *elim* set.

Destruction Rules as Elimination Rules

Every destruction rule $\llbracket RA_1; \dots; RA_n \rrbracket \implies C$ can be re-stated as the elimination rule $\llbracket RA_1; \dots; RA_n; C \implies ?P \rrbracket \implies ?P$. If that is applied by *erule* it has the same effect as if the original rule is applied by method *drule*.

5.4.3 Alternative Syntax for Elimination Rules

The alternative syntax available for case rules described in Section 5.3.3 can be extended for elimination rules. An elimination rule

theorem $\llbracket RA_1; \dots; RA_m; P_1; \dots; P_k \rrbracket \implies P$ $\langle proof \rangle$

where every P_i is a rule of the form

$\bigwedge x_{i,1} \dots x_{i,p_i} \cdot \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \implies P$

can be specified using the alternative syntax

theorem
assumes $RA_1 \dots RA_m$
obtains $C_1 \mid \dots \mid C_k$
 $\langle proof \rangle$

where every C_i is

$x_{i,1} \dots x_{i,p_i}$ **where** $Q_{i,1} \dots Q_{i,q_i}$

The major premise and possibly other additional assumptions are specified using **assumes**, then the assumptions for the cases are specified using **obtains**. As usual, the set RA_1, \dots, RA_m is automatically named **assms** and the set of the P_i is automatically named **that** in the outermost proof context of the theorem, additional names may be specified explicitly.

The example rule *elimexp* from the previous section can alternatively be specified as

theorem
assumes $(x : nat) \leq c$
obtains $x < c \mid x = c$
 $\langle proof \rangle$

5.4.4 Elimination in Structured Proofs

The method *erule* is intended to be used in proof scripts, therefore it does not process input facts and does not create named cases. In structured proofs elimination can be done using the *cases* method.

The *cases* method applies a rule by elimination, if the rule is attributed by *[consumes 1]*. This means the rule will “consume” one assumption by unifying it with its major premise and removing it. A rule may be attributed upon definition in the form

theorem *[consumes 1]*: $prop$ $\langle proof \rangle$

or it may be attributed on the fly when it is applied by the `cases` method:

```
cases "term" rule: name[consumes 1]
```

Since the `cases` method is intended to be used in structured proofs where facts are stored in the proof context it does not unify the rule's major premise with an assumption in the goal, it unifies it with the first input fact (possibly after the rule itself if not specified as method argument).

Now the rule `elimexp` from the previous sections can be defined in the form

```
theorem elimexp[consumes 1]:
  assumes "(x::nat) ≤ c"
  obtains (lower) "x < c" | (equal) "x = c"
  ⟨proof⟩
```

naming the cases by `lower` and `equal`. Then an example for an application in a structured proof with cases is

```
theorem "C" if "(x::nat) ≤ 5"
  using that
proof (cases rule: elimexp)
  case lower ... show ?thesis ⟨proof⟩
next
  case equal ... show ?thesis ⟨proof⟩
qed
```

Note the use of the structured theorem form which puts the assumption $(x::nat) \leq 5$ into the proof context and names it `that` so that it can be input by `using that` into the structured proof and into its initial method `cases` which consumes it.

5.5 Induction

With induction a goal is proved by processing “all possible cases” for certain values which occur in it. If the goal can be proved for all these cases and the cases cover all possibilities, the goal holds generally. A specific technique is to assume the goal for some values and then prove it for other values. In this way it is possible to cover infinite value sets by proofs for only a finite number of values and steps from values to other values.

Perhaps the best known example of induction is a proposition which is proved for the natural number 0 and the step from a number n to its successor $n+1$, which together covers the whole infinite set of natural numbers.

As a (trivial) example consider the proposition $0 \leq n$. To prove that it is valid for all natural numbers n we prove the “base case” where n is 0 , which is true

because $0 \leq 0$. Then we prove the “induction step”, by assuming that $0 \leq n$ (the “induction hypothesis”) and proving that $0 \leq n+1$ follows, which is true because addition increases the value.

5.5.1 Induction Rules

As for case based reasoning (see Section 5.3) a goal is split into these cases by applying a meta rule. For induction the splitting can be done by a meta rule of the form

$$\llbracket P_1 \ ; \ \dots \ ; \ P_n \ \rrbracket \Longrightarrow \ ?P \ ?a$$

where every P_i is a rule of the form

$$\bigwedge y_{i,1} \ \dots \ y_{i,p_i} \cdot \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \Longrightarrow \ ?P \ \mathit{term}_i$$

where the assumptions $Q_{i,j}$ may contain the unknown $?P$ but no other unknowns, in particular not $?a$. A rule for a base case usually has no bound variables $y_{i,j}$ and no assumptions $Q_{i,j}$, at least the $Q_{i,j}$ do not contain $?P$. The remaining rules mostly have only a single assumption $Q_{i,j}$ which contains $?P$. Rules of this form are called “induction rules”. Note that the unknown $?a$ only occurs once in the conclusion of the meta rule and nowhere else. Like the case rules induction rules must be “prepared” for use, this is done by replacing $?a$ by a specific term term . This is the term for which all possible cases shall be processed in the goal. It must have the same type as all term_i in the P_i .

Usually, “all possible cases” means all values of the type of term , then term consists of a single variable which may adopt any values of its type. There are also other forms of induction where more complex terms are used, but they are not presented in this introduction, refer to other Isabelle documentations [7, 5] for them. In the following the unknown $?a$ will always be replaced by a variable x .

Applying Induction Rules

When a prepared induction rule is applied to a goal \mathcal{C} without bound variables and assumptions as described in Section 4.3.1, it unifies $?P \ x$ with the conclusion \mathcal{C} . This has the effect of abstracting \mathcal{C} to a (boolean) function PC by identifying all places where x occurs in \mathcal{C} and replacing it by the function argument. The function PC is then bound to the unknown $?P$, so that applying $?P$ to the argument x again yields \mathcal{C} . The function PC is the property to be proved for all possible argument values. Therefore the cases of the proof can be described by applying $?P$ to the terms term_i for the specific values in the rules P_i for the cases.

The application of the prepared rule results in the n goals

$$\begin{aligned}
& \bigwedge y_{1,1} \dots y_{1,p_1} \cdot \llbracket Q_{1,1}; \dots; Q_{1,q_1} \rrbracket \implies PC \text{ term}_1 \\
& \dots \\
& \bigwedge y_{n,1} \dots y_{n,p_n} \cdot \llbracket Q_{n,1}; \dots; Q_{n,q_n} \rrbracket \implies PC \text{ term}_n
\end{aligned}$$

The induction rule is only valid if the terms term_i cover all possible values of their associated type. If this has been proved the induction rule is available as a fact which can be applied. After preparing the induction rule for application, its conclusion $?P \ x$ matches all propositions which contain the variable x in one or more copies. It depends on the P_i in the rule whether splitting a specific goal with the induction rule is useful for proving the goal. The real power of induction rules emerges, when a $Q_{i,j}$ contains the unknown $?P$. Due to the type associated with $?P$ it must be applied to an argument $\text{term}_{i,j}$ of the same type as x and the term_i . Then the goal resulting from P_i states the property that if $Q_{i,j}$ holds when specialized to $PC \ \text{term}_{i,j}$, PC holds for term_i (an “induction step”). Thus, for covering the possible values of x , the step from $\text{term}_{i,j}$ to term_i can be repeated arbitrarily often which allows to cover some types with infinite value sets.

Example

An induction rule for the natural numbers is

$$\llbracket ?P \ 0; \bigwedge y. ?P \ y \implies ?P \ (y+1) \rrbracket \implies ?P \ ?a$$

P_1 is the base case, it has no variables and assumptions and only consists of the conclusion $?P \ 0$. P_2 binds the variable y , has one assumption $?P \ y$ and the conclusion $?P \ (y+1)$. P_1 covers the value 0 , P_2 covers the step from a value y to its successor $y+1$, together they cover all possible values of type nat .

To apply the rule to the goal $0 \leq n$, it must be prepared by substituting the variable n for the unknown $?a$. Then the rule conclusion $?P \ n$ is unified with the goal which abstracts the goal to the boolean function $PC = (\lambda i. 0 \leq i)$ and substitutes it for all occurrences of $?P$. This results in the rule instance $\llbracket (\lambda i. 0 \leq i) \ 0; \bigwedge y. (\lambda i. 0 \leq i) \ y \implies (\lambda i. 0 \leq i) \ (y+1) \rrbracket \implies (\lambda i. 0 \leq i) \ n$. By substituting the arguments in the function applications its assumption part yields the two goals

$$\begin{aligned}
& 0 \leq 0 \\
& \bigwedge y. 0 \leq y \implies 0 \leq (y+1)
\end{aligned}$$

which correspond to the base case and induction step as described above.

General Form of Induction Rules

Induction rules may even be more general than shown above. Instead of applying $?P$ to a single argument it may have several arguments and the

conclusion becomes $?P \text{ ?a}_1 \dots \text{ ?a}_r$. Also in the P_i every occurrence of $?P$ then has r terms as arguments. Such an induction rule is valid if it covers all possible cases for all combinations of the r argument values. Finally, in addition to the assumptions P_i an induction rule may have assumptions about the argument $?a$ or the arguments $?a_1 \dots ?a_r$.

Note, however, that there is no alternative syntax for induction rules, such as the **obtains** form for case rules.

5.5.2 The *induction* Method

Induction can be performed in a structured proof using the method *induction* in the form

```
induction x rule: name
```

where *name* is the name of a valid induction rule. The method prepares the rule by substituting the specified variable *x* for the unknown $?a$ 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 induction rule. For the general form depicted above the context for the i th case contains the variables $y_{i,1} \dots y_{i,p_i}$ and the assumptions $Q_{i,1}; \dots; Q_{i,q_i}$. The term abbreviation *?case* is defined for the case conclusion $PC \text{ term}_i$ which is to be proved for the case.

Input Facts

The *induction* method treats input facts like the empty method (see Section 4.1) and the *cases* method (see Section 5.3.2) by inserting them as assumptions into the original goal before splitting it.

Automatic Rule Selection

Also like the *cases* method the *induction* method supports automatic rule selection for the induction rule. This is only possible if $?P$ is applied to a single argument, which means that only one variable is specified in the method:

```
induction x
```

Then the rule is selected according to the type of *x*. In Isabelle HOL (see Section 6) most types have an associated induction rule.

Examples

The rule $\llbracket ?P \text{ True}; ?P \text{ False} \rrbracket \implies ?P \text{ ?a}$ is associated with type *bool*. Therefore induction can be applied to every proposition which contains a variable of type *bool*, such as the goal $b \wedge \text{False} = \text{False}$. Applying the method

```
induction b
```

will split the goal into the goals

```
False ∧ False = False
True  ∧ False = False
```

which cover all possible cases for *b*. Here, the type has only two values, therefore induction is not really needed.

As for the *cases* method (see Section 5.3.2) the names used for the contexts created by the *induction* method can be specified by attributing the induction rule. They can be determined from the proof skeleton which is displayed in the interactive editor when the cursor is positioned on the *induction* method (see Section 5.1.2).

If the induction rule $\llbracket ?P \ 0; \bigwedge y. ?P \ y \implies ?P \ (y+1) \rrbracket \implies ?P \text{ ?a}$ for the natural numbers has been proved and named *induct_nat* with case names *Start* and *Step*, a structured proof for the goal $0 \leq n$ may have the form

```
proof (induction n rule: induct_nat)
case Start
...
show ?case <proof>
next
case Step
...
show ?case <proof>
qed
```

The *induction* method creates the named contexts *Start* and *Step*. The former has no local variables and assumptions and binds *?case* to the proposition $0 \leq 0$, the latter has the local variable *y*, the assumption $0 \leq y$ named *Step* and binds *?case* to the proposition $0 \leq y + 1$.

If the rule *induct_nat* has been associated with type *nat* the rule specification may be omitted in the method:

```
proof (induction n)
...
```

5.5.3 Case Assumption Naming and the *induct* Method

As usual, the **case** statement uses the case name as name for the assumptions $Q_{i,1} \dots Q_{i,q_i}$ in the i th case or an explicit name may be specified for them (see Section 5.1.1). Additionally, the *induction* method arranges the named context for a case so that the set of assumptions is split into those which in the rule contain the unknown $?P$ and those which do not. These sets are separately named, so that they can be referenced individually.

The set of assumptions which originally contained $?P$ now contain an application of PC to a value $term_{i,j}$ and allow the step from this value to value $term_i$ by induction. These assumptions are called “induction hypotheses” and are named “*cname.IH*” where *cname* is the case name or the explicit name for the case assumptions. The other assumptions are independent from PC , they are additional hypotheses and are named “*cname.hyps*”. Both forms of names must be enclosed in quotes because the dot is not a normal name constituent.

For an example consider the induction rule $\llbracket ?P\ 0; ?P\ 1; \bigwedge y. \llbracket y \geq 1; ?P\ y \rrbracket \implies ?P\ (y+1) \rrbracket \implies ?P\ ?a$ with an additional base case for the value 1 and a step which always starts at value 1 or greater. If applied to the goal $0 \leq n$ the *induction* method produces the three goals

$$\begin{aligned} 0 &\leq 0 \\ 0 &\leq 1 \\ \bigwedge y. \llbracket y \geq 1; 0 \leq y \rrbracket &\implies 0 \leq (y+1) \end{aligned}$$

If the default case name 3 is used for the third case, the induction hypothesis $0 \leq y$ is named “3.IH” and the additional hypothesis $y \geq 1$ is named “3.hyps”. There is a method *induct* which behaves like *induction* with the only difference that it does not introduce the name “*cname.IH*”, it uses “*cname.hyps*” for all assumptions $Q_{i1} \dots Q_{iq_i}$, whether they contain $?P$ or not.

5.5.4 Goals with Assumptions

If the *induction* method would apply the prepared induction rule in the same way as the *rule* method to a goal $\llbracket A_1; \dots; A_m \rrbracket \implies C$ with assumptions it would unify $?P\ x$ only with the conclusion C and copy the assumptions A_1, \dots, A_m to all resulting goals unchanged. However, if x also occurs in one or more of the A_i this connection with C is lost after applying the prepared induction rule.

Consider the goal

$$4 < n \implies 5 \leq n$$

which is of the form

$$A \Rightarrow C$$

When applying the prepared induction rule for the natural numbers $\llbracket ?P \ 0; \bigwedge y. ?P \ y \Rightarrow ?P \ (y+1) \rrbracket \Rightarrow ?P \ n$ in the way of the *rule* method the conclusion will be matched which leads to the abstracted function $PC \equiv (\lambda i. \ 5 \leq i)$ and the resulting goals are

$$\begin{aligned} 4 < n &\Rightarrow 5 \leq 0 \\ \bigwedge y. \llbracket 4 < n; 5 \leq y \rrbracket &\Rightarrow 5 \leq (y+1) \end{aligned}$$

where the first goal is invalid. Although the second goal is valid, it shows that the relation between the variable n in the assumption and the variable y used in the induction rule has been lost.

For a goal with assumptions every occurrence of $?P$ in the rule, applied to a specific term must be replaced by PC applied to the term together with all assumptions which must also be adapted to the same term. Therefore the *induction* and *induct* methods work in a different way. They unify $?P \ x$ with the conclusion C and separately with every assumption A_l and thus additionally determine an abstracted function PA_l for every A_l . Then they replace every $?P \ term$ in a P_i or in a $Q_{i,j}$ in the rule by an application $PC \ term$ together with assumptions $PA_1 \ term; \dots; PA_m \ term$.

The assumptions for a direct occurrence of $?P \ term_i$ as conclusion in a P_i are added after all $Q_{i,j}$, so that the i th goal created by the *induction* method now has the form

$$\bigwedge y_{i,1} \dots y_{i,p_i}. \llbracket Q_{i,1}; \dots; Q_{i,q_i}; PA_1 \ term_i; \dots; PA_m \ term_i \rrbracket \Rightarrow PC \ term_i$$

Additionally, the assumptions for occurrences of a $?P \ term$ in a $Q_{i,j}$ must be added which usually can be done by replacing $Q_{i,j}$ by the rule $Q_{i,j}'$ of the form $\llbracket PA_1 \ term; \dots; PA_m \ term \rrbracket \Rightarrow Q_{i,j}'$ where $Q_{i,j}'$ results by substituting $?P \ term$ in $Q_{i,j}$ by $PC \ term$. If $?P$ is applied to several different terms in $Q_{i,j}$, several sets of corresponding assumptions are added in $Q_{i,j}'$.

Moreover, the *induction* method (and also the *induct* method) arranges the named contexts in a way that the assumptions $PA_1 \ term_i; \dots; PA_m \ term_i$ which originate from the goal are named by "*cname.prem*s" and can thus be referenced separate from the $Q_{i,j}$ which are named "*cname.hyps*" and possibly "*cname.IH*" as described above.

In the example above the *induction* method additionally unifies $?P \ n$ with the assumption $4 < n$ which yields the abstracted function $PA \equiv (\lambda i. \ 4 < i)$ and produces the goals

$$\begin{aligned} 4 < 0 &\Rightarrow 5 \leq 0 \\ \bigwedge y. \llbracket 4 < y \Rightarrow 5 \leq y; 4 < (y+1) \rrbracket &\Rightarrow 5 \leq (y+1) \end{aligned}$$

Here $4 < (y+1)$ results from applying PA to $(y+1)$ and $4 < y$ results from adding PA applied to y as assumption to the assumption $?P \ y$ from the rule.

If the default case name 2 is used for the second case, the case assumption $4 < y \implies 5 \leq y$ will be named "*2.IH*" and the case assumption $4 < (y+1)$ will be named "*2.prem*" by the **case** statement.

Like the **cases** method the methods *induction* and *induct* insert input facts as assumptions into the goal before they process the goal assumptions in the described way. Therefore both methods can be used both in proof scripts, where facts are stored as rule assumptions in the goal state, and in structured proofs where facts are stored in the proof context and are input to the initial methods of subproofs.

Other than for the **cases** method it is not possible to pass the induction rule as input fact to the methods *induction* and *induct*.

5.5.5 Induction with Elimination

Like case rules, induction rules can be extended to include elimination (see Section 5.4.2). Such induction rules have an additional first assumption which is used as major premise to unify with a goal assumption and eliminate it, before processing the remaining goal assumptions and splitting the goal into cases.

Like case rules such extended induction rules must be attributed by *[consumes 1]*, then the methods *induction* and *induct* apply elimination before doing the rest of processing.

As example consider the rule $\llbracket ?a \geq 42; ?P\ 42; \bigwedge y. ?P\ y \implies ?P\ (y+1) \rrbracket \implies ?P\ ?a$. It uses 42 as its base case and can thus only prove a property for numbers equal to or greater than 42. The major premise restricts *?a* to these cases. If this rule has been proved and named *induct_nat42* it may be applied with elimination as initial method of a structured proof for the goal $4 < n \implies 5 \leq n$ by

```
proof (induction n rule: induct_nat42[consumes 1])
```

If the fact $n \geq 42$ is available as first input fact the application will be successful, consume that fact, and split the goal into the goals

$$4 < 42 \implies 5 \leq 42$$

$$\bigwedge y. \llbracket 4 < y \implies 5 \leq y; 4 < (y+1) \rrbracket \implies 5 \leq (y+1)$$

In contrast to the **cases** method the methods *induction* and *induct* will also consume the first assumption present in the goal, if it unifies with the major premise of the induction rule and no input facts are present.

5.5.6 Arbitrary Variables

If the goal contains bound variables, i.e., is of the form $\bigwedge x_1 \dots x_k. \llbracket A_1; \dots; A_m \rrbracket \implies C$ these variables may occur in the assumptions A_1, \dots, A_m

and the conclusion C . When these are instantiated for the occurrences of $?P$ in the rule as described above, every such instance needs its own separate copy of bound variables $x_1 \dots x_k$.

Consider the goal

$$\bigwedge c. n < c \implies 2 * n < 2 * c$$

When applying the prepared induction rule for the natural numbers $\llbracket ?P \ 0; \bigwedge y. ?P \ y \implies ?P \ (y+1) \rrbracket \implies ?P \ n$ in the way described above, the variable c must be bound separately for the occurrences $?P \ y$ and $?P \ (y+1)$ because it need not denote the same value in both positions. This can be accomplished by creating the goals

$$\begin{aligned} &\bigwedge c. 0 < c \implies 2 * 0 < 2 * c \\ &\bigwedge y \ c. \llbracket \bigwedge c. y < c \implies 2 * y < 2 * c; y+1 < c \rrbracket \implies 2 * (y + 1) < 2 * c \end{aligned}$$

with two nested bindings of c .

The methods *induction* and *induct* treat explicit bindings of variables in the goal in this way. However, if variables are not bound explicitly in the goal (i.e., they are free in the goal), there are two possible meanings: in a theorem all variables are implicitly bound and therefore need a separate binding for every occurrence of $?P$, whereas in a local goal statement variables may have been fixed in the enclosing context, then they must denote the same value for every occurrence of $?P$ and must not be bound separately. Since it is difficult for Isabelle to determine the correct treatment of free variables automatically, it may be specified explicitly by the proof writer. The free variables which do not denote a fixed value from the context but an “arbitrary” value used only locally in the goal can be specified to the *induction* method in the form

induction x *arbitrary*: $x_1 \dots x_k$ *rule*: *name*

and in the same form for the *induct* method.

In particular, if a theorem or goal statement is specified in structured form the free variables are not bound in the goal but in the outermost proof context (see Section 3.1.3) and the goal only consists of the conclusion C . To apply induction as initial proof method the assumptions must be input to it and the variables must be specified as arbitrary:

```
theorem
  fixes  $x_1 \dots x_k$ 
  assumes " $A_1$ " ... " $A_m$ "
  shows " $C$ "
using assms
proof (induction ... arbitrary:  $x_1 \dots x_k$  ...) ... qed
```

Chapter 6

Isabelle/HOL Basics

The basic mechanisms described in the previous Chapters 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 which are available in HOL for arbitrary types.

The abbreviation “HOL” is used in the logics community to denote the general concept of higher order logics. In this document we use it specifically to denote the implementation as object logic in Isabelle which is named Isabelle/HOL.

Isabelle/HOL consists of the session HOL/HOL (see Section 1.1.2) and several extensions. Most theories in HOL/HOL are imported by the theory *Main*. By importing *Main* the content of all these theories becomes available. The most basic theory in HOL/HOL is also named *HOL*. There are additional theories in HOL/HOL which are not imported by *Main*, such as for rational and real numbers.

This introduction only covers some of the theories imported by *Main*.

6.1 Predicates and Relations

Basically HOL uses the type $\text{'a} \Rightarrow \text{'b}$ of functions provided by Isabelle (see Section 2.2.3) to represent predicates and relations in the usual way.

6.1.1 Predicates

A predicate on values of arbitrary types t_1, \dots, t_n is a function of type $t_1 \Rightarrow \dots \Rightarrow t_n \Rightarrow \text{bool}$. It may be denoted by a lambda term $\lambda x_1 \dots x_n.$

bterm where *bterm* is a term of type *bool* which may contain free occurrences of the variables x_1, \dots, x_n . Note that also a predicate defined as a named constant *P* can always be specified by the lambda term $\lambda x_1 \dots x_n. P\ x_1 \dots x_n$.

Note that the concept of predicates actually depends on the specific HOL type *bool*, which is introduced in Section 8.1.

6.1.2 Unary Predicates and Sets

Unary predicates of type $t \Rightarrow bool$ are equivalent to sets of type *t set*. There is a 1-1 correspondence between a predicate and the set of values for which the predicate value is *True*. Actually, the HOL type constructor *set* described in Section 8.4 is defined based on this correspondence. See that section for more information about denoting and using sets.

As a convention HOL often provides a predicate and rules about it in both forms as a set named *name* and as a predicate named *namep* or *nameP*. Then usually every fact *F* containing such predicates in set form can be converted to the corresponding fact containing them in predicate form by applying the attribute *to_pred* as in *F[to_pred]* and vice versa by applying the attribute *to_set*.

6.1.3 Relations

A binary relation between values of arbitrary types t_1 and t_2 is a binary predicate of type $t_1 \Rightarrow t_2 \Rightarrow bool$. As binary functions the relations in HOL often have an alternative operator name of the form *(op)* which supports specifying applications in infix form *x op y* (see Section 2.2.4).

By partial application (see Section 2.2.4) the first argument of a relation *R* can be fixed to yield the unary predicate *(R x)* on the second argument. For operators this must be done using the operator name in the form *((op) x)*, partial application cannot be written by omitting an argument on one side of the infix operator.

Relation Operations

HOL provides the composition operation for relations. It is defined as the function

$$\text{relcomp} :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow ('a \Rightarrow 'c \Rightarrow bool) \equiv \\ \lambda R_1\ R_2\ a\ c. \exists b. R_1\ a\ b \wedge R_2\ b\ c$$

The quantifier \exists and the boolean operator \wedge have the usual meaning, for more information about them see Section 8.1.3. HOL also provides the operator name *oo* (two capital letters O) for infix notation, as in $R_1\ oo\ R_2$.

HOL provides the converse of a relation, defined as the function

```
conversep :: ('a ⇒ 'b ⇒ bool) ⇒ ('b ⇒ 'a ⇒ bool) ≡
  λR b a. R a b
```

with the alternative notation R^{-1-1} for `conversep R`. Note that although the superscript is applied twice it denotes only a single conversion. The single superscript is used for the conversion of a different form of relations (see Section 6.6.2).

HOL provides functions for the domain and range predicates of a relation:

```
Domainp :: ('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ bool) ≡ λR a. ∃ b. R a b
Rangep :: ('a ⇒ 'b ⇒ bool) ⇒ ('b ⇒ bool) ≡ λR b. ∃ a. R a b
```

HOL provides predicates for typical properties of relations:

```
left_total :: ('a ⇒ 'b ⇒ bool) ⇒ bool ≡ λR. ∀ x. ∃ y. R x y
left_unique :: ('a ⇒ 'b ⇒ bool) ⇒ bool ≡
  λR. ∀ x y z. R x z → R y z → x = y
right_total :: ('a ⇒ 'b ⇒ bool) ⇒ bool ≡ λR. ∀ y. ∃ x. R x y
right_unique :: ('a ⇒ 'b ⇒ bool) ⇒ bool ≡
  λR. ∀ x y z. R x y → R x z → y = z
bi_total :: ('a ⇒ 'b ⇒ bool) ⇒ bool ≡
  λR. left_total R ∧ right_total R
bi_unique ('a ⇒ 'b ⇒ bool) ⇒ bool ≡
  λR. left_unique R ∧ right_unique R
```

These predicates test whether a relation R is a function (`right_unique`), surjective (`right_total`), injective (`left_unique`), total (`left_total`), an injective function (`bi_unique`), or surjective and total (`bi_total`). R is a bijection if `bi_total R ∧ bi_unique R`.

Relations As Set-Valued Functions

Since the unary predicate $(R\ x)$ is equivalent to a set, every binary relation of type $t_1 \Rightarrow t_2 \Rightarrow \text{bool}$ is equivalent to a set-valued function of type $t_1 \Rightarrow (t_2\ \text{set})$. It maps every value of type t_1 to the set of related values of type t_2 . HOL extends the convention described above and often provides relations named `namep` or `nameP` also as set-valued functions named `name`.

Note that HOL introduces the specific type constructor `rel` (see Section 8.8), where the values are equivalent to binary relations on a single type t .

More generally, n -ary relations for $n > 2$ are directly represented by n -ary predicates. Every n -ary relation is equivalent to an $(n-1)$ -ary set-valued function.

6.2 Equality, Orderings, and Lattices

6.2.1 The Equality Relation

HOL introduces the equality relation as a function

```
HOL.eq :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
```

with the alternative operator name (=) for infix notation.

Inequality can be denoted by

```
not_equal :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
```

with the alternative operator name (\neq) for infix notation.

Both functions are polymorphic and can be applied to terms of arbitrary type, however, they can only be used to compare two terms which have the same type. Therefore the proposition

```
True  $\neq$  1
```

is syntactically wrong and Isabelle will signal an error for it.

Moreover, in a term such as $term_1 = term_2$ or $term_1 \neq term_2$ no type information can be derived for $term_1$ and $term_2$ other than that they must have the same type. There may be relations with the same semantics but for operands of a specific type, then it is possible to derive the operand types. An example is the relation *iff* with operator name (\longleftrightarrow) which is equal to (=) but only defined for operands of type *bool*. Therefore for the term $term_1 \longleftrightarrow term_2$ HOL automatically derives that $term_1$ and $term_2$ have type *bool*. There is also a restricted form of equality

```
eq_onp :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool
```

which can only be used to compare values which satisfy a given predicate. Thus *eq_onp* *P* *x* *y* will only be *True* if *x* and *y* are equal and additionally *P* *x* holds.

6.2.2 The Ordering Relations

HOL also introduces two ordering relations as functions

```
less :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool  
less_eq :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
```

with the alternative operator names (<) and (\leq) for infix notation and the abbreviations *greater* and *greater_eq* for reversed arguments with operator names (>) and (\geq).

Based on these relations HOL also introduces the functions

```

min :: 'a ⇒ 'a ⇒ 'a
max :: 'a ⇒ 'a ⇒ 'a

```

in the usual way, i.e. if $a \leq b$ then $\text{min } a \ b$ is a , otherwise b . HOL also introduces the functions

```

Min :: 'a set ⇒ 'a
Max :: 'a set ⇒ 'a

```

for the minimum or maximum of all values in a set. The value of *Min* and *Max* is underspecified if applied to an empty or infinite set.

All these functions are polymorphic but are not available for arbitrary types. They are overloaded only for those types which have ordered values. However, this is the case for most basic types introduced by HOL and described in Chapter 8. If an ordering function is used for a type for which it is not available an error message of the form “No type arity ...” is signaled.

Basically, these are only syntactic definitions, no rules about orderings are implied by them. For some of its predefined types, such as type *nat*, HOL provides more specific specifications by overloading.

As for $(=)$ the type of *term*₁ and *term*₂ cannot be derived in a term such as *term*₁ < *term*₂. There are relations with the same semantics but specific operand types such as (\subseteq) for the relation (\leq) on sets (see Section 8.4).

6.2.3 Lattice Operations

HOL introduces the two lattice operations

```

inf :: 'a ⇒ 'a ⇒ 'a
sup :: 'a ⇒ 'a ⇒ 'a

```

with the alternative operator names (\sqcap) and (\sqcup) for infix notation together with the lattice constants

```

top :: 'a
bot :: 'a

```

with the alternative names (\top) and (\perp) (in the interactive editor available in the Symbols panel in the Logic tab).

Additionally there are the lattice operations for all values in a set (see Section 8.4)

```

Inf :: 'a set ⇒ 'a
Sup :: 'a set ⇒ 'a

```

with the alternative operator names (\bigcap) and (\bigcup) for prefix notation.

HOL does not provide the six alternative names automatically. To make them available, the command

unbundle *lattice_syntax*

must be used on theory level. It is available after importing the theory *Main* (see Section 1.1.2).

Like the ordering functions (see Section 6.2.2) the lattice operations and constants are polymorphic but are not available for arbitrary types. They are overloaded only for those types which have a corresponding structure. For example, type *nat* has the *bot* value (which is equal to 0), but no *top* value. If a lattice operation or constant is used for a type for which it is not available an error message of the form “No type arity ...” is signaled.

As for equality and ordering relations, because the lattice operations and constants are overloaded it is not possible to derive the type for valid operands. Again, there are operations and constants with more specific operand types, such as (\cap) for (\sqcap) on sets where HOL automatically derives the operand types.

6.3 Description Operators

A description operator selects a value from all values which satisfy a given unary predicate.

Description operators use the binder syntax of the form *OP x. term* (see Section 2.2.5). Like a lambda term it locally binds a variable *x* which may occur in *term*.

6.3.1 The Choice Operator

An arbitrary value satisfying the given predicate $\lambda x. bterm$ can be denoted by

SOME x. bterm

Only a single variable may be specified after *SOME*, however, as in a lambda term a type may be specified for it.

The value denoted by the term is underspecified in the sense of Section 2.2.2. The only information which can be derived for it is that it satisfies the predicate $\lambda x. bterm$. If there is no value which satisfies the predicate not even this property may be derived.

The operator *SOME* is equivalent to the famous Hilbert choice operator. HOL includes the axiom of choice and provides the operator on this basis.

6.3.2 The Definite Description Operator

If only one value satisfies the given predicate $\lambda x. bterm$ this value can be denoted by

THE $x. bterm$

As for *SOME* only a single variable may be specified with an optional type specification.

The value denoted by the term is also underspecified. However, after proving that there exists a value v which satisfies the predicate $\lambda x. bterm$ and that all values satisfying the predicate are equal it is possible to prove that *THE* $x. bterm = v$.

6.3.3 The Least and Greatest Value Operators

If the values satisfying a predicate $\lambda x. bterm$ are ordered, the least or greatest of these values can be denoted by the terms

LEAST $x. bterm$

GREATEST $x. bterm$

Only a single variable with an optional type specification is useful to be specified.

The operators are defined using *THE* to return the value x which satisfies the predicate and $x \leq y$ or $x \geq y$ holds, respectively, for all values y which also satisfy the predicate.

Also, if the values are ordered but there is no single least or greatest value among them the resulting value is underspecified. For example, the term *GREATEST* $n::nat. n > 0$ is correct and denotes a value of type *nat*, although no information is available about it.

6.4 Undefined Value

HOL introduces the undefined value

undefined $:: 'a$

which is overloaded for arbitrary types. It is completely underspecified as described in Section 2.2.2, i.e., no further information is given about it.

Despite its name it is a well defined value for every type $'a$. It is typically used for values which are irrelevant, such as in the definition

definition $f :: "nat \Rightarrow nat"$ **where** $"f\ x \equiv undefined"$

Although the function f looks like a completely undefined function, it is not possible to define true partial functions this way. Functions in Isabelle are always total. Function f maps every natural number to the (same) value *undefined*, which is of type *nat*, but it cannot be proved to be equal to a specific natural number such as 1 or 5. However, since it is a single value the equality $f\ x = f\ y$ holds for arbitrary x and y of type *nat*.

6.5 Let Terms

HOL extends the inner syntax for terms described in Section 2.2 by terms of the following form

let $x_1 = \text{term}_1; \dots; x_n = \text{term}_n$ *in* *term*

where x_1, \dots, x_n are variables. The variable bindings are sequential, i.e., if $i < j$ variable x_i may occur in term_j and denotes term_i there. In other words, the scope of x_i are the terms term_j with $i < j$ and the *term*. If x_i and x_j are the same variable, the binding of its second occurrence shadows the binding of the first and ends the scope of the first occurrence.

Let terms are useful to introduce local variables as abbreviations for sub-terms.

Don't confuse the *let* term with the **let** statement described in Section 3.12.1. Although they are used for a similar purpose there are several differences: A *let* term belongs to the inner syntax of the object logic HOL, the **let** statement belongs to the meta-level of the outer syntax. Moreover, the **let** statement uses unification to bind sub-terms to unknowns in a term pattern, a *let* term only binds explicitly specified terms to single variables.

The let term specified above is an alternative syntax for the nested let terms

let $x_1 = \text{term}_1$ *in* (...) (*let* $x_n = \text{term}_n$ *in* *term*) ...)

and a single let term

let $x = \text{term}'$ *in* *term*

is an alternative syntax for the function application term

Let *term'* ($\lambda x. \text{term}$)

Here *Let* is the polymorphic function

Let $:: 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b \equiv \lambda x f. f x$

which simply applies its second argument to its first argument.

Occurrences of let terms are usually not automatically resolved by substituting the bound term for the variable. Therefore a proposition like (*let* $x = a+b$ *in* $(x*x)$) = $((a+b)*(a+b))$ cannot be proved by the simplifier (or other methods including simplification like *auto*, see Sections 4.6 and 4.7.1). To resolve it, the simplifier must be configured by adding the definitional equation of *Let* as *simp add: Let_def*.

6.6 Tuples

HOL supports type expressions of the form $t_1 \times \dots \times t_n$ for arbitrary types t_1, \dots, t_n . They denote the type of n -tuples where the i th component has type t_i . Here the \times is the “times operator” available in the interactive editor in the Symbols panel in the Operator tab. An alternative syntax is $t_1 * \dots * t_n$ using the ASCII star character. As an example the type `nat \times bool` is the type of pairs of natural numbers and boolean values. The type `nat \times 'a \times bool` is the polymorphic type of triples of a natural number, a value of the arbitrary type denoted by the type parameter `'a`, and a boolean value. As usual, all these type expressions are part of the inner syntax.

Values for n -tuples of type $t_1 \times \dots \times t_n$ are denoted in inner syntax as terms of the form $(term_1, \dots, term_n)$ where $term_i$ is a term of type t_i and the parentheses and the comma belong to the syntax.

The selector function `fst` can be used to access the first component of a tuple value, the selector function `snd` can be used to access the remaining tuple after removing the first component (which yields the single second component if applied to a pair) (see Section 8.6.2).

6.6.1 Function Argument Tuples

Every n -ary function of type $t_1 \Rightarrow \dots \Rightarrow t_n \Rightarrow t$ (see Section 2.2.4) is equivalent to a function of type $(t_1 \times \dots \times t_n) \Rightarrow t$ with n -tuples as argument values, which is the common way in mathematics to represent a function with n arguments. There is a 1-1 correspondence between functions of these two forms. The first form is called the “curried” form and the second form with tuples as arguments is called the “uncurried” form (named after Haskell Curry who introduced the first form for n -ary functions). Note that for unary functions both forms are the same.

Section 8.7 describes means to convert between both forms and other ways how to work with them.

6.6.2 Relations as Tuple Sets

For an n -ary relation of type $t_1 \Rightarrow \dots \Rightarrow t_n \Rightarrow bool$ (see Section 6.1.3) the uncurried form is a predicate of type $(t_1 \times \dots \times t_n) \Rightarrow bool$. Since all arguments together are represented by a single tuple, this predicate is equivalent to a set of type $(t_1 \times \dots \times t_n)$ `set` (see Section 6.1.2) where the elements in the set are tuples. This is the usual form of representing relations in mathematics.

HOL extends the convention described in Section 6.1.2 of providing relations named `namep` or `nameP` also as set-valued functions named `name` by alterna-

tively using a tuple set named *name*. Moreover it provides the relation operations (see Section 6.1.3) for binary relations as tuple sets: composition as function *relcomp* with operator name *O* (a single capital O), and conversion as function *converse* with alternative notation R^{-1} for *converse R*.

6.7 Inductive Definitions

HOL supports inductive definitions of predicates as content in theories. An inductive definition defines a predicate by derivation rules which allow to derive the predicate values for some arguments from the values for other arguments. An inductive definition for a *k*-ary predicate has the form

inductive *name* :: " $t_1 \Rightarrow \dots \Rightarrow t_k \Rightarrow \text{bool}$ "
where $P_1 \mid \dots \mid P_n$

with derivation rules P_i . The type specification for the predicate may be omitted if it can be derived from the use of the defined predicate in the rules P_i .

6.7.1 The Defining Rules

The P_i are derivation rules which may be specified in inner syntax of the form

$\bigwedge x_{i,1} \dots x_{i,p_i} \cdot \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \Longrightarrow \text{name } \text{term}_{i,1} \dots \text{term}_{i,k}$

where the conclusion is always an application of the defined predicate to *k* argument terms. The defined predicate *name* may occur in arbitrary ways in the rule assumptions $Q_{i,j}$, but not in the argument terms $\text{term}_{i,j}$. The rule separators */* belong to the outer syntax, thus every rule must be separately quoted.

An example is the following inductive definition for an evenness predicate:

inductive *evn* :: " $\text{nat} \Rightarrow \text{bool}$ " **where**
 "*evn*(0)" */* " $\bigwedge n. \text{evn}(n) \Longrightarrow \text{evn}(n+2)$ "

Alternative Rule Forms

As for derivation rules specified in theorems (see Section 2.6.1) the explicit bindings of the variables $x_{i,1}, \dots, x_{i,p_i}$ are optional, variables occurring free in the assumptions or the conclusion are always automatically bound. As usual, types may be specified for (some of) the variables, so explicit bindings can be used as a central place for specifying types for the variables, if necessary. The equivalent example with omitted binding is

```

inductive evn :: "nat  $\Rightarrow$  bool" where
  "evn(0)" | "evn(n)  $\implies$  evn(n+2)"

```

Alternatively a rule P_i may be specified in the structured form described in Section 2.5.6:

```

"name termi,1 ... termi,k" if "Qi,1" ... "Qi,qi" for xi,1 ... xi,pi

```

where the assumptions and variables may be grouped by **and** and types may be specified for (some of) the variable groups, however, no names may be specified for assumption groups, because no proof is specified for the rules. In this form the example is written

```

inductive evn :: "nat  $\Rightarrow$  bool" where
  "evn(0)" | "evn(n+2)" if "evn(n)"

```

The rules P_i are introduction rules (see Section 4.3.4) for the defined predicate **name** in the sense that they introduce a specific application of **name**, possibly depending on other applications of **name**. An inductive definition turns the rules P_i to valid facts “by definition” under the fact set name **name.intros**. Explicit names for (some of) the rules may be specified in the form

```

inductive name :: "t1  $\Rightarrow$  ...  $\Rightarrow$  tk  $\Rightarrow$  bool"
where rname1: P1 | ... | rnamen: Pn

```

Using this form the example can be written as

```

inductive evn :: "nat  $\Rightarrow$  bool" where
  zero: "evn(0)" | step: "evn(n)  $\implies$  evn(n+2)"

```

The fact set **name.intros** together with other facts introduced by an inductive definition can be displayed using the “Print Context” tab in the Query panel as described in Section 2.6.6.

Note that the syntax of the defining rules only allows to specify when the defined predicate is **True**, not when it is **False**. In particular, a rule conclusion may not have the form of a negated application \neg **name** term_{i,1} ... term_{i,k}.

Semantics of an Inductive Definition

To determine whether an application of the predicate to argument values yields the value **True** the defining rules P_i can be applied as introduction rules in backward reasoning steps (see Section 4.3.2) until the predicate is not present anymore. However, depending on the rules, this may not succeed. If the defining rules P_i would be the only information available about the defined predicate, it would be underspecified for such cases. The specific

property of an *inductive* definition is the additional regulation that in all such cases the predicate value is *False*. In other words, if it is not possible to prove that the predicate value is *True* by using the defining rules it is defined to be *False*. This can happen in two ways.

In the first case there is no rule for which the conclusion unifies with the predicate application term. Consider the trivial inductive definition

```
inductive uspec1 :: "nat  $\Rightarrow$  bool" where "uspec1 3" / "uspec1 4"
```

The rules only unify with applications of *uspec*₁ to the argument values 3 and 4, for them it can be derived that the predicate value is *True*. For all other arguments the value is *False*.

In the example the value of (*evn* 3) is *False* because an application of the defining rule *step* as backwards reasoning step leads to (*evn* 1) which does not unify with the conclusion of either rule.

In the second case there are rule conclusions which unify with the predicate application term, but there is no finite sequence of backward rule application steps which removes all occurrences of the predicate. Consider the inductive definition

```
inductive uspec2 :: "nat  $\Rightarrow$  bool" where "uspec2 i  $\Longrightarrow$  uspec2 i"
```

Although the rule is trivially valid it cannot be used to prove that *uspec*₂ is *True* for any argument value. Therefore its value is *False* for all arguments.

Monotonicity Properties

Actually, this way of implicit specification when the predicate value is *False* is only possible if all assumptions $Q_{i,j}$ used in the rules satisfy a “monotonicity” property for the defined predicate. HOL is able to prove these properties for most forms of the assumptions $Q_{i,j}$ automatically and then prove the rules themselves and additional rules to be facts. In the interactive editor these proof activities are displayed in the Output panel.

A common case where a rule assumption $Q_{i,j}$ does not satisfy the monotonicity property is if it contains an application of the defined predicate *name* in negated form. Consider the inductive definition

```
inductive uspec3 :: "nat  $\Rightarrow$  bool" where " $\neg$  uspec3 i  $\Longrightarrow$  uspec3 i"
```

Although if *uspec*₃ is defined as $\lambda i. \text{True}$ it would satisfy the defining rule, the rule cannot be used to prove that *uspec*₃ is *True* for any argument by applying the rule in backward reasoning steps. Isabelle will signal an error for the inductive definition, stating that the monotonicity proof failed for the assumption $\neg \text{uspec}_3 i$. Note that negation may also occur in other syntactic forms like *uspec*₃ *i* = *False* or *uspec*₃ *i* \Longrightarrow *i* > 0.

6.7.2 Fixed Arguments

It may be the case that one or more arguments of the defined predicate are “fixed” , i.e. in all defining rules the values of these arguments in the conclusion are the same as in all assumptions. If this is the case for the first m arguments the inductive definition can be specified in the form

```
inductive name :: "t1 ⇒ ... ⇒ tk ⇒ bool"
for y1 ... ym
where P1 / ... / Pn
```

The variables y_i may be grouped by **and** and types may be specified for (some of) the groups.

Then the defining rules P_i must have the form

$$\bigwedge x_{i,1} \dots x_{i,p_i} . \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \Longrightarrow \textit{name } y_1 \dots y_m \textit{ term}_{i,m+1} \dots \textit{term}_{i,k}$$

and every occurrence of **name** in the $Q_{i,j}$ must also have $y_1 \dots y_m$ as its first m arguments.

Specifying fixed arguments is optional, it does not have any effect on the defined predicate, however it makes the rules provided by HOL about the predicate simpler (see below).

As an example consider the inductive definition of a divides predicate

```
inductive divides :: "nat ⇒ nat ⇒ bool"
for m
where "divides m 0" / "divides m n ⇒ divides m (n+m)"
```

6.7.3 The cases Rule

The general form of inductive definition constructs and automatically proves the additional rule

$$\llbracket \textit{name } ?a_1 \dots ?a_k; RA_1; \dots; RA_n \rrbracket \Longrightarrow ?P$$

where every RA_i has the form

$$\bigwedge x_{i,1} \dots x_{i,p_i} . \llbracket ?a_1 = \textit{term}_{i,1}; \dots; ?a_k = \textit{term}_{i,k}; Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \Longrightarrow ?P$$

and names it **name.cases**.

This rule has the form of an elimination rule for the predicate as described in Section 5.4.2. The major premise is the application **name** $?a_1 \dots ?a_k$ of the defined predicate to arbitrary arguments. When the rule is applied by the methods **erule** or **cases** to a goal it removes a predicate application from the goal assumptions or the input facts, respectively, and splits the goal into cases according to the defining rules of the predicate. The named cases created by the **cases** method are named by numbers starting with 1.

The **cases** rule for the evenness example is **evn.cases**:

$$\llbracket \text{evn } ?a; ?a = 0 \implies ?P; \bigwedge n. \llbracket ?a = n + 2; \text{evn } n \rrbracket \implies ?P \rrbracket \implies ?P$$

Since it is a case rule it can be displayed by **print_statement** in the alternative form (see Section 5.3.3):

fixes $a_1 \dots a_k$
assumes "*name* $a_1 \dots a_k$ "
obtains $C_1 \mid \dots \mid C_n$

where every case C_i has the form

$$x_{i,1} \dots x_{i,p_i} \text{ where } "a_1=\text{term}_{i,1}" \dots "a_k=\text{term}_{i,k}" "Q_{i,1}" \dots "Q_{i,q_i}"$$

Together with *name.cases* the similar rule *name.simps* is provided. It is an equation which substitutes an arbitrary application of *name* by a disjunction of the cases according to the defining P_i and may be used by the simplifier (see Section 4.6.3). It is not added to the simpset automatically, because its application may not terminate (e.g., for *uspec₂* as below).

Effect of Fixed Arguments

If the first m predicate arguments are fixed by *for* $y_1 \dots y_m$ as above, *name.cases* has the simpler form

$$\llbracket \text{name } ?y_1 \dots ?y_m ?a_1 \dots ?a_{(k-m)}; RA_1; \dots; RA_n \rrbracket \implies ?P$$

where every RA_i has the form

$$\bigwedge x_{i1} \dots x_{ip_i}. \llbracket ?a_1=\text{term}_{i(m+1)}; \dots; ?a_{(k-m)}=\text{term}_{ik}; Q_{i1}'; \dots; Q_{iq_i}' \rrbracket \implies ?P$$

and in the Q_{ij}' all applications of *name* have the unknowns $?y_1 \dots ?y_m$ as their first arguments. In other words, the unknowns from the major premise are directly used in place of the first m arguments in all predicate applications without the need of corresponding equations. Therefore the rule *divides.cases* for the divides predicate above is

$$\llbracket \text{dvds } ?m ?a; ?a = 0 \implies ?P; \bigwedge n. \llbracket ?a = n + ?m; \text{dvds } ?m n \rrbracket \implies ?P \rrbracket \implies ?P$$

Using the cases Rule

Due to its form the rule *name.cases* includes information about the implicit definition of *False* predicate values. If a predicate application does not unify with the conclusion of any P_i there will be atleast one assumption $?a_j=\text{term}_{ij}$ in every RA_i which is false and thus makes RA_i solved as a goal. Together that proves that the predicate application implies arbitrary propositions which means that it is *False*.

Consider the predicate *uspec₁* defined as above. The rule *uspec₁.cases* is

$\llbracket \text{uspec}_1 \text{ ?a}; \text{ ?a}=3 \implies \text{ ?P}; \text{ ?a}=4 \implies \text{ ?P} \rrbracket \implies \text{ ?P}$

The proposition $\text{uspec}_1 \ 0 = \text{False}$ is equivalent to $\text{uspec}_1 \ 0 \implies \text{False}$ (because the other direction is trivially valid) which can be proved as follows:

```
theorem "uspec1 0  $\implies$  False"
  apply (erule uspec1.cases)
  by simp_all
```

The application of $\text{uspec}_1.\text{cases}$ creates the goals $0=3 \implies \text{False}$ and $0=4 \implies \text{False}$ which are solved by the simplifier.

Note that this does not work for the predicate uspec_2 . For it $\text{uspec}_2.\text{cases}$ is

$\llbracket \text{uspec}_2 \text{ ?a}; \bigwedge i. \llbracket \text{ ?a} = i; \text{ uspec}_2 \ i \rrbracket \implies \text{ ?P} \rrbracket \implies \text{ ?P}$

and the application by erule to $\text{uspec}_2 \ 0 \implies \text{False}$ yields the goal $\bigwedge i. \llbracket 0 = i; \text{ uspec}_2 \ i \rrbracket \implies \text{False}$ which is equivalent to the original goal. Therefore even an iterated application by erule will not solve the goal in a finite number of steps.

The *cases* Rule in Structured Proofs

The rule $\text{name}.\text{cases}$ is attributed by $[\text{consumes } 1]$, so that it can be applied by the cases method in structured proofs (see Section 5.4.4). The example above can be written

```
theorem "False" if "uspec1 0"
  using that
  proof (cases rule: uspec1.cases) qed simp_all
```

Note the use of the structured theorem form to put $\text{uspec}_1 \ 0$ into the proof context with name *that* so that it is input by **using that** into the structured proof and its initial method *cases* which consumes it.

Moreover, the rule $\text{name}.\text{cases}$ is associated to the defined predicate name in the following way: if the first input fact to the cases method has an inductively defined predicate name as its outermost function, the rule $\text{name}.\text{cases}$ is the default rule applied by the method if no rule is explicitly specified. So the proof for *"False" if "uspec₁ 0"* above can be abbreviated to

```
proof cases qed simp_all
```

or even shorter (see Section 3.3.2):

```
by cases simp_all
```

Note that this only works if the consumed predicate application is taken as input fact, not if it is a goal assumption as in the proof script above.

6.7.4 The Induction Rule

The general form of inductive definition also constructs and automatically proves the additional rule

$$\llbracket \text{name } ?a_1 \dots ?a_k; RA_1; \dots; RA_n \rrbracket \Longrightarrow ?P ?a_1 \dots ?a_k$$

where every RA_i has the form

$$\bigwedge x_{i,1} \dots x_{i,p_i} . \llbracket Q_{i,1}' ; \dots ; Q_{i,q_i}' \rrbracket \Longrightarrow ?P \text{ term}_{i,1} \dots \text{term}_{i,k}$$

and in the $Q_{i,j}'$ every application $\text{name } t_1 \dots t_k$ of the predicate to argument terms is replaced by the conjunction $\text{name } t_1 \dots t_k \wedge ?P t_1 \dots t_k$. The rule is named *name.induct*.

This rule has the form of an induction rule extended by elimination for the predicate as described in Section 5.5.5. The major premise is the application $\text{name } ?a_1 \dots ?a_k$ of the defined predicate to arbitrary arguments. The rule is attributed by *[consumes 1]*. When it is applied by the methods *induct* or *induction* to a goal it consumes a predicate application from the input facts or the goal assumptions and splits the goal into cases according to the defining rules of the predicate. The named cases created by the induction methods are named by numbers starting with 1.

The induction rule for the evenness example is *evn.induct*:

$$\llbracket \text{evn } ?a; ?P 0; \bigwedge n. \llbracket \text{evn } n; ?P n \rrbracket \Longrightarrow ?P (n + 2) \rrbracket \Longrightarrow ?P ?a$$

Using the Induction Rule

The induction rule can be used to prove properties about the defined predicate *name* which may involve an iterated application of the defining rules. It abstracts the goal conclusion to a function with the same arguments as *name* and then splits it together with the predicate according to the predicate's defining rules. See [5] for corresponding proof techniques.

Like *name.cases* the rule *name.induct* includes information about the implicit definition of *False* predicate values. It covers the cases where a predicate application unifies with the conclusion of a P_i but cannot be reduced in a finite number of backwards reasoning steps.

Consider the predicate *uspec₂* defined as above. The rule *uspec₂.induct* is

$$\llbracket \text{uspec}_2 ?x; \bigwedge i. \llbracket \text{uspec}_2 i; ?P i \rrbracket \Longrightarrow ?P i \rrbracket \Longrightarrow ?P ?x$$

The proposition $\text{uspec}_2 0 \Longrightarrow \text{False}$ can be proved as follows:

```
theorem "uspec2 0  $\Longrightarrow$  False"
  apply (induction rule: uspec2.induct)
  by simp
```

The application of `uspec2.induct` creates the goal $\bigwedge i. \llbracket \text{uspec}_2\ i; \text{False} \rrbracket \implies \text{False}$ which is solved by the simplifier. Actually, this works also for the proposition `uspec2 i \implies False` because the conclusion does not depend on the argument of `uspec2`

Note that this does not work for the predicate `uspec1`. For it `uspec1.induct` is

$\llbracket \text{uspec}_1\ ?x; ?P\ 3; ?P\ 4 \rrbracket \implies ?P\ ?x$

and the application by `induction` to `uspec1 0 \implies False` yields two goals `False` which cannot be proved.

The Induction Rule in Structured Proofs

Like `name.cases` the rule `name.induct` can also be applied in structured proofs (see Section 5.5.5). The example above can be written

```
theorem "False" if "uspec2 0"
  using that
proof (induct rule: uspec2.induct) qed simp
```

Moreover, the rule `name.induct` is associated to the defined predicate `name` in the following way: if the first input fact to the `induction` or `induct` method has an inductively defined predicate `name` as its outermost function, the rule `name.induct` is the default rule applied by the methods if no rule is explicitly specified. So the proof for `"False" if "uspec2 0"` above can be abbreviated to

```
proof induction qed simp
```

or even shorter:

```
by induction simp
```

6.7.5 Single-Step Inductive Definitions

A simple case is an inductive definition where no rule assumption $Q_{i,j}$ contains an application of the defined predicate `name`. Then for every defining rule a single backward reasoning step will remove the predicate and determine its value. The rule `name.cases` contains the complete information about the defined predicate and is sufficient for proving arbitrary properties about it.

Every predicate defined by an Isabelle definition (see Section 2.3.1)

```
definition name :: "t1  $\Rightarrow$  ...  $\Rightarrow$  tk  $\Rightarrow$  bool"
  where "name x1 ... xn  $\equiv$  term"
```

is equivalent to the predicate defined by the inductive definition

```
inductive name :: "t1 ⇒ ... ⇒ tk ⇒ bool"
  where "term ⇒ name x1 ... xn"
```

Thus inductive definitions are a generalization of the basic Isabelle definitions for predicates.

Here *name* cannot occur in *term* because Isabelle definitions do not support recursion, therefore it is a single-step inductive definition with only one rule. Vice versa, every single-step inductive definition can be converted to an Isabelle definition, where the *term* is mainly a disjunction of the left sides of all defining rules. Actually, for this case the rule *name.simps* (see above) is equivalent to the defining equation of the corresponding Isabelle definition.

For such predicates it is often simpler to use an Isabelle definition. However, if there are a lot of alternative cases in *term* it may be easier to use the form of an inductive definition. Note also, that there are predicates which cannot be defined by an Isabelle definition but can be defined by a (non-single-step) inductive definition.

6.7.6 Mutually Inductive Definitions

Inductively defined predicates may depend on each other. Then they must be defined by a common inductive definition of the extended form

```
inductive name1 ... namem
where P1 | ... | Pn
```

The *name*_{*i*} may be grouped by *and* and types may be specified for (some of) the groups.

The defining rules *P*_{*i*} have the same form as described above, however, every conclusion may be an application of one of the defined predicates *name*_{*i*} (with arbitrary ordering of the rules) and every rule assumption *Q*_{*i,j*} may contain applications of all defined predicates.

The following example defines predicates for even and odd numbers in a mutually inductive way:

```
inductive evn odd :: "nat ⇒ bool" where
  "evn(0)" | "odd(n) ⇒ evn(n+1)" | "evn(n) ⇒ odd(n+1)"
```

The set of defining rules is named *name*₁...*name*_{*m*}.*intros*. For every *name*_{*i*} separate rules *name*_{*i*}.*cases* and *name*_{*i*}.*simps* are created which cover only the defining rules with *name*_{*i*} in the conclusion. As induction rules the set *name*₁...*name*_{*m*}.*inducts* is created containing for every *name*_{*i*} an induction rule with an application of *name*_{*i*} as major premise. Additionally, a rule *name*₁...*name*_{*m*}.*induct* is generated without elimination where the conclusion is an explicit case distinction for all defined predicates.

For the example the rule `evn_odd.induct` is

$$\begin{aligned} & \llbracket ?P1\ 0; \bigwedge n. \llbracket \text{odd } n; ?P2\ n \rrbracket \implies ?P1\ (n + 1); \\ & \bigwedge n. \llbracket \text{evn } n; ?P1\ n \rrbracket \implies ?P2\ (n + 1) \rrbracket \\ & \implies (\text{evn } ?x1 \longrightarrow ?P1\ ?x1) \wedge (\text{odd } ?x2 \longrightarrow ?P2\ ?x2) \end{aligned}$$

6.8 Well-Founded Relations

A binary relation r of type $t \Rightarrow t \Rightarrow \text{bool}$ (i.e., the related values are of the same type) is called “well-founded” if every non-empty set of values of type t has a “leftmost” element x for r which means that there is no y so that $r\ y\ x$. If the relation $(<)$ (see Section 6.2.2) is well-founded for a type t this is usually described by “every non-empty set has a minimal element”.

HOL provides the predicate

`wfP :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool`

which tests an arbitrary binary relation for being well-founded. The predicate

`wf :: ('a \times 'a) set \Rightarrow bool`

does the same for a binary relation in tuple set form (see Section 6.6.2).

For a well-founded relation of type $t \Rightarrow t \Rightarrow \text{bool}$ also the universal set `UNIV :: t set` (see Section 8.4.1) has a leftmost element, which is the leftmost value of type t . Moreover, from that value all other values of type t can be reached in a finite number of steps to a related value, or, if the relation is transitive, even by a single step.

The best-known well-founded relation is the strict less-order $(<)$ on the natural numbers. Note that (\leq) is not well-founded because for every number n there is n itself so that $n \leq n$.

6.8.1 Induction

For every well-founded relation r the following principle of (transfinite) induction is valid: If a property holds for a value whenever it holds for all values “left” to it, the property holds for all values.

HOL provides the induction rules with elimination (see Section 5.5.5)

`wfp_induct_rule:`
 $\llbracket \text{wfp } ?r; \bigwedge x. (\bigwedge y. ?r\ y\ x \implies ?P\ y) \implies ?P\ x \rrbracket \implies ?P\ ?a$

`wf_induct_rule:`
 $\llbracket \text{wf } ?r; \bigwedge x. (\bigwedge y. (y, x) \in ?r \implies ?P\ y) \implies ?P\ x \rrbracket \implies ?P\ ?a$

Their major premise is well-foundedness of the relation $?r$. The single case corresponds to the induction principle.

`thm wf_induct_rule`

6.8.2 The Accessible Part of a Relation

For an arbitrary binary relation r of type $t \Rightarrow t \Rightarrow \text{bool}$ generally not all values of t can be reached in a finite number of relation steps from a leftmost element. HOL defines the predicate *Wellfounded.accp* by the inductive definition (see Section 6.7)

```
inductive accp :: "('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  bool"
  for r :: "('a  $\Rightarrow$  'a  $\Rightarrow$  bool)"
  where "( $\bigwedge y. r\ y\ x \implies \text{accp}\ r\ y$ )  $\implies \text{accp}\ r\ x$ "
```

Its partial application *accp r* to a binary relation r is the predicate which is *True* for all values of t which can be reached in a finite number of relation steps from a leftmost element. These values are called the “accessible part” of relation r .

HOL also defines the equivalent set-valued function (see Section 6.1.2) for relations represented as tuple sets (see Section 6.6.2)

```
acc :: "('a  $\times$  'a) set  $\Rightarrow$  'a set"
```

which returns the accessible part of a relation as a set.

A binary relation r of type $t \Rightarrow t \Rightarrow \text{bool}$ is well-founded if and only if its accessible part *accp r* covers all values of t . If r is not well-founded there may be several leftmost values in t or no such value (in the latter case the accessible part is empty).

For arbitrary binary relations the induction principle can be used on the accessible part. The corresponding induction rules provided by HOL are

```
accp_induct_rule:
  [| Wellfounded.accp ?r ?a;
     $\bigwedge x. [\text{Wellfounded.accp}\ ?r\ x; \bigwedge y. ?r\ y\ x \implies ?P\ y] \implies ?P\ x$ 
  |]  $\implies ?P\ ?a$ 
```

```
acc_induct_rule:
  [| ?a  $\in$  Wellfounded.acc ?r;
     $\bigwedge x. [x \in \text{Wellfounded.acc}\ ?r; \bigwedge y. (y, x) \in ?r \implies ?P\ y] \implies ?P\ x$ 
  |]  $\implies ?P\ ?a$ 
```

Here the major premise is that the value *?a* for which the property shall be proved belongs to the accessible part of the relation. Also the induction step is only done for values in the accessible part (if a value is in the accessible part then also all values “left” to it).

6.8.3 Measure Functions

A binary relation r of type $t \Rightarrow t \Rightarrow \text{bool}$ is well-founded if it is mapped by a function $f :: t \Rightarrow u$ into a well-founded relation s on type u . This is the case if $r\ x\ y \implies s\ (f\ x)\ (f\ y)$ for all values x and y of type t .

This property is often used to prove well-foundedness for a binary relation r by specifically mapping it into the well-founded order ($<$) on type \mathbf{nat} . In this context the mapping function $f :: t \Rightarrow \mathbf{nat}$ is called a “measure function”.

HOL provides the polymorphic function

```
measure :: ('a  $\Rightarrow$  nat)  $\Rightarrow$  ('a  $\times$  'a) set
```

which turns a measure function for values of an arbitrary type $'a$ to a relation on $'a$ in tuple set form. The relation $\mathbf{measure} \ f$ relates values x and y if $(f \ x) < (f \ y)$.

The usual way to prove that a relation r of type $t \Rightarrow t \Rightarrow \mathbf{bool}$ is well-founded is to design a measure function $f :: t \Rightarrow \mathbf{nat}$ and prove that $r \ x \ y \implies (\mathbf{measure} \ f) \ x \ y$.

6.8.4 The *size* Function

HOL introduces the polymorphic function

```
size :: 'a  $\Rightarrow$  nat
```

which is overloaded for many HOL types. If it is not supported for a type an application to a value of that type will result in an error message “No type arity ...”.

The *size* function is intended as a standard measure function for proving well-foundedness of relations on type $'a$ with the help of the function *measure*. However, there may be relations where this does not work and a different measure function is required.

Note that the concept of *size* and *measure* actually depends on the specific HOL type \mathbf{nat} , which is introduced in Section 8.3.

6.9 The Proof Method *atomize_elim*

As a useful method for proving elimination rules (see Section 5.4.2) including the more specific case rules (see Section 5.3.1) and the goals of **obtain** statements (see Section 3.11) HOL introduces the proof method

```
atomize_elim
```

It only affects the first goal. Assume that this goal has the form of an elimination rule

```
theorem "[RA1; ...; RAm; P1; ...; Pk]  $\implies$  P" <proof>
```

where every P_i is a rule of the form

$$\bigwedge^{x_{i,1} \dots x_{i,p_i}} \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \implies P$$

(see Section 5.4.2). The method converts the goal to the form

$$\llbracket RA_1; \dots; RA_m \rrbracket \implies D_1 \vee \dots \vee D_p$$

where every D_i has the form $\exists x_{i,1} \dots x_{i,p_i} \cdot Q_{i,1} \wedge \dots \wedge Q_{i,q_i}$. For the quantifier \exists and the boolean operators \vee and \wedge see Section 8.1.3. Because the converted goal does not contain the technical variable P any more and uses the boolean operators it is sometimes easier to prove it.

If the method is applied to the initial goal of the example elimination rule theorem

theorem elimexp: " $\llbracket (x::nat) \leq c; x < c \implies P; x = c \implies P \rrbracket \implies P$ "

in Section 5.4.2 it converts the goal to

$$x \leq c \implies x < c \vee x = c$$

which can be solved by method `auto`.

If `atomize_elim` is applied to the initial goal of the example case rule theorem

theorem mycaserule: " $\llbracket n = 0 \implies P; n \neq 0 \implies P \rrbracket \implies P$ "

in Section 5.3.1 it converts the goal to

$$n = 0 \vee n \neq 0$$

which can be solved by method `simp`.

If `atomize_elim` is applied to the initial goal of the example **obtain** statement

obtain $y \ z$ **where** " $x = y - 3$ " **and** " $y + z = 2*x + 8$ "

in Section 3.11.1 it converts the goal to

$$\exists y \ z. x = y - 3 \wedge y + z = 2 * x + 8$$

which can be solved by method `arith` (see Section 4.7.1).

6.10 Recursive Functions

HOL supports the definition of recursive functions in the form

function *name* **::** " $t_1 \Rightarrow \dots \Rightarrow t_k \Rightarrow type$ "
where $eq_1 \mid \dots \mid eq_n \langle proof \rangle$

The type specification for the function may be omitted if it can be derived from the use of the function in the eq_i .

The definition resembles an Isabelle definition as described in Section 2.3.1, but instead of a single defining equation there may be arbitrary many defining equations eq_i followed by a proof. Also other than for an Isabelle definition the part **name** ... **where** cannot be omitted, because the **name** must be known to check whether the equations have the correct form.

Compared with the inductive definitions in Section 6.7 recursive functions can not only be predicates but may have an arbitrary result type and equations are used instead of defining rules.

6.10.1 The Defining Equations

Each of the defining equations eq_i has the general form of a derivation rule (see Section 2.5.2) with an equation as its conclusion:

$$\bigwedge x_{i,1} \dots x_{i,p_i} . \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \Longrightarrow \text{name } term_{i,1} \dots term_{i,k} = term_i$$

It is specified in inner syntax. The separating bars / belong to the outer syntax, therefore each equation must be separately quoted. Note that the conclusion must be an equation using the symbol = instead of \equiv . The left side of the equation is restricted to the form of a (non-partial) application of the defined function to argument terms, i.e., if the defined function has k arguments according to its type, in every equation it must be applied to k terms. If the conclusion is no equation or if the left side has a different form an error is signaled.

Other than in an Isabelle definition the equations may be recursive, i.e. the defined function **name** may be used in $term_i$ on the right side of the equation (but not in the $Q_{i,j}$ or $term_{i,j}$). It may be used in arbitrary ways, also by partial application or by passing it as argument to other functions.

The familiar example of the faculty function can be defined using two defining equations in the form

```
function fac :: "nat  $\Rightarrow$  nat" where
  "fac 0 = 1"
| " $\bigwedge n. n > 0 \Longrightarrow fac\ n = n * fac\ (n-1)$ "
<proof>
```

The bound variables $x_{i,1}, \dots, x_{i,p_i}$ may occur in the assumptions and in the conclusion. However, if a bound variable occurs in $term_i$ on the right side it must also occur in one or more of the $term_{i,j}$ on the left side, otherwise an error is signaled.

After termination has been proved (see Section 6.10.6) for the recursive function definition the defining equations are available as the fact set **name.simps**.

Note that no defining equation *name_def* (see Section 2.6.5) exists for recursively defined functions, instead the set *name.simps* plays the corresponding role.

Alternative Rule Forms

As for derivation rules specified in theorems (see Section 2.6.1) the explicit bindings are optional, variables occurring free in the assumptions or the conclusion are always automatically bound. As usual, types may be specified for (some of) the variables, so explicit bindings can be used as a central place for specifying types for the variables, if necessary. An equivalent definition for the faculty is

```
function fac :: "nat  $\Rightarrow$  nat" where
  "fac 0 = 1"
| "n > 0  $\implies$  fac n = n * fac (n-1)"
<proof>
```

where the binding of *n* is omitted.

Alternatively an equation *eq_i* may be specified in the structured form described for derivation rules in Section 2.5.6:

```
"name termi,1 ... termi,k = termi" if "Qi,1" ... "Qi,qi" for xi,1 ... xi,pi
```

where the assumptions and variables may be grouped by *and* and types may be specified for (some of) the variable groups, however, no names may be specified for assumption groups.

In this form the faculty definition becomes

```
function fac :: "nat  $\Rightarrow$  nat" where
  "fac 0 = 1"
| "fac n = n * fac (n-1)" if "n > 0"
<proof>
```

Note that on the left side of the equations the arguments may be specified by arbitrary terms, not only by variables. Therefore the faculty function can also be defined in the form

```
function fac2 :: "nat  $\Rightarrow$  nat" where
  "fac2 0 = 1"
| "fac2 (n+1) = (n+1) * fac2 n"
<proof>
```

where the assumption is not required anymore for the second equation.

It is also possible to explicitly name (some of) the single equations by specifying the recursive definition in the form

```

function name :: "t1 ⇒ ... ⇒ tk ⇒ type" where
  eqname1 : eq1 / ... / eqnamen : eqn <proof>

```

Every function *name* defined by the Isabelle definition (see Section 2.3.1)

```

definition name :: "t1 ⇒ ... ⇒ tk ⇒ type"
  where "name x1 ... xn ≡ term"

```

is equivalent to the function defined by the recursive definition

```

function name :: "t1 ⇒ ... ⇒ tk ⇒ type"
  where name_def : "name x1 ... xn = term" <proof>

```

which actually does not use recursion.

6.10.2 Covering All Arguments

As described in Section 2.2.3 functions in Isabelle must be total, they must be defined for all possible arguments. It is easy to fail doing so when specifying the defining equations for a recursive function definition. Consider

```

function nototal :: "nat ⇒ nat" where
  "nototal 5 = 6" <proof>

```

The function is only defined for the value 5, no definition is given for the other natural numbers.

Therefore HOL expects a proof that the equations are complete in the sense that they cover all possible cases for the function arguments. In the general case it creates a goal of the form

$$\begin{aligned}
 &\bigwedge^P \mathbf{x}. \\
 &\quad \llbracket \bigwedge^{x_{1,1}} \dots x_{1,p_1}. \llbracket Q_{1,1}; \dots; Q_{1,q_1}; \mathbf{x} = (\mathit{term}_{1,1}, \dots, \mathit{term}_{1,k}) \rrbracket \implies P; \\
 &\quad \dots \\
 &\quad \bigwedge^{x_{n,1}} \dots x_{n,p_n}. \llbracket Q_{n,1}; \dots; Q_{n,q_n}; \mathbf{x} = (\mathit{term}_{n,1}, \dots, \mathit{term}_{n,k}) \rrbracket \implies P \rrbracket \\
 &\implies P
 \end{aligned}$$

which must be proved in the recursive function definition's <*proof*>. This goal has the form of a case rule (see Section 5.3.1) before replacing the variable *P* by an unknown upon turning it to a fact. It specifies that *x* covers all possible cases. The variable *x* is set to the tuples of all the argument terms used on the left side of all equations.

Using tuples here depends on the specific HOL type constructor *prod*, which is introduced in Section 8.6. The goal could also be constructed without using tuples, however, it is more compact like that.

Note that the goal does not mention the defined function *name* at all. It is only about the groups of the argument terms in the equations.

For the faculty function defined as above the goal is

$$\bigwedge P \ (x :: \text{nat}). \llbracket x = 0 \implies P; \bigwedge n. \llbracket 0 < n; x = n \rrbracket \implies P \rrbracket \implies P$$

It can be proved by method `auto` (see Section 4.7.1).

After the proof of the recursive function definition the goal is available as a case rule named `name.cases`. If applied by the proof method `cases` (see Section 5.3.2) it splits a goal and introduces named cases (named by numbers starting at 1) according to the argument regions in the defining equations of function `name`.

If the function `name` has only one argument `name.cases` is a usual case rule for the argument type, otherwise it is a case rule for the type of the argument tuples.

Using Proof Method `atomize_elim`

Since the goal has the form of a case rule the method `atomize_elim` (see Section 6.9) can be applied to it. It converts the general form of the goal to the form

$$\begin{aligned} \bigwedge x. \\ & (\exists \ x_{1,1} \ \dots \ x_{1,p_1}. \ Q_{1,1} \wedge \dots \wedge Q_{1,q_1} \wedge x = (\text{term}_{1,1}, \dots, \text{term}_{1,k}) \\ & \vee \dots \\ & \vee (\exists \ x_{n,1} \ \dots \ x_{n,p_n}. \ Q_{n,1} \wedge \dots \wedge Q_{n,q_n} \wedge x = (\text{term}_{n,1}, \dots, \text{term}_{n,k}) \end{aligned}$$

which directly expresses that the cases together cover all possibilities and may be easier to prove. In many simple cases of recursive function definitions, however, the method is not necessary and the goal can be proved by an automatic method like `auto` or `blast` (see Section 4.7.1).

For the faculty function defined as above the converted goal is

$$\bigwedge x :: \text{nat}. \ x = 0 \vee (\exists n. \ n > 0 \wedge x = n)$$

which can also be proved by method `auto`, so the method `atomize_elim` is not necessary here.

Recursive Definitions vs. Inductive Definitions

In principle an inductively defined predicate `name` (see Section 6.7) could be defined recursively by adding the cases where it has the value `False`, so that all arguments are covered. The evenness predicate from Section 6.7.1 can be defined by

```
function evn :: "nat ⇒ bool" where
  "evn 0 = True" | "evn 1 = False" | "evn (n+2) = evn n"
  <proof>
```

because the cases 0 , 1 , and $n+2$ cover all natural numbers.

However, this is not always possible. It may not be possible to give an explicit specification for the arguments where the predicate value is *False* or the termination proof (see Section 6.10.6) may fail.

Generally, an inductive definition is simpler because it only specifies the positive cases and only these must be proved in proofs of properties of the defined predicate. On the other hand, if the negative cases are of interest they are directly provided by a recursive definition and the defining equations of a recursive definition can be used for rewriting (see Section 4.6).

6.10.3 Uniqueness

As usual, a function must be unique, mapping every argument to only one value. It is easy to fail doing so, consider

```
function nunique :: "nat ⇒ nat" where
  "nunique 5 = 6"
| "nunique 5 = 7"
⟨proof⟩
```

The function maps the value 5 to two different values 6 and 7 (and thus is no function anymore).

Therefore HOL expects a proof that the equations are compatible in the sense that they cover disjoint arguments spaces or, if the spaces of two equations overlap, the equations specify the same function values on the overlapping regions. In the general case HOL creates (after the goal for equation completeness) for every pair of equations eq_i and eq_j where $i \leq j$ a goal of the form

$$\bigwedge x_{i,1} \dots x_{i,p_i} x_{j,1}' \dots x_{j,p_j}'. \\ (term_{i,1}, \dots, term_{i,k}) = (term_{j,1}, \dots, term_{j,k}) \implies term_i = term_j$$

Here $x_{j,b}'$ is a renamed $x_{j,b}$ to avoid a name clash with an $x_{i,a}$ if necessary. The renamed variables are consistently replaced in all terms. Moreover, all occurrences of the defined function *name* in $term_i$ and $term_j$ are replaced by *name_sumC* which is the uncurried (see Section 8.6.3) form of *name* where the arguments are specified as a tuple.

Together these are $\sum_{i=1}^n i = (n+1) * n/2$ goals where n is the number of equations. The proof of these goals depends on the types and functions used in the equations, for many simple cases it can be done by method *auto* which solves all goals together.

For the faculty function defined as above these are the three goals

$$0 = 0 \implies 1 = 1 \\ \bigwedge n. \llbracket 0 < n; 0 = n \rrbracket \implies 1 = n * fac_sumC (n - 1)$$

$$\bigwedge n \text{ na. } \llbracket 0 < n; 0 < \text{na}; n = \text{na} \rrbracket \\ \implies n * \text{fac_sumC } (n - 1) = \text{na} * \text{fac_sumC } (\text{na} - 1)$$

where the first and third are trivial and the second is valid because the two assumptions are a contradiction (because the argument spaces are disjoint). All three goals are solved together by a single application of method `auto` (see Section 4.7.1).

6.10.4 The Domain Predicate

Even if the defining equations cover all possible arguments and define the function values in a unique way the function value may still be underspecified for some arguments. Consider the recursive definition

```
function uspec :: "nat  $\Rightarrow$  nat" where
  "uspec i = uspec i"
by auto
```

The proof of equation completeness and compatibility is successfully done by the `auto` method and the definition introduces the total function *uspec*, however, no information is available about its result values.

Information about result values must either be specified directly on the right side of a non-recursive equation *eq_i* or it must be derivable by a finite number of substitutions using the equations. Such a substitution mainly replaces the function arguments specified on the left side of the equation by the arguments used in the recursive calls on the right side.

Therefore HOL creates for every recursive definition of a function *name* the relation *name_rel* which relates for all defining equations the arguments of every recursive call on the right side to the arguments on the left side. Actually, *name_rel* relates argument tuples, like the arguments of *name_sumC* as described above. It is defined inductively (see Section 6.7) using defining rules of the form

$$\bigwedge x_{i,1} \dots x_{i,p_i}. \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \implies \\ \text{name_rel } (\text{term}_{i,j,1}, \dots, \text{term}_{i,j,k}) (\text{term}_{i,1}, \dots, \text{term}_{i,k})$$

where *term_{i,j,l}* is the *l*-th argument in the *j*-th recursive call of *name* in *term_i*. There is one such rule for every recursive call occurring in the defining equations *eq_i*. The defined relation *name_rel* never occurs in the *Q_{i,j}*, therefore it is a single-step inductive definition (see Section 6.7.5) which could also have been specified as an Isabelle definition.

For the faculty function defined as above the corresponding definition of the relation *fac_rel* is

```
inductive fac_rel :: "nat  $\Rightarrow$  nat  $\Rightarrow$  bool"
where "0 < n  $\implies$  fac_rel (n - 1) n"
```

with only one rule because there is only one recursive call in the definition. This relation relates every natural number with its immediate successor.

Now it is possible to characterize the arguments for which the function value is fully specified by the accessible part *Wellfounded.accp name_rel* of *name_rel* (see Section 6.8.2). Every substitution by an equation *eq_i* corresponds to an application of *name_rel* from right to left, every chain of such applications is finite on its accessible part.

For every recursive definition of a function *name* :: *t₁* ⇒ ... ⇒ *t_k* ⇒ *type* HOL defines the abbreviation

```
abbreviation name_dom :: "(t1 × ... × tk) ⇒ bool" where
  "name_dom ≡ Wellfounded.accp name_rel"
```

called the “domain predicate” for the argument tuples for which *name* is fully specified.

For the faculty function the domain predicate is defined by

```
abbreviation fac_dom :: "nat ⇒ bool" where
  "fac_dom ≡ Wellfounded.accp fac_rel"
```

Since every natural number can be reached by a finite number of steps starting at 0 this predicate is *True* for all natural numbers.

6.10.5 Rules Provided by Recursive Definitions

HOL automatically creates and proves some additional rules from a recursive definition and its proof. The domain predicate is used in these rules to exclude underspecified cases. The rules can be displayed using the “Print Context” tab in the Query panel as described in Section 2.6.6.

Simplification Rules

HOL automatically creates and proves simplification rules which directly correspond to the defining equations *eq_i*. Every such rule is guarded by the domain predicate for the arguments of the substituted function application. For the equation *eq_i* in the general form as given above the rule is

$$\llbracket Q_{i,1}; \dots; Q_{i,q_i}; \text{name_dom } (term_{i,1}, \dots, term_{i,k}) \rrbracket \implies \\ \text{name } term_{i,1} \dots term_{i,k} = term_i$$

where all occurrences of the bound variables *x_{i,1}*, ..., *x_{i,p_i}* have been replaced by corresponding unknowns. The set of all these simplification rules is named *name.psimps*. The rules are not added to the simpset automatically, they can be explicitly used for “unfolding” the recursive definition in one or more steps. If individual names have been specified for (some of) the *eq_i* these names denote the corresponding facts in *name.psimps*.

Elimination Rule

HOL also creates an elimination rule (see Section 5.4.2) *name.pelims* where the major premise has the form *name* ?*x*₁ ... ?*x*_{*k*} = ?*y* and the rule replaces it mainly by case assumptions according to the defining equations.

Induction Rule

HOL also creates and proves a single induction rule *name.pinduct* of the form

$$\llbracket \text{name_dom } (?a_1, \dots, ?a_k); RA_1; \dots; RA_n \rrbracket \Longrightarrow ?P ?a_1 \dots ?a_k$$

where every *RA_i* has the form

$$\begin{aligned} &\bigwedge x_{i,1} \dots x_{i,p_i} \cdot \\ &\llbracket \text{name_dom } (\text{term}_{i,1}, \dots, \text{term}_{i,k}); Q_{i,1}; \dots; Q_{i,q_i}; R_{i,1}; \dots; R_{i,r_i} \rrbracket \\ &\Longrightarrow ?P \text{term}_{i,1} \dots \text{term}_{i,k} \end{aligned}$$

and every *R_{i,j}* has the form *?P term_{i,j,1} ... term_{i,j,k}* where *term_{i,j,l}* is the *l*-th argument in the *j*-th recursive call of *name* in *term_i*.

It can be used for induction with elimination (see Section 5.5.5) where the major premise is the domain predicate for the arguments of the proved property *?P* which is usually a property of the defined function *name* applied to these arguments. The cases correspond to the defining equations and are named by numbers starting with 1. The induction step in every case goes from the arguments of the recursive calls in *term_i* to the arguments on the left side of *eq_i*. This form of induction is also called “computation induction”. For the faculty function defined as above the induction rule *fac.pinduct* is

$$\begin{aligned} &\llbracket \text{fac_dom } ?a; \text{fac_dom } 0 \rrbracket \Longrightarrow ?P 0; \\ &\bigwedge n. \llbracket \text{fac_dom } n; 0 < n; ?P (n - 1) \rrbracket \Longrightarrow ?P n \rrbracket \Longrightarrow ?P ?a \end{aligned}$$

6.10.6 Termination

In programming languages a recursively defined function *name* is only considered correct if its value can be determined for every argument by simplification using *name.psimps* in a finite number of steps (i.e., its computation terminates). This is only the case if the argument (tuple) satisfies the domain predicate. Therefore it is often of interest to prove that this is the case for all possible argument (tuple)s. HOL provides specific support for proving this property.

The corresponding theorem for a recursively defined function *name* :: *t*₁ ⇒ ... ⇒ *t*_{*k*} ⇒ *type* is

theorem " $\forall x :: (t_1 \times \dots \times t_k). \text{ name_dom } x$ " *<proof>*

stating that all possible argument tuples satisfy the domain predicate *name_dom* (for the quantifier \forall see Section 8.1.3). HOL provides the equivalent abbreviated form

termination *name* $\langle proof \rangle$

which is called a “termination proof” for the recursive function *name*. If *name* is omitted it refers to the last previously defined recursive function.

The Proof Method *relation*

Note that this theorem is equivalent to the property that the relation *name_rel* is well-founded. As described in Section 6.8.3 it can be proved by proving that *name_rel* $x\ y \implies (\text{measure } f)\ x\ y$ holds for a measure function $f :: (t_1 \times \dots \times t_k) \Rightarrow \text{nat}$. HOL provides the proof method

relation "M"

which replaces the original goal of a termination proof for *name* by the goals

wf *M*
 R_1
 \dots
 R_r

where every R_h corresponds to a defining rule for *name_rel* (see Section 6.10.4) in *name_rel.intros* (see Section 6.7.1) and has the form

$$\bigwedge x_{i,1} \dots x_{i,p_i} \cdot \llbracket Q_{i,1}; \dots; Q_{i,q_i} \rrbracket \implies ((\text{term}_{i,j,1}, \dots, \text{term}_{i,j,k}), (\text{term}_{i,1}, \dots, \text{term}_{i,k})) \in M$$

where $\text{term}_{i,j,l}$ is the l -th argument in the j -th recursive call of *name* in term_i . The goals R_1, \dots, R_r together are equivalent to the goal *name_rel* $x\ y \implies M\ x\ y$.

The proof method is usually applied in the form

relation "measure *f*"

for an appropriate measure function *f* as above. It must be constructed so that for every tuple $(\text{term}_{i,j,1}, \dots, \text{term}_{i,j,k})$ of arguments for a recursive call the value is strictly lower than for the argument tuple $(\text{term}_{i,1}, \dots, \text{term}_{i,k})$ in the defining equations *eq_i*. The resulting goals *wf* (measure *f*) and R_1, \dots, R_r are often solved by method *auto* (see Section 4.7.1). Then the termination proof has the form

termination *name* **by** (*relation* "measure *f*") *auto*

For the faculty function defined as above the goal of the termination proof is

```
 $\forall x :: \text{nat}. \text{fac\_dom } x$ 
```

which is true as argued above. To prove it, the identity function $\lambda n. n$ is an applicable measure function because $n > 0 \implies n-1 < n$. Applying the *relation* method with it yields the two goals

```
 $wf \ (\text{measure } (\lambda n. n))$   
 $\bigwedge n. 0 < n \implies (n - 1, n) \in \text{measure } (\lambda n. n)$ 
```

They can be solved together by method *auto*, therefore a termination proof for the faculty definition can be specified as

```
termination fac by (relation "measure ( $\lambda n. n$ )") auto
```

The Proof Methods *lexicographic_order* and *size_change*

HOL also provides the proof method

```
lexicographic_order
```

and the stronger method

```
size_change
```

for termination proofs. They try to automatically construct a measure function from the *size* function (see Section 6.8.4) combined in a lexicographic way for argument tuples. They are applied to the original goal and if they are successful they solve it completely, otherwise an error is signaled. Using the first method a termination proof has the form

```
termination name by lexicographic_order
```

Since for type *nat* the *size* function is the identity the termination proof for the faculty function may also be specified as

```
termination fac by lexicographic_order
```

6.10.7 Rules Provided by Termination Proofs

A termination proof for a function *name* using the command **termination** provides the additional rules *name.simps*, *name.elims*, and *name.induct*. They result from the corresponding rules provided by the recursive definition by

removing all applications of the domain predicate *name_dom*. This is valid because the termination proof has shown that it is always *True*.

Specifically, the resulting simplification rules exactly correspond to the defining equations and can be used for unfolding the definition for a function application in one or more steps.

If individual names have been specified for (some of) the *eq_i* the termination proof replaces the associated facts from *name.psimps* by those from *name.simps*.

The resulting induction rule is a plain induction rule without elimination of the form

$$\llbracket RA_1; \dots; RA_n \rrbracket \implies ?P \ ?a_1 \ \dots \ ?a_k$$

where every *RA_i* has the form

$$\bigwedge x_{i,1} \ \dots \ x_{i,p_i} . \llbracket Q_{i,1}; \dots; Q_{i,q_i}; R_{i,1}; \dots; R_{i,r_i} \rrbracket \\ \implies ?P \ term_{i,1} \ \dots \ term_{i,k}$$

and every *R_{i,j}* has the form *?P term_{i,j,1} ... term_{i,j,k}* where *term_{i,j,l}* is the *l*-th argument in the *j*-th recursive call of *name* in *term_i*.

For the faculty function defined as above the induction rule *fac.induct* is

$$\llbracket ?P \ 0; \bigwedge n. \llbracket 0 < n; ?P \ (n - 1) \rrbracket \implies ?P \ n \rrbracket \implies ?P \ ?a$$

To use the rule with the proof methods *induct* and *induction* (see Section 5.5.3) it must always be specified explicitly in the form

induct ... rule: name.induct

All these rules can be displayed using the “Print Context” tab in the Query panel as described in Section 2.6.6, if the cursor is positioned after the termination proof.

6.10.8 Mutual Recursion

If several recursive functions are defined depending on each other they must be defined together in a single recursive definition of the form

function *name₁ ... name_m*
where *eq₁ | ... | eq_n* *<proof>*

The *name_i* may be grouped by *and* and types may be specified for (some of) the groups.

The defining equations *eq_i* have the same form as described above, however, every left side may be an application of one of the defined functions *name_i*

(with arbitrary ordering of the rules) and every right side $term_i$ may contain applications of all defined functions.

A mutual recursive definition of the predicates evn and odd (see Section 6.7.6) is

```
function evn odd :: "nat  $\Rightarrow$  bool"
where "evn 0 = True" | "odd 0 = False"
| "evn (n+1) = odd n" | "odd (n+1) = evn n"
⟨proof⟩
```

The simplification and elimination rules are provided for every defined function as $name_i.psimps$, $name_i.pelims$, $name_i.simps$, and $name_i.elims$. The induction rules are provided for all defined functions together in the sets named $name_1_..._name_m.pinduct$ and $name_1_..._name_m.induct$.

The domain predicate and the corresponding relation are common for all defined functions and are named $name_1_..._name_m.dom$ and $name_1_..._name_m.rel$. They are defined on values of the sum type (see Section 8.9) of the argument tuples of all defined functions, this is also the case for the function $name_1_..._name_m.sumC$ used in the goals for the uniqueness proof. Also a measure function used in a termination proof must be defined on this sum type.

A termination proof can use the name of either defined function to refer to the mutual recursive definition.

6.11 Functors

The content of this and the following section are not required for understanding most of the remaining chapters and may be skipped upon first reading. However, some of the concepts in the remaining chapters are based on functors and these two sections may be useful for fully understanding these concepts.

A type constructor such as set (see Section 2.2.1) can be seen as a mapping from types to types, mapping an arbitrary type T to the type $T\ set$ having as values sets of values of type T . However, the mapping does not specify any details about the values of T . From category theory the concept of a “functor” can be used to extend type constructors in that direction.

A functor for set additionally maps functions on the values of T to functions on the values of $T\ set$.

6.11.1 Mappers

For a type constructor C such a mapping can be specified to lead from functions on a type parameter $'a$ to functions on the polymorphic type $'a\ C$

(see Section 2.2.1). Such a mapping is itself a function of the polymorphic type $(\text{'a} \Rightarrow \text{'a}) \Rightarrow (\text{'a } \mathcal{C} \Rightarrow \text{'a } \mathcal{C})$. Such functions are called “mappers” or “mapper functions” for the type constructor \mathcal{C} . A mapper is said to “lift” functions on 'a to functions on $\text{'a } \mathcal{C}$.

More generally, a mapper may have the type $(\text{'a} \Rightarrow \text{'b}) \Rightarrow (\text{'a } \mathcal{C} \Rightarrow \text{'b } \mathcal{C})$ and lift functions between two types to functions between the corresponding types constructed by \mathcal{C} . Since the type constructor \mathcal{C} is directly determined by the mapper’s type, a functor may be fully specified by the mapper without explicitly naming the type constructor.

To be a mapper for a functor, m must satisfy two additional properties: it must be compatible with function composition and with the identity function. This may be expressed by the two laws

$$\begin{aligned} \text{comp}: (m \ f) \circ (m \ g) &= m \ (f \circ g) \\ \text{id}: m \ \text{id} &= \text{id} \end{aligned}$$

where \circ denotes function composition (see Section 8.7.2) and id denotes the (polymorphic) identity function (see Section 8.7.1).

These properties allow to lift an arbitrary term built from function applications by lifting the individual functions and building the corresponding term from the results. For this reason functors provide a versatile basic tool for transferring terms between types.

As an example the function $\text{image} :: (\text{'a} \Rightarrow \text{'b}) \Rightarrow \text{'a } \text{set} \Rightarrow \text{'b } \text{set}$ (see Section 8.4.3) maps every function f to $\text{image } f$ which returns for a set A of members of type 'a the set of f -values of these members, i.e., $\text{image } f$ applies f to all members of its argument sets. image is a mapper for the type constructor set , because the facts comp and id are satisfied for it. Therefore set is a functor with image as its mapper. image lifts every function $f :: \text{'a} \Rightarrow \text{'b}$ to a function of type $\text{'a } \text{set} \Rightarrow \text{'b } \text{set}$.

Not every function with the type of a mapper is a mapper, for example the function $\lambda f. (\lambda A. \{\})$ which maps every function f to the constant function returning the empty set is not a mapper because it does not satisfy the law id .

Mappers are not unique for type constructors. For example, the function $\lambda f. \text{id}$ which maps every function to the identity function is a (trivial) mapper for all type constructors.

There is a second kind of mapper type for type constructors \mathcal{C} . It has the form $(\text{'b} \Rightarrow \text{'a}) \Rightarrow \text{'a } \mathcal{C} \Rightarrow \text{'b } \mathcal{C}$ (i.e., the direction of 'a and 'b is reversed). Mappers of this type are called “contravariant mappers”. To distinguish both the other kind of mappers is also called “covariant mappers”. The law comp for a contravariant mapper m has the slightly different form

$$\text{comp}: (m \ f) \circ (m \ g) = m \ (g \circ f)$$

As an example, the function `vimage` (see Section 8.4.3) which maps every function to its reverse image on sets is a contravariant mapper for `set`.

Often the intended mapper which causes a type constructor \mathcal{C} to be a functor is assumed to be known implicitly, then \mathcal{C} is simply said to be a functor without explicitly specifying the mapper.

6.11.2 Multivariate Functors

The concept of functors can be extended to type constructors with more than one type parameters. For a type constructor \mathcal{C} with n parameters the mapper type has the form

$$('a_1 \Rightarrow 'b_1) \Rightarrow \dots \Rightarrow ('a_n \Rightarrow 'b_n) \Rightarrow ('a_1, \dots, 'a_n) \mathcal{C} \Rightarrow ('b_1, \dots, 'b_n) \mathcal{C}$$

A corresponding mapper lifts n functions to a single function between the types constructed by \mathcal{C} from the argument or result types, respectively.

The functor laws are extended accordingly:

$$\begin{aligned} \text{comp}: (m \ f_1 \ \dots \ f_n) \circ (m \ g_1 \ \dots \ g_n) &= m \ (f_1 \circ g_1) \ \dots \ (f_n \circ g_n) \\ \text{id}: m \ \text{id} \ \dots \ \text{id} &= \text{id} \end{aligned}$$

In category theory the resulting functors are also called “multivariate functors”.

As an example consider the type constructor `prod` (see Section 8.6) for the type of pairs. It has two type parameters and the polymorphic type $('a_1, 'a_2) \text{ prod}$ may be abbreviated by $'a_1 \times 'a_2$ (see Section 6.6). Therefore the mapper type for `prod` has the form $('a_1 \Rightarrow 'b_1) \Rightarrow ('a_2 \Rightarrow 'b_2) \Rightarrow ('a_1, 'a_2) \text{ prod} \Rightarrow ('b_1, 'b_2) \text{ prod}$. A possible mapper is the function `map_prod` $\equiv \lambda f_1 \ f_2. \lambda (x,y). (f_1 \ x, f_2 \ y)$ (see Section 8.6.3). It applies f_1 to the first component of the pair (x,y) and f_2 to the second component. In this way it lifts f_1 and f_2 to a function on pairs. It can easily be verified that `map_prod` satisfies the functor laws and causes `prod` to be a functor.

A mapper type for n parameters may also be contravariant in some parameters and covariant in the other parameters. Then the `comp` law must be modified accordingly, by reversing the function compositions for those parameters.

As an example consider the type constructor `fun` (see Section 2.2.3) for function types. It has two type parameters and the polymorphic type $('a_1, 'a_2) \text{ fun}$ may be abbreviated by $'a_1 \Rightarrow 'a_2$. Therefore a mapper type for `fun` may have the form $('b_1 \Rightarrow 'a_1) \Rightarrow ('a_2 \Rightarrow 'b_2) \Rightarrow ('a_1, 'a_2) \text{ fun} \Rightarrow ('b_1, 'b_2) \text{ fun}$ which is contravariant in the first parameter and covariant in the second. A possible mapper of this type is the function `map_fun` $\equiv \lambda f_1 \ f_2. \lambda f. f_2 \circ f \circ f_1$ (see Section 8.7.2). It can easily be verified that it satisfies the functor laws and causes `fun` to be a functor.

Similar to a partial function application (see Section 2.2.4) some of the parameters of a type constructor with n parameters may be filled by specific constant types, resulting in a type constructor with the remaining type parameters only. If one of n parameters of a type constructor is filled by a constant type the resulting type constructor has $n-1$ parameters. Whenever a type constructor is a functor all type constructors resulting from filling some parameters by constant types are also functors. The mappers for them are constructed by partially applying the original mapper to the identity function in the corresponding arguments.

As an example the polymorphic type $(nat, 'a) \text{ fun}$ is equivalent to a type constructor with one parameter. For every type $'a$ its values are the functions from natural numbers to result values of type $'a$. Since fun is a functor this type constructor is also a functor and its mapper is map_fun id which is $\lambda f_2. \lambda f. f_2 \circ f \circ id$ and thus is equivalent to the function composition \circ .

Even more generally an arbitrary polymorphic type expression built from several type constructors may be considered as a mapping from the type parameters to the resulting type instances and thus may satisfy the properties of a functor if combined with a corresponding mapper. Whenever all constructors used for the type are functors, then the resulting type is also a functor and the mapper can always be constructed from the mappers of the constructors. This property is called “composability” of functors.

As an example the polymorphic type expression $('a_1 \text{ set}, 'a_2) \text{ fun}$ denotes a functor because set and fun are functors. The mapper is of type $('b_1 \Rightarrow 'a_1) \Rightarrow ('a_2 \Rightarrow 'b_2) \Rightarrow ('a_1 \text{ set}, 'a_2) \text{ fun} \Rightarrow ('b_1 \text{ set}, 'b_2) \text{ fun}$ and may be defined as $\lambda f_1 f_2. \text{map_fun (image } f_1) f_2$ from the mappers image and map_fun .

6.11.3 Registering Functors

Functors are used by HOL for several purposes. Therefore HOL provides an outer syntax construct for “registering” a functor, i.e., making known to HOL that a function is a mapper for a functor:

functor *"term"* $\langle proof \rangle$

where *term* is a term for a function of a mapper type for a type constructor \mathcal{C} . It is mainly an abbreviation for

theorem *"c &&& i"* $\langle proof \rangle$

where *c* and *i* are the propositions for the *comp* and *id* laws according to the mapper type. The proof must prove that *term* satisfies these laws and is thus a mapper for \mathcal{C} . As proved facts the laws are automatically named

C.comp and *C.id*. Using these names the facts may then be used in proofs of other theorems. The facts can be displayed using the “Print Context” tab in the Query panel as described in Section 2.6.6.

If more than one mapper shall be registered for the same type constructor the fact name prefix may be specified explicitly in the form

```
functor pre: "term" <proof>
```

resulting in the fact names *pre.comp* and *pre.id*.

Both forms of the registration command provide the laws in two additional alternative forms named *pre.compositionality* and *pre.identity*.

The registration works for arbitrary multivariate functors with arbitrary contravariant parameters. The corresponding forms of *c* and *i* are automatically determined from the type of the *term*. The registration does not work for arbitrary polymorphic type expressions built from several constructors because their functor property always follows from those of the single constructors.

6.12 Bounded Natural Functors

The most important use of functors in HOL is their special case of “bounded natural functors” (BNF). It is closely related to the support of type definitions in HOL described in Chapter 7.

6.12.1 Natural Functors

HOL calls a single-parameter functor *C* with mapper *m* “natural” if there is a function *s* of type *'a C* \Rightarrow *'a set* which satisfies the laws

```
set_map: s  $\circ$  (m f) = (image f)  $\circ$  s
map_cong0: ( $\bigwedge z. z \in s\ x \implies f\ z = g\ z$ )  $\implies m\ f\ x = m\ g\ x$ 
```

Note that *set* is also a single-parameter functor, thus *s* is a mapping from one functor type to another. In category theory such mappings are called “natural transformation” if they satisfy the law *set_map*. For this reason the functor *C* is called “natural” in HOL. The transformation *s* is usually called a “set-function” in HOL.

Container Types

Natural functors model container types such as lists or trees which are often used as data types in programming. They consist of a “shape” (the container) and a “content” (the contained values). The type of the contained values is specified by the functor’s type parameter, the container shape is determined

by the functor. Thus the same shape may be instantiated to have content of arbitrary type.

In this view the set-function s returns the content of a container value as the set of the content values. The function $m f$ replaces content values by applying a function f to them. The law **set_map** states that the content after applying f lifted by the mapper is the same as applying f to each value of the original content. The law **map_cong0** states that a function lifted by m has no effect on the shape and only affects the content: if f and g have the same effect on the content of x the lifted functions $m f$ and $m g$ already have the same effect on the whole container x .

As an example consider the functor $(nat, 'a) fun$ with mapper \circ (see Section 6.11.2). Its values are functions from natural numbers to the values of $'a$. They can be viewed to be infinite containers of numbered values of $'a$. The common shape is the infinite sequence of slots numbered by natural numbers. Together with the set-function **range** $:: ('a \Rightarrow 'b) \Rightarrow 'b set$ (see Section 8.7.2), which maps every function to the set of all its possible result values, this is a natural functor. Viewed as a container the content of a function is the set of its possible result values, which is retrieved by **range**. It is easily verified that the laws **set_map** and **map_cong0** are satisfied.

Another example is the functor **set** itself. Using the identity id as set-function, the laws are trivially satisfied, therefore **set** is a natural functor. The container shape is the set, the content consists of the set elements.

Natural Multivariate Functors

As for functors (see Section 6.11.2) the concept of natural functors can be extended to type constructors with more than one parameter. Then a separate set-function s_i is required for every type parameter $'a_i$. A multivariate functor C with n type parameters $'a_1, \dots, 'a_n$ is natural, if there are functions $s_i :: ('a_1, \dots, 'a_n) C \Rightarrow 'a_i set$ for $i = 1, \dots, n$ satisfying the laws

```

set_map:
   $s_1 \circ (m f_1 \dots f_n) = (image f_1) \circ s_1$ 
  ...
   $s_n \circ (m f_1 \dots f_n) = (image f_n) \circ s_n$ 
map_cong0:
   $\llbracket \bigwedge z_1. z_1 \in s_1 x \implies f_1 z = g_1 z;$ 
  ...;
   $\bigwedge z_n. z_n \in s_n x \implies f_n z = g_n z \rrbracket$ 
   $\implies (m f_1 \dots f_n) x = (m g_1 \dots g_n) x$ 

```

Seen as container type, a natural multivariate functor has content of different types corresponding to its type parameters. Every set-function s_i retrieves the set of content values of type $'a_i$. The function $(m f_1 \dots f_n)$ lifted by

the mapper selectively applies its n argument functions to the corresponding content values.

As an example consider the multivariate functor *prod* with mapper *map_prod* (see Section 6.11.2). Using the functions $\lambda(x,y). \{x\}$ and $\lambda(x,y). \{y\}$ as set-functions, which return the singleton sets of the first or second component, respectively, *prod* is a natural (multivariate) functor. The type $(\text{'a}_1, \text{'a}_2)$ *prod* is a container type, having as content a single value of type 'a_1 and a single value of type 'a_2 .

As for functors the concept of natural functors can immediately be extended to arbitrary polymorphic type expressions. Like functors, natural functors are composable: if all type constructors used in a type expression are natural functors then the resulting type is also a natural functor and the set-functions can be constructed from those of the type constructors.

Live and Dead Type Parameters

The multivariate functor *fun* with mapper *map_fun* (see Section 6.11.2) is no natural functor because it has a contravariant type parameter. The definition of natural functors in HOL requires that all its type parameters must be covariant. However, if the first type parameter is instantiated by an arbitrary type, such as in $(\text{nat}, \text{'a})$ *fun* (see above), the resulting type expression always denotes a natural functor.

This situation is described by saying that *fun* is a natural functor with one “dead” type parameter (the first one). The second type parameter is called “live”. In general a natural functor may have several dead and live type parameters. A set-function only exists for each live type parameter and the mapper lifts only functions for the live type parameters.

Seen as container type, the dead type parameters contribute to the container shape, the live type parameters specify the types of content values. In the example $(\text{nat}, \text{'a})$ *fun* the type *nat* determines the shape as being an infinite sequence of numbered slots. If it is replaced by *bool* the type $(\text{bool}, \text{'a})$ *fun* is a container with only two slots for content values of type 'a .

A dead type parameter may also be covariant, then the corresponding values are also considered to be parts of the shape instead of being content. Whether a type parameter is live or dead is determined by the type of the mapper and by the number and types of the set-functions.

If natural functors are composed all type parameters of a constructor used on a dead parameter position of another constructor become dead for the resulting type. As an example the type $(\text{'a}_1 \text{ set}, \text{'a}_2)$ *fun* (see Section 6.11.2) has the dead parameter 'a_1 because *set* occurs on the position of the dead parameter of *fun*. The parameter 'a_2 is live and the corresponding set-function is *range*.

6.12.2 Predicators and Relators

There are two other useful functions for every natural functor C with mapper m and set-function s . These are called “predicator” and “relator”.

The Predicator

A predicator is a function $p :: ('a \Rightarrow bool) \Rightarrow ('a\ C \Rightarrow bool)$, i.e., it lifts predicates (see Section 6.1.1) on type $'a$ to predicates on type $'a\ C$. It must satisfy the law

$$pred_set: p\ P = (\lambda x. \forall v \in (s\ x). P\ v)$$

which directly specifies a definition for p : for a container value x the lifted predicate $p\ P$ is valid if the original predicate P is valid for all content values retrieved by $s\ x$ (for the syntax see Section 8.4.3). Since p can be constructed in this way from the set-function s , a predicator is available for every natural functor.

The predicator for $(nat, 'a)\ fun$ is the function $\lambda P. \lambda f. \forall v \in (range\ f). P\ v$. The lifted predicate tests the possible result values of f whether the predicate P is valid for all of them.

For a natural multivariate functor the predicator has the type $('a_1 \Rightarrow bool) \Rightarrow \dots \Rightarrow ('a_n \Rightarrow bool) \Rightarrow (('a_1, \dots, 'a_n)\ C \Rightarrow bool)$. It lifts n predicates on the type parameters to a combined predicate on the functor type. The law $pred_set$ is accordingly extended to apply to each set of content values the predicate corresponding to their type.

The predicator for $prod$ is the function $pred_prod$ (see Section 8.6.3). It lifts two predicates P_1, P_2 on $'a_1$ and $'a_2$, respectively, to the predicate on pairs of type $('a_1, 'a_2)\ prod$ which tests the first component by P_1 and the second component by P_2 .

Predicators are composable: if a type expression consists of composed natural functors, the predicators can be composed in the same way to yield a predicator for the resulting type. As example the predicator for the type $('a_1, ('a_2, 'a_3)\ prod)\ prod$ is $\lambda P_1\ P_2\ P_3. pred_prod\ P_1\ (pred_prod\ P_2\ P_3)$.

The Relator

A relator is a function $r :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a\ C \Rightarrow 'b\ C \Rightarrow bool)$, i.e., it lifts relations (see Section 6.1.3) between types $'a$ and $'b$ to relations between types $'a\ C$ and $'b\ C$. It must satisfy the law

$$\begin{aligned} in_rel: r\ R = \\ (\lambda x\ y. \exists z :: ('a \times 'b)\ C. \forall (a, b) \in (s\ z). R\ a\ b \\ \wedge (m\ fst)\ z = x \wedge (m\ snd)\ z = y) \end{aligned}$$

which specifies a definition for r using both s and m (for the inner syntax used here see Chapter 8). It tests two containers $x :: 'a\ C$ and $y :: 'b\ C$ whether they are related by the relation lifted from $R :: 'a \Rightarrow 'b \Rightarrow \text{bool}$. For this it uses a container z of type $('a \times 'b)\ C$ where all contained values are pairs (see Section 6.6). Using the set-function s it retrieves these pairs and tests whether their components are related by R . Using the mapper m applied to the selector functions fst and snd for the components of a pair (see Section 6.6) it replaces every contained pair in z by its first or second component, respectively, which must result in the containers x and y tested for being related. Since r can be constructed in this way from s and m , a relator is available for every natural functor.

The construction of the container z of type $('a \times 'b)\ C$ for the containers x and y of type $'a\ C$ and $'b\ C$ is a generalization of the well-known function “zip” in functional programming, which constructs the list of pairs of corresponding elements from two lists. Therefore this construction is called “zip construction” in this introduction, z is called a “zipped container”, and x and y are called “unzipped containers”.

Trivially, a relator r always lifts equality to equality: $r\ (=)$ has type $'a\ C \Rightarrow 'a\ C \Rightarrow \text{bool}$ and relates two containers if they have the same shape and their content is the same.

The relator for $(\text{nat}, 'a)\ \text{fun}$ is the function $\lambda R. \lambda f\ g. \forall i :: \text{nat}. R\ (f\ i)\ (g\ i)$. The lifted relation tests for all natural numbers i whether the result values $f\ i$ and $g\ i$ are related by R .

For a natural multivariate functor the relator has the type $('a_1 \Rightarrow 'b_1 \Rightarrow \text{bool}) \Rightarrow \dots \Rightarrow ('a_n \Rightarrow 'b_n \Rightarrow \text{bool}) \Rightarrow (('a_1, \dots, 'a_n)\ C \Rightarrow ('b_1, \dots, 'b_n)\ C \Rightarrow \text{bool})$. It lifts n relations between types $'a_i$ and $'b_i$ to a combined relation between the functor types constructed from the $'a_i$ and $'b_i$, respectively. The law in_rel is extended accordingly, using $(m\ \text{fst} \dots \text{fst})$ and $(m\ \text{snd} \dots \text{snd})$ to extract the unzipped containers x and y from the zipped container z .

The relator for prod is the function rel_prod (see Section 8.6.3). It lifts two relations R_1, R_2 between $'a_1, 'b_1$ and $'a_2, 'b_2$, respectively, to the relation between pairs of type $('a_1, 'a_2)\ \text{prod}$ and $('b_1, 'b_2)\ \text{prod}$ which compares the first components by R_1 and the second components by R_2 .

Like predicates relators are composable: if a type expression consists of composed natural functors, the relators can be composed in the same way to yield a relator for the resulting type. As example the relator for the type $('a_1, ('a_2, 'a_3)\ \text{prod})\ \text{prod}$ is $\lambda R_1\ R_2\ R_3. \text{rel_prod}\ R_1\ (\text{rel_prod}\ R_2\ R_3)$.

Predicate and Relator for Functions

Although fun has no set-function for its contravariant first type parameter, a predicate and relator can be defined for it. HOL provides the functions

$$\begin{aligned} \text{pred_fun} &:: ('p_1 \Rightarrow \text{bool}) \Rightarrow ('p_2 \Rightarrow \text{bool}) \Rightarrow ('p_1, 'p_2) \text{ fun} \Rightarrow \text{bool} \\ &\equiv \lambda p_1 p_2 f. \forall x. p_1 x \longrightarrow p_2 (f x) \end{aligned}$$

and

$$\begin{aligned} \text{rel_fun} &:: ('p_1 \Rightarrow 'q_1 \Rightarrow \text{bool}) \Rightarrow ('p_2 \Rightarrow 'q_2 \Rightarrow \text{bool}) \\ &\Rightarrow ('p_1, 'p_2) \text{ fun} \Rightarrow ('q_1, 'q_2) \text{ fun} \Rightarrow \text{bool} \\ &\equiv \lambda r_1 r_2 f g. \forall x y. r_1 x y \longrightarrow r_2 (f x) (g y) \end{aligned}$$

which lift predicates or relations, respectively, from argument and result type to the function type.

As an example using the predicate `evn` from Section 6.7.1, the partial application `pred_fun evn evn` is the predicate of type $(\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{bool}$ which tests whether a function on natural numbers maps all even numbers to even numbers.

As for the mapper `map_fun` the predicator and relator for an instantiation $(T, 'a) \text{ fun}$ such as $(\text{nat}, 'a) \text{ fun}$ (see Section 6.11.2) can be obtained by partial application. The predicator for $(\text{nat}, 'a) \text{ fun}$ (see above) is equal to `pred_fun` $(\lambda_. \text{True})$, the relator for $(\text{nat}, 'a) \text{ fun}$ (see above) is equal to `rel_fun` $(=)$. Note that these functions are also polymorphic for the argument type, so they can be used for functions of arbitrary types $'p_1 \Rightarrow 'p_2$.

Since functions with multiple arguments in curried form (see Section 6.6.1) have functions as intermediate result values the lifting can be iterated over multiple arguments. For example, a relation r on the result type t can be lifted to binary functions of type $t_1 \Rightarrow t_2 \Rightarrow t$ by `rel_fun` $(=)$ `(rel_fun (=) r)` which is equivalent to the relation on functions $\lambda f g. \forall x y. r (f x y) (g x y)$.

6.12.3 Bounded Natural Functors

A natural functor is called “bounded”, if the size of its content is restricted. The shape is not arbitrarily “elastic” to admit content value sets of arbitrary large cardinality. For a multivariate functor to be bounded, the content sets for all live type parameters must be bounded. In this introduction (and also in other HOL documentation) bounded natural functors are usually abbreviated as BNF.

The natural functor `set` (see Section 6.12.1) is not bounded. Depending on the type substituted for the type parameter $'a$, a container value of type $'a$ `set` may have arbitrary large cardinality, because the universal set `UNIV` (see Section 8.4.1) always contains all values of type $'a$.

The natural functor `prod` is bounded, because every container value has only two content values, independent of their types. The natural functor $(\text{nat}, 'a) \text{ fun}$ is bounded, because every container has countably many content

values, even if `'a` is instantiated by a type with more values, such as the type of real numbers.

More generally, the bound may not depend on the live type parameters, but it may depend on the dead type parameters, because they contribute to the shape instead of the content. Therefore the natural functor `fun` with one dead and one live type parameter is bounded by the cardinality of the function argument type which is specified by the dead type parameter.

There is a second requirement for a bounded natural functor, stated as the law

$$(r \ R_1 \dots R_n) \ OO \ (r \ S_1 \dots S_n) \leq r \ (R_1 \ OO \ S_1) \ \dots \ (R_n \ OO \ S_n)$$

where `r` is the relator (see Section 6.12.2) of a natural functor with `n` live type parameters. The operator `OO` is relation composition (see Section 6.1.3), the ordering \leq on relations is the usual “stronger as” ordering (see Section 8.7.2). The other direction \geq of the ordering is automatically satisfied, together the law states distributivity of the relator `r` with respect to relation composition. Therefore the law is also called “subdistributivity of the relator”.

BNFs are composable: Whenever in a type expression all type constructors occurring on live type parameter positions are bounded, then the type denoted by the expression is bounded as well. The type `('a1 set, 'a2) fun` (see Section 6.12.1) is a BNF because the constructor `set`, which is not bounded, occurs on the position of the dead parameter of `fun`.

Cardinalities

HOL represents cardinalities by specific binary ordering relations of type `('a × 'a) set` (see Section 6.6.2). The predicate `card_order :: ('a × 'a) set ⇒ bool` determines such relations. The predicate `regularCard :: ('a × 'a) set ⇒ bool` test the specific property of being “regular”, the predicate `cinfinite :: ('a × 'a) set ⇒ bool` tests a cardinality for being infinite.

The cardinality of the natural numbers, usually denoted by \aleph_0 in mathematics, corresponds to the ordering (\leq) (see Section 6.2.2) on type `nat`, HOL defines the constant name `natLeq` for it. The function `card_suc` maps a cardinality to the next higher cardinality. The cardinality of a set can be determined by the function `card_of :: 'a set ⇒ ('a × 'a) set` and cardinalities can be compared by `ordLess2 :: ('a × 'a) set ⇒ ('a × 'a) set ⇒ bool` for being strictly ranked. The syntax `/A/` for `card_of A` and the operator name `<o` for `ordLess2` are available after using the command

```
unbundle cardinal_syntax
```

on theory level, which is available after importing the theory `Main` (see Section 1.1.2).

The theories *HOL.BNF_Cardinal_Order_Relation* and *HOL.BNF_Cardinal_Arithmetic* introduce many rules which can be used to prove goals about cardinalities.

6.12.4 Registering Bounded Natural Functors

Some type definition mechanisms in HOL, as described in Chapter 7, require the information whether a type constructor is a BNF. Therefore HOL supports registering BNFs (i.e., adding them to internal HOL data structures so that they can be retrieved by HOL mechanisms). In most cases this need not be done explicitly, because the type definition mechanisms automatically register newly defined types as bounded natural functor, if applicable.

The **bnf** Command

A polymorphic type with n type parameters is registered as BNF using the outer syntax command

```
bnf name: "type"
    map: "term" sets: "term1" ... "termm" bd: "bterm" <proof>
```

where *term* is a function of a mapper type for the *type*, the *term*_{*i*} are set functions for the *type* and *bterm* is an infinite regular cardinality which is strictly larger than that of all sets returned by the set functions. As usual, the terms need not be quoted, if they consist of a single identifier. The *map*:, *sets*:, and *bd*: are keywords and are part of the command syntax.

The type is registered with m live type parameters, the remaining parameters are considered dead. The live type parameters are determined from the types of the set functions, the type of the mapper function must match exactly, otherwise an error is signaled.

The command generates $7+2*m$ goals which must be proved in the <*proof*>. These are the two functor laws *comp* and *id* (see Section 6.11.1), the law *map_cong0* and m laws *set_map* for a natural functor (see Section 6.12.1), m laws of the form $|term_i| <_o bterm$ for the cardinality bound, the goals *card_order bterm*, *regularCard bterm*, and *cinfinite bterm*, and the law for the subdistributivity of the (automatically constructed) relator (see Section 6.12.3).

After the proof of these facts is completed HOL uses them to construct and automatically prove about 40 other named facts. They can be displayed using the “Print Context” tab in the Query panel as described in Section 2.6.6, if the cursor is positioned after the proof of the **bnf** command.

All fact names use the *name* specified in the **bnf** command as prefix. In some cases *name* may be omitted and is automatically constructed from the *type*. This is possible if the *type* is a single type constructor *C* applied to type variables, then *C* is used as prefix.

As an example, the natural multivariate functor *prod* (see Section 6.12.1) can be registered as a BNF using the command

```
bnf myprod: "('a, 'b) prod"
  map: map_prod sets: "λ(x,y).{x}" "λ(x,y).{y}" bd: natLeq <proof>
```

Since both set functions return finite sets (singletons) the cardinality *natLeq* of the natural numbers can be used as strictly larger infinite bound. A name must be specified because HOL already registers this type as BNF under its type name *prod*.

The natural functor *(nat, 'a) fun* (see Section 6.12.1) can be registered as a BNF by

```
bnf natfun: "((nat, 'a) fun)"
  map: "(o) :: ('b ⇒ 'c) ⇒ ((nat, 'b) fun) ⇒ ((nat, 'c) fun)"
  sets: "range :: ((nat, 'a) fun) ⇒ 'a set"
  bd: "card_suc natLeq" <proof>
```

The types for the mapper and set function must be specified to restrict the general functions *(o)* and *range* to the specific functions on natural numbers. The cardinality of the content set is not larger than that of the natural numbers, therefore its successor *card_suc natLeq* can be used as strictly larger bound.

For an example that registers *fun* as a BNF with one dead and one live parameter including the proof see [1, Section 6.2, introductory examples]

Registering Relator and Predicate

If required, HOL automatically constructs the relator and predicate (see Section 6.12.2) for a registered bounded natural functor, using names of the form *pred_name* and *rel_name*. However, it is also possible to define these as named functions and add them to the registration, so that they can be used by HOL. This is done by an extended **bnf** command of the form

```
bnf ... rel: "rterm" pred: "pterm" <proof>
```

where *rterm* is a term for the relator and *pterm* is a term for the predicate. The predicate is optional, the relator may only be omitted together with the predicate.

In this extended form the command generates the additional goals *in_rel* and *pred_set* which must be proved to show that the specified functions actually correspond to the definition of a relator or predicate, respectively. As an example *prod* can be registered together with its relator and predicate by

```

bnf myprod: "('a, 'b) prod"
  map: map_prod sets: " $\lambda(x,y). \{x\}$ " " $\lambda(x,y). \{y\}$ " bd: natLeq
  rel: rel_prod pred: pred_prod <proof>

```

Registering the Mapper Function

The command **bnf** *name*: "*type*" *map*: "*term*" ... does not register the *term* specified for the mapper function as a functor (see Section 6.11.3). This may cause problems when HOL needs only the functor property of *type* later. An example is the definition of a quotient type (see Section 7.3). It signals a warning if a quotient type is defined for a type that has not been registered as a functor.

To make a registered BNF known as a functor it must explicitly be registered as functor in the form

```

functor "term" <proof>

```

where *term* denotes the mapper function specified in the **bnf** command.

Although the **bnf** command does not register the type as functor, it provides the facts *comp* and *id* which must be proved for the **functor** command, named *map_comp* and *map_id0* prefixed by the *name* specified in the **bnf** command. Therefore it is easiest to register a type as functor after registering it as BNF, reusing the facts provided by the **bnf** command for the proof of the **functor** command. After type *prod* has been registered as BNF as above it can be registered as functor using

```

functor map_prod
  using myprod.map_comp by fastforce (simp add: myprod.map_id0)

```

Displaying Registered Bounded Natural Functors

All registered BNFs can be displayed by the command

```

print_bnfs

```

It prints a list of short descriptions of all BNFs which have been registered at the position of the command to the Output panel (see Section 1.2.3).

6.13 Transfer

This section describes the transfer mechanism in detail and is only intended for the interested reader. Usually the transfer mechanism can be used without knowing the details, as described in Section 6.14.5.

The relator `rel_fun` for the function type (see Section 6.12.2) lifts a relation R_a between two argument types and a relation R_r between two result types to a relation between the corresponding function types. It is defined in a way that the following law is valid:

$$\llbracket (\text{rel_fun } R_a \ R_r) \ f \ g; \ R_a \ x \ y \rrbracket \implies R_r \ (f \ x) \ (g \ y)$$

It states that if two functions f and g are related by `rel_fun` R_a R_r and the arguments x and y are related by R_a then the function results are related by R_r .

Thus it is possible to replace the function application term $(f \ x)$ by $(g \ y)$ and to repeat this for enclosing function applications in a complex term while changing the overall result value to a related one. If that is done for the body of a lambda term (see Section 2.2.5) the resulting lambda term is also related (as function) to the original one.

In Isabelle terms of the inner syntax are always built from function applications and lambda terms (all other binder syntax forms are defined as specific combinations of both). Thus the replacement schema described above can be used for arbitrary terms. By replacing all constants (for values and functions) occurring in a term by related ones, a new term results which usually has another type, but its value is related to that of the original term. This is called “transfer”: the term is “transferred” from a type to a different target type.

In this context the relations used for the transfer between corresponding types are called “transfer relations”.

As an example consider a relation `rnb` between type `bool` and the target type `nat` which relates `True` with `1` and `False` with `0`. Then `(rel_fun rnb rnb)` is a transfer relation between operations on `bool` and operations on `nat`. It can easily be verified that it relates the boolean conjunction \wedge with multiplication $*$ and the boolean disjunction \vee with the maximum function `max` (see Section 6.2.2). Then the boolean term `False \vee (True \wedge False)` can be transferred to the term `max 0 (1 * 0)`.

Note that this is an artificial example, transfer is not intended to relate types such as `nat` and `bool`.

6.13.1 Transfer Rules

To transfer a term a replacement must be found for every constant occurring in it. This is done with the help of simple theorems of the form

$$R \ \text{term} \ \text{term}'$$

where R is a transfer relation applied to the two terms. It states that `term` and `term'` are related by R . In this context such theorems are called “transfer rules”.

To find a replacement for a constant it is matched with the *term*' of every known transfer rule. A match yields both the *term* for the replacement and the transfer relation *R* which is needed for determining the arguments relation for the next enclosing function application. Backtracking is used to determine a consistent set of transfer rules for all constants in a term. If for atleast one constant no transfer rule matches the term cannot be transferred. Note that HOL always matches constants with the second relation argument term, thus the transfer rules are applied "from right to left". The target type is always the left argument type of the transfer relation.

Transfer Rules for Functions

In a transfer rule for functions the transfer relation *R* is constructed with the help of *rel_fun*, so the rule has the form

```
(rel_fun Ra Rr) term term'
```

As an alternative syntax HOL provides the operator name (*====>*) for *rel_fun* so that the rule can be specified as

```
(Ra ====> Rr) term term'
```

The operator name (*====>*) is only available after using the command

```
unbundle lifting_syntax
```

on theory level.

This syntax is intended to make the term for the relation syntactically resemble the (function) types of *term* :: $T_a \Rightarrow T_r$ and *term'* :: $T_a' \Rightarrow T_r'$; *R_a* is the transfer relation between the argument types *T_a* and *T_a'*, *R_r* is the transfer relation between the result types *T_r* and *T_r'*.

If the functions have types of the form $T_{a,1} \Rightarrow \dots \Rightarrow T_{a,n} \Rightarrow T_r$ with *n* arguments the relator *rel_fun* can be iterated as described in Section 6.12.2, yielding a transfer rule of the form

```
(Ra,1 ====> ... ====> Ra,n ====> Rr) term term'
```

Example

For transferring from *bool* to *nat* the transfer relation *rnb* can be defined inductively (see Section 6.7) as

```
inductive rnb :: "nat ⇒ bool ⇒ bool"
where "rnb 1 True" | "rnb 0 False"
```

Then the defining rules are transfer rules which can be used to replace the value constants *True* and *False*. The transfer rules for replacing the boolean operations \wedge and \vee are

$$\begin{aligned} (rnb \implies rnb \implies rnb) \ (*) \ (\wedge) \\ (rnb \implies rnb \implies rnb) \ max \ (\vee) \end{aligned}$$

With all four rules together the term $False \vee (True \wedge False)$ can be transferred to the term $max \ 0 \ (1 * 0)$.

Relating by Equality

A specific case occurs if the equality relation $(=)$ (see Section 6.2.1) is used as transfer relation. It is polymorphic, but always relates a type with itself and also every value with the same value. Thus the transfer rule $(R_a \implies (=)) \ f \ g$ states that if x and y are related by R_a then $(f \ x) = (g \ y)$. If $(g \ y)$ is transferred using this rule, the value of this term remains unchanged. More generally, an arbitrary term can be transferred with preserving its value if the outermost function or operator is replaced using a transfer relation of the form $A \implies (=)$.

If transfer is applied in this way to formulas (terms of type *bool*, see Section 2.5.1), their truth value is preserved by the transfer. This makes transfer to a very powerful tool for constructing proofs: to prove a formula about values of one or more types it is transferred to a formula about values of target types related by transfer relations, then this formula is proved. Often this proof already exists and can thus be reused.

Transfer can be extended to arbitrary propositions involving meta operators using the equivalence of derivation rules and boolean terms described in Section 8.1.4.

Transferring Equations

If the term to be transferred is an equation of two terms the outermost operator is the equality as a binary predicate on the type of both terms. It can be replaced by the equality on a related type if the transfer rule

$$(R_a \implies R_a \implies (=)) \ (=) \ (=)$$

is valid. The first occurrence of $(=)$ states that the related functions preserve their values, the second and third occurrences denote the related instances of equality on the target type and the original type.

For the example relation *rnb* the corresponding transfer rule

$$(rnb \implies rnb \implies (=)) \ (=) \ (=)$$

together with the four transfer rules above can be used to transfer the equation $(False \vee (True \wedge False)) = False$ to the equation $\max 0 (1 * 0) = 0$ while preserving the validity of the equations.

Transferring Propositions with Variables

If a proposition contains universally bound variables it has the form $\bigwedge x_1 \dots x_n. P$ (see Section 2.5.3). This is equivalent to the formula $\forall x_1 \dots x_n. P$, as described in Section 8.1.4. Here the outermost operator is the quantifier \forall which also corresponds to a function (see Section 8.1.3) and can be replaced for transfer.

HOL has a predefined transfer rule which replaces \forall on a type by \forall on any target type. However, this is only valid if the transfer relation R between both types is bi-total. Therefore the property *bi_total* R (see Section 6.1.3) must be provided as a fact, only then the transfer rule can be used.

If the relation is only total on the original type the quantifier \forall must be replaced by a bounded quantifier over the domain of R , i.e., only the values of the target type which are related with a value in the original type. It has the form $\forall x. \text{Domainp } R \ x \longrightarrow \dots$ which restricts x to the domain of R (see Section 6.1.3). HOL has a predefined transfer rule for this replacement, to be used the property *right_total* R must be provided as a fact. If R is not right-total the (unbounded) quantifier \forall cannot be transferred.

HOL uses facts of the form

Domainp $R = (\lambda x. \dots)$

specifying the domain as an explicit predicate. If available, the explicit predicate is used instead of *Domainp* R in a transferred term.

Note that the example relation *rn**b* is right-total, but not bi-total. Therefore the additional rules

right_total *rn**b*
Domainp *rn**b* = $(\lambda n. n \leq 1)$

can be used to transfer the proposition

$\bigwedge a \ b \ c. (a \vee (b \wedge c)) = ((a \vee b) \wedge (a \vee c))$

to the proposition

$\bigwedge a \ b \ c. \llbracket a \leq 1; b \leq 1; c \leq 1 \rrbracket \implies$
 $\max a (b * c) = \max a \ b * \max a \ c$

6.13.2 The *transfer* Method

HOL supports transfer by the proof method

transfer

The method only affects the first goal, by transferring it. The method uses the dynamic fact set (see Section 2.6.3) *transfer_rule* for matching the constants in the goal. If no consistent set of matches for all constants can be found in *transfer_rule* the goal will be partially transferred with additional goals for the replacement of the unmatched constants.

Facts in *transfer_rule* may have additional assumptions, i.e., they may be genuine derivation rules with a transfer relation application as conclusion. However, a constant only matches such a rule successfully if all goals resulting from the assumptions can be proved by applying other rules in *transfer_rule* (which need not have transfer relation applications as conclusions).

Typical assumptions in transfer rules are properties such as *right_total* or *bi_unique* (see Section 6.1.3) for the transfer relation. Thus facts stating these properties for specific relations must also be added to the *transfer_rule* set to be used for transfer.

HOL uses another dynamic fact set *transfer_domain_rule* for explicit specifications of the domain of transfer relations.

Note that the *transfer* method has no arguments, its effect is fully determined by the facts in the sets *transfer_rule* and *transfer_domain_rule*. Usually these sets are populated automatically by HOL (see Section 6.14) so that the *transfer* method is applicable for goals about values of certain types.

Example

The *transfer* method can be applied to the example using the transfer relation *rnb* as follows.

In the definition of *rnb* the defining rules are directly added to the *transfer_rule* set:

```
inductive rnb :: "nat  $\Rightarrow$  bool  $\Rightarrow$  bool"
where [transfer_rule]: "rnb 1 True"
      | [transfer_rule]: "rnb 0 False"
```

The transfer rules for the operations and the equality must be added and proved as theorems:

```
unbundle lifting_syntax
theorem [transfer_rule]: "(rnb  $\implies$  rnb  $\implies$  rnb) (*) ( $\wedge$ )"
  using rnb.simps by fastforce
```

```

theorem [transfer_rule]: "(rnb ==> rnb ==> rnb) max (V)"
  using rnb.simps by fastforce
theorem [transfer_rule]: "(rnb ==> rnb ==> (=)) (=) (=)"
  using rnb.simps by fastforce

```

Then it is possible to use *transfer* on goals without variables:

```

theorem "(False ∨ (True ∧ False)) = False"
  apply(transfer) by simp

```

Here the *transfer* method replaces the goal by the transferred goal

```

max 0 (1 * 0) = 0

```

which can be proved (like the original goal) by *simp*.

For goals with variables the fact that *rnb* is right-total must be added to the *transfer_rule* set:

```

theorem [transfer_rule]: "right_total rnb"
  using rnb.simps right_total_def by metis

```

This fact allows to replace the quantifier \forall and makes it possible to use *transfer* on goals with variables:

```

theorem "(a ∨ (b ∧ c)) = ((a ∨ b) ∧ (a ∨ c))" apply(transfer)
  using rnb.simps by fastforce

```

Here the *transfer* method replaces the goal by the transferred goal

```

∧ a b c. [Domainp rnb a; Domainp rnb b; Domainp rnb c] ==>
  max a (b * c) = max a b * max a c

```

which can be proved using *fastforce*.

The domain predicate for *rnb* can be made explicit by using the dynamic fact set *transfer_domain_rule*:

```

theorem [transfer_domain_rule]: "Domainp rnb = (λn. n ≤ 1)"
  using rnb.simps by force

```

Then the *transfer* method in

```

theorem "(a ∨ (b ∧ c)) = ((a ∨ b) ∧ (a ∨ c))" apply(transfer)
  using le_eq_less_or_eq by fastforce

```

replaces the goal by the transferred goal

```

∧ a b c. [a ≤ 1; b ≤ 1; c ≤ 1] ==>
  max a (b * c) = max a b * max a c

```

which can be proved by *fastforce* using the additional fact *le_eq_less_or_eq*.

6.13.3 Polymorphic Types

Assume we have a target type T , related by a transfer relation R to a type T' for transfer from T' to T . It may happen that a term containing subterms of type T' also contains subterms of a type $T' \ C$ where C is a type constructor applied to T' , such as $T' \ \text{set}$. Then these subterms must also be transferred, which requires a transfer relation between $T' \ C$ and a corresponding target type. A natural candidate is the type $T \ C$. The transfer relation between $T \ C$ and $T' \ C$ can be constructed from R by using a relator $\text{rel_}C$ (see Section 6.12.2) for C as the lifted relation $\text{rel_}C \ R$.

Whenever C is a natural functor (see Section 6.12.1) such a relator is available and transfer can immediately be applied to all instances of the polymorphic type $'a \ C$, if $'a$ is instantiated by a type related by a transfer relation to a target type.

Using the example transfer relation rnb , the relation $\text{rel_set} \ \text{rnb}$ relates the target type nat set with the type bool set and can be used to specify corresponding transfer rules.

Multivariate Functors

As usual, this can be extended to multivariate natural functors C with n type parameters. An instance $(T_1', \dots, T_n') \ C$ is related to a corresponding target type by the transfer relation $(\text{rel_}C \ R_1 \ \dots \ R_n)$ where R_i is the transfer relation relating T_i' to a target type T_i . If a parameter T_i' of C is not to be transferred, the equality ($=$) is used as R_i and T_i is the same as T_i' .

Note that lifting transfer relations R_a and R_r between argument and result types to a function type as $(\text{rel_fun} \ R_a \ R_r)$ or $(R_a \implies R_r)$, which is the basis of the transfer mechanism, is a special application of this.

Since relators can be composed according to the structure of a type expression (see Section 6.12.2) it is possible to systematically construct relators and thus transfer relations for arbitrary types which are composed from natural functors.

Transferring Polymorphic Functions

A function with a type instance $T' \ C$ as argument or result type is often itself polymorphic and defined for the polymorphic type $'a \ C$, i.e., for arbitrary types as parameters. Then it is not possible to specify a transfer rule for the function in the way described above, because no common transfer relation for the type parameter is available (other than trivial polymorphic relations like $=$).

However, there are polymorphic functions for which the definition does not depend on the type parameter(s) at all. An example is the function `image`

(see Section 6.11.1). For a function $f :: 'a \Rightarrow 'b$ and a set A of argument values $\text{image } f \ A$ is the set of result values of f applied to all values in A . This definition is completely independent of the actual types substituted for the type parameters $'a$ and $'b$.

Such functions are called “parametrically polymorphic” . Polymorphic functions with a direct, inductive, or recursive definition are usually parametrically polymorphic, whereas functions defined by overloading (see Section 2.4.1) are usually not, because different definitions may be used for different type parameters.

Since parametrically polymorphic functions work in the same way for all values of its type parameters it is possible to replace these values in the argument in arbitrary ways and they will be replaced in the result in related ways. Therefore it is possible to specify a transfer rule for the function using *all possible relations* on *all possible types* substituted for the type parameters, i.e., the transfer relations for the type parameters occur as universally bound variables in the transfer rule.

An example is the transfer rule

```
theorem image_transfer [transfer_rule]:
  " $\bigwedge R_1 \ R_2. ((R_1 \implies R_2) \implies \text{rel\_set } R_1 \implies \text{rel\_set } R_2) \text{ image image}$ "
  by (simp add: rel_fun_def rel_set_def) blast
```

which is provided by HOL. It states that for the function *image* an instance for an element type T is always related to an instance of *image* for any other element type. Thus transfer will syntactically preserve occurrences of *image* although its type instance is transferred. Note how the relation $((R_1 \implies R_2) \implies \text{rel_set } R_1 \implies \text{rel_set } R_2)$ is constructed according to the structure of the type expression $('a_1 \Rightarrow 'a_2) \Rightarrow ('a_1 \text{ set}) \Rightarrow ('a_2 \text{ set})$ for the type of *image* from the corresponding relators. The type parameters $'a_1$ and $'a_2$ are replaced by the universally bound variables R_1 and R_2 for the arbitrary relations.

Such transfer rules for a single polymorphic function are called “parametricity rules” because they are equivalent to the proposition that the function is parametrically polymorphic. HOL provides such parametricity rules for most of the predefined polymorphic functions, such as most functions for sets described in Section 8.4.3.

Normal transfer rules between two polymorphic functions may be specified using universally bound variables in the same way if both functions are parametrically polymorphic.

Some parametrically polymorphic functions are independent of the type parameters but cannot be related by arbitrary relations. Then their parametricity rule has additional assumptions which restrict the transfer relation. An example is the equality as a polymorphic function. It can only be transferred by bi-unique transfer relations, otherwise it would identify values in

one type which are different in the other. The corresponding parametricity rule provided by HOL is

```
theorem eq_transfer [transfer_rule]:
  "bi_unique R ==> (R ==> R ==> (=)) (=) (=)"
  by (auto simp add: bi_unique_def rel_fun_def)
```

6.14 Quotients and Lifting

Assume a relation (see Section 6.1.3) $R :: T \Rightarrow T' \Rightarrow \text{bool}$ between types T and T' which is right-total and right-unique. This implies the existence of a surjective function $\text{abs}_{T'} :: T \Rightarrow T'$ mapping every value of T to the single value of T' which is related by R .

Since R need not be left-unique $\text{abs}_{T'}$ need not be injective and gives rise to the definition of the equivalence relation $E :: T \Rightarrow T \Rightarrow \text{bool}$ which relates all values mapped by $\text{abs}_{T'}$ to a common value in T' . Since R need not be left-total $\text{abs}_{T'}$ is underspecified (see Section 2.2.2) for all values of T which are not related to a value of T' by R . This gives rise to the definition of the set $D :: T \text{ set}$ of all values in T which are related to a value of T' . The set D is the domain of R , represented as a set. Also, E is in general a partial equivalence relation and does not relate values outside of D , not even with themselves (which is the case for the values in D since as an equivalence relation E must be reflexive).

The predicate *reflp* (see Section 8.8.1) may be used to specify that E is reflexive on the whole type T , thus the proposition *reflp* E is equivalent to the proposition *left_total* R and also to $D = (\text{UNIV} :: T)$ where $(\text{UNIV} :: T)$ is the universal set of type T (see Section 8.4.1).

6.14.1 Abstract and Raw Types

In this situation the values of type T' are considered to be “more abstract” than those of T because there may be several distinct related values in T so that they may represent more details and there may be values in T which have no related values in T' at all. Therefore the type T' is called an “abstract type” and T is called its associated “raw type”. Usually, in HOL for an abstract type only one relation R is considered and thus only one associated raw type, whereas T may be the raw type for several abstract types.

The function $\text{abs}_{T'}$ which maps values of the raw type to values of the abstract type is called “abstraction function” in this context.

The interrelation between R , $\text{abs}_{T'}$ and E can be formally described by the laws

$$E \ v_1 \ v_2 = (E \ v_1 \ v_1 \wedge E \ v_2 \ v_2 \wedge \text{abs}_{T'} \ v_1 = \text{abs}_{T'} \ v_2)$$

$$R \ v \ v' = E \ v \ v \ \wedge \ abs_T' \ v = v')$$

where $E \ v \ v$ is equivalent to the property that v has a related value in T' , i.e., $v \in D$.

Note that the relation R can also be used as transfer relation (see Section 6.13). Then terms are transferred from the abstract type to the raw type.

Example

The relation rnb used as example in Section 6.13 is right-total and right-unique. In this case $bool$ is the abstract type and nat is the raw type. If defined as above the function abs_bool maps 0 to $False$ and 1 to $True$ and is underspecified for all other natural numbers. Thus the equivalence classes are trivially $\{0\}$ and $\{1\}$ (because rnb is also left-unique) and the corresponding equivalence relation E is the relation $rnbE$ which can be defined inductively (see Section 6.7) as

```
inductive rnbE :: "nat  $\Rightarrow$  nat  $\Rightarrow$  bool"
where "rnbE 1 1" | "rnbE 0 0"
```

It is partial in the sense that it only relates values in the set defined as

```
definition rnbD :: "nat set"
where "rnbD  $\equiv$  {0,1}"
```

which are the only values in nat related to values in $bool$ by rnb . Using the restricted equality eq_onp (see Section 6.2.1) $rnbE$ can also be represented as $eq_onp \ (\lambda x. x \in rnbD)$.

Alternatively consider the relation $rnb2$ which relates all natural numbers which are not 0 to $True$. Then abs_bool is fully specified (because $rnb2$ is left-total), it maps 0 to $False$ and all other numbers to $True$. The equivalence classes of $rnb2E$ are $\{0\}$ and the set of all numbers greater 0 , the set $rnb2D$ contains all natural numbers.

6.14.2 Quotient Theorems

Note that there is a 1-to-1 relation between the equivalence classes of E and the values of T' . Therefore an abstract type is also called a “quotient” of its raw type, because in algebra, the notion of a “quotient” is used for sets of equivalence classes.

Representation Values

Since R is right-total every abstract value v' in T' has at least one related value in T . Thus it is possible to select one of these values as “representation value” for v' . A “representation function” $rep_{T'} :: T' \Rightarrow T$ maps every value of T' to a selected representation value.

Due to its definition $abs_{T'}$ maps each representation value back to its abstract value, i.e., the law $abs_{T'} (rep_{T'} v') = v'$ holds for every abstract value v' . Moreover, $E (rep_{T'} v') (rep_{T'} v')$ holds because E is reflexive on the image of $rep_{T'}$. The functions $abs_{T'}$ and $rep_{T'}$ are called “morphisms” of the quotient T' .

In the *rnb* example the only possible representation function rep_{bool} maps *False* to 0 and *True* to 1. In the *rnb2* example rep_{bool} may map *True* to any number greater 0, so there are infinitely many possible representation functions.

The Quotient Predicate

HOL provides the predicate

Quotient :: $(T \Rightarrow T \Rightarrow bool) \Rightarrow (T \Rightarrow T') \Rightarrow (T' \Rightarrow T) \Rightarrow (T \Rightarrow T' \Rightarrow bool) \Rightarrow bool$

It is defined as the conjunction of the two laws about R , $abs_{T'}$, and E above and the two laws about $rep_{T'}$, so that *Quotient* E $abs_{T'}$ $rep_{T'}$ R states that T' is a quotient of T with the specified relations and morphisms.

Theorems stating that *Quotient* holds for four specific arguments are called “quotient theorems”.

The quotient theorem for the *rnb* example is

theorem qrnrb: “*Quotient* *rnbE* *abs_bool* *rep_bool* *rnb*” *<proof>*

Unique Representation Values

If R is also left-unique, every abstract value v' in T' has exactly one related value in T . Thus the representation function $rep_{T'}$ is uniquely determined by R and has the set D as its image, thus the law $rep_{T'} v' \in D$ holds for every abstract value v' .

Also, since every abstract value in T' has only one single representation value in T , $rep_{T'}$ is the inverse of $abs_{T'}$ on the set D , i.e., the law $v \in D \implies rep_{T'} (abs_{T'} v) = v$ holds for every value v in the raw type.

Note that together with the law $abs_{T'} (rep_{T'} v') = v'$ (see above) these two laws uniquely determine the relation R and cause it to be right-total and bi-unique, i.e., they cause T' to be a quotient of T with the uniquely determined representation function $rep_{T'}$.

Moreover, in this case the equivalence relation E can always be represented by `eq_onp` ($\lambda x. x \in D$) using the restricted equality described in Section 6.2.1. In the `rnb` example `rnb` is left-unique and the representation function `rep_bool` is uniquely determined.

The `type_definition` Predicate

For the case of a left-unique transfer relation R HOL provides the predicate

`type_definition :: (T' \Rightarrow T) \Rightarrow (T \Rightarrow T') \Rightarrow (T set) \Rightarrow bool`

It is defined as the conjunction of the three laws about `abs_T'`, `rep_T'`, and D mentioned in the previous section, so that `type_definition rep_T' abs_T' D` states that T' is a quotient of T with unique representation function `rep_T'`. For an explanation of the predicate name see Section 7.2.5.

Theorems stating that `type_definition` holds for three specific arguments are called “type-definition theorems”.

The type-definition theorem for the `rnb` example is

`theorem trnb: "type_definition rep_bool abs_bool rnbD" <proof>`

6.14.3 Polymorphic Quotients

If the raw and abstract type are polymorphic types `'a C` and `'a C'` the transfer relation R is also polymorphic with type parameter `'a` and so are the equivalence relation E and the morphisms `abs_C'` and `rep_C'`.

As described in Section 6.13.3, if a relator `rel_C` is available for the raw type `'a C`, it is possible to construct a transfer relation between `'a C` and `'b C` from an arbitrary transfer relation R_a between the type parameters `'a` and `'b` as `rel_C R_a`. Composed (see Section 6.1.3) with R it yields the parameterized transfer relation `pcr = $\lambda R_a. (rel_C R_a) \circ R$` (where `pcr` stands for “parameterized correspondence relation”).

The totality and uniqueness properties of the argument relation R_a are preserved by the relator `rel_C` and by composition and thus also by `pcr_C'`. Therefore the properties

`right_total R_a \implies right_total (pcr R_a)`
`right_unique R_a \implies right_unique (pcr R_a)`

are always satisfied which implies that `pcr R_a` is actually a transfer relation if R_a is a transfer relation.

In other words, if T' is a quotient of T then $T' C'$ is a quotient of $T C$. Using transfer rules with transfer relation `pcr R_a`, transfer works for T' when it is used as type parameter of a polymorphic quotient.

As usual, if C and C' are multivariate functors with n type parameters the relator $\mathit{rel_C}$ takes n relations as arguments and the parameterized transfer relation has the form $\mathit{pcr} = \lambda R_1 \dots R_n. (\mathit{rel_C} R_1 \dots R_n) \text{ OO } R$.

6.14.4 Registering Quotients

HOL supports a mechanism for automatically transferring functions between a quotient and its raw type. It is called the “lifting package”.

The first step of using the package consists of registering a type T' as quotient, together with its raw type T and the morphisms $\mathit{abs_T'}$ and $\mathit{rep_T'}$. Registering means that the fact that T' is a quotient of T is stored together with the morphisms in internal HOL data structures so that they can be retrieved by HOL mechanisms. All the information to be stored is contained in a quotient or type-definition theorem for T' , therefore a quotient is registered by specifying the name of a fact which is such a theorem. This is done using a command of the form

setup_lifting *name*

where *name* is the name of a quotient theorem or a type-definition theorem. In the case of a type-definition theorem the command internally defines the relation R named $\mathit{cr_T'}$ using the definition

$\mathit{cr_T_def}: \mathit{cr_T'} \equiv \lambda(x::T) y::T'. x = \mathit{rep_T'} y$

Note that in this case the equivalence relation E is always the equality $\mathit{eq_onp} (\lambda x. x \in D)$ restricted to the domain set D .

Moreover, the command always tries to define the parameterized transfer relation $\mathit{pcr_T'}$ (see Section 6.14.3) using a definition of the form

$\mathit{pcr_T_def}: \mathit{pcr_T'} \equiv \lambda R_1 \dots R_n. (\mathit{rel_T} R_1 \dots R_n) \text{ OO } R$

where n is the number of type parameters for a polymorphic quotient T' . If T' is not polymorphic $\mathit{pcr_T'}$ has no arguments and is equal to R .

Note that for a polymorphic quotient the definition of $\mathit{pcr_T'}$ depends on the existence of the relator $\mathit{rel_T}$. If the relator is not known to HOL the command signals a warning “Generation of a parametrized correspondence relation failed. Reason: No relator for the type T found.” and omits the definition of $\mathit{pcr_T'}$, which results in weaker transfer support (see below). If the raw type T has been registered as BNF (see Section 6.12.4) the relator is always known to HOL, otherwise usually not. However, if T has been registered as BNF with dead type parameters (see Section 6.12.1) the relator has fewer parameters than expected by **setup_lifting** and cannot be used for constructing $\mathit{pcr_T'}$. Then **setup_lifting** will still signal the warning about not finding the relator.

Basic Support for Transfer

The **setup_lifting** command supports transfer in the following way. First it adds the facts

```
T'.right_total: right_total pcr_T'
T'.right_unique: right_unique pcr_T'
```

to the set **transfer_rule** (see Section 6.13.2). For a polymorphic T' they have the form

```
T'.right_total: [[right_total R_1, ..., right_total R_n]]
                ==> right_total (pcr_T' R_1 ... R_n)
T'.right_unique: [[right_unique R_1, ..., right_unique R_n]]
                ==> right_unique (pcr_T' R_1 ... R_n)
```

They state that pcr_T' is a transfer relation if all R_i are transfer relations and provide the base for introducing transfer rules with pcr_T' as transfer relation. Additionally, the rule $T'.right_total$ allows the **transfer** method to replace the universal quantification on T' by quantification on T restricted to the domain D of pcr_T' (see Section 6.13.1). For a polymorphic T' it allows to replace the universal quantification on $(\text{'a}_1, \dots, \text{'a}_n) T'$ by quantification on $(\text{'b}_1, \dots, \text{'b}_n) T$ restricted to the domain of $(pcr_T' R_1 \dots R_n)$ for arbitrary types $\text{'b}_1, \dots, \text{'b}_n$ and $\text{'a}_1, \dots, \text{'a}_n$ related as quotients by arbitrary relations R_1, \dots, R_n .

If pcr_T' could not be defined because the relator of T is not known, these and all subsequently described transfer rules are defined using R as transfer relation instead of pcr_T' . This results in weaker effects when the **transfer** method is applied to terms involving type T' .

The **setup_lifting** command also adds a rule $T'.domain$ to the set **transfer_domain_rule** which constructs an explicit representation of the domain of $pcr_T' R_1 \dots R_n$ from the domains of the R_i (see Section 6.13.2).

Note that all fact names are qualified by the type name T' of the quotient, therefore the names are specific for the quotient registered by **setup_lifting**. If two **setup_lifting** commands are specified for the same quotient T' an error is signaled.

Support for Equality Transfer

If the quotient is registered using a quotient theorem the **setup_lifting** command also adds the fact

```
T'.rel_eq_transfer:
  (pcr_T' (=)...(=) ==> pcr_T' (=)...(=) ==> (=)) E (=)
```

to the set *transfer_rule*. It allows the *transfer* method to replace the equality on T' by the equivalence E on T . If the quotient is registered by a type-definition theorem no such rule is required, the standard HOL transfer rules for transferring equations, such as *eq_transfer* (see Section 6.13.3) are sufficient.

Since the relator lifts equality to equality (see Section 6.12.2), the application $\text{pcr_}T' (=) \dots (=)$ to n equality relations is always equal to the transfer relation R . Thus the rule $T'.\text{rel_eq_transfer}$ is equivalent to the form $(R \implies R \implies (=)) E (=)$. In the polymorphic case the rule only supports transferring equality on $(\text{'a}_1, \dots, \text{'a}_n) T'$ to the equivalence E on $(\text{'a}_1, \dots, \text{'a}_n) T$ where the type parameters are the same.

However, if E is parametrically polymorphic (see Section 6.13.3) it can further be transferred to a different instance $(\text{'b}_1, \dots, \text{'b}_n) T$. Therefore, when using a quotient theorem, the *setup_lifting* command supports the extended form

setup_lifting *name* *parametric pname*

where *pname* is the name of a fact of the form

$$(\text{rel_}T R_1 \dots R_n \implies \text{rel_}T R_1 \dots R_n \implies (=)) E E$$

which is a parametricity rule (see Section 6.13.3) stating that E is parametrically polymorphic. In this form the *setup_lifting* command generates the stronger transfer rule

$$\begin{aligned} T'.\text{rel_eq_transfer}: \\ (\text{pcr_}T' R_1 \dots R_n \implies \text{pcr_}T' R_1 \dots R_n \implies (=)) E (=) \end{aligned}$$

with arbitrary relations R_1, \dots, R_n relating the different instances of the type parameters. It supports transferring equality on $(\text{'a}_1, \dots, \text{'a}_n) T'$ to the equivalence E on $(\text{'b}_1, \dots, \text{'b}_n) T$.

When using a type-definition theorem for *setup_lifting* no parametricity rule may be specified. It is not required because then the equivalence E is always the (possibly restricted) equality which is always parametric for a bi-unique transfer relation (see below).

In the *rnv* example after the command

setup_lifting *qrnv*

where *qrnv* is the name of the quotient theorem as defined in Section 6.14.2, the *transfer* method will transfer equations of the form $\bigwedge x::\text{bool}. x = \dots$ to propositions of the form

$$\bigwedge x::\text{nat}. \text{rnvE } x \ x \implies \text{rnvE } x \ \dots$$

After the command

setup_lifting *trnb*

where *trnb* is the name of the type-definition theorem as defined in Section 6.14.2, the *transfer* method will transfer equations of the form $\bigwedge x :: \text{bool}. x = \dots$ to propositions of the form

$$\bigwedge x :: \text{nat}. x \in \text{rnbD} \implies x = \dots$$

Left-Total Transfer Relations

If it uses a quotient theorem the **setup_lifting** command may be specified in the extended form

setup_lifting *name rname*

where *rname* is the name of a fact of the form *reflp E* and *E* is the equivalence relation used in the quotient theorem *name*. The fact *rname* is called a “reflexivity rule” for *E*. As described at the beginning of Section 6.14 it is equivalent with *left_total R*. This property allows HOL to introduce stronger transfer rules as follows.

The command in this form adds the additional rule

$$T'.\text{bi_total}: \llbracket \text{bi_total } R_1; \dots \text{bi_total } R_n \rrbracket \implies \text{bi_total } (\text{pcr_T}' R_1 \dots R_n)$$

to the set *transfer_rule*. It allows the *transfer* method to replace quantification on $(\text{'a}_1, \dots, \text{'a}_n)$ *T'* to unrestricted quantification on $(\text{'b}_1, \dots, \text{'b}_n)$ *T* (see Section 6.13.1).

It also adds the rule

$$T'.\text{id_abs_transfer}: (\text{rel_T } R_1 \dots R_n \implies \text{pcr_T}' R_1 \dots R_n) (\lambda x. x) \text{abs_T}'$$

It allows to replace the morphism *abs_T'* by the identity function on $(\text{'b}_1, \dots, \text{'b}_n)$ *T*.

In the *rnb2* example from Section 6.14.1 after the theorems

theorem *qrbn2*: "Quotient *rnb2E abs_bool rep_bool rnb2*" *<proof>*

theorem *rrnb2*: "*reflp rnb2E*" *<proof>*

with a corresponding equivalence relation *rnb2E* and accordingly modified morphisms *abs_bool* and *rep_bool* the command

setup_lifting *qrbn2 rrnb2*

introduces the transfer rules as described. Then the *transfer* method will transfer the (invalid) equation $\bigwedge (x :: \text{bool}) (n :: \text{nat}). x = (\text{abs_bool } n)$ to the proposition

$$\bigwedge (x :: \text{nat}) (n :: \text{nat}). \text{rnb2E } x n$$

Left-Unique Transfer Relations

HOL provides special support for a quotient with left-unique transfer relation only if the quotient is specified by a **setup_lifting** command using a type-definition theorem.

Then the command always adds the additional rule

```
T'.bi_unique:
  [[bi_unique R1; ... bi_unique Rn]] ==> bi_unique (pcr_T' R1...Rn)
```

to the set *transfer_rule*. Together with the standard HOL transfer rules for transferring equations, such as *eq_transfer* (see Section 6.13.3), it allows the *transfer* method to transfer equality from $(\text{'a}_1, \dots, \text{'a}_n) T'$ to $(\text{'b}_1, \dots, \text{'b}_n) T$.

It also adds the rule

```
T'.rep_transfer: (pcr_T' R1...Rn ==> rel_T R1...Rn) (λx. x) rep_T'
```

It allows to replace the function *rep_T'* by the identity function on $(\text{'b}_1, \dots, \text{'b}_n) T$.

In the *trnb* example the command

```
setup_lifting trnb
```

introduces the transfer rules as described. Then the *transfer* method will transfer the (invalid) equation $\bigwedge (x::\text{bool}) (n::\text{nat}). (\text{rep_bool } x) = n$ to the proposition

```
 $\bigwedge (x::\text{nat}) (n::\text{nat}). x = n$ 
```

If the transfer relation is also left-total, *T'* and *T* are isomorphic and *D* is the universal set (see Section 8.4.1) *UNIV*. HOL detects type-definition theorems of the form *type-definition rep_T' abs_T' UNIV* and also adds the rules *T'.bi_total* and *T'.id_abs_transfer* as described above for left-total transfer relations.

6.14.5 Lifting

Whenever *T'* is a quotient of *T* it is possible to “lift” operations from *T* to *T'* by first mapping the argument of type *T* to its representation value of type *T'* using *rep_T'*, then applying the original operation, and finally mapping the result back using *abs_T'*.

Moreover, since the relation *R* can always be used as a transfer relation (see Section 6.13) it is possible to transfer terms with occurrences of the lifted operation to a corresponding term where it is replaced by the original operation. The lifting package supports this by automatically configuring the required transfer rules for lifted operations.

Respectful Functions

A function $f :: T \Rightarrow T$ “respects” the equivalence relation E if it preserves E for its arguments in the sense that the law

$$\bigwedge x y. E\ x\ y \implies E\ (f\ x)\ (f\ y)$$

is satisfied. A theorem stating such a law is called a “respectfulness theorem” for f . In mathematics in this case E is also called a “congruence relation” for f .

If two functions f and g are lifted as described above, the composition of the lifted functions is $\text{abs_}T' (f\ (\text{rep_}T' (\text{abs_}T' (g\ (\text{rep_}T' x))))$. Here $\text{rep_}T' (\text{abs_}T' y)$ may map y to a different value in T , but that must be equivalent. If f respects E the result is always the same as the result $\text{abs_}T' (f\ (g\ (\text{rep_}T' x)))$ of applying the lifted composition $f \circ g$, i.e., lifting is compatible with function composition. Therefore HOL supports lifting only for functions which respect E .

If the corresponding transfer relation R is left-unique the equivalence relation E is the equality $\text{eq_onp}\ (\lambda x. x \in D)$ restricted to the domain D of R (see Section 6.14.4). Then the respectfulness theorem for a function f and E is equivalent to the proposition

$$\bigwedge x. x \in D \implies (f\ x) \in D$$

which means that the domain D is closed under f .

Defining Functions by Lifting

After registering a type T' as quotient of T with relations R and E and morphisms $\text{abs_}T'$ and $\text{rep_}T'$ it is possible to lift operations on T respecting E to T' using the command

lift_definition *name* *::* “ $T' \Rightarrow T'$ ” *is* “*term*” *<proof>*

where *term* is a term of type $T \Rightarrow T$. The command introduces a definition *name_def* for *name* as $\text{name}\ x \equiv \text{abs_}T' (\text{term}\ (\text{rep_}T' x))$. Note that this is equivalent to $\text{name} \equiv \text{map_fun}\ \text{rep_}T'\ \text{abs_}T'\ \text{term}$ using the mapper *map_fun* (see Section 6.11.2), which is the form actually used by HOL.

Every **lift_definition** command creates a goal and includes the *<proof>* for the respectfulness theorem for the given *term* and the quotient’s equivalence relation E . If T' has been registered using a type-definition theorem the goal is generated in the form that D must be closed under *term*.

In the *rnv* example after registering type *bool* as quotient of type *nat* a negation operation can be defined using lifting in the form

lift_definition *mynot* *::* “*bool* \Rightarrow *bool*” *is* “ $\lambda n. 1-n$ ” *<proof>*

If the quotient has been registered using the quotient theorem *qrb*, the command generates the respectfulness theorem

$$\bigwedge x y. \text{ rnbE } x y \implies \text{ rnbE } (1-x) (1-y)$$

as goal to be proved. If it has been registered using the type-definition theorem *trnb*, the command generates the closedness theorem

$$\bigwedge n. n \in \text{ rnbD } \implies 1-n \in \text{ rnbD }$$

as goal to be proved.

Generalized Lifting

More generally, lifting is supported for respectful functions *f* of a type $T_1 \Rightarrow T_2$ with different types for argument and result. If both types are registered quotients the lifted function is defined using the corresponding morphisms *abs*_{*T*}₂ and *rep*_{*T*}₁. If only one of both types is a registered quotient, the other is retained by lifting, then the corresponding application of *abs*_{*T*}₂ or *rep*_{*T*}₁ is omitted. The respectfulness theorem has the form $E_1 \ x \ y \implies E_2 \ (f \ x) \ (f \ y)$ where *E_i* is replaced by equality if *T_i* is not a registered quotient. Since functions with multiple arguments are represented by types of the form $T_1 \Rightarrow \dots \Rightarrow T_n \Rightarrow T$ lifting is also applicable to them. Finally, a single value is lifted from *T* to *T'* by application of *abs*_{*T'*}. Together, the general form of the command is

lift_definition *name* :: *type* is "*term*" *<proof>*

where *type* is an arbitrary type and the type of *term* must result from *type* by replacing all occurrences of registered quotients by their corresponding raw types.

The definition of *name* can be systematically constructed from *type* by replacing occurrences of registered quotients by their corresponding morphism *abs*_{*T*} or *rep*_{*T*} and replacing all other type constructors by their mapper function (see Section 6.11.1) and all other types by the identity function *id*. This is similar as constructing predicates and relators according to the structure of a type expression consisting of composed natural functors (see Section 6.12.2). HOL defines the operator name (*--->*) for *map_fun* to make mappers for function types resemble visually to the corresponding type. Like the operator name (*==>*) for *rel_fun* (see Section 6.13.1) (*--->*) is only available after using the command

unbundle *lifting_syntax*

on theory level.

As examples, if `type` is $T_1' \Rightarrow T_2'$ the definition generated for `name` is `name` \equiv `(rep_T1' ---> abs_T2') term` and if `type` is $T_1' \Rightarrow T_2' \Rightarrow \text{nat}$ the definition generated for `name` is `name` \equiv `(rep_T1' ---> rep_T2' ---> id) term`.

In the `rnb` example after registering type `bool` as quotient of type `nat` boolean connectives can be defined using lifting in the form

```
lift_definition myand :: "bool  $\Rightarrow$  bool  $\Rightarrow$  bool" is "(*)" <proof>
lift_definition myor :: "bool  $\Rightarrow$  bool  $\Rightarrow$  bool" is "max" <proof>
```

A predicate on natural numbers can be defined using lifting in the form

```
lift_definition is_zero :: "nat  $\Rightarrow$  bool" is "\n. 1-n" <proof>
```

Note how the type determines how the specified term is lifted. Here only the result type is lifted to `bool`, in the definition of `mynot` above the argument type is lifted as well.

Support for Transfer

To support transfer from the lifted function to the original function a `lift_definition` command for `name`, `type`, and `term` adds the transfer rule

```
name.transfer: REL' term name
```

where `REL'` is the relator constructed according to the structure of `type` as described in Section 6.12.2. As usual, it allows the `transfer` method to replace occurrences of `name` by `term`.

If `type` is polymorphic, the type parameters are replaced by `(=)` when constructing `REL'`, thus the transfer rule only supports transfer when type parameters are instantiated by the same type for the quotient and the raw type. HOL supports the extended form

```
lift_definition name :: type is "term" parametric pname <proof>
```

where `pname` is the name of a fact of the form

```
REL term term
```

where `REL` is the relator constructed according to the structure of the type of `term`. This is a parametricity rule (see Section 6.13.3) stating that `term` is parametrically polymorphic. In this form the `lift_definition` command replaces type parameters by arbitrary relations R_1, \dots, R_n when constructing `REL'`. The resulting stronger transfer rule allows to replace `name` by `term` even if different types are substituted for the type parameters in the quotient and in the raw type.

In the `rnb` example the transfer rules generated by the definitions above are


```

mynot.transfer: (pcr_bool ==> pcr_bool) (λn. 1-n) mynot
myand.transfer: (pcr_bool ==> pcr_bool ==> pcr_bool) (*) myand
myor.transfer: (pcr_bool ==> pcr_bool ==> pcr_bool) max myor
is_zero.transfer: ((=) ==> pcr_bool) (λn. 1-n) is_zero

```

If the quotient has been registered using the type-definition theorem *trnb*, these rules cause the *transfer* method in

```

theorem "mynot (myor a b) = myand (mynot a) (mynot b)"
  apply (transfer)
  by (simp add: mult_eq_if)

```

to replace the goal by the transferred goal

$$\bigwedge a\ b. \llbracket a \in \text{rnbD}; b \in \text{rnbD} \rrbracket \implies 1 - \max a\ b = (1-a) * (1-b)$$

which can be proved by the simplifier using the additional fact *mult_eq_if*.

6.14.6 Quotients as Bounded Natural Functors

As described in Section 6.14.4 a polymorphic quotient T' should have a registered BNF (see Section 6.12) without dead type parameters as its raw type T , so that the relator *rel_T* is available for constructing *pcr_T'* and benefitting from full transfer support. Moreover, in this case the quotient T' may again be a BNF.

Since the cardinality of a quotient is never greater than that of the raw type, the quotient is bounded as soon as the raw type is bounded.

The mapper m of T can be lifted to T' as

$$m' = \lambda f_1 \dots f_n. \text{abs}_{T'} \circ (m\ f_1 \dots f_n) \circ \text{rep}_{T'}$$

if it respects the equivalence relation E of the quotient (see Section 6.14.5), i.e., if the proposition

$$\bigwedge x\ y. E\ x\ y \implies E\ ((m\ f_1 \dots f_n)\ x)\ ((m\ f_1 \dots f_n)\ y)$$

is valid for all possible functions $f_1 \dots f_n$ where n is the number of type parameters of the BNF T . In this case the lifted mapper m' is compatible with function composition (see Section 6.14.5), thus T' is a functor with mapper m' (see Section 6.11.1).

The remaining conditions for T' to be a BNF are the existence of the set-function(s), the satisfying of the laws of a natural functor (see Section 6.12.1), and the subdistributivity of the relator (see Section 6.12.3).

Quotients as Natural Functors

If the quotient T' has been registered by a type-definition theorem (see Section 6.14.4) the set-functions s_1, \dots, s_n can also be lifted:

$$s_i' = s_i \circ \text{rep_}T'$$

because they trivially respect E which is the equality restricted to the domain D of the transfer relation. In a similar way the predicator and relator of T can be lifted to T' .

To satisfy the laws of a natural functor only two properties must be satisfied: the domain D of the transfer relation must be closed under the mapper m :

$$x \in D \implies (m \ f_1 \ \dots \ f_n) \ x \in D$$

and under the zip construction (see Section 6.12.2) using the mapper m and the set-functions s_i :

$$\begin{aligned} & \llbracket (m \ \text{fst} \dots \text{fst}) \ z \in D, (m \ \text{snd} \dots \text{snd}) \ z \in D \rrbracket \\ & \implies \exists \text{zz} \in D. \\ & \quad s_1 \ \text{zz} \subseteq s_1 \ z \wedge \dots \wedge s_n \ \text{zz} \subseteq s_n \ z \wedge \\ & \quad (m \ \text{fst} \dots \text{fst}) \ \text{zz} = (m \ \text{fst} \dots \text{fst}) \ z \wedge \\ & \quad (m \ \text{snd} \dots \text{snd}) \ \text{zz} = (m \ \text{snd} \dots \text{snd}) \ z \end{aligned}$$

Here for every zipped container z (see Section 6.12.2) for which the unzipped containers are in D there must be a zipped container zz in D with no additional content and the same unzipped containers. The subdistributivity of the relator (see Section 6.12.3) directly carries over from T .

If the quotient T' has been registered by a quotient theorem the set-functions could only be lifted if they respect the equivalence relation E , which means that values of the BNF T can only be equivalent if they have the same content, differing only in their shape. To support more general equivalence relations HOL constructs the set-functions, the relator and the predicator for T' from E and m in a different way.

For this construction the laws of a natural functor can be satisfied by proving a single goal about the equivalence relation E . A second goal corresponds to the subdistributivity of the relator. More information about these conditions can be found in [3].

The lift_bnf Command

HOL supports registering a quotient of a BNF as BNF using the command

```
lift_bnf ('name1, ..., 'namen) name <proof>
```

where the parentheses may be omitted if $n = 1$. **name** must be the type constructor name of a registered quotient (see Section 6.14.4) and the **'name₁**

must be type variables for its type parameters. If **name** has not been registered as a quotient using **setup_lifting** the command signals an error.

The command generates two goals which must be proved in the $\langle proof \rangle$. The goals are as described above and differ depending whether the quotient has been registered using a type-definition theorem or a quotient theorem.

After the $\langle proof \rangle$ of both goals the **lift_bnf** command generates definitions for the mapper m , the set-functions s_i , the predicate p , and the relator r , with the usual names **map_name**, **seti_name**, **pred_name**, and **rel_name**. Alternative names for (some of) these functions may be specified in the form

```
lift_bnf (sname1: 'name1, ..., snamen: 'namen) name
  for map: mname pred: pname rel: rname
```

After the definition of the BNF functions the command generates and proves the same named facts as the **bnf** command (see Section 6.12.4) and registers the quotient **name** as BNF.

If the quotient T' has been registered using a quotient theorem the **lift_bnf** command requires that the quotient's transfer relation R is left-total, i.e., a reflexivity theorem **reflp** E must have been specified in the **setup_lifting** command used to register the quotient (see Section 6.14.4). Otherwise the **lift_bnf** command signals an error.

If the quotient T' has been registered using a type-definition theorem the **lift_bnf** command may complain that a “nonemptiness witness of the raw type's BNF was lost”. This means that it must be proved that the domain D of R is not empty by specifying an actual element (a “witness”) in it. This is done by the extended form

```
lift_bnf ('name1, ..., 'namen) name [wits: "term"]  $\langle proof \rangle$ 
```

of the **lift_bnf** command where **term** is a function with arguments of types **'name₁, ..., 'name_n**. Remember that the raw type is a BNF, i.e., a type of container values consisting of shape and content. The function arguments denote content values, the function result is a corresponding container. In this form the command generates additional goals to be proved: one that the resulting container is always in D , and n goals stating that the container value has no other content of type **'name_i** than the argument values of that type.

For examples of using **lift_bnf** see Sections 7.2.7 and 7.3.5.

Like the **bnf** command the **lift_bnf** command does not register the generated mapper m as functor. If needed this must be done explicitly using the **functor** command (see Section 6.12.4).

The `copy_bnf` Command

If the transfer relation R for a quotient T' is left-total and left-unique T' is isomorphic to the raw type T and can be registered as quotient using a type-definition theorem with the universal set $UNIV$ as domain D of the transfer relation (see Section 6.14.4).

If T is a BNF the closedness properties for the mapper and the zip construction required for T' to also be a BNF (see above) are trivially satisfied. HOL supports registering such a quotient as BNF using the command

```
copy_bnf ('name1, ..., 'namen) name
```

where the parentheses may be omitted if $n = 1$. **name** must be the name of the quotient and the **'name₁** must be type variables for its type parameters. No proof is required, the two goals are proved automatically by HOL. As for **lift_bnf** **name** must have been registered as a quotient using **setup_lifting**, otherwise the command signals an error.

The **copy_bnf** command generates definitions for the mapper m , the set-functions s_i , the predicator p , and the relator r , with the usual names **map_name**, **seti_name**, **pred_name**, and **rel_name**. Alternative names for (some of) these functions may be specified similar as for **lift_bnf** in the form

```
copy_bnf (sname1: 'name1, ..., snamen: 'namen) name  
  for map: mname pred: pname rel: rname
```

After the definition of the BNF functions the command generates and proves most of the named facts provided by the **lift_bnf** command and registers the quotient **name** as BNF.

For an example of using **copy_bnf** see Section 7.2.7.

Like the **bnf** command the **copy_bnf** command does not register the generated mapper m as functor. If needed this must be done explicitly using the **functor** command (see Section 6.12.4).

Chapter 7

Isabelle/HOL Type Definitions

This chapter introduces 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 in outer syntax.

7.1 Algebraic Types

Roughly an algebraic type is equivalent to a union of tuples with support for recursion, which allows nested tuples. In this way most data types used in programming languages can be covered, such as records, unions, enumerations, and pointer structures. Therefore HOL also uses the notion “datatype” for algebraic types.

7.1.1 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_{*i,1*}" ... "type_{*i,k_i*}"

The *cname*_{*i*} are names and the type_{*i,j*} 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.

As a convention, capitalized names are used in 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.

Recursive Algebraic Types

Recursion is supported for the types, i.e., the name *name* of the defined type may occur in the type specifications $\text{type}_{i,j}$. However, there must be at least 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.

An example for a recursive datatype definition with two constructor specifications is

```

datatype tree =
  Leaf nat
/ Tree nat tree tree

```

Its value set is equivalent to the set of all binary trees with a natural number in every node.

Moreover, recursive occurrences of *name* are only allowed on live parameter positions (see Section 6.12.1) of a bounded natural functor $\text{type}_{i,j}$ (see Section 6.12.3). Due to composability of BNFs this means that all type constructors in which the recursive occurrence is nested must be bounded natural functors. If this is not the case an error is signaled.

In the definition of *tree* above the recursive occurrences are not nested in other type constructors, therefore the condition is satisfied. If the second constructor specification is modified to

```

... / Tree nat "tree set" tree

```

an error will result because *set* is not registered as BNF (it cannot, because it is none, see Section 6.12.3). If the constructor specification is modified to

```

... / Tree nat "tree  $\Rightarrow$  bool" tree

```

another error will result because $\text{tree} \Rightarrow \text{bool}$ is an abbreviation of $(\text{tree}, \text{bool}) \text{ fun}$ and *fun* is a BNF (see Section 6.12.3), but *tree* occurs on its dead parameter’s position. If the constructor specification is modified to

```

... / Tree nat "nat  $\Rightarrow$  tree" tree

```

everything is fine because now the recursive occurrence is on the live parameter position.

Parameterized Algebraic Types

Like declared types algebraic types may be parameterized (see Section 2.2.1):

```
datatype ('name1, ..., 'namem) name = alt1 | ... | altn
```

where the '*name*_{*i*}' are the type parameters. They may occur in the type specifications *type*_{*i,j*}, i.e., the *type*_{*i,j*} may be polymorphic (see Section 2.2.1). Recursive occurrences of *name* must always be applied to the type parameters, no other types are allowed as arguments. As usual, the parentheses may be omitted if there is only one type parameter.

An example for a parameterized datatype definition with one type parameter is

```
datatype 'a coordx =  
  Dim2 'a 'a  
| Dim3 'a 'a 'a
```

Its value set is equivalent to the union of pairs and triples of values of the type parameter. The type *coord* is equivalent to the type *nat coordx*. The type *real coordx* is equivalent to the union of pairs and triples of values of type *real* of the real numbers.

An example for a parameterized recursive datatype definition with one type parameter is

```
datatype 'a treex =  
  Leaf 'a  
| Tree 'a "'a treex" "'a treex"
```

With this definition the type *tree* is equivalent to the type *nat treex*.

7.1.2 Constructors

Every *cname*_{*i*} is used by the definition to introduce a “(value) constructor function”, i.e., a constant

```
cnamei :: "typei,1 ⇒ ... ⇒ typei,ki ⇒ name"
```

which is a function with *k*_{*i*} 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* \Rightarrow *nat* \Rightarrow *coord* and *Dim3* :: *nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *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 tuples of the value sets of *type_{i,1}* ... *type_{i,k_i}*: 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 *term₁* ... *term_{k_i}*

where each *term_j* is of type *type_{i,j}* and specifies an argument for the constructor function application.

Together, datatypes have what is called “free constructors” in Isabelle: the constructors are injective, disjoint, and exhaustive (they cover all values of the type).

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 function 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 constructor argument tuples. 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.

For more comfortable record implementations in Isabelle see Section 7.4.

7.1.3 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 $\text{type}_{i,j}$ the selector function is a function of type $\text{name} \Rightarrow \text{type}_{i,j}$. For every value constructed by $\text{cname}_i \text{ term}_1 \dots \text{term}_{k_i}$ 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

```
cnamei (snamei,1 : "typei,1") ... (snamei,ki : "typei,ki")
```

where the $\text{sname}_{i,j}$ 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 $x :: \text{coord} \Rightarrow \text{nat}$ 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 2.2.2). 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 $z :: \text{coord} \Rightarrow \text{nat}$ 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 = alt1 | ... | altn
where "prop1" | ... | "propm"
```

where every prop_p is a proposition of the form

```
snamei,j (cnameq var1 ... varkq) = termp
```

and specifies term_p as the default value of selector $\text{sname}_{i,j}$ for values constructed by cname_q . If a default value is specified for a cname_i where the selector is already defined, the default specification is ignored.

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 $name \Rightarrow bool$ 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 = dname1: alt1 / ... / dnamen: altn
```

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 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 7.1.1 has the form

```
case term of  
  cname1 var1,1 ... var1,k1  $\Rightarrow$  term1  
/ ...  
/ cnamen varn,1 ... varn,kn  $\Rightarrow$  termn
```

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_{1,1} ... var_{1,k₁}* 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_{i,j}* is the same as the value selected by *sname_{i,j}* *term*.

Actually, a *case* term is only an alternative syntax for the function application term

```
case_name  
  (λ var1,1 ... var1,k1. term1)  
  ...  
  (λ varn,1 ... varn,kn. termn)  
term
```

Here `case_name` is the “case combinator” function for the datatype `name`. It takes as arguments `n` functions which map the corresponding constructor arguments to the term variant (or the term variant itself if the constructor has no arguments) and the `term` of type `name` as final argument. Note that the constructor names `cnamei` do not occur here, the constructor corresponding to a term variant is only determined by the argument position `i` compared with the position of the constructor in the datatype definition.

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. The corresponding case combinator application term is

```
case_coord (λ a b. a+b) (λ a b c. a+b+c) cv
```

A `case` term is useful even for a datatype with a single constructor. If `rv` is of type `recrd` as defined in Section 7.1.3 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 3.12.1). 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 and occurs as argument of the case combinator function.

7.1.4 Parameterized Algebraic Types as Bounded Natural Functors

Every algebraic type can be considered to be a disjoint union (specified by the alternatives) of tuples (denoted by the constructor specifications). A disjoint union corresponds to a sum type (see Section 8.9), a tuple to a product type (see Section 8.6). Seen as type constructors both are BNFs (see Section 6.12.3). Due to composability of BNFs this implies that an algebraic type is a bounded natural functor as soon as all constructor argument types `typei,j` are BNFs. This is even true for recursive algebraic types.

HOL checks every datatype definition whether it satisfies this condition by checking whether all type constructors used in the `typei,j` have been registered as BNF (see Section 6.12.4) or are a recursive occurrence of the defined datatype. If that is the case the datatype is itself registered as BNF, all required goals are automatically proved. If it is not the case the datatype is registered as if all parameters are dead (see Section 6.12.1).

Like the **bnf** command a datatype definition does not register the generated mapper *m* as functor. If needed this must be done explicitly using the **functor** command (see Section 6.12.4).

Both example types `coordx` and `treex` specified in Section 7.1.1 are detected and registered by HOL as BNFs with one live type parameter. The datatype

```
datatype 'a nobnf = Dim2 'a 'a / Dimset "'a set"
```

is not recognized by HOL as BNF, because it uses the type constructor `set` in its second constructor specification, which is not a BNF (see Section 6.12.3). Since (bounded) natural functors model container types (see Section 6.12.1), the same is true for corresponding datatypes. As an example, every value of the polymorphic datatype `'a coordx` contains two or three values of the type parameter `'a` as its content. For a recursive datatype the content consists of the values at all direct occurrences of the type parameters united with the content of all recursive occurrences of datatype values.

BNF Functions

If HOL detects that a datatype is a BNF, it also generates definitions for the mapper (see Section 6.11.1), the set-function(s) (see Section 6.12.1), the predictor and the relator (see Section 6.12.2). The names are generated from the datatype name as `map_name`, `seti_name`, `pred_name`, and `rel_name`, where *i* is the position number of the corresponding type parameter `'namei` (see Section 7.1.1). So, if there are two type parameters the set-functions are named `set1_name` and `set2_name`. If the datatype has only one type parameter the (single) set-function is named `set_name`.

If a datatype is not recognized as a BNF, none of these functions are defined by HOL.

For the type *coordx* the mapper *map_coordx* has type $(p \Rightarrow q) \Rightarrow p \text{ coordx} \Rightarrow q \text{ coordx}$. For instance, if $f :: \text{real} \Rightarrow \text{nat}$ is the function that rounds every real number to the next natural number, the application *map_coordx* f *cv* replaces the real coordinates in *cv* of type *real coordx* by rounded natural coordinates, resulting in a value of type *nat coordx*.

For *coordx* the only set function is *set_coordx*. It maps every value of a type $T \text{ coordx}$ to the set of either two or three coordinate values of type T .

The predicate *pred_coordx* has type $(p \Rightarrow \text{bool}) \Rightarrow p \text{ coordx} \Rightarrow \text{bool}$. For instance, if *cv* is of type *nat coordx* the term *pred_coordx* $(\lambda n. n=0)$ *cv* tests whether all coordinates of *cv* are 0.

The relator *rel_coordx* has type $(p \Rightarrow q \Rightarrow \text{bool}) \Rightarrow p \text{ coordx} \Rightarrow q \text{ coordx} \Rightarrow \text{bool}$. For instance, if *cv₁* and *cv₂* are of type *nat coordx* the term *rel_coordx* (\leq) *cv₁* *cv₂* tests whether *cv₁* and *cv₂* have the same dimension and every coordinate in *cv₁* is lower or equal to the corresponding coordinate in *cv₂*.

Similar as for the command *lift_bnf* (see Section 6.14.6) the form

```
datatype (sname1: 'p1..., snamem: 'pm) name = alt1 | ... | altn
  for map: mname pred: pname rel: rname
```

of a datatype definition allows to define alternate names *sname_i*, *mname*, *pname*, *rname* for (some of) the set-, mapper, predicate, and relator functions. If the datatype is not recognized as BNF by HOL the names are ignored.

7.1.5 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 4.6.3). Other rules must be explicitly used by referring them by their name.

Only some basic rules are described here, for more information refer to [1]. All introduced rules can be displayed using the “Print Context” tab in the Query panel as described in Section 2.6.6, if the cursor is positioned after the **datatype** definition.

Simplifier Rules

The rules added by a definition for a datatype *name* to the simpset (see Section 4.6.3) support many ways for the simplifier to process terms with constructors and destructors.

The rule set *name.inject* states that every non-constant constructor is injective, the rules are of the form

$$((cname_i \ ?x_1 \ \dots \ ?x_{k_i}) = (cname_i \ ?y_1 \ \dots \ ?y_{k_i})) = \\ (\ ?x_1 = ?y_1 \ \wedge \ \dots \ \wedge \ ?x_{k_i} = ?y_{k_i})$$

The rule set *name.distinct* states that values constructed by different constructors are different, the rules are of the form

$$(cname_i \ ?x_1 \ \dots \ ?x_{k_i}) \neq (cname_j \ ?y_1 \ \dots \ ?y_{k_j})$$

where $i \neq j$.

The rule set *name.sel* provides “defining equations” for the selectors of the form

$$sname_{ij} \ (cname_i \ ?x_1 \ \dots \ ?x_{k_i}) = ?x_j$$

The rule set *name.case* provides equations for simplifying *case* terms where the discriminating term is directly specified by a constructor application. They have the form

$$(case \ (cname_i \ ?x_1 \ \dots \ ?x_{k_i}) \ of \\ \quad cname_1 \ var_{1,1} \ \dots \ var_{1,k_1} \Rightarrow ?f_1 \ var_{1,1} \ \dots \ var_{1,k_1} \\ \quad / \ \dots \\ \quad / \ cname_n \ var_{n,1} \ \dots \ var_{n,k_n} \Rightarrow ?f_n \ var_{n,1} \ \dots \ var_{n,k_n}) \\ = ?f_i \ ?x_1 \ \dots \ ?x_{k_i}$$

Note that each branch is specified as an application of a function $?f_i$ so that the variables bound in the branch can be substituted by the arguments of the constructor application.

Depending on the datatype definition there may be additional simplifier rules. In particular, if the datatype is parameterized, simplifier rules are generated for the functions described in Section 7.1.4. The set of all rules added to the simpset is named *name.simps*. By displaying it using the **thm** command (see Section 2.6.6) it can be inspected to get an idea how the simplifier processes terms for a specific datatype.

Case Rule

Every definition for a datatype *name* introduces a rule corresponding to the exhaustiveness of the free constructors (see Section 7.1.2). It has the form

```

name.exhaust:

$$\llbracket \bigwedge x_1 \dots x_{k_1}. ?y = \text{cname}_1 x_1 \dots x_{k_1} \implies ?P; \dots; \bigwedge x_1 \dots x_{k_n}. ?y = \text{cname}_n x_1 \dots x_{k_n} \implies ?P \rrbracket \implies ?P$$


```

According to Section 5.3.1 this rule is a case rule. It is automatically associated with the datatype for use by the `cases` method. 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 `cnamei`. Therefore a structured proof using case based reasoning for a `term` of datatype `name` has the form

```

proof (cases "term")
  case (cname1 x1 ... xk1) ... show ?thesis <proof>
next
...
next
  case (cnamen x1 ... xkn) ... show ?thesis <proof>
qed

```

The names `xi` of the locally fixed variables can be freely selected, they denote the constructor arguments of the corresponding constructor. Therefore the case specification `(cnamei x1 ... xki)` looks like a constructor application to variable arguments, although it is actually a context name together with locally fixed variables.

Split Rule

A `case` term (see Section 7.1.3) is only processed automatically by the simplifier, if the discriminating term is a constructor application (see the `name.case` rule set above). Otherwise it is only processed if a corresponding split rule is configured for it (see Section 4.6.4). Every definition for a datatype `name` introduces such a split rule. It has the form

```

name.split:

$$\begin{aligned} & ?P(\text{case } ?t \text{ of} \\ & \quad \text{cname}_1 x_{1,1} \dots x_{1,k_1} \Rightarrow ?t_1 x_{1,1} \dots x_{1,k_1} \\ & \quad / \dots \\ & \quad / \text{cname}_n x_{n,1} \dots x_{n,k_n} \Rightarrow ?t_n x_{n,1} \dots x_{n,k_n}) = \\ & ( (?t = \text{cname}_1 x_{1,1} \dots x_{1,k_1} \longrightarrow ?P(?t_1 x_{1,1} \dots x_{1,k_1})) \\ & \quad \wedge \dots \\ & \quad \wedge (?t = \text{cname}_n x_{n,1} \dots x_{n,k_n} \longrightarrow ?P(?t_n x_{n,1} \dots x_{n,k_n}))) \end{aligned}$$


```


As described in Section 4.6.5 the rule splits a goal with a `case` term for type `name` in the conclusion into goals where the `case` term is replaced by the terms in the cases. Note that the sub-terms of the `case` term are specified by unknowns, so the rule unifies with arbitrary `case` terms for type `name`. Also note, that the $?t_i$ are specified with arguments, so that they will be matched by functions depending on the constructor arguments $x_{i,1}, \dots, x_{i,k_i}$, as described in Section 7.1.3.

As an example, let `cv` be a variable or constant of type `coord`, as above. Then the goal

$$\text{sum} = (\text{case } cv \text{ of } \text{Dim2 } a \ b \Rightarrow a + b \mid \text{Dim3 } a \ b \ c \Rightarrow a + b + c)$$

is split by the split rule `coord.split` into the goals

$$\begin{aligned} cv = \text{Dim2 } a \ b &\Longrightarrow \text{sum} = a + b \\ cv = \text{Dim3 } a \ b \ c &\Longrightarrow \text{sum} = a + b + c \end{aligned}$$

Induction Rule

Every definition for a datatype `name` introduces an induction rule (see Section 5.5.1) of the form

$$\begin{aligned} \text{name.induct:} \\ \llbracket \bigwedge x_1 \dots x_{k_1} . \llbracket ?P \ x_{l_1} ; \dots ?P \ x_{l_{m_1}} \rrbracket \Longrightarrow ?P \ (cname_1 \ x_1 \dots x_{k_1}) ; \\ \dots ; \\ \bigwedge x_1 \dots x_{k_n} . \llbracket ?P \ x_{l_n} ; \dots ?P \ x_{l_{m_n}} \rrbracket \Longrightarrow ?P \ (cname_n \ x_1 \dots x_{k_n}) \rrbracket \\ \Longrightarrow ?P \ ?a \end{aligned}$$

where the $x_{l_1} \dots x_{l_{m_i}}$ are those $x_1 \dots x_{k_i}$ which have type `name` (i.e., the recursive occurrences of the type name). Like the case rule it is valid because the constructor applications cover all possibilities of constructing a value `?a` of the datatype.

If the datatype `name` is not recursive there are no $x_{l_1} \dots x_{l_{m_i}}$ and the assumptions of all inner rules are empty, then the induction rule is simply a specialization of the case rule and is redundant. However, for a recursive datatype `name` induction using rule `name.induct` is the standard way of proving a property to hold for all values.

The rule `name.induct` is associated with datatype `name` for use by the methods `induction` and `induct` (see Section 5.5.3). Therefore the application of the method

$$\text{induction } x$$

where `x` is a variable of type `name` splits a goal into `n` subgoals where every subgoal uses a different constructor term in the place of `x`.

As for the case rule and the *cases* method, the names for the named contexts created by the methods *induction* and *induct* are simply the constructor names *cname_i*. Therefore a structured proof using induction for a variable *x* of datatype *name* has the form

```
proof (induction x)
  case (cname1 x1 ... xk1) ... show ?case <proof>
next
...
next
  case (cnamen x1 ... xkn) ... show ?case <proof>
qed
```

In the rule *name.induct* all inner assumptions are of the form *?P x_{l_i}*, i.e., they are induction hypotheses and are named "*cname_i.IH*" by the *induction* method, the assumption set "*cname_i.hyps*" is always empty. The *induct* method instead names all inner assumptions by "*cname_i.hyps*".

As an example, the induction rule for the recursive datatype *tree* defined in Section 7.1.1 is

```
tree.induct:
  [[ $\bigwedge x. ?P (\text{Leaf } x);$ 
 $\bigwedge x_1 x_2 x_3. [[?P x_2; ?P x_3]] \implies ?P (\text{Tree } x_1 x_2 x_3)]]$ 
 $\implies ?P ?a$ 
```

If *p :: tree \Rightarrow bool* is a predicate for values of type *tree* the goal *p x* which states that *p* holds for all values is split by applying the method (*induction x*) into the goals

```
 $\bigwedge x. p (\text{Leaf } x)$ 
 $\bigwedge x_1 x_2 x_3. [[p x_2; p x_3]] \implies p (\text{Tree } x_1 x_2 x_3)$ 
```

A structured proof for goal *p x* has the form

```
proof (induction x)
  case (Leaf x) ... show ?case <proof>
next
  case (Tree x1 x2 x3) ... show ?case <proof>
qed
```

and in the second case the assumptions *p x₂*, *p x₃* are named *Tree.IH*.

BNF Rules

If HOL detects a datatype to be a BNF (see Section 7.1.4) it also provides the rules which are provided by the **bnf** command (see Section 6.12.4) with their names in the namespace of the datatype.

7.1.6 Recursive Functions on Algebraic Types

A term is called a “constructor pattern” if it only consists of variables and constructor function applications. Note that this includes terms consisting of a single variable. More generally, a constructor pattern may be a sequence of such terms used as arguments in a function application. A constructor pattern is called “linear” if every variable occurs only once.

HOL provides specific support for recursive definitions (see Section 6.10) of functions $\text{name} :: t_1 \Rightarrow \dots \Rightarrow t_k \Rightarrow \text{type}$ where every defining equation eq_i has the form

$$\bigwedge x_{i,1} \dots x_{i,p_i} . \text{name } \text{term}_{i,1} \dots \text{term}_{i,k} = \text{term}_i$$

without assumptions $Q_{i,j}$ and the sequence $\text{term}_{i,1} \dots \text{term}_{i,k}$ on the left side is a linear constructor pattern.

Since the type of a constructor application term is always an algebraic type, an argument type t_j may only be a non-algebraic type if the corresponding $\text{term}_{i,j}$ is a single variable in all eq_i .

The Proof Method *pat_completeness*

For recursive definitions where all eq_i are without assumptions and using a linear constructor pattern on their left side HOL provides the proof method

pat_completeness

for the proof of equation completeness (see Section 6.10.2), i.e. for solving the first goal created by the recursive definition. The remaining goals for uniqueness can be solved by using the injectivity of the constructor functions which is usually done by the proof method *auto*. Therefore if a recursive definition uses only linear patterns its proof can be specified as

by pat_completeness auto

Automatic Uniqueness by Sequential Equations

A constructor pattern \mathbf{a} is more specific than a constructor pattern \mathbf{b} if \mathbf{a} can be constructed from \mathbf{b} by replacing some variables by constructor applications (and renaming variables). If corresponding function arguments $\text{term}_{i,j}$ are specified by such patterns \mathbf{a} and \mathbf{b} in two defining equations, the spaces of both equations overlap and the defined value must be the same for uniqueness. Alternatively, uniqueness can be guaranteed by replacing the equation with the more general pattern \mathbf{b} by equations with more specific patterns using the remaining constructors, so that the argument spaces do not overlap anymore.

This can be done automatically for a recursive definition where all eq_i are without assumptions and using a linear constructor pattern on their left side by specifying it in the form

```
function (sequential) name :: " $t_1 \Rightarrow \dots \Rightarrow t_k \Rightarrow type$ "
where  $eq_1 \mid \dots \mid eq_n \langle proof \rangle$ 
```

Here the defining equations must be ordered such that equations with more specific patterns precede those with more general patterns. HOL automatically replaces the latter equations so that the argument spaces of all equations are pairwise disjoint. Note that the resulting equations are used for the completeness and compatibility proofs and also in the rules provided for the recursive definition such as *name.cases* or *name.psimps*.

If the equations do not cover all possible arguments because some constructors are omitted, additional equations are added with patterns using the omitted constructors. These equations use *undefined* (see Section 6.4) as $term_i$ on the right side.

The completeness and compatibility proof must still be specified explicitly, although it always works in the form *by pat_completeness auto*.

The *fun* Command

HOL supports the abbreviation

```
fun name :: " $t_1 \Rightarrow \dots \Rightarrow t_k \Rightarrow type$ "
where  $eq_1 \mid \dots \mid eq_n$ 
```

for the recursive definition

```
function (sequential) name :: " $t_1 \Rightarrow \dots \Rightarrow t_k \Rightarrow type$ "
where  $eq_1 \mid \dots \mid eq_n$ 
by pat_completeness auto
termination by lexicographic_order
```

which includes the completeness and compatibility proof and a termination proof. If the termination proof cannot be done by the proof method *lexicographic_order* (see Section 6.10.6) an error is signaled, then the long form must be used to specify a different termination proof.

The faculty function definitions in Section 6.10.1 are not of the required form: the definition(s) of *fac* use an assumption in their second equation, the definition of *fac2* uses the argument term $n+1$ on the left side which is no constructor pattern because the function (+) is not a constructor. However, type *nat* is actually defined in a way equivalent to an algebraic type with constructors *0* and *Suc* (see Section 8.3) where the term *Suc n* is equivalent to $n+1$. Therefore the faculty function can be defined as

```

fun fac3 :: "nat  $\Rightarrow$  nat" where
  "fac3 0 = 1"
  | "fac3 (Suc n) = (Suc n) * fac3 n"

```

because 0 and (*Suc* *n*) are linear constructor patterns.

Primitive Recursion

A linear constructor pattern consisting of a sequence of terms is called “primitive” if exactly one term is a constructor application and all constructor arguments in this term are single variables. Thus a primitive constructor pattern has the general form

$$x_1 \dots x_{i-1} \text{ (cname } x_{i,1} \dots x_{i,n}) x_{i+1} \dots x_k$$

where all x_i and $x_{i,j}$ are variables. In particular, if a linear constructor pattern consists of a single term it is always primitive.

A recursive function definition

```

fun name :: "t1  $\Rightarrow$  ...  $\Rightarrow$  tk  $\Rightarrow$  type"
where eq1 / ... / eqn

```

is called primitive if all defining equations *eq*_{*i*} use a primitive constructor pattern on their left side and all arguments of recursive calls in *term*_{*i*} on the right side are single variables.

Note that since the equations must be ordered so that equations with more specific patterns precede equations with more general patterns all constructor application terms must occur at the same argument position in the patterns. Therefore only one argument type *t*_{*i*} must be an algebraic type, all others may be arbitrary types because the corresponding arguments are denoted by single variables in all patterns.

For primitive recursive function definitions HOL provides the alternative syntax

```

primrec name :: "t1  $\Rightarrow$  ...  $\Rightarrow$  tk  $\Rightarrow$  type"
where eq1 / ... / eqn

```

Its difference to the form using the **fun** command is that it only provides the rule set *name.simps* and none of the other rules. The reason is that the *cases* and *induct* rules are mainly equivalent to the case and induction rules provided for the algebraic type of the used constructors (see Section 7.1.5).

Since every such definition can also be written using the **fun** command the use of **primrec** is mainly a documentation that the definition is primitive recursive.

Since the factorial function has only one argument the constructor patterns in the definition of *fac3* above are both primitive. Moreover, since *fac3* is

only applied to the plain variable n on the right side of the second equation it is a primitive recursive function definition and may be written

```
primrec fac3 :: "nat  $\Rightarrow$  nat" where
  "fac3 0 = 1"
| "fac3 (Suc n) = (Suc n) * fac3 n"
```

7.2 Subtypes

A subtype specifies the values of a new type T' by a set of values of an existing type T . 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.

A subtype is defined in the form

```
typedef name = "term" <proof>
```

where *name* is the name of the new type T' and *term* is a term for the representing set. The <proof> must prove that the representing set is not empty. A subtype definition implies that for every value in the representing set there is a unique value in the defined subtype.

Note that the concept of subtypes actually depends on the specific HOL type *set* for specifying the representing set. See Section 8.4 for how to denote terms for this set. Also note that the set is always of type $T \text{ set}$ where T is the common type of all set elements. This implies that the representing set is always a subset of the set of all values of T which explains the designation of type T' as “subtype” of T .

A simple example is the type

```
typedef threesub = "{0::nat,1,2}" by auto
```

which has three values. The representing set contains three natural numbers. As usual, their type *nat* must be explicitly specified because the constants 0, 1, 2 may also denote values of other types. However, they do not denote the values of the new type *threesub*, the type definition does not introduce constants for them.

7.2.1 Subtypes as Quotients

The values of type T in the representing set can be related to their corresponding values of the new type T' by a relation $R :: T \Rightarrow T' \Rightarrow \text{bool}$. Since there is a unique value in T' for every set member and all values of T' are related with a set member, R is right-unique and right-total.

As described in Section 6.14 this means that T' is a quotient of T with transfer relation R . This observation implies the existence of all other items related to a quotient: the morphisms $abs_{T'}$, $rep_{T'}$, the domain D of R and the equivalence relation E (see Section 6.14). As usual for a quotient, T' is called “abstract type” and T is called “raw type”.

Since the relation between D and T' is 1-1, the transfer relation R is also left-unique (see Section 6.14.2)

The domain D of R is the set of all values in T related to a value in T' , so it is exactly the representing set used to define the subtype. The abstraction function $abs_{T'} :: T \Rightarrow T'$ maps all values in D to their corresponding value in T' and is underspecified otherwise. The representation function $rep_{T'} :: T' \Rightarrow T$ is uniquely determined and maps all values of T' to their corresponding value in D . As usual, the values in D are called representation values for the values of T' . The equivalence relation E is the equality restricted to the representing set $eq_{onp} (\lambda x. x \in D)$.

The Morphism Names

The subtype definition `typedef T' = D <proof>` introduces the constants $Abs_{T'}$ and $Rep_{T'}$ for the morphisms (note the capitalization of the names). The function $Abs_{T'}$ can be used to denote the values of the subtype. Thus, $Abs_{T'}$ plays the role of a constructor for type T' , whereas $Rep_{T'}$ can be thought of being a destructor for T' .

In the example the morphisms are $Abs_{threesub} :: nat \Rightarrow threesub$ and $Rep_{threesub} :: threesub \Rightarrow nat$. The values of type $threesub$ may be denoted as $(Abs_{threesub} 0)$, $(Abs_{threesub} 1)$, and $(Abs_{threesub} 2)$. The term $(Abs_{threesub} 42)$ is a valid term of type $threesub$, however, no information about its value is available.

Alternative names may be specified for the morphisms in the form

```
typedef name = "term" morphisms rname aname <proof>
```

where $rname$ replaces Rep_name and $aname$ replaces Abs_name .

7.2.2 Parameterized Subtypes

Like declared types subtypes may be parameterized (see Section 2.2.1):

```
typedef ('name1, ..., 'namen) 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 2.2.2). Then it does not denote a fixed set of values, it denotes for every instantiation of the type parameters a different set of values from a different type.

As an example the type

```
typedef 'a eqpair = "{(x::'a,y::'a). x = y}" by auto
```

has the set of all pairs (see Section 6.6) as its representing set, where the first and second components are equal. The type of both components is the type parameter 'a, thus the raw type is the type 'a × 'a of pairs of values of the same type. If 'a is instantiated by type *bool* the representing set is $\{(True, True), (False, False)\}$ whereas if 'a is instantiated by type *nat* the representing set is infinite.

7.2.3 Type Copies

A type copy is the special case of a subtype definition where the representing set is the universal set (see Section 8.4.1) of the raw type:

```
typedef name = "UNIV :: T set" by auto
```

The non-emptiness proof can always be performed by the *auto* method, since the universal set covers all values in type *T* and types are always non-empty. As described at the end of Section 6.14.4 a type copy is the case where the transfer relation *R* is left-unique and left-total and thus the new type *name* is “isomorphic” to *T*. The values are in 1-1 relation, although, as usual for distinct types, the value sets are disjoint. The equivalence relation *E* is the equality (=) on *T*.

For a parameterized type copy the raw type *T* is some type expression constructed from the type parameters. An example is the type copy

```
typedef ('a, 'b) cpyfun = "UNIV :: ('a ⇒ 'b) set" by simp
```

of the type of functions from 'a to 'b.

7.2.4 Subtype Rules

A subtype definition only introduces a small number of rules, no rules are added to the simpset. All introduced rules can be displayed using the “Print Context” tab in the Query panel as described in Section 2.6.6, if the cursor is positioned after the subtype definition.

Basic Morphism Rules

As described in Section 6.14.2 for a left-unique transfer relation the morphisms are inverse of each other. For a subtype definition **typedef** *T'* = *D* *<proof>* this is expressed by two rules of the form

```
Abs_T'_inverse:
  ?y ∈ D ⇒ Rep_T' (Abs_T' ?y) = ?y
Rep_T'_inverse:
  Abs_T' (Rep_T' ?x) = ?x
```


This also implies that both morphisms are injective which is stated explicitly by two rules of the form

Abs_T'_inject:

$$\llbracket ?y_1 \in D; ?y_2 \in D \rrbracket \implies (Abs_T' \ ?y_1 = Abs_T' \ ?y_2) = (?y_1 = ?y_2)$$
Rep_T'_inject:

$$(Rep_T' \ ?x_1 = Rep_T' \ ?x_2) = (?x_1 = ?x_2)$$

Since all values of type T' can be denoted as $Abs_T' \ y$ for some y in the representing set D , the rule **Abs_T'_inject** can be used to prove equality or inequality for values of type T' based on the equality for values in D .

Case Rules

Every subtype definition **typedef** $T' = D \ \langle proof \rangle$ introduces a case rule (see Section 5.3.1) of the form

Abs_T'_cases:

$$(\bigwedge y. \llbracket ?x = Abs_T' \ y; y \in D \rrbracket \implies ?P) \implies ?P$$

It is valid because the Abs_T' application covers all possibilities of constructing a value $?x$ of T' .

The rule **Abs_T'_cases** is associated with the new subtype T' for use by the **cases** method (see Section 5.3.2). Therefore the application of the method

cases "*term*"

where *term* is of type T' applies **Abs_T'_cases** to replace the current goal. Since the rule has only one case, it does not split the goal. Applying it to a goal $\bigwedge x_1 \dots x_m. \llbracket A_1; \dots; A_n \rrbracket \implies C$ as described in Section 5.3.2 results in the single new goal

$$\bigwedge x_1 \dots x_m \ y. \llbracket A_1; \dots; A_n; term = Abs_T' \ y; y \in D \rrbracket \implies C$$

where the variable y and the two assumptions from the case rule have been added. Together the new goal provides a representation of *term* by applying Abs_T' to a value y from the representing set D . This may allow to use facts about y to prove the goal.

The name for the named context created by the **cases** method is simply the morphism name Abs_T' . Therefore a structured proof using case based reasoning for a *term* of subtype T' has the form

```
proof (cases "term")
  case ( $Abs\_T' \ y$ ) ... show ?thesis  $\langle proof \rangle$ 
qed
```

The name y of the locally fixed variable can be freely selected, it denotes the morphism argument, i.e., the representation value for *term*.

Every subtype definition **typedef** $T' = D \langle proof \rangle$ also introduces an elimination rule (see Section 5.4.2) of the form

```
Rep_T'_cases:
   $\llbracket ?y \in D; \bigwedge x. ?y = \text{Rep\_}T' \ x \implies ?P \rrbracket \implies ?P$ 
```

It is valid because the *Rep_T'* application covers all possibilities to determine a representation value $?y$ in D .

With the help of this rule it is possible to introduce an abstraction value x corresponding to a representation value $?y$, consuming an assumption or input fact that $?y$ is in D . For application by the method *cases* the rule is annotated by *[consumes 1]* and the name for the created named context is the morphism name *Rep_T'*. As described in Section 5.4.4 a pattern for using the rule in a structured proof is

```
theorem "C" if "y ∈ D"
  using that
proof (cases rule: Rep_T'_cases)
  case (Rep_T' x) ... show ?thesis <proof>
qed
```

Induction Rules

Every subtype definition **typedef** $T' = D \langle proof \rangle$ introduces two induction rules (see Section 5.5.1) of the form

```
Abs_T'_induct:
   $(\bigwedge y. y \in D \implies ?P \ (\text{Abs\_}T' \ y)) \implies ?P \ ?a$ 
Rep_T'_induct:
   $\llbracket ?a \in D; \bigwedge x. ?P \ (\text{Rep\_}T' \ x) \rrbracket \implies ?P \ ?a$ 
```

The former rule is a plain induction rule, the latter is an induction rule with elimination where the major premise states that the value $?a$ is in D . Both rules only contain a “base case” and no “induction step” with a recursive occurrence of values of the defined type T' . As for the case rules they are valid because the morphism applications cover all possibilities of constructing values of T' or values in D , respectively.

Since the rules only consist of a base case they are mainly equivalent to the case rules. However, when applied by the *induct* method, they not only provide a representation by a morphism for a specified variable, they also substitute every occurrence of the variable by the morphism representation.

The rule *Abs_T'_induct* is associated with subtype T' for use by the methods *induction* and *induct* (see Section 5.5). Therefore the application of the method

induction x

where x is a variable of type T' replaces a goal by a goal where every occurrence of x is substituted by the term $Abs_T' \ y$ and y is a new bound variable with the additional assumption $y \in D$ named $Abs_T'.hyps$. As usual for the induction methods, x is substituted in the goal conclusion and also in all goal assumptions.

As for the case rule and the *cases* method, the name for the named context created by the methods *induction* and *induct* is simply the morphism name Abs_T' . Therefore a structured proof using induction for a variable x of subtype T' has the form

```
proof (induction x)
  case (Abs_T' y) ... show ?case <proof>
qed
```

As an example, the induction rule for the subtype *threesub* defined above is

```
Abs_threesub_induct:
  "\y. y \in {0, 1, 2} \implies ?P (Abs_threesub y)) \implies ?P ?a
```

By applying the method (*induction x*) the goal $x = Abs_threesub \ 0 \implies x \neq Abs_threesub \ 1$ is replaced by the goal $\bigwedge y. \llbracket y \in \{0, 1, 2\}; Abs_threesub \ y = Abs_threesub \ 0 \rrbracket \implies Abs_threesub \ y \neq Abs_threesub \ 1$ (which does not help for the proof, but shows the effect of the induction rule).

The rule *Rep_T'_induct* is annotated by [*consumes 1*] for application by the methods *induction* and *induct* and the name for the created named context is the morphism name Rep_T' . As described in Section 5.5.5 a pattern for using the rule in a structured proof is

```
theorem "C" if "y \in D"
  using that
proof (induction rule: Rep_T'_induct)
  case (Rep_T' x) ... show ?case <proof>
qed
```

As an example, the induction rule with elimination for the subtype *threesub* defined above is

```
Rep_threesub_induct:
  \llbracket ?a \in \{0, 1, 2\}; \bigwedge x. ?P (Rep_threesub x) \rrbracket \implies ?P ?a
```

7.2.5 Registering Subtypes as Quotients

Although every subtype defined by `typedef T' = D <proof>` is a quotient, HOL does not automatically register it as such (see Section 6.14.4). However, it generates and proves the fact

```

type_definition_T':
  type_definition Rep_T' Abs_T' D

```

This is a type-definition theorem (see Section 6.14.2) and can be used for registering T' as quotient using the command

```

setup_lifting type_definition_T'

```

As usual, the command prepares for defining functions on the subtype by lifting them from the raw type (see Section 6.14.5) and introduces transfer rules for transferring terms from the subtype to the raw type (see Section 6.14.4). This use of the predicate `type_definition` for characterizing types introduced by the `typedef` command is the reason for its name.

As described in Section 6.14.4, if the raw type T of a polymorphic subtype is no BNF or has dead type parameters, the relator `rel_T` cannot be used to provide full transfer support, the `setup_lifting` command will issue a warning in this case.

7.2.6 Lifting Functions to Subtypes

After registering a subtype as quotient as described in Section 7.2.5, functions can be lifted from the raw type to the subtype as described in Section 6.14.5. Since the transfer relation for a subtype is left-unique, the respectfulness property required for a function to be lifted is based on closedness of the representing set. For a subtype defined by `typedef T' = D <proof>` the respectfulness property for an operation $f :: T \Rightarrow T$ is

$$\bigwedge x :: T. x \in D \implies f\ x \in D$$

and for a function $f :: u \Rightarrow T$ where only the results shall be lifted it is

$$\bigwedge x :: u. f\ x \in D$$

For a function $f :: T \Rightarrow u$ where only the arguments shall be lifted respectfulness is always satisfied.

For the example type `threesub` as defined above, after registering it as quotient by

```

setup_lifting type_definition_threesub

```

example functions can be lifted from type `nat` as follows:

```

lift_definition suc3 :: "threesub  $\Rightarrow$  threesub"
  is "\x. (x+1) mod 3" by auto
lift_definition mod3 :: "nat  $\Rightarrow$  threesub"
  is "\x. x mod 3" by auto
lift_definition evn3 :: "threesub  $\Rightarrow$  bool"
  is "\x. x mod 2 = 0" done

```

The first example defines a cyclic successor function on *threesub*. Here the argument and result type are lifted, the goal to be proved for respectfulness is $\bigwedge n. n \in \{0, 1, 2\} \implies (n + 1) \bmod 3 \in \{0, 1, 2\}$ which can be proved by the *auto* method. The second example calculates the remainder modulo 3 and represents the result in type *threesub*, the goal to be proved is $\bigwedge n. n \bmod 3 \in \{0, 1, 2\}$. The third example defines a predicate for evenness on *threesub*, no goal needs to be proved because only the argument type is lifted, therefore the proof is empty and can immediately be terminated by *done*.

As described in Section 6.14.5 HOL provides automatic support for transferring terms containing functions defined by lifting to the raw type. As an example in the theorem

```
theorem "evn3 (suc3 (mod3 7))" apply(transfer) by simp
```

the *transfer* method replaces the goal by the proposition $(7 \bmod 3 + 1) \bmod 3 \bmod 2 = 0$ which is simply a property of natural numbers and can be proved by the simplifier using the existing rules about the natural numbers. No rules about the subtype *threesub* are required.

This is a typical way of working with a subtype using the lifting package: All functions on the subtype are introduced by lifting, then all theorems specified for the subtype can be proved using transfer and the existing facts about the raw type.

In the case of a type copy (see Section 7.2.3) the representing set contains all values of the raw type and is closed for all functions. Therefore arbitrary functions can be lifted from the raw type to a type copy using an empty proof.

7.2.7 Subtypes of Bounded Natural Functors

As described in Section 6.14.6 a subtype T' of a BNF T (the type of the elements of the representing set) may again be a BNF. In that case the **lift_bnf** command can be used to register T' as BNF. Note that before using the command it is necessary to explicitly register T' as quotient by the command **setup_lifting type_definition_** T' (see Section 7.2.5).

If the representing set does not contain all values of T the nonemptiness witness of T (see Section 6.14.6) may get lost in T' . Therefore usually a new witness which belongs to the representing set must be specified in the command:

```
lift_bnf ('a1, ..., 'an)  $T'$  [wits: "term"] <proof>
```

where *term* specifies a function from arguments of types $'a_1, \dots, 'a_n$ to values which must be in the representing set.

After registering the example type `eqpair` defined in Section 7.2.2 as quotient using

```
setup_lifting type_definition_eqpair
```

it can be registered as BNF using the command

```
lift_bnf 'a eqpair [wits: "λ(x::'a). (x,x)"] by auto
```

because the pair (x, x) has equal components and is thus in the representing set of `eqpair`.

For a type copy T' (see Section 7.2.3) of a BNF T the transfer relation is also left-total and the `copy_bnf` command (see Section 6.14.6) can be used to register T' as BNF (after registering it as quotient). No proof is required, a type copy of a BNF is always a BNF.

After registering the example type `cpyfun` defined in Section 7.2.3 as quotient using

```
setup_lifting type_definition_cpyfun
```

it can be registered as BNF using the command

```
copy_bnf ('a, 'b) cpyfun
```

7.3 Quotient Types

A quotient type specifies the values of a new type T' by equivalence classes of an equivalence relation E on an existing type T . As usual, the equivalence classes are not directly the values of the new type, instead, there is a 1-1 relation between them.

A quotient type is defined in the form

```
quotient_type name = "type" / "term" ⟨proof⟩
```

where `name` is the name of the new type T' , `type` specifies the existing type T , and `term` specifies the equivalence relation E which must be of type $T \Rightarrow T \Rightarrow \text{bool}$. The `⟨proof⟩` must prove that E is an equivalence relation which is stated by the predicate application `equivp E` (see Section 8.8.1).

A simple example is the quotient type

```
quotient_type rem3 = nat / "λx y. x mod 3 = y mod 3"
  by (simp add: equivp_def) metis
```

which has three values related to the remainder classes of natural numbers modulo 3.

If the `type` specified in a `quotient_type` definition has not been registered as functor (see Section 6.11.3) the `quotient_type` definition signals a warning, that no map function has been defined for it. As described in Section 6.12.4 this also happens for types registered as BNF, they have a defined mapper, but it is not registered.

7.3.1 Quotient Types as Quotients

As the name suggests, a quotient type T' defined for T and E is a quotient of T . The corresponding transfer relation $R :: T \Rightarrow T' \Rightarrow \text{bool}$ relates all values in an equivalence class of E with the value in T' which corresponds to that class. Since there is a unique value in T' for every member of a class and all values of T' are related with a class member, R is right-unique and right-total, i.e., T' is a quotient of T .

This observation implies the existence of all other items related to a quotient: the morphisms `abs_T'`, `rep_T'`, the domain D of R and the equivalence relation E (which is used in the definition) (see Section 6.14). As usual for a quotient, T' is called “abstract type” and T is called “raw type”.

In contrast to a subtype definition (see Section 7.2.5) a quotient type definition immediately registers the quotient type T' as quotient of its raw type T using an internal `setup_lifting` command (see Section 6.14.4).

The property `equivp E` implies the reflexivity rule `reflp E` (see Section 6.14), i.e., for a quotient type definition as above the transfer relation R is always left-total. HOL automatically proves the reflexivity rule and uses it in the internal `setup_lifting` command as described for left-total transfer relations in Section 6.14.4.

A more general quotient type where R needs not be left-total can be defined in the form

```
quotient_type name = "type" / partial: "term" <proof>
```

Here the `<proof>` has only to prove the weaker property `part_equivp E` (see Section 8.8.1) which does not imply `reflp E` and allows E to be partial in the sense that there may be values in T which are not even equivalent by E to themselves. In this case the domain D of R is the set of values which are equivalent to themselves: $D = \{x. E\ x\ x\}$. Such a quotient type corresponds to the most general case of quotient described in Section 6.14. It is also automatically registered as quotient by an internal `setup_lifting` command but with omitted reflexivity rule.

The Morphism Names

The quotient type definition `quotient_type T' = T / E` introduces the constants `abs_T'` and `rep_T'` for the morphisms (note the lowercase names in contrast to the morphism names described for subtypes in Section 7.2.1).

Additionally it introduces the constants `Abs_T' :: (T set) ⇒ T'` and `Rep_T' :: T' ⇒ (T set)`. The former maps each equivalence class of E to the corresponding value in the quotient type, it is underspecified for all other sets of values of type T . The latter maps all values in the quotient type to the corresponding equivalence class of E .

Quotient Types and Subtypes

In general, for a quotient type the equivalence classes of E are no singletons, therefore the transfer relation R is not left-unique. Only if E is the equality ($=$) or the restricted equality `eq_onp` ($\lambda x. x \in D$) (see Section 6.2.1) this would be the case.

Since every subtype (see Section 7.2) is a quotient with a left-unique transfer relation, for every subtype defined by `typedef T' = D` with a representing set `D :: T set` the quotient type definition

```
quotient_type T' = T / partial: "eq_onp ( $\lambda x. x \in D$ )" <proof>
```

has nearly the same effect as the subtype definition together with the `setup_lifting` command for it. The main differences are the morphism names and the rules generated for them (see Section 7.3.3).

In the same way for every type copy `typedef T' = (UNIV::T)` (see Section 7.2.3) the quotient type definition

```
quotient_type T' = T / "(=)" <proof>
```

has nearly the same effect as the type copy definition together with the `setup_lifting` command for it.

7.3.2 Parameterized Quotient Types

Like declared types quotient types may be parameterized (see Section 2.2.1):

```
quotient_type ('name1, ..., 'namen) name = "type" / "term" <proof>
```

where the `'namei` are the type parameters. They may occur in `type` i.e., the `type` (and also the `term`) may be polymorphic (see Section 2.2.2).

As an example the type

```
quotient_type 'a quotpair = "'a × 'a" / " $\lambda(x_1, x_2) (y_1, y_2). x_2 = y_2$ "
  by(simp add: equivp_def split_beta) (metis case_prod_conv)
```


is defined using the equivalence relation on pairs (see Section 6.6), where pairs are equivalent if they have the same second component (i.e., the first component is ignored). Note that this means the resulting quotient type is isomorphic to the type parameter `'a` itself.

As described in Section 6.14.4, if the raw type T of a polymorphic quotient type is no BNF or has dead type parameters, the relator `rel_T` cannot be used to provide full transfer support. The quotient type definition will issue a warning in this case, caused by the internal `setup_lifting` command.

7.3.3 Quotient Type Rules

Since a quotient type definition includes the `setup_lifting` command, it introduces all corresponding rules as described in Section 6.14.4. Additionally, it introduces similar rules like a subtype definition (see Section 7.2.4), these are described in the following subsections. No rules are added to the simpset. All introduced rules can be displayed using the “Print Context” tab in the Query panel as described in Section 2.6.6, if the cursor is positioned after the `quotient_type` definition.

Basic Morphism Rules

Other than for subtypes the basic morphism rules for a quotient type defined by `quotient_type T' = T / E <proof>` are not directly defined for the morphisms, but instead for the functions `Abs_T'` and `Rep_T'` which map between values of T' and equivalence classes of E . Whenever `Abs_T'` is applied to a variable in a rule, the rule must contain the additional assumption that the variable belongs to the set of equivalence classes of E . This set is specified in the rules by a complex term, for better presentation this term is abbreviated by $\mathcal{C}(E)$ in the following subsections.

As mappings to/from equivalence classes the functions are inverse of each other. This is expressed by two rules of the form

Abs_T'_inverse:
 $?y \in \mathcal{C}(E) \implies \text{Rep_T'} } (\text{Abs_T'} } ?y) = ?y$
Rep_T'_inverse:
 $\text{Abs_T'} } (\text{Rep_T'} } ?x) = ?x$

This also implies that both functions are injective which is stated explicitly by two rules of the form

Abs_T'_inject:
 $\llbracket ?y_1 \in \mathcal{C}(E); ?y_2 \in \mathcal{C}(E) \rrbracket \implies (\text{Abs_T'} } ?y_1 = \text{Abs_T'} } ?y_2) = (?y_1 = ?y_2)$
Rep_T'_inject:
 $(\text{Rep_T'} } ?x_1 = \text{Rep_T'} } ?x_2) = (?x_1 = ?x_2)$

Since all values of type T' can be denoted as $Abs_T' \ y$ for some equivalence class y , the rule $Abs_T'_inject$ can be used to prove equality or inequality for values of type T' based on the equality of equivalence classes of E .

Case Rules and Induction Rules

Every quotient type definition **quotient_type** $T' = T / E \ \langle proof \rangle$ introduces case rules (see Section 5.3.1) and induction rules (see Section 5.5.1) similar to those described for subtypes (see Section 7.2.4), where again the single values in the representing set D are replaced by equivalence classes of E . All other remarks given for the subtype rules also apply here. The rules are

Abs_T'_cases:
 $(\bigwedge y. \llbracket ?x = Abs_T' \ y; \ y \in C(E) \rrbracket \implies ?P) \implies ?P$
Rep_T'_cases:
 $\llbracket ?y \in C(E); \bigwedge x. ?y = Rep_T' \ x \implies ?P \rrbracket \implies ?P$
Abs_T'_induct:
 $(\bigwedge y. y \in C(E) \implies ?P \ (Abs_T' \ y)) \implies ?P \ ?a$
Rep_T'_induct:
 $\llbracket ?a \in C(E); \bigwedge x. ?P \ (Rep_T' \ x) \rrbracket \implies ?P \ ?a$

Additionally the quotient type definition introduces the induction rule

$T'.abs_induct: (\bigwedge y. ?P \ (abs_T' \ y)) \implies ?P \ ?a$

It can be used to replace values of T' based on values of T instead of equivalence classes.

Morphism Definitions

For the actual morphisms the quotient type definition generates definitions based on Abs_T' and Rep_T' .

The morphism abs_T' is defined in the form

$abs_T'_def: abs_T' \ x \equiv Abs_T' \ \{y. E \ x \ y\}$

Here for the argument x the class $\{y. E \ x \ y\}$ of all equivalent values is mapped by Abs_T' to the corresponding value in T' .

The morphism rep_T' is defined in the form

$rep_T'_def: rep_T' \ x \equiv SOME \ y. y \in Rep_T' \ x$

using the choice operator (see Section 6.3.1). In this way, although the representation function is normally not unique, HOL can provide a single definition without the need to decide which specific representation values to use.

Quotient Theorem

Every quotient type definition **quotient_type** $T' = T / E \langle proof \rangle$ introduces the quotient theorem (see Section 6.14.2)

Quotient_T': *Quotient* E *abs_T'* *rep_T'* *cr_T'*

and provides the definition

cr_T'_def: $cr_T' = \lambda x\ y. abs_T'\ x = y$

for the transfer relation R named *cr_T'*. The quotient theorem is then used in the internal **setup_lifting** command to register T' as quotient.

The quotient type definition also introduces the type-definition theorem

type_definition_T': *type_definition* Rep_T' Abs_T' $\mathcal{C}(E)$

which allows to view T' as subtype of the type T **set** with the equivalence classes $\mathcal{C}(E)$ of E as representing set. However, since every type can only be registered once as a quotient, the type-definition theorem cannot be used for a corresponding registration.

7.3.4 Lifting Functions to Quotient Types

Since the definition of a quotient type includes its registration as quotient, functions can be lifted immediately after the definition from the raw type to the quotient type as described in Section 6.14.5.

As described in Section 6.14.5 the respectfulness property required for a function to be lifted corresponds to preserving the equivalence used to define the quotient type. For a quotient type defined by **quotient_type** $T' = T / E \langle proof \rangle$ the respectfulness property for an operation $f :: T \Rightarrow T$ is

$$\bigwedge x\ y. E\ x\ y \implies E\ (f\ x)\ (f\ y)$$

and for a function $f :: T \Rightarrow u$ where only the arguments shall be lifted it is

$$\bigwedge x\ y. E\ x\ y \implies (f\ x) = (f\ y)$$

For a function $f :: u \Rightarrow T$ where only the results shall be lifted respectfulness is always satisfied.

For the example type *rem3* as defined above, example functions can be lifted from type *nat* as follows:

```
lift_definition add3 :: "rem3  $\Rightarrow$  rem3  $\Rightarrow$  rem3"
  is "(+)" using mod_add_cong by blast
lift_definition zro3 :: "rem3  $\Rightarrow$  bool"
  is "\x. x mod 3 = 0" by simp
lift_definition sub3 :: "nat  $\Rightarrow$  nat  $\Rightarrow$  rem3"
  is "(-)" done
```

The first example defines addition of remainder classes. Here the argument and result types are lifted, the goal to be proved for respectfulness is

$$\bigwedge n_1 \ n_2 \ m_1 \ m_2. \llbracket n_1 \bmod 3 = n_2 \bmod 3; m_1 \bmod 3 = m_2 \bmod 3 \rrbracket \\ \implies (n_1 + m_1) \bmod 3 = (n_2 + m_2) \bmod 3$$

which can be proved using the fact *mod_add_cong* provided by HOL. The second example defines a predicate for being the zero remainder class, the goal to be proved is $\bigwedge n_1 \ n_2. n_1 \bmod 3 = n_2 \bmod 3 \implies (n_1 \bmod 3 = 0) = (n_2 \bmod 3 = 0)$. The third example calculates the difference and represents the result in type *rem3*, no goal needs to be proved because only the result type is lifted, therefore the proof is empty and can immediately be terminated by *done*.

Support for Transfer

As described in Section 6.14.5 HOL provides automatic support for transferring terms containing functions defined by lifting to the raw type. As an example in the theorem

theorem "zro3 (add3 (sub3 10 5) (sub3 3 2))" **apply**(*transfer*) **by simp**

the *transfer* method replaces the goal by the proposition $(10 - 5 + (3 - 2)) \bmod 3 = 0$ which is simply a property of natural numbers and can be proved by the simplifier using the existing rules about the natural numbers. No rules about the quotient type *rem3* are required.

This is a typical way of working with a quotient type using the lifting package: All functions on the quotient type are introduced by lifting, then all theorems specified for the quotient type can be proved using transfer and the existing facts about the raw type.

Lifting Functions to Partial Quotient Types

If the quotient type has been defined by **quotient_type** $T' = T / \text{partial} : E \langle \text{proof} \rangle$ lifting functions works mainly in the same way as described above. However, in the case of a function $f : u \Rightarrow T$ where only the results shall be lifted, respectfulness is no more automatically satisfied. The respectfulness property to be proved here is $\bigwedge x. E (f \ x) (f \ x)$ which means that all result values $(f \ x)$ must belong to the domain D of the transfer relation (see Section 6.14.1).

7.3.5 Quotient Types for Bounded Natural Functors

As described in Section 6.14.6 a quotient type T' of a BNF T may again be a BNF. If T' has been defined by **quotient_type** $T' = T / E \langle \text{proof} \rangle$ the

`lift_bnf` command can be used to register T' as BNF. If, instead, T' has been defined by `quotient_type $T' = T / \text{partial}: E \langle \text{proof} \rangle$` where the transfer relation need not be left-total, the `lift_bnf` command is not supported, it signals an error message about a missing reflexivity rule (which is only available for a left-total transfer relation).

The example type `quotpair` defined in Section 7.3.2 can be registered as BNF using the command

```
lift_bnf 'a quotpair by force auto
```

7.4 Record Types

Record types resemble algebraic types in that they are roughly equivalent to tuple types, however, they are defined in a completely different way. They do not support recursion, instead they support a simple form of inheritance. They can be used to model “record types” in programming languages and object data in object oriented programming languages.

7.4.1 Record Definitions

A record type is defined in the form

```
record rname = fname1 :: "ftype1" ... fnamen :: "ftypen"
```

where *rname* is the name of the new type, the *fname*_{*i*} are the pairwise distinct names of the record components (also called “fields”) and the *ftype*_{*i*} are the corresponding component types. Atleast one component must be specified. The resulting type *rname* is mainly equivalent to the tuple type *ftype*₁ × ... × *ftype*_{*n*} (see Section 6.6), however, every record type definition introduces a new type, even if the component names and types are the same.

The record type defined as above can either be denoted by its *rname* or by its “record type expression” (in inner syntax)

```
(fname1 :: ftype1, ..., fnamen :: ftypen)
```

Note that a record type expression may only be used after a corresponding record type definition. If there are several record type definitions with the same field names and types the record type expression refers to the syntactically latest previous matching record type definition.

If a field name *fname*_{*i*} occurs in several record type definitions it may be referred uniquely as a qualified name by prepending the record type name in the form *rname.fname*_{*i*}.

An example record definition is

```

record recrd =
  num :: nat
  nums :: "nat set"
  nice :: bool

```

It has an equivalent structure as the datatype *recrd* with the single constructor defined in Section 7.1.2. The record type expression for it is

```

(num :: nat, nums :: nat set, nice :: bool)

```

Alternatively the field names may be qualified:

```

(recrd.num :: nat, recrd.nums :: nat set, recrd.nice :: bool)

```

Record Type Schemes

To be able to extend a record type by additional fields, a record type definition **record** *rname* = *fname*₁ :: "*ftype*₁" ... *fname*_{*n*} :: "*ftype*_{*n*}" actually defines a parameterized type *rname_scheme* with a single type parameter and an additional component of that type which is called the “more part”. Every instantiation of (*'a rname_scheme*) is called a record type scheme, the most general one is (*'a rname_scheme*) where the more part has an arbitrary type *'a*. For the defined record type the more part has type *unit* (see Section 8.2), i.e., type *rname* is the same as (*unit rname_scheme*).

Like record types, record type schemes may be denoted by record type expressions. They have the same form, the more part is denoted by the pseudo field name ... (three-dot symbol). Therefore the polymorphic type scheme (*'a rname_scheme*) can be denoted by the record type expression

```

(fname1 :: ftype1, ..., fnamen :: ftypen, ... :: 'a)

```

and the record type expression (*fname*₁ :: *ftype*₁, ..., *fname*_{*n*} :: *ftype*_{*n*}) is simply an abbreviation for (*fname*₁ :: *ftype*₁, ..., *fname*_{*n*} :: *ftype*_{*n*}, ... :: *unit*).

The polymorphic record type scheme for the record type *recrd* defined above is denoted by *'a recrd_scheme* or by the record type expression (*num* :: *nat*, *nums* :: *nat set*, *nice* :: *bool*, ... :: *'a*).

Record type schemes may be used to define functions which are applicable to a record and all its extensions, similar to methods in object oriented programming.

Extending Record Types

A record type is extended by instantiating the more part to a “record fragment type”. Like a record type it consists of a sequence of one or more fields,

however it cannot be used on its own, it must always be embedded as more part in another record.

A record type is extended by the definition:

```
record rname = "rtype"
      + efname1 :: "eftype1" ... efnamem :: "eftypem"
```

where *rtype* is a previously defined record type. The fields specified by the *efname*_{*i*} and *eftype*_{*i*} comprise the record fragment which is appended after the fields of *rtype*, even if (some of) the same field names have already been used for *rtype*. Type *rtype* is called the “parent type” of type *rname*. A record type which does not extend another record type (has no parent type) is called a “root record type”.

If necessary, the field names *efname*_{*i*} must be qualified by *rname*, whereas the field names of the parent type *rtype* must be qualified by the name of *rtype*, even if they occur in an extension of *rtype*.

The record fragment type defined by the extension above may be denoted by the record type expression $\langle \text{efname}_1 :: \text{eftype}_1, \dots, \text{efname}_m :: \text{eftype}_m \rangle$. If $\langle \text{fname}_1 :: \text{ftype}_1, \dots, \text{fname}_n :: \text{ftype}_n \rangle$ is an expression for the parent type, the extended record type may be denoted by the type scheme expression

$$\langle \text{fname}_1 :: \text{ftype}_1, \dots, \text{fname}_n :: \text{ftype}_n, \\ \dots :: \langle \text{efname}_1 :: \text{eftype}_1, \dots, \text{efname}_m :: \text{eftype}_m \rangle \rangle$$

or by the type expression

$$\langle \text{fname}_1 :: \text{ftype}_1, \dots, \text{fname}_n :: \text{ftype}_n, \\ \text{efname}_1 :: \text{eftype}_1, \dots, \text{efname}_m :: \text{eftype}_m \rangle$$

The example record type defined above can be extended by

```
record recrd2 = recrd + full :: bool num :: nat
```

Note that the resulting record type has two fields with name *num* and type *nat*, they must always be referred by the qualified field names *recrd.num* and *recrd2.num*. A record type expression for *recrd2* is

$$\langle \text{recrd.num} :: \text{nat}, \text{nums} :: \text{nat set}, \text{nice} :: \text{bool}, \\ \text{full} :: \text{bool}, \text{recrd2.num} :: \text{nat} \rangle$$

Parameterized Record Types

Like declared types record types may be parameterized (see Section 2.2.1):

```
record ('name1, ..., 'namem) rname = fname1 :: "ftype1" ...
      fnamen :: "ftypen"
```

where the $'name_j$ are the type parameters. They may occur in the component types f_{type_i} , i.e., the f_{type_i} may be polymorphic (see Section 2.2.1). As usual, the parentheses may be omitted if there is only one type parameter. For a parameterized record type a record type expression may be specified for every possible instance. As usual, a record type expression with type variables denotes a polymorphic type.

In the record type scheme the parameter for the more part follows all other type parameters.

As an example, a parameterized record type with two type parameters is defined by

```
record ('a, 'b) recrdp = f1 :: 'a f2:: "'a set" f3 :: 'b
```

The most general record type scheme is ('a, 'b, 'c) *recrdp_scheme* where 'c is the parameter for the more part.

After this definition valid record type expressions are ($f1::nat, f2::nat\ set, f3::bool$) which is equivalent to (*nat, bool*) *recrdp*, and ($f1::bool, f2::bool\ set, f3::'a$) which is equivalent to (*bool, 'a*) *recrdp*, but not ($f1::nat, f2::bool\ set, f3::bool$), because here the type parameter 'a has been substituted by the two different types *nat* and *bool*.

7.4.2 Record Constructors

Every record type definition **record** *rname* = *fname₁* :: "*f_{type1}*" ... *fname_n* :: "*f_{type_n}*" defines the record constructor function

```
make :: ftype1 ⇒ ... ⇒ ftypen ⇒ rname
```

which constructs values of the record type from values to be used for all fields. If more than one record type has been defined the name of the constructor function must be qualified by the record type name as *rname.make*.

For an extended record type defined by **record** *rname2* = *rname* + *efname₁* :: "*e_{f_{type1}}*" ... *efname_m* :: "*e_{f_{type_m}}*" the constructor function is

```
make :: ftype1 ⇒ ... ⇒ ftypen ⇒ eftype1 ⇒ ... ⇒ eftypem ⇒ rname2
```

It takes values for *all* fields and constructs a full record of type *rname2*.

Every definition for a (root or extended) record type *rname* also defines the constructor function

```
extend :: rname ⇒ 'a ⇒ ('a rname_scheme)
```

It replaces the more part of a record of type *rname* (which is the unit value) by a value of an arbitrary type 'a. The result will only be a proper record if 'a is a record fragment type for a defined extension of *rname*.

Additionally the definition of the extended record type *rname2* defines the constructor function

$$\begin{aligned} \text{fields} &:: \text{ftype}_{n+1} \Rightarrow \dots \Rightarrow \text{ftype}_m \Rightarrow \\ &\quad (\text{fname}_{n+1} :: \text{ftype}_{n+1}, \dots, \text{fname}_m :: \text{ftype}_m) \end{aligned}$$

for the record fragment used as the more part of *rname*. For a root record type *fields* is the same function as *make*.

Constructing Values

As for a datatype every record constructor function is 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 *make* is equivalent to the set of all tuples of values of the field types, which is equivalent to the set of all possible values of the record type *rname*. Thus every value of *rname* may be denoted by a term

$$\text{rname.make } \text{term}_1 \dots \text{term}_n$$

where each *term_i* is of type *ftype_i* and specifies the value for field *i*.

There is an alternative Syntax for applications of record constructors. The record expression

$$(\text{fname}_1 = \text{term}_1, \dots, \text{fname}_n = \text{term}_n)$$

denotes the same record value as the constructor application above. If the name *fname₁* of the first field has been used in more than one record type it must be qualified. The record schema expression

$$(\text{fname}_1 = \text{term}_1, \dots, \text{fname}_n = \text{term}_n, \dots = \text{mterm})$$

denotes a value for the record type scheme where *mterm* denotes the value for the more part.

Values of the record fragment type defined for *rname2* are constructed by the application

$$\text{rname2.fields } \text{term}_1 \dots \text{term}_m$$

or equivalently by the record fragment expression $(\text{efname}_1 = \text{term}_1, \dots, \text{efname}_m = \text{term}_m)$.

Values of the extended record type *rname2* itself can be constructed from a value *r* of type *rname* by *rname.extend r (rname2.fields term₁ ... term_m)* or *rname.extend r (rname2.efname₁ = term₁, ..., efname_m = term_m)*.

Values of type *recrd* as defined above are denoted by terms such as *recrd.make 2 {5,7} True* or the equivalent record expression $(\text{num} = 2, \text{nums} = \{5,7\},$

`nice = True`) or the record scheme expression $\langle \text{num} = 2, \text{nums} = \{5,7\}, \text{nice} = \text{True}, \dots = () \rangle$. Values of the extension `recrd2` as defined above are denoted by terms such as `recrd2.make 2 {5,7} True True 42` or $\langle \text{recrd.num} = 2, \text{nums} = \{5,7\}, \text{nice} = \text{True}, \text{full} = \text{True}, \text{recrd2.num} = 42 \rangle$ or $\langle \text{recrd.num} = 2, \text{nums} = \{5,7\}, \text{nice} = \text{True}, \dots = \langle \text{full} = \text{True}, \text{recrd2.num} = 42 \rangle \rangle$.

7.4.3 Record Destructors

The only record destructors available are selectors which correspond to the field names. Every record type definition `record rname = fname1 :: "ftype1" ... fnamen :: "ftypen"` defines the record selector functions

```
fname1 :: 'a rname_scheme ⇒ ftype1
...
fnamen :: 'a rname_scheme ⇒ ftypen
```

Note that instead of `rname` their argument type is the record type scheme `'a rname_scheme`. Thus a record selector function for a field is polymorphic and may also be applied to every extended record to return the field value. However, to make a field name unique, it must be qualified by the name of the record type where it has been introduced.

Additionally a record type definition always defines the selector named `more` for the more part.

If `r` is a variable of type `recrd` as defined above, the term `nums r` selects the value of the second field. The same works if `r` has the extended type `recrd2`. The term `recrd.more r` selects the value of the more part which is the unit value if `r` has type `recrd` and the extension fragment if `r` has type `recrd2`.

A field selector cannot be applied directly to a record fragment. The fields of the fragment can only be selected if the fragment is embedded in the extended record.

7.4.4 Record Updates

In addition to the constructor and selector functions a record type definition `record rname = fname1 :: "ftype1" ... fnamen :: "ftypen"` defines the record update functions

```
fname1_update :: (ftype1 ⇒ ftype1) ⇒ 'a rname_scheme ⇒ 'a rname_scheme
...
fnamen_update :: (ftypen ⇒ ftypen) ⇒ 'a rname_scheme ⇒ 'a rname_scheme
```

Each record update function `fnamei_update` takes as argument an update function for values of type `ftypei` (which maps an old value to a new value) and a record value. It returns the record where the value of field `fnamei` is

the result of applying the update function to its old value and the values of all other fields are unchanged.

Like the selector functions the update functions are defined for the polymorphic record type scheme and can thus be also applied to all extended records.

A field may be set to a specific value *term* without regarding the old value by using the term *fname_i_update* (λ_term) *r* where a constant function (see Section 2.2.5) is used as update function for the field value. For record update applications of this form the alternative syntax

r (*fname_i := term*)

is available. Further notation for repeated updates is also available: *r* (*x := a*) (*y := b*) (*z := c*) may be written

r (*x := a, y := b, z := c*)

Note that the former term is equivalent to *z_update* (λ_c) (*y_update* (λ_b) (*x_update* (λ_a) *r*)), so the fields are actually set in the order in which they occur in the alternative notation from left to right.

7.4.5 Record Rules

A record type definition also introduces a number of named facts about the constructors, selectors and update functions. All fact names belong to the namespace of the record definition. The facts are named automatically in the same way for all record types, therefore the fact names must always be qualified by prefixing the record 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 4.6.3). Other rules must be explicitly used by referring them by their name.

Only some basic rules are described here, for more information refer to [7, Section 11.6 on records].

Simplifier Rules

The rules for injectivity of the record constructor have the form

$$\begin{aligned} & ((fname_1 = ?x_1, \dots, fname_n = ?x_n, \dots = ?x) = \\ & \quad (fname_1 = ?y_1, \dots, fname_n = ?y_n, \dots = ?y)) \\ & = (?x_1 = ?y_1 \wedge \dots \wedge ?x_n = ?y_n \wedge ?x = ?y) \end{aligned}$$

and are named *rname.iffs* for the record type *rname*.

Other rules added by a record definition to the simpset process terms where selectors or update functions are applied to constructed record values. They have the form of equations

$$\begin{aligned}
& fname_i \langle fname_1 = ?x_1, \dots, fname_n = ?x_n \rangle = ?x_i \\
& fname_i_update \ ?f \ \langle fname_1 = ?x_1, \dots, fname_n = ?x_n \rangle = \\
& \quad \langle fname_1 = ?x_1, \dots, fname_i = ?f \ ?x_i, \dots, fname_n = ?x_n \rangle
\end{aligned}$$

The set of all these rules is named *rname.simps* for the record type *rname*. Additional internal simplifier rules process selectors applied to updated records such as

$$\begin{aligned}
& fname_i \ (fname_i_update \ ?f \ ?r) = ?f \ (fname_i \ ?r) \\
& fname_i \ (fname_j_update \ ?f \ ?r) = fname_i \ ?r
\end{aligned}$$

where $i \neq j$.

Constructor Rules

Additional rules provide definitional equations for the constructors of the defined record type, such as

$$rname.make \ x_1 \ \dots \ x_n = \langle fname_1 = x_1, \dots, fname_n = x_n \rangle$$

Every rule provides a representation of a constructor application as a record expression. The set of these rules is named *rname.defs* for the record type *rname*. It is not added to the simpset, the rules must be explicitly applied by adding them to the simplifier method when needed (as described in Section 4.6.4).

The simplifier rules are mainly defined for record expressions. To apply them to record terms specified by the constructor functions the constructor rules must be used to convert the terms to record expressions.

Case Rules

Every definition for a record type *rname* introduces case rules (see Section 5.3.1) of the form

$$\begin{aligned}
& rname.cases: \\
& \quad \llbracket \bigwedge x_1 \ \dots \ x_n. \ ?y = \langle fname_1 = x_1, \dots, fname_n = x_n \rangle \implies ?P \rrbracket \implies ?P \\
& rname.cases_scheme: \\
& \quad \llbracket \bigwedge x_1 \ \dots \ x_n \ m. \ ?y = \langle fname_1 = x_1, \dots, fname_n = x_n, \dots = m \rangle \implies ?P \rrbracket \\
& \quad \implies ?P
\end{aligned}$$

They are valid because the record expressions cover all possibilities of constructing a value *?y* of the record type or record scheme type, respectively. Both rules are associated with the record type for use by the *cases* method (see Section 5.3.2), the method automatically selects the most sensible of them. Therefore the application of the method

`cases "term"`

where `term` is a record of type `rname` replaces an arbitrary goal by a goal where `term` is set equal to a record constructed from explicit field values x_1, \dots, x_n and possibly a more part.

The name used for the named context created by the `cases` method is “fields”. Therefore a structured proof using case based reasoning for a `term` of a record type `rname` has the form

```
proof (cases "term")
  case (fields x1 ... xn) ... show ?thesis <proof>
qed
```

The names x_i of the locally fixed variables can be freely selected, they denote the field values of the record.

The main purpose of applying the case rules is to provide a record expression for a record term which is not specified as such or by a constructor function, such as a variable of a record type. Providing the record expression makes the simplifier rules applicable, therefore a proof for a record often consists of an application of the `cases` method followed by an application of the `simp` method.

Induction Rule

Every definition for a record type `rname` introduces induction rules (see Section 5.5.1) of the form

```
rname.induct:
  ( $\bigwedge x_1 \dots x_n. ?P \ (fname_1=x_1, \dots, fname_n=x_n)$ )  $\implies$  ?P ?a
rname.induct_scheme:
  ( $\bigwedge x_1 \dots x_n m. ?P \ (fname_1=x_1, \dots, fname_n=x_n, \dots=m)$ )  $\implies$  ?P ?a
```

Like the case rule it is valid because the record expressions cover all possibilities of constructing a value `?a` of the record type `rname`.

The rules are associated with the record type for use by the methods `induction` and `induct` (see Section 5.5.3), the methods automatically select the most sensible of them. Therefore the application of the method

`induction x`

where `x` is a variable of type `rname` replaces a goal by a goal which uses a record expression in the place of `x`.

As for the case rule and the `cases` method, the names used for the named contexts created by the methods `induction` and `induct` are “fields”. Therefore a structured proof using induction for a variable `x` of record type `rname` has the form

```

proof (induction x)
  case (fields x1 ... xn) ... show ?case <proof>
qed

```

As an example, the induction rules for the record type *recrd* defined in Section 7.4.1 are

```

recrd.induct:
  ( $\bigwedge x_1 x_2 x_3. ?P \ (\text{num}=x_1, \text{nums}=x_2, \text{nice}=x_3) \implies ?P \ ?a$ )
recrd.induct_scheme:
  ( $\bigwedge x_1 x_2 x_3 m. ?P \ (\text{num}=x_1, \text{nums}=x_2, \text{nice}=x_3, \dots=m) \implies ?P \ ?a$ )

```

By applying the method (*induction* x) the goal (*num* x) = y is replaced by the goal $\bigwedge x_1 x_2 x_3. (\text{num} \ (\text{num} = x_1, \text{nums} = x_2, \text{nice} = x_3)) = y$.

Like the cases methods a transformation of this kind may enable the application of the simplifier methods.

7.4.6 Records as Bounded Natural Functors

As described in Section 7.4.1 a record type is mainly equivalent to the tuple type for all fields. Seen as type constructors, tuple types are BNFs (see Section 6.12.3). Due to composability of BNFs this implies that a (polymorphic) record type is a bounded natural functor as soon as all (polymorphic) field types are BNFs. However, HOL does not register record types as BNFs automatically, this must be done manually.

Record Type Constructors

The actual type constructor introduced by a record definition **record** ('a₁, ..., 'a_m) *rname* = ... is named *rname_ext*. The record type scheme name *rname_scheme* is a synonym for it, whereas ('a₁, ..., 'a_m) *rname* is a synonym for ('a₁, ..., 'a_m, *unit*) *rname_ext* (see Section 7.4.1).

The type constructor *rname_ext* is internally defined as a type copy (see Section 7.2.3) of a tuple type with all record field types as components. This tuple type is constructed from pairs (see Section 6.6) in a balanced way, so that the number of steps to access a component using the selectors *fst* and *snd* (see Section 6.6) is always logarithmic in the number of components.

Using a type copy implies that different records with the same component types are different, although isomorphic. It also makes it possible to define the field selector (see Section 7.4.3) and update (see Section 7.4.4) functions specifically for each record type.

Registering Record Types as BNF

As described in Section 6.12.4 the type constructor *prod* for pairs is a BNF. Due to composability of BNFs this is also true for every tuple type and the BNF property can be lifted to the type copy for the record scheme from the underlying tuple type for the components using the command **copy_bnf** (see Section 6.14.6). As described in Section 7.2.7 the type copy must before be registered as quotient using the command **setup_lifting**.

Together, a record type defined by *record* (*'a*₁, ..., *'a*_{*m*}) *rname* = *fname*₁ :: "*ftype*₁" ... *fname*_{*n*} :: "*ftype*_{*n*}" is registered as BNF by the command sequence

```
setup_lifting type_definition_rname_ext  
copy_bnf ('a1, ..., 'am, 'more) rname_ext
```

Note the additional type parameter *'more* for the more part of the record type scheme. Its name may be arbitrarily selected, it must only be distinct from the *'a*₁, ..., *'a*_{*m*}.

If a polymorphic component type *ftype*_{*i*} is not a registered BNF it is treated as BNF with all type parameters as dead (see Section 6.12.1). Every type parameter *'a*_{*j*} of the record type occurring on a dead parameter position in an *ftype*_{*i*} is registered as dead by the **copy_bnf** command. Since the more part always consists directly of the corresponding type parameter, that parameter is always registered as live.

After registering the record type scheme *rname_ext* as BNF the record type (*'a*₁, ..., *'a*_{*m*}) *rname* is immediately recognised as BNF because it is a synonym for the instantiation (*'a*₁, ..., *'a*_{*m*}, *unit*) *rname_ext*. Extensions of *rname* (see Section 7.4.1) are usually not recognised as BNF, because they instantiate the more part differently, for them the **copy_bnf** command must be applied separately to the extended type scheme.

The polymorphic example record type (*'a*, *'b*) *recrdp* defined in Section 7.4.1 is registered as (quotient and) BNF by

```
setup_lifting type_definition_recrdp_ext  
copy_bnf ('a, 'b, 'more) recrdp_ext
```

The type parameters *'b* and *'more* are registered as live, whereas *'a* is registered as dead because the second field has type *'a set* which is not a BNF.

Subtypes and Quotients of Record Types

Defining a subtype (see Section 7.2) of a record type *rname* is straightforward. If this subtype shall be registered as quotient using **setup_lifting** the record type should be registered as BNF before, so that the relator

`rel_rname_ext` is defined for it and known to HOL. Only then full transfer support is available for the subtype and no warning is issued by `setup_lifting` (see Section 6.14.4).

If, however, the record type is registered as BNF with dead parameters (see above), the relator cannot be used by `setup_lifting`. Then `setup_lifting` will still warn about not finding the relator, although it has been defined by `copy_bnf`.

Note that due to the type parameter for the more part the record type scheme is always polymorphic, even if the defined record type has no type parameters. Therefore the `setup_lifting` command always expects the relator for `rel_rname_ext` for subtypes of arbitrary record types.

Since a quotient type definition includes the `setup_lifting` command (see Section 7.3.1) a record type `rname` should be registered as BNF before a quotient type is defined for it. Additionally, the mapper `map_rname_ext`, introduced when registering `rname` as BNF should be registered as functor (see Section 6.12.4) so that the `quotient_type` command does not signal a warning about an undefined map function.

The polymorphic example record type `('a, 'b) recrdp` has a dead type parameter when registered as BNF (see above), therefore subtypes and quotient types for it cannot provide full transfer support.

Chapter 8

Isabelle/HOL Types

This chapter introduces a small basic part of the types available in HOL.

Most of the types are algebraic types (see Section 7.1). 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.

A type basically provides the type name or type constructor, if the type is parametric. Together with a type, HOL introduces functions on the values of the type, either implicitly by using type definition mechanisms described in Section 7, or explicitly by using definitions (see Sections 2.3, 6.7, and 6.10). In this way HOL populates the object level of the inner syntax with a rich language for expressing mathematical content. Additionally, HOL provides facts (usually derivation rules) about these functions which can be used in proofs. For every type this introduction describes its mathematical meaning, it gives a short description of (most of) the defined functions, and it lists some exemplary rules with explanations.

If applicable, the functions described for a type include the specific forms of ordering relations and lattice operations (see Section 6.2) and functions for binder syntax (see Section 6.3).

The semantics of the described functions is either given informally for well-known functions or by a description of the form *name* :: *type* \equiv *lambda-term*. The latter is often not the actual definition used by HOL for the function, it is only used here for documentation purpose.

At the beginning of each of the following sections the theories in session HOL are listed where the type, functions, and rules described in the section are actually defined.

8.1 Boolean Values

Theories: *HOL, Orderings, Complete_Lattices*

The type of boolean values is specified equivalent to an algebraic type of the form of the enumeration type (see Section 7.1.2)

```
datatype bool = True | False
```

The type *bool* plays a special role in HOL since it is the type of all terms which are used as formulas (see Section 2.5.1) in Isabelle. Every object logic used in Isabelle must define a type which plays this role.

8.1.1 Values

Values of type *bool* can directly be denoted by the parameterless constructors *True* and *False*.

The lattice constants *top* and *bot* (see Section 6.2.3) are available for type *bool* and denote the values *True* and *False*, respectively.

8.1.2 Destructors

Since both constructors are constant no selectors can be defined. Discriminators are not required since the constants are already boolean values.

A *case* term for type *bool* has the form

```
case term of True  $\Rightarrow$  term1 | False  $\Rightarrow$  term2
```

where *term* is a term of type *bool*.

As an alternative syntax HOL provides the usual form

```
if term then term1 else term2
```

8.1.3 Functions

The usual logical functions are defined for type *bool*: *conj*, *disj*, *implies*, *iff* of type *bool* \Rightarrow *bool* \Rightarrow *bool* with operator names (\wedge) , (\vee) , (\longrightarrow) , (\longleftrightarrow) and the unary negation *Not* of type *bool* \Rightarrow *bool* and operator name \neg . The function (\longleftrightarrow) is the specific instance of $(=)$ for type *bool* (see Section 6.2.1).

Functions for Orderings and Lattices

The ordering relations (see Section 6.2.2) and the lattice operations (see Section 6.2.3) are defined for type *bool* so that *False* < *True*. This implies that (\leq) is equivalent to (\longrightarrow) and (\sqcap) , (\sqcup) and also *min*, *max* are equivalent to (\wedge) , (\vee) .

The lattice operators (\sqcap) and (\sqcup) on sets are provided for *bool* by definitions $\sqcap A \equiv (\text{False} \notin A)$ and $\sqcup A \equiv (\text{True} \in A)$, so they correspond to conjunction and disjunction over sets, respectively. Note that the meta-logic quantifier \wedge (see Section 2.5.3) does *not* denote a conjunction operation on sets of boolean values. For nonempty sets of boolean values the functions *Min* and *Max* are equivalent to (\sqcap) and (\sqcup) .

Functions for Binder Syntax

The quantifiers are defined as “predicates on predicates”:

```

All :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool  $\equiv$   $\lambda P. (P = (\lambda x. \text{True}))$ 
Ex  :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool  $\equiv$   $\lambda P. (\neg \text{All } (\lambda x. \neg P x))$ 
Uniq :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool  $\equiv$ 
     $\lambda P. (\text{All } (\lambda x. (\text{All } (\lambda y. P x \longrightarrow P y \longrightarrow y = x))))$ 
Ex1  :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool  $\equiv$ 
     $\lambda P. (\text{Ex } (\lambda x. P x \wedge (\text{All } (\lambda y. P y \longrightarrow y = x))))$ 

```

with the alternative binder syntax $\forall x. \text{bterm}$ for the application *All* $(\lambda x. \text{bterm})$, $\exists x. \text{bterm}$ for *Ex* $(\lambda x. \text{bterm})$, $\exists_{\leq 1} x. \text{bterm}$ for *Uniq* $(\lambda x. \text{bterm})$, and $\exists ! x. \text{bterm}$ for *Ex1* $(\lambda x. \text{bterm})$. The *Uniq* quantifier states that there is atmost one value satisfying the predicate, the *Ex1* quantifier states that there is exactly one such value.

An iterated application for an n-ary predicate $\lambda x_1 \dots x_n. \text{bterm}$ can be written in the form $\forall x_1 \dots x_n. \text{bterm}$ for all quantifiers. As for lambda terms (see Section 2.2.5) types may be specified for (some of) the variables as in $\forall (x_1 :: \text{type}_1) \dots (x_n :: \text{type}_n). \text{bterm}$.

HOL also provides several functions which support binder syntax where a single bound variable is restricted (“bounded”) by an ordering (see Section 6.2.2) or inequality (see Section 6.2.1) relation to some value.

HOL provides functions similar to *All* and *Ex* to support the syntax of the bounded quantifiers

```

 $\forall x < y. \text{bterm} \equiv \forall x. x < y \longrightarrow \text{bterm}$ 
 $\exists x < y. \text{bterm} \equiv \exists x. x < y \wedge \text{bterm}$ 
 $\forall x \leq y. \text{bterm} \equiv \forall x. x \leq y \longrightarrow \text{bterm}$ 
 $\exists x \leq y. \text{bterm} \equiv \exists x. x \leq y \wedge \text{bterm}$ 
 $\forall x > y. \text{bterm} \equiv \forall x. x > y \longrightarrow \text{bterm}$ 
 $\exists x > y. \text{bterm} \equiv \exists x. x > y \wedge \text{bterm}$ 
 $\forall x \geq y. \text{bterm} \equiv \forall x. x \geq y \longrightarrow \text{bterm}$ 
 $\exists x \geq y. \text{bterm} \equiv \exists x. x \geq y \wedge \text{bterm}$ 
 $\forall x \neq y. \text{bterm} \equiv \forall x. x \neq y \longrightarrow \text{bterm}$ 
 $\exists x \neq y. \text{bterm} \equiv \exists x. x \neq y \wedge \text{bterm}$ 

```

8.1.4 Rules

The rules described here are the usual rules for an algebraic type and introduction / elimination / destruction rules for the functions, and some specific rules for negation. HOL provides many additional rules, see the Isabelle documentation [7, 5, 6] for how to use them in proofs.

Complex proofs using these rules can often be done automatically by the proof method *blast* (see Section 4.7.1).

Algebraic Type Rules

Since there are no selectors for *bool* and all constructors are constant the main simplifier rules are the rule sets *bool.distinct* and *bool.case* (see Section 7.1.5):

```
True ≠ False
False ≠ True
(case True of True ⇒ ?t1 | False ⇒ ?t2) = ?t1
(case False of True ⇒ ?t1 | False ⇒ ?t2) = ?t2
```

The case, split, and induction rules are

```
bool.exhaust:
  [[?y = True ⇒ ?P; ?y = False ⇒ ?P]] ⇒ ?P
bool.split:
  ?P (case ?t of True ⇒ ?t1 | False ⇒ ?t2) =
    ((?t = True → ?P ?t1) ∧ (?t = False → ?P ?t2))
bool.induct:
  [[?P True; ?P False]] ⇒ ?P ?a
```

Actually, as automatic case rule for type *bool* instead of *bool.exhaust* the slightly different rule

```
case_split:
  [[?Q ⇒ ?P; ¬ ?Q ⇒ ?P]] ⇒ ?P
```

is used, as described in Section 5.3.1.

For the alternate case term form described in Section 8.1.2 there is also a split rule:

```
if_split:
  ?P (if ?t then ?t1 else ?t2) =
    ((?t → ?P ?t1) ∧ (¬ ?t → ?P ?t2))
```

Other than *bool.split* this rule is automatically applied by the simplifier (see Section 4.6.5) for splitting *if*-terms.

Rules About Boolean Functions

For the functions described in Section 8.1.3 corresponding introduction rules (see Section 4.3.4), destruction rules (see Section 4.4.5), and elimination rules (see Section 5.4.2) are available. They are present in the rule sets *intro*, *dest*, or *elim*, respectively.

The introduction rules are:

$conjI: \llbracket ?P; ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$
 $disjI1: ?P \Longrightarrow ?P \vee ?Q$
 $disjI2: ?Q \Longrightarrow ?P \vee ?Q$
 $notI: (?P \Longrightarrow False) \Longrightarrow \neg ?P$
 $impI: (?P \Longrightarrow ?Q) \Longrightarrow ?P \longrightarrow ?Q$
 $iffI: \llbracket ?P \Longrightarrow ?Q; ?Q \Longrightarrow ?P \rrbracket \Longrightarrow ?P = ?Q$
 $allI: (\bigwedge x. ?P\ x) \Longrightarrow \forall x. ?P\ x$
 $exI: ?P\ ?x \Longrightarrow \exists x. ?P\ x$

The destruction rules are:

$conjunct1: ?P \wedge ?Q \Longrightarrow ?P$
 $conjunct2: ?P \wedge ?Q \Longrightarrow ?Q$
 $mp: \llbracket ?P \longrightarrow ?Q; ?P \rrbracket \Longrightarrow ?Q$
 $spec: \forall x. ?P\ x \Longrightarrow ?P\ ?x$
 $iffD1: \llbracket ?Q = ?P; ?Q \rrbracket \Longrightarrow ?P$
 $iffD2: \llbracket ?P = ?Q; ?Q \rrbracket \Longrightarrow ?P$

The rule *mp* is the well known logic rule “modus ponens”.

The elimination rules are:

$conjE: \llbracket ?P \wedge ?Q; \llbracket ?P; ?Q \rrbracket \Longrightarrow ?R \rrbracket \Longrightarrow ?R$
 $disjE: \llbracket ?P \vee ?Q; ?P \Longrightarrow ?R; ?Q \Longrightarrow ?R \rrbracket \Longrightarrow ?R$
 $notE: \llbracket \neg ?P; ?P \rrbracket \Longrightarrow ?R$
 $impE: \llbracket ?P \longrightarrow ?Q; ?P; ?Q \Longrightarrow ?R \rrbracket \Longrightarrow ?R$
 $iffE: \llbracket ?P = ?Q; \llbracket ?P \longrightarrow ?Q; ?Q \longrightarrow ?P \rrbracket \Longrightarrow ?R \rrbracket \Longrightarrow ?R$
 $allE: \llbracket \forall x. ?P\ x; ?P\ ?x \Longrightarrow ?R \rrbracket \Longrightarrow ?R$
 $exE: \llbracket \exists x. ?P\ x; \bigwedge x. ?P\ x \Longrightarrow ?Q \rrbracket \Longrightarrow ?Q$

Additionally, the following four rules can be used for “proofs by contradiction”:

$contrapos_pn: \llbracket ?Q; ?P \Longrightarrow \neg ?Q \rrbracket \Longrightarrow \neg ?P$
 $contrapos_pp: \llbracket ?Q; \neg ?P \Longrightarrow \neg ?Q \rrbracket \Longrightarrow ?P$
 $contrapos_nn: \llbracket \neg ?Q; ?P \Longrightarrow ?Q \rrbracket \Longrightarrow \neg ?P$
 $contrapos_np: \llbracket \neg ?Q; \neg ?P \Longrightarrow ?Q \rrbracket \Longrightarrow ?P$

Equivalence of Derivation Rules and Boolean Terms

Using these rules together with the rules about the meta-logic operators

$atomize_all: (\bigwedge x. ?P\ x) \equiv \forall x. ?P\ x$

$\text{atomize_imp}: (?A \implies ?B) \equiv ?A \longrightarrow ?B$
 $\text{atomize_conjL}: (\llbracket ?A; ?B \rrbracket \implies ?C) \equiv (?A \wedge ?B \implies ?C)$
 $\text{atomize_conj}: (?A \&\&\& ?B) \equiv ?A \wedge ?B$

every meta-logic derivation rule with possibly multiple conclusions (see Section 2.5.4)

$\bigwedge x_1 \dots x_m. \llbracket A_1; \dots; A_n \rrbracket \implies C_1 \&\&\& \dots \&\&\& C_h$

can be converted to the boolean term

$\forall x_1 \dots x_m. (A_1' \wedge \dots \wedge A_n') \longrightarrow C_1' \wedge \dots \wedge C_h'$

(where each A_i' and C_i' is converted in the same way if it is again a derivation rule), and vice versa.

In principle every theorem may be specified in either of both forms. However, its application by proof methods in other proofs is usually only possible if it is specified in derivation rule form. Therefore it is usually preferable to specify theorems in this form.

8.2 The Unit Type

Theories: *Product_Type*

The unit type has only one value. It is specified equivalent to an algebraic type of the form of the enumeration type (see Section 7.1.2)

datatype *unit* = *Unity*

A typical use of the unit type is for instantiating a parameter of a parameterized type when you actually don't care about it. For example, if you have to use binary predicates of type `'a \Rightarrow 'b \Rightarrow bool` but do not care about the second argument you can use the type `'a \Rightarrow unit \Rightarrow bool`. Syntactically its values are still binary predicates, however, since there is only one possible value for the second argument, it cannot affect the result, so semantically they are unary predicates. Another use of the unit type is for the more part in unextended record types (see Section 7.4.1).

So the unit type plays the role of an “empty type” (which does not exist for formal reasons).

8.2.1 Values

Values of type *unit* can directly be denoted by the parameterless constructor *Unity*. The usual way of denoting it is by its alternative form `()`, which has been chosen because the unit type is also used to represent empty tuples (see Section 8.6).

8.2.2 Destructors

A *case* term for type *unit* has the form

case term of () \Rightarrow *term*₁

where *term* is a term of type *unit*. It is equivalent to *term*₁.

8.2.3 Rules

The usual rules for an algebraic type are also defined for *unit*. They are of no much use, but here they are shown for the interested reader:

unit.exhaust: $(?y = () \Rightarrow ?P) \Rightarrow ?P$
unit.split: $?P \text{ (case ?t of () } \Rightarrow ?t_1) = (?t = () \longrightarrow ?P ?t_1)$
unit.induct: $?P () \Rightarrow ?P ?a$

The split rule is required if during a proof a *case* term for type *unit* (see Section 8.2.2) occurs for some reason. Although it is trivially equivalent to its subterm *term*₁, the simplifier will not use that equivalence since it never splits *case* terms automatically. It must be configured as *simp split: unit.split* to do so.

Since there are no functions for *unit* there are no introduction / destruction / elimination rules.

8.3 Natural Numbers

Theories: *Nat, Groups, Rings*

The type of natural numbers is specified equivalent to a recursive algebraic type (see Section 7.1) of the form

datatype *nat* = *0* / *Suc nat*

It is not really defined in this way, because *0* is syntactically not a constructor, but it can mainly be used in the same way.

The type *nat* denotes the mathematical concept of natural numbers, it has infinitely many values, there is no upper limit.

8.3.1 Values

Values of type *nat* can be denoted in the usual way using constructor expressions such as *Suc 0*, *Suc (Suc 0)*, ...

Alternatively they can be denoted by decimal number literals such as *0*, *1*, *2*, ... of arbitrary size.

However, the decimal number literals are overloaded and may also denote values of other numerical types, such as type *int* for the integer numbers. Therefore the type of an isolated decimal number literal is not determined, which may cause unexpected effects. To denote a value of type *nat* its type may be explicitly specified as described in Section 2.2.2, such as *1::nat*.

The lattice constant *bot* (see Section 6.2.3) is available for type *nat* and denotes the value 0. The constant *top* is not available for type *nat*.

8.3.2 Destructors

Since *Suc* plays the role of a constructor, corresponding destructors can be defined. The selector function which inverts *Suc* is defined as *nat.pred* where *nat.pred x* is equivalent to *x - 1* and *nat.pred 0 = 0*. This selector is not intended to be used directly, use the subtraction function described below instead. There are no discriminators, instead the equality terms *x = 0* and *x ≠ 0* can be used.

A *case* term for type *nat* has the form

case term of 0 ⇒ term₁ | Suc n ⇒ term₂

where *term* is a term of type *nat*.

8.3.3 Functions

The usual basic arithmetic functions are defined for type *nat*: *plus*, *minus*, *times*, *divide*, *modulo* of type *nat ⇒ nat ⇒ nat* with operator names (+), (-), (*), (div), (mod) and alternate operator name (/) for (div). Subtraction is truncated at 0, e.g. 4 - 7 evaluates to 0. Also defined is the binary “divides” relation *dvd_class.dvd* with operator name (dvd).

Like decimal number literals all these functions are overloaded and not restricted to natural numbers. As a consequence, a proposition such as

4 - 7 = 0

is not valid and cannot be proved. To become a provable fact it must be specified in a form like

(4::nat) - 7 = 0

which can be proved by method *simp*. Note that it is sufficient to specify the type for a single literal, because then the type of the function (-) is derived to be *nat ⇒ nat ⇒ nat* (there are no “mixed-typed” arithmetic functions) from which the type of the second literal is derived and similar for the equality.

For type *nat* HOL defines the *size* function (see Section 6.8.4) to be the identity function.

Functions for Orderings and Lattices

The ordering relation ($<$) (see Section 6.2.2) is defined to be the usual ordering on natural numbers. This implies that also *min* and *max* have their usual meaning. The lattice operations (see Section 6.2.3) are overloaded for type *nat* so that (\sqcap) , (\sqcup) are equivalent to *min*, *max*.

The functions *Min* and *Max* on sets are also defined as expected on finite nonempty sets of natural numbers, otherwise their result is underspecified. The lattice operators (\sqcap) and (\sqcup) are equivalent to *Min* and *Max*, additionally $\sqcup \{\}$ is specified to be 0 and (\sqcap) is also specified to return the minimal value for infinite sets of natural numbers.

8.3.4 Rules

HOL provides a large number of rules applicable for proofs about values of type *nat*. Here we only show rules for an algebraic type and introduction / elimination / destruction rules for the functions, as for other types. They are usually not sufficient for proofs about natural numbers and should only give an impression about the type *nat* in comparison with other types.

Proofs for linear arithmetic properties of *nat* values using these and other rules can often be done automatically by the proof methods *linarith* or *arith* (see Section 4.7.1).

Algebraic Type Rules

The simplifier rules for the constructors of type *nat* are the rule sets *nat.inject*, *nat.distinct* and *nat.case* (see Section 7.1.5):

```
(Suc ?x = Suc ?y) = (?x = ?y)
0 ≠ Suc ?x
Suc ?x ≠ 0
(case 0 of 0 ⇒ ?t1 | Suc x ⇒ ?t2 x) = ?t1
(case Suc ?x of 0 ⇒ ?t1 | Suc x ⇒ ?t2 x) = ?t2 ?x
```

Note the difference between the unknown *?x* and the locally bound *x* in the last rule.

The case, split, and induction rules are

```
nat.exhaust:
  ⟦?y = 0 ⇒ ?P; ∧x. ?y = Suc x ⇒ ?P⟧ ⇒ ?P
nat.split:
  ?P (case ?t of 0 ⇒ ?t1 | Suc x ⇒ ?t2 x) =
  ((?t = 0 ⟶ ?P ?t1) ∧ (∀x. ?t = Suc x ⟶ ?P (?t2 x)))
nat.induct:
  ⟦?P 0; ∧i. ?P i ⇒ ?P (Suc i)⟧ ⇒ ?P ?n
```

For the case and induction rule the cases are named 0 and *Suc*.

Other Induction Rules

The rule *nat.induct* is the induction rule associated with type *nat*. Additionally, there are several other induction rules for values of type *nat* which may be used for other types of induction by specifying them explicitly to the *induction* or *induct* method:

```

nat_less_induct:
  
$$\llbracket \bigwedge i. \forall j < i. ?P\ j \implies ?P\ i \rrbracket \implies ?P\ ?n$$

infinite_descent0:
  
$$\llbracket ?P\ 0; \bigwedge i. \llbracket 0 < i; \neg ?P\ i \rrbracket \implies \exists j < i. \neg ?P\ j \rrbracket \implies ?P\ ?n$$

diff_induct:
  
$$\begin{aligned} & \llbracket \bigwedge i. ?P\ i\ 0; \bigwedge j. ?P\ 0\ (Suc\ j); \bigwedge i\ j. ?P\ i\ j \implies ?P\ (Suc\ i)\ (Suc\ j) \rrbracket \\ & \implies ?P\ ?m\ ?n \end{aligned}$$

less_Suc_induct [consumes 1]:
  
$$\begin{aligned} & \llbracket ?m < ?n; \bigwedge i. ?P\ i\ (Suc\ i); \\ & \bigwedge i\ j\ k. \llbracket i < j; j < k; ?P\ i\ j; ?P\ j\ k \rrbracket \implies ?P\ i\ k \rrbracket \\ & \implies ?P\ ?m\ ?n \end{aligned}$$

nat_induct_at_least [consumes 1]:
  
$$\llbracket ?m \leq ?n; ?P\ ?m; \bigwedge i. \llbracket ?m \leq i; ?P\ i \rrbracket \implies ?P\ (Suc\ i) \rrbracket \implies ?P\ ?n$$

nat_induct_non_zero [consumes 1]:
  
$$\llbracket 0 < ?n; ?P\ 1; \bigwedge i. \llbracket 0 < i; ?P\ i \rrbracket \implies ?P\ (Suc\ i) \rrbracket \implies ?P\ ?n$$

inc_induct [consumes 1]:
  
$$\begin{aligned} & \llbracket ?n \leq ?m; ?P\ ?m; \bigwedge i. \llbracket ?n \leq i; i < ?m; ?P\ (Suc\ i) \rrbracket \implies ?P\ i \rrbracket \\ & \implies ?P\ ?n \end{aligned}$$

strict_inc_induct [consumes 1]:
  
$$\begin{aligned} & \llbracket ?n < ?m; \bigwedge i. ?m = Suc\ i \implies ?P\ i; \\ & \bigwedge i. \llbracket i < ?m; ?P\ (Suc\ i) \rrbracket \implies ?P\ i \rrbracket \\ & \implies ?P\ ?n \end{aligned}$$

dec_induct [consumes 1]:
  
$$\begin{aligned} & \llbracket ?m \leq ?n; ?P\ ?m; \bigwedge i. \llbracket ?m \leq i; i < ?n; ?P\ i \rrbracket \implies ?P\ (Suc\ i) \rrbracket \\ & \implies ?P\ ?n \end{aligned}$$


```

Note that the last six rules combine induction with elimination, as described in Section 5.5.5, and are therefore attributed by *[consumes 1]*.

Rules About Functions

The constructor function *Suc* has result type *nat* and therefore cannot occur as outermost operator in a proposition. Therefore introduction, destruction, and elimination rules for *Suc* must embed the application term in a proposition with the help of a predicate. The ordering relations are used for this purpose, thus only ordering properties can be proved using these rules.

Introduction rules (see Section 4.3.4) for *Suc* are

```

le_SucI: ?m ≤ ?n ⟹ ?m ≤ Suc ?n
less_SucI: ?m < ?n ⟹ ?m < Suc ?n
Suc_leI: ?m < ?n ⟹ Suc ?m ≤ ?n
Suc_lessI:  $\llbracket ?m < ?n; Suc\ ?m \neq ?n \rrbracket \implies Suc\ ?m < ?n$ 

```

Destruction rules (see Section 4.4.5) for *Suc* are

Suc_leD: $\text{Suc } ?m \leq ?n \implies ?m \leq ?n$
Suc_lessD: $\text{Suc } ?m < ?n \implies ?m < ?n$
Suc_less_SucD: $\text{Suc } ?m < \text{Suc } ?n \implies ?m < ?n$
Suc_le_lessD: $\text{Suc } ?m \leq ?n \implies ?m < ?n$

An elimination rule (see Section 5.4.2) for *Suc* is

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

For the arithmetic functions described in Section 8.3.3 most properties are specified by equations such as

add_Suc_right: $?m + \text{Suc } ?n = \text{Suc } (?m + ?n)$
mult_Suc_right: $?m * \text{Suc } ?n = ?m + ?m * ?n$
diff_Suc_1: $\text{Suc } ?n - 1 = ?n$
min_Suc_Suc: $\min (\text{Suc } ?m) (\text{Suc } ?n) = \text{Suc } (\min ?m ?n)$
max_Suc_Suc: $\max (\text{Suc } ?m) (\text{Suc } ?n) = \text{Suc } (\max ?m ?n)$
dvd_diff_nat: $\llbracket ?k \text{ dvd } ?m; ?k \text{ dvd } ?n \rrbracket \implies ?k \text{ dvd } ?m - ?n$

Many of these equations are members of the simpset, therefore many simple arithmetic properties can be proved by the simplifier. More complex properties may need the methods *linarith* or *arith*, or can be proved by induction or a combination thereof.

8.4 Sets

Theories: *Set*, *Boolean_Algebras*, *Relation*, *BNF_Def*, *Finite_Set*

You may think of the type constructor *set* as being specified equivalent to the parameterized algebraic type

```
datatype 'a set = Collect (contains: 'a  $\Rightarrow$  bool)
```

Thus a set is simply a wrapper for a unary predicate of type $'a \Rightarrow \text{bool}$ (see Section 6.1.1).

The selector *contains* is not intended for general use and can only be specified in qualified form as *Predicate_Compile.contains*.

8.4.1 Values

Basically, values of type $'a \text{ set}$ are denoted by constructor application terms of the form *Collect* $(\lambda x. \text{ bterm})$ where $(\lambda x. \text{ bterm})$ is a unary predicate (see Section 6.1.1).

For such a term HOL provides the alternative “set comprehension” syntax (which is a special form of binder syntax)

`{x. bterm}`

HOL also provides two standard abbreviations:

- `{}` for the empty set, which is written `{x. False}` in comprehension syntax, and
- `UNIV` for the universal set `{x. True}`.

Both abbreviations are available for arbitrary types `'a set`. The universal set is the set of all values of the type `'a`. Examples are `UNIV :: bool set` which is the set `{True, False}` and `UNIV :: nat set` which is the set of all natural numbers.

The lattice constants `top` and `bot` (see Section 6.2.3) are available for sets and denote the universal set `UNIV` and the empty set `{}`, respectively.

For a set comprehension of the form

`{x. ∃ x1 ... xn. x = term ∧ bterm}`

HOL provides the alternative syntax

`{term / x1 ... xn. bterm}`

Note that the bindings of the variables `x1 ... xn` follow the `/` although the scope of the bindings includes the `term` before the `/`.

An example for this syntax is the set

`{2*x / x::nat. x < 5}`

which contains the numbers 0, 2, 4, 6, 8. Note that types may be specified in the bindings in the same way as for the `∃` quantifier.

8.4.2 Destructors

For use instead of the selector

`Predicate_Compile.contains :: 'a set ⇒ 'a ⇒ bool`

HOL provides the function with reversed arguments

`Set.member :: 'a ⇒ 'a set ⇒ bool`

with alternate operator name (`∈`). It combines unwrapping and applying the predicate and corresponds to the usual member test predicate for sets.

Since there is only one constructor, discriminators and case terms are not available for sets.

8.4.3 Functions

In addition to the basic function *Set.member* and its negation *Set.not_member* with operator name (\notin) HOL provides the other usual functions on sets: the relations *subset*, *subset_eq*, *supset*, *supset_eq* with operator names (\subset), (\subseteq), (\supset), (\supseteq) and the operations *inter*, *union*, *minus* with operator names (\cap), (\cup), ($-$). The function *minus* is set difference, there is also the unary set complement (relative to the universal set *UNIV*, see Section 8.4.1) *uminus* which also has the operator name ($-$) for prefix application. The functions *minus* and *uminus* are overloaded for other types as well, therefore Isabelle will not automatically derive that their arguments are sets.

Intersection and union of a family of sets is supported by the functions

```
Inter :: 'a set set  $\Rightarrow$  'a set
Union :: 'a set set  $\Rightarrow$  'a set
```

with operator names (\cap) and (\cup) for prefix notation.

HOL also provides the function

```
insert :: 'a  $\Rightarrow$  'a set  $\Rightarrow$  'a set  $\equiv \lambda a B. \{x. x = a \vee x \in B\}$ 
```

which inserts a value into a set. HOL uses it to introduce the set enumeration syntax

```
{x1, ..., xn}
```

as abbreviation for *insert* *x*₁ (... (*insert* *x*_{*n*} {}) ...).

Moreover HOL provides the functions

```
Pow :: 'a set  $\Rightarrow$  'a set set  $\equiv \lambda A. \{B. B \subseteq A\}$ 
```

```
Powp :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\equiv \lambda P B. \forall x \in B. P x$ 
```

```
is_singleton :: 'a set  $\Rightarrow$  bool  $\equiv \lambda A. (\exists x. A = \{x\})$ 
```

```
the_elem :: 'a set  $\Rightarrow$  'a  $\equiv \lambda A. (THE x. A = \{x\})$ 
```

```
pairwise :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\equiv$ 
```

```
   $\lambda f A. (\forall x \in A. \forall y \in A. x \neq y \longrightarrow f x y)$ 
```

```
disjnt :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\equiv \lambda A B. A \cap B = \{\}$ 
```

Pow is the powerset operator, *Powp* is the same for a predicate instead of a set. As usual, the result of *the_elem* *A* is underspecified (see Section 6.3.2) if *A* is not known to be a singleton set. The relation application *pairwise* *f* *A* is satisfied if *f* is satisfied for all pairs of different elements of *A*. The relation *disjnt* tests two sets for being disjoint.

As a convention, variables for sets are usually denoted by uppercase letters.

Functions for Orderings and Lattices

The ordering relations (see Section 6.2.2) are defined for sets so that $(<)$ is equivalent to the subset relation (\subset) and analogously for (\leq) , $(>)$, (\geq) . The lattice operations (\cap) , (\cup) , (\bigcap) , and (\bigcup) (see Section 6.2.3) are overloaded to be equivalent to (\cap) , (\cup) , (\bigcap) , and (\bigcup) .

Since (\subset) is not a total ordering the functions *min* and *max* are *not* equivalent to (\cap) and (\cup) and the functions *Min* and *Max* are not available for sets of sets.

Functions for Binder Syntax

HOL provides many functions which support binder syntax where a single bound variable is restricted (“bounded”) by a member or subset relation to some set.

HOL provides functions similar to *All* and *Ex* (see Section 8.1.3) to support the syntax of bounded quantifiers

$$\begin{aligned} \forall x \in \text{sterm}. \text{bterm} &\equiv \forall x. x \in \text{sterm} \longrightarrow \text{bterm} \\ \exists x \in \text{sterm}. \text{bterm} &\equiv \exists x. x \in \text{sterm} \wedge \text{bterm} \\ \exists !x \in \text{sterm}. \text{bterm} &\equiv \exists !x. x \in \text{sterm} \wedge \text{bterm} \\ \forall x \subset \text{sterm}. \text{bterm} &\equiv \forall x. x \subset \text{sterm} \longrightarrow \text{bterm} \\ \exists x \subset \text{sterm}. \text{bterm} &\equiv \exists x. x \subset \text{sterm} \wedge \text{bterm} \\ \forall x \subseteq \text{sterm}. \text{bterm} &\equiv \forall x. x \subseteq \text{sterm} \longrightarrow \text{bterm} \\ \exists x \subseteq \text{sterm}. \text{bterm} &\equiv \exists x. x \subseteq \text{sterm} \wedge \text{bterm} \\ \exists !x \subseteq \text{sterm}. \text{bterm} &\equiv \exists !x. x \subseteq \text{sterm} \wedge \text{bterm} \end{aligned}$$

and the bounded descriptor (see Section 6.3.3)

$$\text{LEAST } x \in \text{sterm}. \text{bterm} \equiv \text{LEAST } x. x \in \text{sterm} \wedge \text{bterm}$$

Note that there is no such form for *GREATEST*. Other than for the plain quantifier only one bounded variable may be specified for this form. If there are more, the quantifiers must be nested as in $\forall x \in \text{sterm}_1. \forall y \in \text{sterm}_2. \text{bterm}$.

As set comprehension syntax for the special case of a predicate which includes a member test HOL provides the syntax

$$\{x \in \text{sterm}. \text{bterm}\}$$

for a term of the form $\{x. x \in \text{sterm} \wedge \text{bterm}\}$.

For the operations (\bigcap) , (\bigcup) HOL provides the alternative syntax of the form

$$\bigcap_{x \in \text{term}_1}. \text{term}_2$$

where both terms must have a `set` type and `x` may occur free in `term2`. This form is equivalent to $\bigcap \{term_2 \mid x. x \in term_1\}$ which is the intersection over all sets returned by `term2` when `x` adopts all values in the set `term1`.

For both operators HOL also provides the abbreviated syntax of the form

$\bigcap x. term_2$

for $\bigcap x \in UNIV. term_2$ and the further abbreviation

$\bigcap x_1 \dots x_n. term_2$

for $\bigcap x_1. \dots \bigcap x_n. term_2$ (which is not available for the form with \in).

Note that `MIN x::nat. x < 5` is *not* the minimum of the numbers which are less than 5, instead it is the minimum of the boolean values which occur as the result of `x < 5` if `x` adopts all possible values of type `nat`, i.e. it is equal to the value `False`. The minimum of the numbers which are less than 5 is denoted by `LEAST x::nat. x < 5` which is equal to the value 0.

BNF Functions

As described in Section 6.12.1 the type constructor `set` is a natural functor, although not bounded. The corresponding mapper is provided as the function

`image :: ('a \Rightarrow 'b) \Rightarrow 'a set \Rightarrow 'b set $\equiv \lambda f A. \{y. \exists x \in A. y = f x\}$`

with operator name `(')`. As set-function the identity `id` is used.

The predicate (see Section 6.12.2) is the function `Powp` (see above), the relator is defined as

`rel_set :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a set \Rightarrow 'b set \Rightarrow bool`
 $\equiv \lambda R A B. (\forall x \in A. \exists y \in B. R x y) \wedge (\forall y \in B. \exists x \in A. R x y)$

Two sets are related if every element in the first set is related with an element in the second set and vice versa.

HOL also provides the function

`vimage :: ('a \Rightarrow 'b) \Rightarrow 'b set \Rightarrow 'a set $\equiv \lambda f A. \{x. f x \in A\}$`

with operator name `(-')` for the reverse image of a function. As described in Section 6.11.1 it is a contravariant mapper for `set`.

Functions for Finite Sets

HOL defines the predicate `finite` by the inductive definition (see Section 6.7)

```

inductive finite :: "'a set  $\Rightarrow$  bool" where
  "finite {}"
| "finite A  $\Rightarrow$  finite (insert a A)"

```

So a set is finite if it can be constructed from the empty set by a finite sequence of inserting single elements.

For iterating through the elements of a finite set HOL introduces the function

```

Finite_Set.fold :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$  'a set  $\Rightarrow$  'b

```

where $\text{fold } f \ z \ \{x_1, \dots, x_n\} = f \ x_1 \ (\dots (f \ x_n \ z) \dots)$ if the resulting value is independent of the order of the x_i , i.e., the function f must be “left-commutative” on the values in the set. If it is not, its result is underspecified. If the set is not finite the result is always the starting value z .

The function *Finite_Set.fold* is not intended for direct use, it is used by HOL to provide other functions. The most basic is

```

card :: 'a set  $\Rightarrow$  nat  $\equiv \lambda A. \text{Finite\_Set.fold } (\lambda\_ n. \text{Suc } n) \ 0 \ A$ 
```

for the cardinality of sets. For finite sets it is the number of elements, for infinite sets, due to the way *fold* is defined, it is always 0. Note the difference to the cardinalities described in Section 6.12.3, which are represented by ordering relations and also support infinite cardinalities.

8.4.4 Rules

Algebraic Type Rules

Since HOL does not define the type *'a set* as an algebraic type it does not provide the standard rules and rule sets. However, the relevant properties are available in slightly different form.

Injectivity of the constructor is not specified as a simplifier equation, instead there is the rule

```

Collect_inj: Collect ?P = Collect ?Q  $\Rightarrow$  ?P = ?Q

```

Note that the other direction is always satisfied for arbitrary functions (see Section 8.7.3). It is provided in the form where the predicates are applied to arguments:

```

Collect_eqI: ( $\bigwedge x. ?P \ x = ?Q \ x$ )  $\Rightarrow$  Collect ?P = Collect ?Q

```

where its name reflects that it has the form of an introduction rule (see Section 4.3.4) for the equality relation (=) (see Section 6.2.1) on type *'a set*.

Equivalent to a “defining equation” for the selector there is the rule

```

mem_Collect_eq: Set.member ?a (Collect ?P) = ?P ?a

```


which is

$$(\lambda a. \lambda x. ?P\ x) = ?P\ \lambda a$$

in alternative syntax. This rule is also provided in the form of two separate rules for both directions:

$$\begin{aligned} \text{CollectI: } ?P\ \lambda a &\implies \lambda x. ?P\ x \\ \text{CollectD: } \lambda x. ?P\ x &\implies ?P\ \lambda a \end{aligned}$$

having the form of an introduction and a destruction rule (see Section 4.4.5). Due to the simple wrapper structure of type `'a set` no other algebraic type rules apply.

Rules About Functions

Using the rules described above and the definitions of the functions on sets described in Section 8.4.3 it is possible to convert every proposition about sets to an equivalent proposition about predicates and boolean operations (see Section 8.1.3) and prove it in this form. Most automatic proof methods described in Section 4.7.1 use this conversion and can thus prove many goals which involve sets and set operations, in particular the method `blast`.

Additionally HOL provides rules which directly work for functions on sets such as the introduction rules (see Section 4.3.4)

$$\begin{aligned} \text{subsetI: } (\lambda x. x \in ?A \implies x \in ?B) &\implies ?A \subseteq ?B \\ \text{IntI: } [\lambda c. c \in ?A; \lambda c. c \in ?B] &\implies \lambda c. c \in ?A \cap ?B \\ \text{UnI1: } \lambda c. c \in ?A &\implies \lambda c. c \in ?A \cup ?B \\ \text{PowI: } ?A \subseteq ?B &\implies ?A \in \text{Pow } ?B \end{aligned}$$

the destruction (see Section 4.4.5) and elimination (see Section 5.4.2) rules

$$\begin{aligned} \text{subsetD: } [\lambda c. c \in ?A; \lambda c. c \in ?B] &\implies \lambda c. c \in ?B \\ \text{IntD1: } \lambda c. c \in ?A \cap ?B &\implies \lambda c. c \in ?A \\ \text{UnE: } [\lambda c. c \in ?A \cup ?B; \lambda c. c \in ?A \implies ?P; \lambda c. c \in ?B \implies ?P] &\implies ?P \\ \text{PowD: } ?A \in \text{Pow } ?B &\implies ?A \subseteq ?B \end{aligned}$$

and the simplifier equations (see Section 4.6.2)

$$\begin{aligned} \text{subset_eq: } (?A \subseteq ?B) &= (\forall x \in ?A. x \in ?B) \\ \text{Int_def: } ?A \cap ?B &= \{x. x \in ?A \wedge x \in ?B\} \\ \text{Un_def: } ?A \cup ?B &= \{x. x \in ?A \vee x \in ?B\} \\ \text{Pow_def: } \text{Pow } ?A &= \{B. B \subseteq ?A\} \end{aligned}$$

where, despite their names, (\cap) and (\cup) are not really defined in this way.

There are also many rules about `finite` and `card` such as

$$\begin{aligned} \text{finite_subset: } [\lambda c. c \in ?A; \lambda c. c \in ?B] &\implies \text{finite } ?A \\ \text{finite_Un: } \text{finite } (?F \cup ?G) &= (\text{finite } ?F \wedge \text{finite } ?G) \\ \text{finite_Int: } \text{finite } ?F \vee \text{finite } ?G &\implies \text{finite } (?F \cap ?G) \end{aligned}$$

```

card_mono:  $\llbracket \text{finite } ?B; ?A \subseteq ?B \rrbracket \implies \text{card } ?A \leq \text{card } ?B$ 
card_Diff_subset_Int:
   $\text{finite } (?A \cap ?B) \implies \text{card } (?A - ?B) = \text{card } ?A - \text{card } (?A \cap ?B)$ 
card_Un_Int:
   $\llbracket \text{finite } ?A; \text{finite } ?B \rrbracket$ 
   $\implies \text{card } ?A + \text{card } ?B = \text{card } (?A \cup ?B) + \text{card } (?A \cap ?B)$ 

```

Many rules about *finite* are known by the automatic proof methods (see Section 4.7.1). Rules about *card* must often be specified explicitly.

8.5 Optional Values

Theories: *Option*

A function argument is optional if it can be omitted. In Isabelle, however, every function has a fixed number of arguments, it is not possible to omit one or more of them. Instead, the idea is to have a special value with the meaning of “no value”. The presence of such a value is no more a property of the function, it is a property of the function argument’s *type*.

Normally, types do not include a “no value” value, it must be introduced separately and it must be different from all existing values. For a type τ this can be done by defining a new type which has one more value. However, since the values of the new type are always different from those of τ (see Section 2.2.1), the new type has to include the values of τ in a “wrapped” form.

All this is done by the algebraic type

```

datatype 'a option =
  None
/ Some (the: 'a)

```

It is polymorphic with one type parameter *'a* (see Section 2.2.1), for every type τ it provides the new type τ *option*.

In this way the type *nat option* can be used to add a “no value” to the natural numbers.

The type constructor *option* can also be used to emulate partial functions (which do not exist in Isabelle) by functions with a result type τ *option*.

8.5.1 Values

Type *option* provides two constructors. The parameterless constructor *None* denotes the “no value” value. The unary constructor *Some* provides the value *Some v* for every value v of the type parameter *'a*. In this way the type τ *option* includes all values v of τ wrapped as *Some v*. Since constructors of

algebraic types are injective (see Section 7.1.2) different values stay different when wrapped. Since different constructors denote different values the value *None* is different from all wrapped values. Note also, that for different types *t1* and *t2* the *None* values in *t1 option* and *t2 option* are different, because they belong to different types.

The type *nat option* provides the value *None* and wrapped natural numbers of the form *Some 5*.

8.5.2 Destructors

The selector *the* is used to “unwrap” the wrapped values: *the (Some v) = v*. As described in Section 7.1.3 the selector can also be applied to *None*, but the result is underspecified. The term *the None* denotes a unique value of the parameter type *'a*, but no information is available about that value.

Although not introduced by the type definition, the function *Option.is_none* plays the role of a discriminator and tests values for being *None*. Alternatively the equality terms *x = None* and *x ≠ None* can be used.

A *case* term for type *t option* has the form

```
case term of None ⇒ term1 / Some v ⇒ term2
```

where *term* is a term of type *t option* and *v* is a variable of type *t* which may occur in *term₂*. Such *case* terms can be used to process a *term* for an optional value. It is tested whether a (wrapped) value is present, if not *term₁* specifies a default, otherwise *term₂* specifies a use of the unwrapped value *v*. The same effect can be achieved by the term

```
if term = None then term1 else term2'
```

where *term₂'* uses (*the term*) instead of *v*. However, proofs may differ depending on which of both forms is used.

8.5.3 Functions

The function

```
Option.these :: 'a option set ⇒ 'a set ≡  
λA. image the {x∈A. x≠None}
```

extends the selector *the* to sets. It returns the set of unwrapped values from a set of optional values, ignoring *None* if present in the set.

No orderings or lattice functions (see Section 6.2) are specified for values of type *'a option*.

BNF Functions

The type constructor *option* is a bounded natural functor as described in Section 6.12. Values of type *'a option* can be viewed as containers of a single value of type *'a*.

The corresponding BNF functions are generated as:

```
set_option :: 'p option ⇒ 'p set ≡
  λx. case x of None ⇒ {} | Some x' ⇒ {x'}
map_option :: ('p ⇒ 'q) ⇒ 'p option ⇒ 'q option ≡
  λf x. case x of None ⇒ None | Some x' ⇒ Some (f x')
pred_option :: ('p ⇒ bool) ⇒ 'p option ⇒ bool ≡
  λp x. case x of None ⇒ True | Some x' ⇒ p x'
rel_option :: ('p ⇒ 'q ⇒ bool) ⇒ 'p option ⇒ 'q option ⇒ bool ≡
  λr x y. case x of None ⇒ y = None | Some x' ⇒
    (case y of None ⇒ False | Some y' ⇒ r x' y')
```

Additionally there is the function

```
combine_options ::
  ('p ⇒ 'p ⇒ 'p) ⇒ 'p option ⇒ 'p option ⇒ 'p option ≡
  λf x y. case x of None ⇒ y | Some x' ⇒
    (case y of None ⇒ x | Some y' ⇒ Some (f x' y'))
```

which extends *map_option* to binary functions. It applies its first argument (a binary function) to two wrapped values without unwrapping them. If one argument is *None* the result is the other argument.

8.5.4 Rules

Algebraic Type Rules

The simplifier rules for the constructors of type *'a option* are the rule sets *option.inject*, *option.distinct* and *option.case* (see Section 7.1.5):

```
(Some ?x = Some ?y) = (?x = ?y)
None ≠ Some ?x
Some ?x ≠ None
(case None of None ⇒ ?t1 | Some x ⇒ ?t2 x) = ?t1
(case Some ?x of None ⇒ ?t1 | Some x ⇒ ?t2 x) = ?t2 ?x
```

The case, split, and induction rules are

```
option.exhaust:
  [[?y = None ⇒ ?P; ∧x. ?y = Some x ⇒ ?P]] ⇒ ?P
option.split:
  ?P (case ?t of None ⇒ ?t1 | Some x ⇒ ?t2 x) =
    ((?t = None → ?P ?t1) ∧ (∀x. ?t = Some x → ?P (?t2 x)))
option.induct:
  [[?P None; ∧x. ?P (Some x)]] ⇒ ?P ?a
```

For the case and induction rule the cases are named *None* and *Some*.

Rules About Functions

For the functions described in Section 8.5.3 most properties are specified by equations such as

```

elem_set: (?x ∈ set_option ?xo) = (?xo = Some ?x)
map_option_case:
  map_option ?f ?y = (case ?y of None ⇒ None | Some x ⇒ Some (?f x))
Option.is_none_def: Option.is_none ?x = (?x = None)
rel_option_unfold:
  rel_option ?R ?x ?y =
    (Option.is_none ?x = Option.is_none ?y ∧
     (¬ Option.is_none ?x →
      ¬ Option.is_none ?y → ?R (the ?x) (the ?y)))

```

Many of these equations are members of the simpset, therefore many properties about optional values can be proved by the simplifier. Remember to configure it with `split: option.split` if `case` terms occur for type `'a option`.

8.6 Tuples

Theories: *Product_Type, Basic_BNFs*

Tuples are represented by HOL as nested pairs. The type of pairs is specified equivalent to an algebraic type (see Section 7.1) of the form

```
datatype ('a, 'b) prod = Pair (fst: 'a) (snd: 'b)
```

As described in Section 7.1, this type is equivalent to the type of pairs of values of the type parameters `'a` and `'b`. It is also called the “product type” of the types `'a` and `'b`.

HOL supports an alternative syntax for instances of type `('a, 'b) prod`. The type instance `(t1, t2) prod` where `t1` and `t2` are arbitrary types may be denoted by the type expression `t1 × t2` or `t1 * t2` (see Section 6.6).

A tuple with more than two components is represented in HOL by a pair where the second component is again a pair or tuple. Hence the type of 4-tuples with component types `t1, t2, t3, t4` can be denoted by the type expression `t1 × (t2 × (t3 × t4))`. Since `×` is right associative the parentheses may be omitted as in the equivalent type expression `t1 × t2 × t3 × t4`.

As an example the type `(bool × nat × nat) option` is the type of optional triples of one boolean value and two natural numbers.

Note that the type *unit* (see Section 8.2) can be used to represent a “0-tuple”, as the alternative form *()* of its single value suggests.

8.6.1 Values

All values of type $(\text{'a}, \text{'b}) \text{ prod}$ are denoted using the single constructor *Pair*. HOL supports an alternative syntax for these constructor application terms: the term *Pair term₁ term₂* may also be specified in the form *(term₁, term₂)* (see Section 6.6).

Iterating this syntax a 4-tuple value may be specified as *(term₁, (term₂, (term₃, term₃)))*. Since the *(...,...)* is also right associative the inner parentheses may be omitted resulting in the equivalent term *(term₁, term₂, term₃, term₃)*.

Thus an example value of type $(\text{bool} \times \text{nat} \times \text{nat}) \text{ option}$ can be specified by the term *Some (True, 5, 42)*.

8.6.2 Destructors

The selectors *fst* and *snd* can be used to access the first or second component in a pair, respectively.

Note that in a tuple of more than two components *fst* still selects the first component, whereas *snd* selects the nested tuple with all other components. To access the third component in a 4-tuple *x* the selectors must be combined accordingly: *fst (snd (snd x))*. The third component in a triple *x* must be accessed by *snd (snd (snd x))*. Therefore there is no general way of accessing the *i*th component in an arbitrary tuple. An “easier” way for it is to use a *case* term.

A *case* term for type $t_1 \times t_2$ (which is equivalent to $(t_1, t_2) \text{ prod}$) has the form

case term of Pair x₁ x₂ \Rightarrow term₁

where *term* is a term of type $t_1 \times t_2$ and *x₁*, *x₂* are variables of type t_1 and t_2 , respectively, which may occur in *term₁*. Using the alternative syntax for pair values the *case* term may also be specified as

case term of (x₁, x₂) \Rightarrow term₁

Such *case* terms can be used to “take apart” a *term* for a pair where the components are not explicitly specified, such as in the application of a function which returns a pair. The components are bound to the variables *x₁*, *x₂* and can be used separately in *term₁*. An example is the case term

case coordinate of (x, y) \Rightarrow x + y

where *coordinate* is a single variable of type `nat × nat`.

As usual, this also works for tuples with more than two components. The general form of a *case* term for an *n*-tuple is

`case term of (x1, ..., xn) ⇒ term1`

where the variables *x*₁, ..., *x*_{*n*} may occur in *term*₁.

HOL provides an alternative form for such *case* terms for tuples:

`let (x1, ..., xn) = term in term1`

Although this form looks like a generalized *let* term (see Section 6.5) this does not imply that *let* terms support arbitrary patterns on the left side of the = sign, like the *let* statement (see Section 3.12.2). As “patterns” only nested tuples of variables may be used, such *let* terms are always equivalent to a *case* term for a tuple.

The same variable tuple patterns can also be used in other kinds of terms where variables are bound such as in lambda terms (e.g., $\lambda(a,b) \ c. \ a+b+c$), in description operators (e.g., *SOME* $(a,b). \ a+b=5$), (but not in logic quantifiers) and in set comprehensions (e.g., $\{(a,b). \ a=b*b\}$). Note that the last example is equivalent to $\{(a,b) \mid a \ b. \ a=b*b\}$, only in this form an arbitrary term may be used instead of (a,b) .

8.6.3 Functions

As described in Section 7.1.3 a *case* term for a pair is only an alternative syntax for the function application term *case_prod* $(\lambda \ x_1 \ x_2. \ term_1) \ term$. The polymorphic function *case_prod* is provided as

`case_prod :: ('a ⇒ 'b ⇒ 't) ⇒ 'a × 'b ⇒ 't ≡
λf (x,y). f x y`

where *'t* is the type of *term*₁ and thus of the whole *case* term.

This function is of interest on its own. It converts a binary function *f* specified in the style described in Section 2.2.4 to a function which takes the two arguments in the form of a single pair. This conversion is generally called “uncurrying” (see Section 6.6.1). Tuple patterns in variable bindings, as described in Section 8.6.2, are implemented in this way: The term $\lambda(a,b). \ term$ is just an alternative syntax for *case_prod* $(\lambda a \ b. \ term)$.

HOL provides the reverse conversion as the function

`curry :: ('a × 'b ⇒ 't) ⇒ 'a ⇒ 'b ⇒ 't ≡
λf x y. f (x,y)`

Additionally there are the functions

```

apfst :: ('a ⇒ 'c) ⇒ 'a × 'b ⇒ 'c × 'b ≡ λf (x,y). (f x,y)
apsnd :: ('b ⇒ 'c) ⇒ 'a × 'b ⇒ 'a × 'c ≡ λf (x,y). (x,f y)

```

which apply their first argument to the first or second component of the second argument, respectively, and the function

```

prod.swap :: 'a × 'b ⇒ 'b × 'a ≡ λ(x,y). (y,x)

```

which interchanges the components of its argument.

The function

```

Product_Type.Times :: 'a set ⇒ 'b set ⇒ ('a × 'b) set ≡
  λA B. {(x, y). x∈A ∧ y∈B}

```

with operator name (\times) constructs the cartesian product of two sets.

No orderings or lattice functions (see Section 6.2) are specified for values of type $('a, 'b)$ *prod*.

BNF Functions

The type constructor *prod* is a bounded natural functor as described in Section 6.12. Values of type $('a, 'b)$ *prod* can be viewed as containers of a single value of type *'a* and a single value of type *'b*.

The corresponding BNF functions are generated as:

```

map_prod ::
  ('a1 ⇒ 'a2) ⇒ ('b1 ⇒ 'b2) ⇒ ('a1 × 'b1) ⇒ ('a2 × 'b2) ≡
  λf g (x,y). (f x, g y)

```

```

Basic_BNFs.fsts :: ('a × 'b) ⇒ 'a set ≡ λ(x,y). {x}
Basic_BNFs.snds :: ('a × 'b) ⇒ 'b set ≡ λ(x,y). {y}

```

```

pred_prod ::
  ('a ⇒ bool) ⇒ ('b ⇒ bool) ⇒ ('a × 'b) ⇒ bool ≡
  λf g (x,y). f x ∧ g y
rel_prod ::
  ('a1 ⇒ 'a2 ⇒ bool) ⇒ ('b1 ⇒ 'b2 ⇒ bool)
  ⇒ ('a1 × 'b1) ⇒ ('a2 × 'b2) ⇒ bool ≡
  λf g (x1,y1) (x2,y2). f x1 x2 ∧ g y1 y2

```


8.6.4 Rules

Algebraic Type Rules

Since there is only one constructor there are no distinctness rules. The simplifier rules for the constructor of type $(\text{'a'}, \text{'b'}) \text{ prod}$ are the rule sets *prod.inject*, *prod.sel* and *prod.case* (see Section 7.1.5):

```
((?x1, ?x2) = (?y1, ?y2)) = (?x1 = ?y1 ∧ ?x2 = ?y2)
fst (?x1, ?x2) = ?x1
snd (?x1, ?x2) = ?x2
(case (?x1, ?x2) of (x, y) ⇒ ?f x y) = ?f ?x1 ?x2
```

The case, split, and induction rules are

```
prod.exhaust:
  (∧x1 x2. ?y = (x1, x2) ⇒ ?P) ⇒ ?P
prod.split:
  ?P (case ?t of (x, y) ⇒ ?f x y) =
  (∀x1 x2. ?t = (x1, x2) → ?P (?f x1 x2))
prod.induct:
  (∧x1 x2. ?P (x1, x2)) ⇒ ?P ?a
```

For the case and induction rule the single case is named *Pair*.

The case and induction rules are also provided for tuples of size 3 to 7 in the form

```
prod_cases3:
  (∧x1 x2 x3. ?y = (x1, x2, x3) ⇒ ?P) ⇒ ?P
prod_induct3:
  (∧x1 x2 x3. ?P (x1, x2, x3)) ⇒ ?P ?a
```

In all of them the single case is named *fields*.

Since all these case and induction rules are associated with the corresponding tuple types a proof of the form

```
proof (cases x)
case (fields x1 x2 x3)
...
show ?thesis <proof>
qed
```

replaces the variable x of type $t_1 \times t_2 \times t_3$ by tuple terms $(x1, x2, x3)$ and allows to reason about the components of x .

Rules About Functions

As described in Section 8.6.2 components of a tuple are easiest accessed by using a tuple pattern which is equivalent to applying the function *case_prod*

(see Section 8.6.3). In proofs it is often necessary to convert a proposition involving `case_prod` to a proposition for the components. The simplifier rule

```
split_def: case_prod = (λc p. c (fst p) (snd p))
```

allows to directly replace occurrences of `case_prod` by rewriting. It is not part of the simpset (see Section 4.6.3), its use by the simplifier must be explicitly configured.

If tuples are not specified with the help of tuple patterns but by single variables of a tuple type the case, split, and induction rules described above can be used to convert such variables to explicit tuples where variables denote the components. Alternatively HOL provides the rule

```
split_paired_all: (λx. ?P x) ≡ (λa b. ?P (a, b))
```

Rewriting by this rule replaces all bound variables of a type $t_1 \times t_2$ by two variables for the components. Since tuples are nested pairs an iterated rewriting does the same for arbitrary tuples. Again, this rule is not part of the simpset and it should be added with care, an iterated rewriting is best done by the method (`simp only: split_paired_all`) without combining it with other simplifier rules.

Instead, the rules

```
split_paired_All: (∀x. ?P x) = (∀a b. ?P (a, b))
split_paired_Ex: (∃x. ?P x) = (∃a b. ?P (a, b))
```

are in the simpset and are thus automatically applied by the simplifier to replace variables of tuple type bound by logic quantifiers (see Section 8.1.3). If that is not intended these rules must be deactivated in the form (`simp del: split_paired_Ex`).

8.7 Functions

Theories: *Fun, Orderings, Lattices, Complete_Lattices, Basic_BNFs, Nat*

The type `('a, 'b) fun` with alternative syntax `'a ⇒ 'b` (see Section 2.2.3) is not specific for HOL, it is provided by Isabelle at the basis of the whole Isabelle type system. It is polymorphic and denotes the type of functions with argument type `'a` and result type `'b`.

However, HOL complements this basic support for functions by many functions and rules for the type `('a, 'b) fun`.

Note that functions introduced by type definitions such as constructors and destructors of algebraic types (see Section 7.1) or the morphisms of a subtype (see Section 7.2) are normal functions of type `('a, 'b) fun` and all mechanisms described here for functions apply to them as well.

The type $(\text{'a}, \text{'b}) \text{fun}$ is not equivalent to an algebraic type. Since functions are always total, the arguments of a constructor function would have to specify function values for all arguments of type 'a which may be infinitely many.

8.7.1 Values

The values of type $\text{'a} \Rightarrow \text{'b}$ are denoted by lambda terms (see Section 2.2.5). HOL introduces the name *id* for the polymorphic identity function $\lambda x. x$.

8.7.2 Functions on Functions

The functions *image* and *vimage* have already been described in Section 8.4.3, as mappers for *set* they can be viewed to “lift” functions on values to functions on value sets.

Functions of arbitrary types can be composed by the polymorphic function

```
comp :: ('b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c
      ≡ λf g x. f (g x)
```

if the “intermediate” type 'b matches. The operator name for infix notation is (\circ) .

There is also the variant *fcomp* with reversed arguments where the left argument is applied first and the function

```
scomp :: ('a ⇒ 'b × 'c) ⇒ ('b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'd
       ≡ λf g x. (case_prod g) (f x)
```

which composes a binary function with a function with pairs as values, using the uncurry function *case_prod* to convert the binary function to a function with argument pairs (see Section 8.6.3). The operator names $(\circ>)$ for *fcomp* and $(\circ\rightarrow)$ for *scomp* are only available after using the command

```
unbundle state_combinator_syntax
```

on theory level.

Finite iteration of a function of type $\text{'a} \Rightarrow \text{'a}$ can be specified by the polymorphic function

```
compow :: nat ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a
```

with operator name (\sim) (with reversed arguments) for infix notation. Thus $\text{compow } 3 \ f = f \sim 3 = f \circ f \circ f$.

Function Updates

HOL provides the function

$$\begin{aligned} \text{fun_upd} &:: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b) \\ &\equiv \lambda f \ a \ b \ x. \text{ if } x = a \text{ then } b \text{ else } f \ x \end{aligned}$$

which returns a function where the value of a single argument a of f has been changed to b . Note that this “function update” does not “change” the function f , it returns a new function which differs from f only for the argument a .

HOL provides the alternative syntax

$$f(\text{terma}_1 := \text{termb}_1, \dots, \text{terma}_n := \text{termb}_n)$$

for the term $\text{fun_upd} \dots (\text{fun_upd } f \text{ terma}_1 \text{ termb}_1) \dots \text{terma}_n \text{ termb}_n$. The changes are applied from left to right, i.e. $f(x := y, x := z) = f(x := z)$.

HOL also provides an update on a set of arguments, where the new values are specified by another function:

$$\begin{aligned} \text{override_on} &:: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'a \Rightarrow 'b \\ &\equiv \lambda f \ g \ A \ x. \text{ if } x \in A \text{ then } g \ x \text{ else } f \ x \end{aligned}$$

BNF Functions

As described in Sections 6.11.2 and 6.12.1 the type constructor fun is a bounded natural functor with one dead and one live type parameter. The mapper is the function

$$\begin{aligned} \text{map_fun} &:: ('b_1 \Rightarrow 'a_1) \Rightarrow ('a_2 \Rightarrow 'b_2) \Rightarrow ('a_1 \Rightarrow 'a_2) \Rightarrow ('b_1 \Rightarrow 'b_2) \\ &\equiv \lambda f_1 \ f_2. \lambda f. f_2 \circ f \circ f_1 \end{aligned}$$

and the set-function for the live type parameter is the function

$$\text{range} :: ('a \Rightarrow 'b) \Rightarrow 'b \text{ set} \equiv \lambda f. \text{ image } f \text{ UNIV}$$

which returns the set of all result values of a function (note that the set of all argument values of a function is always UNIV because functions are total in Isabelle).

There are the predicator and relator (see Section 6.12.2)

$$\begin{aligned} \text{pred_fun} &:: ('p_1 \Rightarrow \text{bool}) \Rightarrow ('p_2 \Rightarrow \text{bool}) \Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow \text{bool} \\ &\equiv \lambda p_1 \ p_2 \ f. \forall x. p_1 \ x \longrightarrow p_2 \ (f \ x) \\ \text{rel_fun} &:: ('p_1 \Rightarrow 'q_1 \Rightarrow \text{bool}) \Rightarrow ('p_2 \Rightarrow 'q_2 \Rightarrow \text{bool}) \\ &\Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow ('q_1 \Rightarrow 'q_2) \Rightarrow \text{bool} \\ &\equiv \lambda r_1 \ r_2 \ f \ g. \forall x \ y. r_1 \ x \ y \longrightarrow r_2 \ (f \ x) \ (g \ y) \end{aligned}$$

which include the dead type parameter. The predicator and relator for the live parameter can be obtained by partial application $\text{pred_fun } (\lambda_ . \text{True}) = (\lambda p_2 f. \forall x. p_2 (f x))$ and $\text{rel_fun } (=) = (\lambda r_2 f g. \forall x. r_2 (f x) (g x))$. For these functions HOL does not define separate names. They lift a predicate or relation by applying it to all function values.

Injectivity and Surjectivity

For the basic function properties of injectivity, surjectivity, and bijectivity HOL provides the predicates

```
inj :: ('a ⇒ 'b) ⇒ bool ≡ λf. ∀x y. f x = f y ⟶ x = y
surj :: ('a ⇒ 'b) ⇒ bool ≡ λf. range f = UNIV
bij :: ('a ⇒ 'b) ⇒ bool ≡ λf. inj f ∧ surj f
```

and also the domain-restricted forms

```
inj_on :: ('a ⇒ 'b) ⇒ 'a set ⇒ bool
  ≡ λf A. ∀x∈A. ∀y∈A. f x = f y ⟶ x = y
bij_betw :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set ⇒ bool
  ≡ λf A B. inj_on f A ∧ image f A = B
```

Injective functions can be inverted on their range. HOL provides the more general inversion function

```
the_inv :: ('a ⇒ 'b) ⇒ ('b ⇒ 'a)
  ≡ λf y. THE x. f x = y
```

which returns the argument mapped by function f to the value y . Note the use of the definite description operator *THE* (see Section 6.3.2). If there is no such argument (because y is not in the range of f) or if it is not unique (because f is not injective), it causes the function to be underspecified. Thus the partial application $(\text{the_inv } f)$ is the inverse of an arbitrary function f . It is total but only specified for values of type $'b$ where the inversion exists and is unique. It is only fully specified if $\text{bij } f$.

Additionally there is the domain-restricted form

```
the_inv_into :: 'a set ⇒ ('a ⇒ 'b) ⇒ ('b ⇒ 'a)
  ≡ λA f y. THE x. x ∈ A ∧ f x = y
```

where the partial application $(\text{the_inv_into } A f)$ is the inverse of f restricted to arguments in set A . It is only fully specified if $\text{image } f A = \text{UNIV}$ and $\text{inj_on } f A$.

Functions for Orderings and Lattices

The ordering relations (\leq) , (\geq) (see Section 6.2.2) are defined for functions by lifting the ordering relations for the function values. Thus the ordering (\leq) on functions is equivalent to $rel_fun\ (=)\ (\leq)$ and analogously for (\geq) . In other words, $f \leq g$ holds if $\forall x. (f\ x) \leq (g\ x)$. Note that even if (\leq) is a total ordering on the function values, the lifted ordering is partial, because for some arguments the function values may be less or equal and for other arguments not.

The strict ordering relations $(<)$, $(>)$ on functions, instead, are not lifted, they are derived from (\leq) , (\geq) according to $(f < g) = (f \leq g) \wedge \neg (g \leq f)$.

In a similar way the lattice operations (\sqcap) , (\sqcup) , (\bigsqcap) , and (\bigsqcup) (see Section 6.2.3) are lifted from the function values to functions:

$$\begin{aligned} f \sqcap g &\equiv \lambda x. f\ x \sqcap g\ x \\ f \sqcup g &\equiv \lambda x. f\ x \sqcup g\ x \\ \bigsqcap A &\equiv \lambda x. \bigsqcap_{f \in A.} f\ x \\ \bigsqcup A &\equiv \lambda x. \bigsqcup_{f \in A.} f\ x \end{aligned}$$

Since $(<)$ is not a total ordering on functions the functions *min* and *max* are *not* equivalent to (\sqcap) and (\sqcup) and the functions *Min* and *Max* are not available for sets of functions.

Since orderings and lattice operations are defined for type *bool* (see Section 8.1.3) so that (\leq) is equivalent to the implication (\longrightarrow) , the lifting works specifically for predicates and relations (see Section 6.1). Predicates are ordered by the “stronger as” relation where $(p_1 \leq p_2) \longleftrightarrow (\forall x. p_1\ x \longrightarrow p_2\ x)$ and the lattice operations correspond to point-wise conjunction or disjunction, such as $(p_1 \sqcap p_2) = (\lambda x. p_1\ x \wedge p_2\ x)$. As described above for functions with multiple arguments the boolean operations can be lifted in the same way to binary or n-ary relations. Binary relations are ordered by the “stronger as” relation where $(r_1 \leq r_2) \longleftrightarrow (\forall x\ y. r_1\ x\ y \longrightarrow r_2\ x\ y)$ and the lattice operations are as above, such as $(r_1 \sqcap r_2) = (\lambda x\ y. r_1\ x\ y \wedge r_2\ x\ y)$.

Monotonicity

Another way to derive predicates on functions is by applying *rel_fun* to compare a function to itself. HOL supports this by the function

$$\begin{aligned} monotone &:: ('p_1 \Rightarrow 'p_1 \Rightarrow bool) \Rightarrow ('p_2 \Rightarrow 'p_2 \Rightarrow bool) \\ &\Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow bool \\ &\equiv \lambda r_1\ r_2\ f. rel_fun\ r_1\ r_2\ f\ f \end{aligned}$$

The partial application *monotone* $r_1\ r_2$ is the predicate which tests a function whether arguments related by r_1 produce results related by r_2 :

$\text{monotone } r_1 \ r_2 = (\lambda f. \forall x \ y. \ r_1 \ x \ y \longrightarrow r_2 \ (f \ x) \ (f \ y))$

The usual notion of monotonicity results when *monotone* is applied to the ordering relations. HOL defines the specific predicates on functions

$\text{mono} :: ('p_1 \Rightarrow 'p_2) \Rightarrow \text{bool} \equiv \text{monotone } (\leq) (\leq)$
 $\text{strict_mono} :: ('p_1 \Rightarrow 'p_2) \Rightarrow \text{bool} \equiv \text{monotone } (<) (<)$
 $\text{antimono} :: ('p_1 \Rightarrow 'p_2) \Rightarrow \text{bool} \equiv \text{monotone } (\leq) (\geq)$

HOL also provides the domain-restricted variants:

$\text{monotone_on} :: 'p_1 \ \text{set} \Rightarrow ('p_1 \Rightarrow 'p_1 \Rightarrow \text{bool}) \Rightarrow ('p_2 \Rightarrow 'p_2 \Rightarrow \text{bool})$
 $\quad \Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow \text{bool}$
 $\text{mono_on} :: 'p_1 \ \text{set} \Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow \text{bool}$
 $\text{strict_mono_on} :: 'p_1 \ \text{set} \Rightarrow ('p_1 \Rightarrow 'p_2) \Rightarrow \text{bool}$

8.7.3 Rules

Since functions are the basic type of Isabelle there are several rules which are already built in and need not be provided by HOL. Some of them are implicitly applied to propositions without the need of using proof methods, such as the rewriting rule

$(\lambda x. \ ?P \ x) \ ?y = ?P \ ?y$

Many others are known by the automatic proof methods (see Section 4.7.1) so that properties relating to functions can usually be proved by them.

The most basic rules provided by HOL about functions are

$\text{ext}: (\bigwedge x. \ ?f \ x = ?g \ x) \Longrightarrow ?f = ?g$
 $\text{fun_cong}: ?f = ?g \Longrightarrow ?f \ ?x = ?g \ ?x$
 $\text{arg_cong}: ?x = ?y \Longrightarrow ?f \ ?x = ?f \ ?y$
 $\text{cong}: \llbracket ?f = ?g; \ ?x = ?y \rrbracket \Longrightarrow ?f \ ?x = ?g \ ?y$

Rules About Functions on Functions

Reasoning about the functions described in Section 8.7.2 can often be done by substituting their definition and then using automatic proof methods. Alternatively HOL provides many rules for direct reasoning about these functions, such as

$\text{comp_assoc}: ?f \circ ?g \circ ?h = ?f \circ (?g \circ ?h)$
 $\text{image_comp}: ?f \ ' \ ?g \ ' \ ?r = (?f \circ ?g) \ ' \ ?r$

and in particular introduction rules (see Section 4.3.4) such as

$\text{injI}: (\bigwedge x \ y. \ ?f \ x = ?f \ y \Longrightarrow x = y) \Longrightarrow \text{inj } ?f$
 $\text{surjI}: (\bigwedge x. \ ?g \ (?f \ x) = x) \Longrightarrow \text{surj } ?g$

$\text{bijI}: \llbracket \text{inj } ?f; \text{surj } ?f \rrbracket \implies \text{bij } ?f$
 $\text{monoI}: (\bigwedge x y. x \leq y \implies ?f x \leq ?f y) \implies \text{mono } ?f$
 the destruction (see Section 4.4.5) and elimination (see Section 5.4.2) rules
 $\text{comp_eq_dest}: ?a \circ ?b = ?c \circ ?d \implies ?a (?b ?v) = ?c (?d ?v)$
 $\text{injD}: \llbracket \text{inj } ?f; ?f ?x = ?f ?y \rrbracket \implies ?x = ?y$
 $\text{monoD}: \llbracket \text{mono } ?f; ?x \leq ?y \rrbracket \implies ?f ?x \leq ?f ?y$
 $\text{comp_eq_elim}:$
 $\llbracket ?a \circ ?b = ?c \circ ?d; (\bigwedge v. ?a (?b v) = ?c (?d v)) \rrbracket \implies ?R \rrbracket \implies ?R$
 $\text{surjE}: \llbracket \text{surj } ?f; \bigwedge x. ?y = ?f x \implies ?C \rrbracket \implies ?C$
 $\text{monoE}: \llbracket \text{mono } ?f; ?x \leq ?y; ?f ?x \leq ?f ?y \implies ?thesis \rrbracket \implies ?thesis$
 and the simplifier equations (see Section 4.6.2)
 $\text{comp_apply}: (?f \circ ?g) ?x = ?f (?g ?x)$
 $\text{fun_upd_apply}:$
 $(?f(?x := ?y)) ?z = (\text{if } ?z = ?x \text{ then } ?y \text{ else } ?f ?z)$
 $\text{bij_id}: \text{bij id}$
 $\text{override_on_apply_in}:$
 $?a \in ?A \implies \text{override_on } ?f ?g ?A ?a = ?g ?a$

8.8 Relations

****todo****

8.8.1 Functions

8.9 The Sum Type

****todo****

Bibliography

- [1] J. Biendarra, J. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel. *Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL*. <https://isabelle.in.tum.de/doc/datatypes.pdf>.
- [2] J. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. *Defining Nonprimitively (Co)recursive Functions in Isabelle/HOL*. <https://isabelle.in.tum.de/doc/corec.pdf>.
- [3] B. Fürer, A. Lochbihler, J. Schneider, and D. Traytel. Quotients of bounded natural functors. *Logical Methods in Computer Science*, Volume 18, Issue 1, Feb 2022.
- [4] D. Matichuk, M. Wenzel, and T. Murray. *The Eisbach User Manual*. <https://isabelle.in.tum.de/doc/eisbach.pdf>.
- [5] T. Nipkow. *Programming and Proving in Isabelle/HOL*. <https://isabelle.in.tum.de/doc/prog-prove.pdf>.
- [6] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL A Proof Assistant for Higher-Order Logic*. <https://isabelle.in.tum.de/doc/tutorial.pdf>.
- [7] M. Wenzel. *The Isabelle/Isar Reference Manual*. <https://isabelle.in.tum.de/doc/isar-ref.pdf>.
- [8] M. Wenzel. *Isabelle/jEdit*. <https://isabelle.in.tum.de/doc/jedit.pdf>.

Index

- () (constant), 237, 253
- (method), 70
- b (option), 10
- c (option), 10
- d (option), 10
- l (option), 10
- n (option), 10
- v (option), 11
- . (keyword), 80
- .. (keyword), 76
- ... (field name), 221
- ... (term abbreviation), 68
- &&&** (meta-logic operator), 26
- \wedge (meta-logic binder), 26
- \neq (operator), 120
- \sqcap (operator), 121
- \cap (operator), 244
- \sqcup (operator), 121
- \cup (operator), 244
- \forall (binder), 234
- \wedge (operator), 233
- $/$ (inner keyword), 243
- $/$ (keyword), 139, 188
- \perp (operator), 121
- \circ (operator), 151, 258
- \sim (operator), 258
- $/$ (operator), 239
- $\{ \}$ (constant), 243
- \exists (binder), 234
- $\exists !$ (binder), 234
- $\circ >$ (operator), 258
- \longleftrightarrow (operator), 233
- ‘ (operator), 246
- \longrightarrow (operator), 233
- \sqcap (operator), 121
- \cap (operator), 122, 244
- λ (binder), 20
- \dashrightarrow (operator), 182
- \in (operator), 243
- $-$ (operator), 239, 244
- \neg (operator), 233
- \notin (operator), 244
- \vee (operator), 233
- $< \circ$ (operator), 160
- $+$ (operator), 239
- \implies (operator), 165
- $\circ \rightarrow$ (operator), 258
- \subset (operator), 244
- \subseteq (operator), 121, 244
- \sqcup (operator), 121
- \supset (operator), 244
- \supseteq (operator), 244
- $*$ (operator), 239, 252
- \top (operator), 121
- \cup (operator), 244
- $\exists_{\leq 1}$ (binder), 234
- $- \text{‘}$ (operator), 246
- \times (operator), 125, 252, 255
- $:=$ (inner keyword), 259
- $<$ (operator), 120
- \longleftrightarrow (operator), 120
- \leq (operator), 120
- $=$ (operator), 22, 120
- \equiv (meta-logic operator), 22
- \rightleftharpoons (syntactic operator), 24
- \implies (meta-logic operator), 25
- \Rightarrow (operator), 19, 257
- $>$ (operator), 120

\geq (operator), 120
 ?case (term abbreviation), 94, 96, 111
 ?thesis (term abbreviation), 49, 62, 100
 abbreviation
 for constant, 23
 for function, 23
 for term, 35, 62
 abbreviation (keyword), 23
 Abs_ (constant name prefix), 206
 abs_ (constant name prefix), 172
 abstract type, 172
 abstraction
 function, 172
 acc (constant), 136
 acc_induct_rule (fact name), 136
 accp (constant), 136
 accp_induct_rule (fact name), 136
 add_Suc_right (fact name), 242
 add: (method argument), 88
 adhoc_overloading (keyword), 24
 algebraic type, 188
 All (constant), 234
 alle (fact name), 236
 allI (fact name), 236
 also (keyword), 68
 and (keyword), 27
 antimonio (constant), 262
 apfst (constant), 255
 apply (keyword), 47
 apsnd (constant), 255
 arbitrary: (method argument), 116
 arg_cong (fact name), 262
 arith (method), 92
 assms (assumption name), 37
 assume (keyword), 55
 assumes (keyword), 32
 assumption, 25
 group, 27, 37
 name, 37, 94, 113
 assumption (method), 71, 73
 atomize_all (fact name), 236
 atomize_conj (fact name), 237
 atomize_conjL (fact name), 237
 atomize_elim (method), 137, 142
 atomize_imp (fact name), 237
 auto (method), 92
 backward reasoning, 38, 73
 bd: (keyword), 161
 bi_total (constant), 119
 bi_unique (constant), 119
 bij (constant), 260
 bij_betw (constant), 260
 bij_id (fact name), 263
 bijI (fact name), 263
 binder, 20, 26, 234
 binder syntax, 20, 122
 body in \sim , 20
 blast (method), 91, 235
 block structure, 45
 BNF, 154
 bnf (keyword), 161
 body in binder syntax, 20
 bool (type), 17, 24, 118, 233
 bot (constant), 121
 bounded natural functor, 154, 159
 build (isabelle tool), 10
 by (keyword), 43
 calculation (fact set name), 65
 card (constant), 247
 card_mono (fact name), 249
 card_Diff_subset_Int (fact name), 249
 card_of (constant), 160
 card_order (constant), 160
 card_suc (constant), 160
 card_Un_Int (fact name), 249
 cardinal_syntax (bundle), 160
 cardinality, 160, 247
 cartesian product, 255
 case, 93, 95, 100
 combinator function, 195

- name, 94, 96, 102
- rule, 98
 - associated \sim , 101
- split, 97, 99, 104, 109
- term, 194
- .case (fact name suffix), 198
- case (inner keyword), 194
- case (keyword), 94
- case name, 100
- case_ (constant name prefix), 195
- case_prod (constant), 254
- case_split (fact name), 100, 235
- _cases (fact name suffix), 208, 217
- .cases (fact name suffix), 129, 134, 142, 227
- cases (method), 99, 107
- cases rule, 129
- .cases_scheme (fact name suffix), 227
- choice operator, 122
- c infinite (constant), 160
- clarify (method), 91
- clarsimp (method), 92
- Collect (constant), 242
- Collect_eqI (fact name), 247
- Collect_inj (fact name), 247
- CollectD (fact name), 248
- CollectI (fact name), 248
- combine_options (constant), 251
- comma, 125
- comments, 15
- comp (constant), 258
- comp (fact name), 151
- comp_apply (fact name), 263
- comp_assoc (fact name), 262
- comp_eq_dest (fact name), 263
- comp_eq_elim (fact name), 263
- compow (constant), 258
- conclusion, 25
 - group, 27, 31
- conditional definition, 28
- cong (fact name), 262
- conj (constant), 233
- conjE (fact name), 236
- conjI (fact name), 236
- conjunct1 (fact name), 236
- conjunct2 (fact name), 236
- conjunction
 - meta \sim , 26, 37
- consider (keyword), 103
- constant, 18
 - abbreviation, 23
 - definition, 22
 - name, 18
 - polymorphic \sim , 18
 - search \sim , 21
 - underspecified \sim , 18, 22
- constructor, 190
 - free \sim , 191
 - function, 190
 - pattern, 202
 - linear \sim , 202
 - primitive linear \sim , 204
 - specification, 188
- consts (keyword), 18
- consumes (attribute), 108, 115
- container type, 154
- contains (constant), 242
- contrapos_nn (fact name), 236
- contrapos_np (fact name), 236
- contrapos_pn (fact name), 236
- contrapos_pp (fact name), 236
- conversep (constant), 119
- coord (example type), 189
- coordx (example type), 190
- copy_bnf (keyword), 187
- corollary (keyword), 28
- cpyfun (example type), 207
- curried function, 125
- curry (constant), 254
- datatype (keyword), 188
- dec_induct (fact name), 241
- _def (fact name suffix), 33, 60
- default selector, 193
- define (keyword), 59
- definite description operator, 123

- definition
 - algebraic type \sim , 188
 - as fact, 33
 - conditional \sim , 28
 - datatype \sim , 188
 - for constant, 22
 - for function, 22
 - for variable, 59
 - inductive \sim , 126, 143
 - mutual recursive \sim , 149
 - name, 33
 - primitive recursive \sim , 204
 - quotient type \sim , 213
 - record type \sim , 220
 - record type extension \sim , 222
 - recursive \sim , 138, 202
 - subtype \sim , 205
- definition (keyword), 22
- .defs (fact name suffix), 227
- del: (method argument), 88
- derivation rule, 25
- description operator, 122, 254
- dest (fact set name), 83
- destructor, 192
- destructor function, 192
- diff_induct (fact name), 241
- diff_Suc_1 (fact name), 242
- Dim2 (example constant), 189, 190
- Dim3 (example constant), 189, 190
- discriminator, 194
- discriminator function, 194
- disj (constant), 233
- disjE (fact name), 236
- disjI1 (fact name), 236
- disjI2 (fact name), 236
- disjnt (constant), 244
- .distinct (fact name suffix), 198
- div (operator), 239
- divide (constant), 239
- divides (example constant), 129
- documentation
 - panel, 13
- _dom (constant name suffix), 145
- domain predicate, 145
- Domainp (constant), 119
- drule (method), 82
- dvd_diff_nat (fact name), 242
- dynamic fact set, 32
- elem_set (fact name), 252
- elim (fact set name), 106
- elim (method), 106
- elimexp (example fact), 106, 108, 138
- elimination, 104, 115
- .elims (fact name suffix), 148
- else (inner keyword), 233
- empty tuple, 237, 253
- enumeration type, 191
- eq_onp (constant), 120
- eq_transfer (fact name), 172
- eqpair (example type), 207
- equality, 120
 - meta \sim , 22
- equational reasoning, 67
- equivp (constant), 213
- erule (method), 104
- evn (example constant), 126, 127, 134, 143
- Ex (constant), 234
- Ex1 (constant), 234
- example1 (example fact), 31
- example2 (example fact), 31
- examples (example fact set), 31
- exE (fact name), 236
- .exhaust (fact name suffix), 199
- exI (fact name), 236
- _ext (type name suffix), 229
- ext (fact name), 262
- extend (constant), 223
- fac (example constant), 139, 140
- fac2 (example constant), 140
- fac3 (example constant), 204, 205
- fact, 24, 28
 - accumulation, 65

- assumed \sim , 55, 57
- chaining, 52, 103
- collection, 31
- current \sim , 38, 52, 65
- display, 33
- export, 49, 56, 58, 60, 62
- external \sim , 41
- group, 54
- input \sim , 50
- local \sim , 35
- name, 31
- search \sim , 34
- set, 32, 65
- fact (method), 77
- fake proof, 44, 49
- False (case name), 100
- False (constant), 18, 233
- fastforce (method), 92
- fcomp (constant), 258
- fields (case name), 228, 256
- fields (constant), 224
- finally (keyword), 68
- find_consts (keyword), 21
- find_theorems (keyword), 34
- finite (constant), 247
- finite_Int (fact name), 248
- finite_subset (fact name), 248
- finite_Un (fact name), 248
- fix (keyword), 58
- fixed
 - argument of inductive
 - definition, 129
- fixed variable, 35, 57
 - name, 94
- fixes (keyword), 32
- fold (constant), 247
- for (keyword), 27, 129, 186, 187, 197
- force (method), 92
- formula, 25
- forward reasoning, 38, 76
- free occurrence, 20, 25, 29, 37, 57
- from (keyword), 52
- frule (method), 81
- fst (constant), 252
- fsts (constant), 255
- fun (keyword), 203
- fun (type), 19, 152, 257
- fun_cong (fact name), 262
- fun_upd (constant), 259
- fun_upd_apply (fact name), 263
- function, 19
 - abbreviation, 23
 - abstraction \sim , 172
 - application, 20
 - partial \sim , 20
 - argument, 19
 - optional \sim , 249
 - constant \sim , 20
 - constructor \sim , 190
 - curried \sim , 125
 - definition, 22
 - destructor \sim , 192
 - discriminator \sim , 194
 - identity \sim , 151, 258
 - mapper \sim , 151
 - measure \sim , 137, 147
 - multiple arguments, 19
 - n-ary \sim , 125
 - parametrically polymorphic
 - \sim , 171
 - predicator \sim , 157
 - primitive recursive \sim , 204
 - record update \sim , 225
 - recursive \sim , 138
 - relator \sim , 157
 - representation \sim , 174
 - respects a relation, 181
 - selector \sim , 192
 - set-, 154
 - type, 19
 - uncurried \sim , 125
 - underspecified \sim , 19
 - update \sim , 259
- function (keyword), 139
- functor, 150
 - bounded natural \sim , 154, 159
 - composability of \sim , 153

- multivariate \sim , 152
 - natural \sim , 154
- functor (keyword), 153
- goal
 - current \sim , 38
 - state, 12, 35, 36, 39, 49
 - statement, 48, 49, 61, 103
- goal_cases (method), 95
- greater (constant), 120
- greater_eq (constant), 120
- GREATEST (binder), 123
- greatest value operator, 123
- have (keyword), 48
- heap file, 9
- HOL (object logic), 16, 117, 188, 232
- HOL (session), 9, 10, 117
- .hyps (fact name suffix), 113
- id (constant), 151, 258
- id (fact name), 151
- identifier
 - definition, 12
- identity function, 151, 258
- if (inner keyword), 233
- if (keyword), 27
- if_split (fact name), 235
- if-term, 233, 235
- iff (constant), 120, 233
- iffD1 (fact name), 236
- iffD2 (fact name), 236
- iffE (fact name), 236
- iffI (fact name), 236
- .iffs (fact name suffix), 226
- .IH (fact name suffix), 113
- image (constant), 151, 246
- image_comp (fact name), 262
- impE (fact name), 236
- impI (fact name), 236
- implication
 - meta \sim , 25
- implies (constant), 233
- imports (keyword), 16
- in (inner keyword), 124, 254
- in_rel (fact name), 158
- inc_induct (fact name), 241
- _induct (fact name suffix), 209, 217
- .induct (fact name suffix), 132, 134, 148, 200, 228
- induct (method), 113
- induction, 108
 - base case, 108
 - hypothesis, 109
 - rule, 109, 110, 132, 200
 - step, 109, 110
 - transfinite \sim , 135
- induction (method), 111
- inductive (keyword), 126
- inductive definition, 126, 143
 - fixed argument of \sim , 129
 - mutually \sim , 134
 - single-step \sim , 133
- .inducts (fact name suffix), 134
- Inf (constant), 121
- inf (constant), 121
- infinite_descent0 (fact name), 241
- infix notation, 20
- inheritance, 220
- inj (constant), 260
- inj_on (constant), 260
- injD (fact name), 263
- _inject (fact name suffix), 208, 217
- .inject (fact name suffix), 198
- injI (fact name), 262
- input (example fact set), 54
- input facts, 50
- insert (constant), 244
- Int_def (fact name), 248
- IntD1 (fact name), 248
- Inter (constant), 244
- inter (constant), 244
- internal fact set, 32
- IntI (fact name), 248
- intro (fact set name), 74

- intro (method), 75
- .intros (fact name suffix), 127, 134
- _inverse (fact name suffix), 208, 216
- is (keyword), 65, 181
- is_none (constant), 250
- is_none_def (fact name), 252
- is_singleton (constant), 244
- Isar language, 43
- jedit (isabelle tool), 10
- lambda term, 20, 254, 258
- lattice_syntax (bundle), 122
- le_SucI (fact name), 241
- Leaf (example constant), 189
- LEAST (binder), 123, 245
- least value operator, 123
- left_total (constant), 119
- left_unique (constant), 119
- lemma (keyword), 28
- less (constant), 120
- less_eq (constant), 120
- less_Suc_induct (fact name), 241
- less_SucE (fact name), 242
- less_SucI (fact name), 241
- Let (constant), 124
- let (inner keyword), 124, 254
- let (keyword), 62, 124
- let term, 124, 254
- level
 - meta-, 16
 - object-, 16
- lexicographic_order (method), 148
- lift_bnf (keyword), 185
- lift_definition (keyword), 181
- lifting, 151
- lifting_syntax (bundle), 165, 182
- linarith (method), 92
- line breaks, 15
- linear constructor pattern, 202
 - primitive \sim , 204
- logic
 - meta-, 15
 - object \sim , 15, 117, 233
- Main (theory), 9, 117
- major premise, 81, 105, 115
- .make (fact name suffix), 227
- make (constant), 223
- map_cong0 (fact name), 154, 155
- map_fun (constant), 152, 259
- map_option_case (fact name), 252
- map_prod (constant), 152, 255
- map: (keyword), 161
- mapper, 151
 - contravariant \sim , 151
 - covariant \sim , 151
- mapper function, 151
- Max (constant), 121
- max (constant), 121
- max_Suc_Suc (fact name), 242
- measure (constant), 137
- measure function, 137, 147
- mem_Collect_eq (fact name), 248
- member (constant), 243
- meta-level, 16
- meta-logic, 15
- method
 - application, 41, 47
 - argument, 47
 - composed \sim , 83, 96
 - empty \sim , 70
 - expression, 83
 - initial \sim , 47, 50
 - name, 47
- metis (method), 92
- Min (constant), 121
- min (constant), 121
- min_Suc_Suc (fact name), 242
- minus (constant), 239, 244
- MkRecrd (example constant), 192
- mkroot (isabelle tool), 9
- mod (operator), 239
- modulo (constant), 239

- modus ponens, 236
- mono (constant), 262
- mono_on (constant), 262
- monoD (fact name), 263
- monoE (fact name), 263
- monoI (fact name), 263
- monotone (constant), 261
- monotone_on (constant), 262
- more part, 221
- moreover (keyword), 66
- morphisms
 - of a quotient, 174
- morphisms (keyword), 206
- mp (fact name), 236
- mult_Suc_right (fact name), 242
- multivariate functor, 152
- mutual recursive definition, 149
- mutually inductive definition, 134
- mycaserule (example fact), 99, 138
- myset (example constant), 19
- mystery (example constant), 19
- mystery2 (example constant), 22
- name
 - for assumption, 37, 94, 113
 - for case, 94, 96, 100, 102
 - for constant, 18
 - for fact, 31
 - for fact collection, 31
 - for fixed variable, 94
 - for method, 47
 - for term, 62
 - for type, 17
 - for variable, 19
 - of defining equation, 33, 60
 - shadowed \sim , 45
- named context, 93, 100, 111
- nat (type), 17, 238
- nat_induct_at_least (fact name), 241
- nat_induct_non_zero (fact name), 241
- nat_less_induct (fact name), 241
- nat.pred (constant), 239
- natLeq (constant), 160
- natural functor, 154
- natural numbers, 238
- next (keyword), 45
- None (constant), 249
- Not (constant), 233
- not_member (constant), 244
- notE (fact name), 236
- note (keyword), 51
- notI (fact name), 236
- nototal (example constant), 141
- nounique (example constant), 143
- number literal, 18, 238
- object logic, 15, 117, 233
- object-level, 16
- obtain (keyword), 60, 99
- obtains (keyword), 102, 107
- odd (example constant), 134
- of (inner keyword), 194
- One (example constant), 191
- only: (method argument), 88
- oops (keyword), 44
- operator name, 20
- operator symbol, 20
- option (type), 249
- ordLess2 (constant), 160
- overloading, 23, 33
 - ad-hoc \sim , 24
- overloading (keyword), 24
- override_on (constant), 259
- override_on_apply_in (fact name), 263
- pair, 125, 252
- Pair (constant), 252
- pairwise (constant), 244
- panel, 11
 - documentation \sim , 13
 - output \sim , 12, 34, 36, 91, 95, 163
 - query \sim , 13, 21, 34, 36, 75, 90, 95, 127, 145, 149,

- 154, 161, 197, 207, 216
- sidekick \sim , 12
- state \sim , 12, 36
- symbols \sim , 12
- theories \sim , 13
- parameterized
 - correspondence relation, 175
 - transfer relation, 175
 - type, 17
- parametric (keyword), 178
- parametrically polymorphic
 - function, 171
- parametricity rule, 171
- parent type, 222
- parentheses, 18, 25, 84, 125
- part_equivp (constant), 214
- partial: (keyword), 214
- pat_completeness (method), 202
- .pelims (fact name suffix), 146
- .pinduct (fact name suffix), 146
- plus (constant), 239
- polymorphic, 17, 18
- Pow (constant), 244
- Pow_def (fact name), 248
- PowD (fact name), 248
- PowI (fact name), 248
- Powp (constant), 244
- pred_fun (constant), 159, 260
- pred_prod (constant), 157, 255
- pred_set (fact name), 157
- pred: (keyword), 162
- predicate, 117
 - binary \sim , 118
 - domain \sim , 145
 - evenness \sim , 126
 - inductive definition for \sim , 126
 - n-ary \sim , 119
 - unary \sim , 118
- predicator function, 157
- premises (keyword), 46
- .prems (fact name suffix), 114
- prems (assumption name), 47
- presume (keyword), 57
- primitive recursive definition, 204
- primrec (keyword), 204
- print-bnfs
 - print_bnfs (keyword), 163
- print_cases (keyword), 95
- print_simpset (keyword), 91
- print_statement (keyword), 34
- prod (type), 152, 252
- prod_cases3 (fact name), 256
- prod_induct3 (fact name), 256
- product type, 252
- proof, 28, 41
 - aborted \sim , 44
 - by contradiction, 236
 - case based \sim , 94
 - skeleton, 95
 - context, 35, 39
 - named \sim , 93
 - nested \sim , 44, 54, 66, 93
 - outermost \sim , 45
 - fake \sim , 44, 49
 - method, 47, 70
 - input, 50
 - mode, 41
 - nested \sim , 44
 - script, 43
 - state, 35
 - initial \sim , 36, 40
 - structured \sim , 43
 - sub-, 46
 - termination \sim , 147
- proof (keyword), 42
- proof(chain) (proof mode), 51
- proof(prove) (proof mode), 41
- proof(state) (proof mode), 41
- proposition, 24
- proposition (keyword), 28
- .psimps (fact name suffix), 145
- Pure (session), 9, 15
- qed (keyword), 42, 71
- qrnb (example fact), 174
- quantifier, 234
 - bounded \sim , 234
- question mark, 30

- quotient, 173
 - morphisms of a \sim , 174
 - theorem, 174
 - type, 213
- Quotient (constant), 174
- quotient_type (keyword), 213
- quotpair (example type), 216

- range (constant), 155, 259
- Rangep (constant), 119
- raw type, 172
- reasoning
 - backward \sim , 38, 73
 - case based \sim , 97
 - equational \sim , 67
 - forward \sim , 38, 76
- record
 - component, 220
 - expression, 224
 - field, 220
 - fragment type, 221
 - schema expression, 224
 - type, 220
 - parameterized \sim , 222
 - type expression, 220
 - type scheme, 221
 - update function, 225
- record (keyword), 220
- record type, 191
- record type extension, 222
- recrd (example type), 192, 221
- recrd2 (example type), 222
- recrdp (example type), 223
- recursion, 138, 189
 - primitive \sim , 204
- recursive definition, 138, 202
 - mutual \sim , 149
- refinement step, 41, 49
- reflexivity rule, 179
- reflp (constant), 172
- regularCard (constant), 160
- _rel (constant name suffix), 144
- rel_fun (constant), 159, 260
- rel_option_unfold (fact name), 252
- rel_prod (constant), 255
- rel_set (constant), 246
- rel: (keyword), 162
- relation, 118
 - accessible part of \sim , 136
 - as tuple set, 125
 - equality \sim , 120
 - n-ary \sim , 125
 - ordering \sim , 120
 - transfer \sim , 164
 - parameterized \sim , 175
 - well-founded \sim , 135
- relation (method), 147
- relator function, 157
- relcompp (constant), 118
- rem3 (example type), 214
- Rep_ (constant name prefix), 206
- rep_ (constant name prefix), 174
- representation
 - function, 174
 - value, 174
- respectful
 - function, 181
- respectfulness theorem, 181
- rewriting, 84
- right_total (constant), 119
- right_unique (constant), 119
- rnbd (example constant), 166, 168
- rnbd (example constant), 173
- rnbdE (example constant), 173
- ROOT (file), 9
- root record type, 222
- rule
 - application, 72
 - case \sim , 98
 - associated \sim , 101
 - cases \sim , 129
 - derivation \sim , 25
 - destruction \sim , 82, 104, 106
 - elimination \sim , 105
 - induction \sim , 109, 110, 132, 200

- introduction \sim , 74
- meta \sim , 25
- parametricity \sim , 171
- reflexivity \sim , 179
- split \sim , 89, 199
- transfer \sim , 164
- transitivity \sim , 67
- rule (method), 72, 77
- rule: (method argument), 99, 111
- `_scheme` (type name suffix), 221
- `scomp` (constant), 258
- `Scratch.thy` (file), 10
- search
 - for constants, 21
 - for destruction rules, 83
 - for elimination rules, 106
 - for facts, 34
 - for introduction rules, 75
 - for simplifier equations, 90
 - specification, 13, 21, 34
- sections, 15
- `.sel` (fact name suffix), 198
- selector, 192
 - default \sim , 193
- selector function, 192
- sequential (keyword), 203
- session, 9
 - directory, 9
 - document, 9
 - parent \sim , 9
- set (type), 242
- set comprehension, 242, 254
- set enumeration, 244
- set-function, 154
- `set_map` (fact name), 154, 155
- `sets:` (keyword), 161
- `setup_lifting` (keyword), 176, 179
- `show` (keyword), 49
- `shows` (keyword), 32
- `simp` (fact set name), 87
- `simp` (method), 87
- `simp_all` (method), 90
- simplification, 86
- simplifier, 87
 - trace, 91
- `.simps` (fact name suffix), 130, 134, 139, 148, 198, 227
- `simpset`, 87
- single-step inductive definition, 133
- size (constant), 137
- `size_change` (method), 148
- `snd` (constant), 252
- `snds` (constant), 255
- `SOME` (binder), 122
- `Some` (constant), 249
- `sorry` (keyword), 44
- `spec` (fact name), 236
- `.split` (fact name suffix), 200
- `split_def` (fact name), 257
- `split_paired_All` (fact name), 257
- `split_paired_all` (fact name), 257
- `split_paired_Ex` (fact name), 257
- `split:` (method argument), 89
- standard (method), 75
- state
 - panel, 12
- `state_combinator_syntax` (bundle), 258
- statement, 41
- `strict_inc_induct` (fact name), 241
- `strict_mono` (constant), 262
- `strict_mono_on` (constant), 262
- structured form, 27, 32, 48
- structured proof, 43
- subgoal (keyword), 46
- subproof, 46
- subset (constant), 244
- `subset_eq` (constant), 244
- `subset_eq` (fact name), 248
- `subsetD` (fact name), 248
- `subsetI` (fact name), 248
- `subst` (method), 84
- substitution, 30
- subtype, 205
- `Suc` (constant), 238

- Suc_le_lessD (fact name), 242
- Suc_leD (fact name), 242
- Suc_leI (fact name), 241
- Suc_less_SucD (fact name), 242
- Suc_lessD (fact name), 242
- Suc_lessI (fact name), 241
- _sumC (constant name suffix), 143
- Sup (constant), 121
- sup (constant), 121
- supset (constant), 244
- supset_eq (constant), 244
- surj (constant), 260
- surjE (fact name), 263
- surjI (fact name), 262
- swap (constant), 255
- symbol
 - abbreviation, 13
 - operator \sim , 20
 - panel, 12
- syntax
 - alternative \sim
 - for case rules, 102
 - for elimination rules, 107
 - for function update, 259
 - for Pair, 253
 - for record updates, 226
 - for rules, 26
 - for theorems, 32
 - binder \sim , 20, 26
 - inner \sim , 14, 16
 - LaTeX \sim , 15
 - outer \sim , 15, 16
 - set comprehension \sim , 242
 - set enumeration \sim , 244
 - structured \sim
 - for rules, 27
 - for theorems, 32
- term, 16
 - abbreviation, 35
 - casual \sim , 64
 - parameterized \sim , 63
 - case \sim , 194
 - if \sim , 233, 235
 - lambda \sim , 20, 254, 258
 - let \sim , 124, 254
 - pattern, 34, 63
 - splitting, 89
 - synonym, 23
- term abbreviation, 62
- termination (keyword), 147
- termination proof, 147
- text area, 11
- that (assumption name), 37, 102
- THE (binder), 123
- the (constant), 249
- the_elem (constant), 244
- the_inv (constant), 260
- the_inv_into (constant), 260
- then (inner keyword), 233
- then (keyword), 51
- theorem, 28
 - quotient \sim , 174
 - respectfulness \sim , 181
 - type-definition \sim , 175
- theorem (keyword), 28
- theories
 - imported \sim , 16
 - panel, 13
- theory, 14
 - file, 9, 16
 - name, 16
- theory (keyword), 16
- these (constant), 250
- this (fact set name), 51
- this (method), 79
- thm (keyword), 33
- three (example type), 191
- threesub (example type), 205
- Times (constant), 255
- times (constant), 239
- to_pred (attribute), 118
- to_set (attribute), 118
- top (constant), 121
- trans (fact set name), 69
- transfer, 164
 - relation, 164

- rule, 164
- transfer (method), 168
- Tree (example constant), 189
- tree (example type), 189
- treex (example type), 190
- trnb (example fact), 175
- True (case name), 100
- True (constant), 18, 25, 233
- try (keyword), 92
- tuple, 125, 191, 252
 - empty \sim , 237, 253
- Two (example constant), 191
- type, 16
 - abstract \sim , 172
 - algebraic \sim , 188
 - constructor, 17
 - container \sim , 154
 - copy, 207
 - enumeration \sim , 191
 - expression, 17
 - record \sim , 220
 - instance, 17
 - name, 17
 - parameter, 17
 - dead \sim , 156
 - live \sim , 156
 - parameterized \sim , 17
 - parent \sim , 222
 - polymorphic \sim , 17
 - product \sim , 252
 - quotient \sim , 213
 - raw \sim , 172
 - record \sim , 191, 220
 - parameterized \sim , 222
 - root \sim , 222
 - scheme, 221
 - record fragment \sim , 221
 - specification, 19, 27
 - synonym, 18
 - variables, 17
- type_definition (constant), 175
- type-definition theorem, 175
- type_synonym (keyword), 18
- typedekl (keyword), 17
- typedef (keyword), 205
- ultimately (keyword), 66
- uminus (constant), 244
- Un_def (fact name), 248
- unbundle (keyword), 122
- uncurried function, 125
- uncurrying, 254
- undefined (constant), 123, 203
- underspecified
 - constant, 18, 22
 - function, 19
- UnE (fact name), 248
- UnI (fact name), 248
- unification, 38, 62
 - higher order \sim , 63
- Union (constant), 244
- union (constant), 244
- Uniq (constant), 234
- unit (type), 237, 253
- Unity (constant), 237
- UNIV (constant), 243
- unknown, 30, 39, 49, 58
 - bound \sim , 63
 - dummy \sim , 63
- Unsorted (chapter name), 9
- _update (constant name suffix), 225
- using (keyword), 51
- uspec (example constant), 144
- uspec₁ (example constant), 128
- uspec₂ (example constant), 128
- uspec₃ (example constant), 128
- variable, 18, 19
 - binding, 29, 32
 - bound \sim , 20, 26, 29, 36, 57, 59
 - definition, 59
 - fixed \sim , 35, 49, 57
 - free occurrence of \sim , 20, 25, 29, 37, 57
 - group, 27
 - local \sim , 58

vimage (constant), 152 , 246	where (keyword), 22 , 23 , 59 , 60 , 102 , 126 , 139
well-founded relation, 135	wildcard, 20
wf (constant), 135	with (keyword), 53
wf_induct_rule (fact name), 135	wits (keyword), 186
wfP (constant), 135	Zero (example constant), 191
wfp_induct_rule (fact name), 135	zip construction, 158