



The Isabelle/Isar Reference Manual

Markus Wenzel
TU München

8 March 2002

Contents

1	Introduction	1
1.1	Overview	1
1.2	Quick start	2
1.2.1	Terminal sessions	2
1.2.2	Proof General	2
1.3	Isabelle/Isar theories	4
1.3.1	Document preparation	5
1.3.2	How to write Isar proofs anyway?	6
2	Syntax primitives	7
2.1	Lexical matters	8
2.2	Common syntax entities	9
2.2.1	Names	9
2.2.2	Comments	10
2.2.3	Type classes, sorts and arities	10
2.2.4	Types and terms	11
2.2.5	Mixfix annotations	13
2.2.6	Proof methods	14
2.2.7	Attributes and theorems	14
2.2.8	Term patterns and declarations	16
2.2.9	Antiquotations	18
3	Basic language elements	21
3.1	Theory commands	21
3.1.1	Defining theories	21
3.1.2	Markup commands	23
3.1.3	Type classes and sorts	24
3.1.4	Primitive types and type abbreviations	25
3.1.5	Constants and simple definitions	26
3.1.6	Syntax and translations	26
3.1.7	Axioms and theorems	27
3.1.8	Name spaces	28
3.1.9	Incorporating ML code	29
3.1.10	Syntax translation functions	30

3.1.11	Oracles	31
3.2	Proof commands	32
3.2.1	Markup commands	32
3.2.2	Context elements	33
3.2.3	Facts and forward chaining	34
3.2.4	Goal statements	36
3.2.5	Initial and terminal proof steps	38
3.2.6	Fundamental methods and attributes	40
3.2.7	Term abbreviations	43
3.2.8	Block structure	44
3.2.9	Emulating tactic scripts	44
3.2.10	Meta-linguistic features	46
3.3	Other commands	47
3.3.1	Diagnostics	47
3.3.2	Inspecting the context	48
3.3.3	History commands	49
3.3.4	System operations	50
4	Generic tools and packages	51
4.1	Theory specification commands	51
4.1.1	Axiomatic type classes	51
4.1.2	Locales and local contexts	52
4.2	Derived proof schemes	56
4.2.1	Generalized elimination	56
4.2.2	Calculational reasoning	57
4.3	Proof tools	59
4.3.1	Miscellaneous methods and attributes	59
4.3.2	Further tactic emulations	62
4.3.3	The Simplifier	64
4.3.4	The Classical Reasoner	68
4.3.5	Proof by cases and induction	73
5	Object-logic specific elements	79
5.1	General logic setup	79
5.2	HOL	80
5.2.1	Primitive types	80
5.2.2	Adhoc tuples	82
5.2.3	Records	82
5.2.4	Datatypes	86
5.2.5	Recursive functions	87
5.2.6	(Co)Inductive sets	89

5.2.7	Arithmetic proof support	90
5.2.8	Cases and induction: emulating tactic scripts	90
5.2.9	Executable code	91
5.3	HOLCF	92
5.3.1	Mixfix syntax for continuous operations	92
5.3.2	Recursive domains	93
5.4	ZF	93
5.4.1	Type checking	93
5.4.2	(Co)Inductive sets and datatypes	94
A	Isabelle/Isar quick reference	98
A.1	Proof commands	98
A.1.1	Primitives and basic syntax	98
A.1.2	Abbreviations and synonyms	99
A.1.3	Derived elements	99
A.1.4	Diagnostic commands	99
A.2	Proof methods	100
A.3	Attributes	101
A.4	Emulating tactic scripts	101
A.4.1	Commands	101
A.4.2	Methods	102
B	Isabelle/Isar conversion guide	103
B.1	No conversion	103
B.1.1	Referring to theorem and theory values	103
B.1.2	Storing theorem values	104
B.1.3	ML declarations in Isar	104
B.2	Porting theories and proof scripts	105
B.2.1	Theories	105
B.2.2	Goal statements	106
B.2.3	Tactics	108
B.2.4	Declarations and ad-hoc operations	112
B.3	Writing actual Isar proof texts	113

Introduction

1.1 Overview

The *Isabelle* system essentially provides a generic infrastructure for building deductive systems (programmed in Standard ML), with a special focus on interactive theorem proving in higher-order logics. In the olden days even end-users would refer to certain ML functions (goal commands, tactics, tacticals etc.) to pursue their everyday theorem proving tasks [9, 10].

In contrast *Isar* provides an interpreted language environment of its own, which has been specifically tailored for the needs of theory and proof development. Compared to raw ML, the Isabelle/Isar top-level provides a more robust and comfortable development platform, with proper support for theory development graphs, single-step transactions with unlimited undo, etc. The Isabelle/Isar version of the *Proof General* user interface [1, 2] provides an adequate front-end for interactive theory and proof development in this advanced theorem proving environment.

Apart from the technical advances over bare-bones ML programming, the main purpose of the Isar language is to provide a conceptually different view on machine-checked proofs [15, 17]. “Isar” stands for “Intelligible semi-automated reasoning”. Drawing from both the traditions of informal mathematical proof texts and high-level programming languages, Isar offers a versatile environment for structured formal proof documents. Thus properly written Isar proofs become accessible to a broader audience than unstructured tactic scripts (which typically only provide operational information for the machine). Writing human-readable proof texts certainly requires some additional efforts by the writer to achieve a good presentation, both of formal and informal parts of the text. On the other hand, human-readable formal texts gain some value in their own right, independently of the mechanistic proof-checking process.

Despite its grand design of structured proof texts, Isar is able to assimilate the old tactical style as an “improper” sub-language. This provides an easy upgrade path for existing tactic scripts, as well as additional means for interactive experimentation and debugging of structured proofs. Isabelle/Isar

supports a broad range of proof styles, both readable and unreadable ones.

The Isabelle/Isar framework is generic and should work reasonably well for any Isabelle object-logic that conforms to the natural deduction view of the Isabelle/Pure framework. Major Isabelle logics like HOL [7], HOLCF [5], FOL [11], and ZF [12] have already been set up for end-users. Nonetheless, much of the existing body of theories still consist of old-style theory files with accompanied ML code for proof scripts; this legacy will be gradually converted in due time.

1.2 Quick start

1.2.1 Terminal sessions

Isar is already part of Isabelle. The low-level `isabelle` binary provides option `-I` to run the Isabelle/Isar interaction loop at startup, rather than the raw ML top-level. So the most basic way to do anything with Isabelle/Isar is as follows:

```
isabelle -I HOL
> Welcome to Isabelle/HOL (Isabelle2002)

theory Foo = Main:
constdefs foo :: nat  "foo == 1";
lemma "0 < foo" by (simp add: foo_def);
end
```

Note that any Isabelle/Isar command may be retracted by `undo`. See the Isabelle/Isar Quick Reference (appendix A) for a comprehensive overview of available commands and other language elements.

1.2.2 Proof General

Plain TTY-based interaction as above used to be quite feasible with traditional tactic based theorem proving, but developing Isar documents really demands some better user-interface support. The Proof General environment by David Aspinall [1, 2] offers a generic Emacs interface for interactive theorem provers that organizes all the cut-and-paste and forward-backward walk through the text in a very neat way. In Isabelle/Isar, the current position within a partial proof document is equally important than the actual proof state. Thus Proof General provides the canonical working environment for Isabelle/Isar, both for getting acquainted (e.g. by replaying existing Isar documents) and for production work.

Proof General as default Isabelle interface

The Isabelle interface wrapper script provides an easy way to invoke Proof General (including XEmacs or GNU Emacs). The default configuration of Isabelle is smart enough to detect the Proof General distribution in several canonical places (e.g. `$ISABELLE_HOME/contrib/ProofGeneral`). Thus the capital **Isabelle** executable would already refer to the **ProofGeneral/isar** interface without further ado.¹ The Isabelle interface script provides several options; pass `-?` to see its usage.

With the proper Isabelle interface setup, Isar documents may now be edited by visiting appropriate theory files, e.g.

```
Isabelle {isabellehome}/src/HOL/Isar_examples/Summation.thy
```

Beginners may note the tool bar for navigating forward and backward through the text (this depends on the local Emacs installation). Consult the Proof General documentation [1] for further basic command sequences, in particular “C-c C-return” and “C-c u”.

Proof General may be also configured manually by giving Isabelle settings like this (see also [18]):

```
ISABELLE_INTERFACE=$ISABELLE_HOME/contrib/ProofGeneral/isar/interface
PROOFGENERAL_OPTIONS=""
```

You may have to change `$ISABELLE_HOME/contrib/ProofGeneral` to the actual installation directory of Proof General.

Apart from the Isabelle command line, defaults for interface options may be given by the `PROOFGENERAL_OPTIONS` setting. For example, the Emacs executable to be used may be configured in Isabelle’s settings like this:

```
PROOFGENERAL_OPTIONS="-p xemacs-nomule"
```

Occasionally, a user’s `~/.emacs` file contains code that is incompatible with the (X)Emacs version used by Proof General, causing the interface startup to fail prematurely. Here the `-u false` option helps to get the interface process up and running. Note that additional Lisp customization code may reside in `proofgeneral-settings.el` of `$ISABELLE_HOME/etc` or `$ISABELLE_HOME_USER/etc`.

¹There is also a **ProofGeneral/isa** interface for old tactic scripts written in ML.

The X-Symbol package

Proof General provides native support for the Emacs X-Symbol package [13], which handles proper mathematical symbols displayed on screen. Pass option `-x true` to the Isabelle interface script, or check the appropriate Proof General menu setting by hand. In any case, the X-Symbol package must have been properly installed already.

Contrary to what you may expect from the documentation of X-Symbol, the package is very easy to install and configures itself automatically. The default configuration of Isabelle is smart enough to detect the X-Symbol package in several canonical places (e.g. `$ISABELLE_HOME/contrib/x-symbol`).

Using proper mathematical symbols in Isabelle theories can be very convenient for readability of large formulas. On the other hand, the plain ASCII sources easily become somewhat unintelligible. For example, \implies would appear as `\<Longrightarrow>` according the default set of Isabelle symbols. Nevertheless, the Isabelle document preparation system (see §1.3.1) will be happy to print non-ASCII symbols properly. It is even possible to invent additional notation beyond the display capabilities of Emacs and X-Symbol.

1.3 Isabelle/Isar theories

Isabelle/Isar offers the following main improvements over classic Isabelle.

1. A new *theory format*, occasionally referred to as “new-style theories”, supporting interactive development and unlimited undo operation.
2. A *formal proof document language* designed to support intelligible semi-automated reasoning. Instead of putting together unreadable tactic scripts, the author is enabled to express the reasoning in way that is close to usual mathematical practice. The old tactical style has been assimilated as “improper” language elements.
3. A simple document preparation system, for typesetting formal developments together with informal text. The resulting hyper-linked PDF documents are equally well suited for WWW presentation and as printed copies.

The Isar proof language is embedded into the new theory format as a proper sub-language. Proof mode is entered by stating some **theorem** or **lemma** at the theory level, and left again with the final conclusion (e.g. via **qed**). A few theory specification mechanisms also require some proof, such as HOL’s **typedef** which demands non-emptiness of the representing sets.

New-style theory files may still be associated with separate ML files consisting of plain old tactic scripts. There is no longer any ML binding generated for the theory and theorems, though. ML functions `theory`, `thm`, and `thms` retrieve this information from the context [10]. Nevertheless, migration between classic Isabelle and Isabelle/Isar is relatively easy. Thus users may start to benefit from interactive theory development and document preparation, even before they have any idea of the Isar proof language at all.

! Proof General does *not* support mixed interactive development of classic Isabelle theory files or tactic scripts, together with Isar documents. The “isa” and “isar” versions of Proof General are handled as two different theorem proving systems, only one of these may be active at the same time.

Manual conversion of existing tactic scripts may be done by running two separate Proof General sessions, one for replaying the old script and the other for the emerging Isabelle/Isar document. Also note that Isar supports emulation commands and methods that support traditional tactic scripts within new-style theories, see appendix B for more information.

1.3.1 Document preparation

Isabelle/Isar provides a simple document preparation system based on existing PDF- \LaTeX technology, with full support of hyper-links (both local references and URLs), bookmarks, and thumbnails. Thus the results are equally well suited for WWW browsing and as printed copies.

Isabelle generates \LaTeX output as part of the run of a *logic session* (see also [18]). Getting started with a working configuration for common situations is quite easy by using the Isabelle `mkdir` and `make` tools. First invoke

```
isatool mkdir Foo
```

to initialize a separate directory for session `Foo` — it is safe to experiment, since `isatool mkdir` never overwrites existing files. Ensure that `Foo/ROOT.ML` holds ML commands to load all theories required for this session; furthermore `Foo/document/root.tex` should include any special \LaTeX macro packages required for your document (the default is usually sufficient as a start).

The session is controlled by a separate `IsaMakefile` (with crude source dependencies by default). This file is located one level up from the `Foo` directory location. Now invoke

```
isatool make Foo
```

to run the `Foo` session, with browser information and document preparation enabled. Unless any errors are reported by Isabelle or \LaTeX , the output

will appear inside the directory `ISABELLE_BROWSER_INFO`, as reported by the batch job in verbose mode.

You may also consider to tune the `usedir` options in `IsaMakefile`, for example to change the output format from `pdf` to `dvi`, or activate the `-D` option to retain a second copy of the generated \LaTeX sources.

See *The Isabelle System Manual* [18] for further details on Isabelle logic sessions and theory presentation. The Isabelle/HOL tutorial [8] also covers theory presentation issues.

1.3.2 How to write Isar proofs anyway?

This is one of the key questions, of course. First of all, the tactic script emulation of Isabelle/Isar essentially provides a clarified version of the very same unstructured proof style of classic Isabelle. Old-time users should quickly become acquainted with that (slightly degenerative) view of Isar.

Writing *proper* Isar proof texts targeted at human readers is quite different, though. Experienced users of the unstructured style may even have to unlearn some of their habits to master proof composition in Isar. In contrast, new users with less experience in old-style tactical proving, but a good understanding of mathematical proof in general, often get started easier.

The present text really is only a reference manual on Isabelle/Isar, not a tutorial. Nevertheless, we will attempt to give some clues of how the concepts introduced here may be put into practice. Appendix A provides a quick reference card of the most common Isabelle/Isar language elements. Appendix B offers some practical hints on converting existing Isabelle theories and proof scripts to the new format (without restructuring proofs).

Further issues concerning the Isar concepts are covered in the literature [15, 19, 3, 4]. The author’s PhD thesis [17] presently provides the most complete exposition of Isar foundations, techniques, and applications. A number of example applications are distributed with Isabelle, and available via the Isabelle WWW library (e.g. <http://isabelle.in.tum.de/library/>). As a general rule of thumb, more recent Isabelle applications that also include a separate “document” (in PDF) are more likely to consist of proper Isabelle/Isar theories and proofs.

Syntax primitives

The rather generic framework of Isabelle/Isar syntax emerges from three main syntactic categories: *commands* of the top-level Isar engine (covering theory and proof elements), *methods* for general goal refinements (analogous to traditional “tactics”), and *attributes* for operations on facts (within a certain context). Here we give a reference of basic syntactic entities underlying Isabelle/Isar syntax in a bottom-up manner. Concrete theory and proof language elements will be introduced later on.

In order to get started with writing well-formed Isabelle/Isar documents, the most important aspect to be noted is the difference of *inner* versus *outer* syntax. Inner syntax is that of Isabelle types and terms of the logic, while outer syntax is that of Isabelle/Isar theory sources (including proofs). As a general rule, inner syntax entities may occur only as *atomic entities* within outer syntax. For example, the string “ $x + y$ ” and identifier z are legal term specifications within a theory, while $x + y$ is not.

! Old-style Isabelle theories used to fake parts of the inner syntax of types, with
• rather complicated rules when quotes may be omitted. Despite the minor drawback of requiring quotes more often, the syntax of Isabelle/Isar is somewhat simpler and more robust in that respect.

Printed theory documents usually omit quotes to gain readability (this is a matter of L^AT_EX macro setup, say via `\isabellestyle`, see also [18]). Experienced users of Isabelle/Isar may easily reconstruct the lost technical information, while mere readers need not care about quotes at all.

Isabelle/Isar input may contain any number of input termination characters “;” (semicolon) to separate commands explicitly. This is particularly useful in interactive shell sessions to make clear where the current command is intended to end. Otherwise, the interpreter loop will continue to issue a secondary prompt “#” until an end-of-command is clearly recognized from the input syntax, e.g. encounter of the next command keyword.

Advanced interfaces such as Proof General [1] do not require explicit semicolons, the amount of input text is determined automatically by inspecting

the present content of the Emacs text buffer. In the printed presentation of Isabelle/Isar documents semicolons are omitted altogether for readability.

! Proof General requires certain syntax classification tables in order to achieve properly synchronized interaction with the Isabelle/Isar process. These tables need to be consistent with the Isabelle version and particular logic image to be used in a running session (common object-logics may well change the outer syntax). The standard setup should work correctly with any of the “official” logic images derived from Isabelle/HOL (including HOLCF etc.). Users of alternative logics may need to tell Proof General explicitly, e.g. by giving an option `-k ZF` (in conjunction with `-l ZF` to specify the default logic image).

2.1 Lexical matters

The Isabelle/Isar outer syntax provides token classes as presented below. Note that some of these coincide (by full intention) with the inner lexical syntax as presented in [10].

$$\begin{aligned}
 \textit{ident} &= \textit{letter quasiletter}^* \\
 \textit{longident} &= \textit{ident}.\textit{ident} \dots \textit{ident} \\
 \textit{symident} &= \textit{sym}^+ \mid \textit{symbol} \\
 \textit{nat} &= \textit{digit}^+ \\
 \textit{var} &= ?\textit{ident} \mid ?\textit{ident}.\textit{nat} \\
 \textit{typefree} &= '\textit{ident} \\
 \textit{typevar} &= ?\textit{typefree} \mid ?\textit{typefree}.\textit{nat} \\
 \textit{string} &= " \dots " \\
 \textit{verbatim} &= \{ * \dots * \} \\
 \\
 \textit{letter} &= \text{a} \mid \dots \mid \text{z} \mid \text{A} \mid \dots \mid \text{Z} \\
 \textit{digit} &= 0 \mid \dots \mid 9 \\
 \textit{quasiletter} &= \textit{letter} \mid \textit{digit} \mid _ \mid ' \\
 \textit{sym} &= ! \mid \# \mid \$ \mid \% \mid \& \mid * \mid + \mid - \mid / \mid : \mid \\
 &\quad < \mid = \mid > \mid ? \mid @ \mid ^ \mid _ \mid ' \mid | \mid \sim \\
 \textit{symbol} &= \forall \mid \exists \mid \wedge \mid \vee \mid \dots
 \end{aligned}$$

The syntax of *string* admits any characters, including newlines; “`”` (double-quote) and “`\`” (backslash) need to be escaped by a backslash. Note that ML-style control characters are *not* supported. The body of *verbatim* may consist of any text not containing “`*}`”; this allows convenient inclusion of quotes without further escapes.

Comments take the form `(* ... *)` and may in principle be nested, just as in ML. Note that these are *source* comments only, which are stripped after lexical analysis of the input. The Isar document syntax also provides *formal comments* that are considered as part of the text (see §2.2.2).

! Proof General does not handle nested comments properly; it is also unable to keep `(*/{*` and `*/}` apart, despite their rather different meaning. These are inherent problems of Emacs legacy. Users should not be overly aggressive about nesting or alternating these delimiters.

Mathematical symbols such as “ \forall ” are represented in plain ASCII as “`\<forall>`”. Concerning Isabelle itself, any sequence of the form `\<ident>` (or `\\<ident>`) is a legal symbol. Display of appropriate glyphs is a matter of front-end tools, say the user-interface of Proof General plus the X-Symbol package, or the L^AT_EX macro setup of document output. A list of predefined Isabelle symbols is given in [18, appendix A].

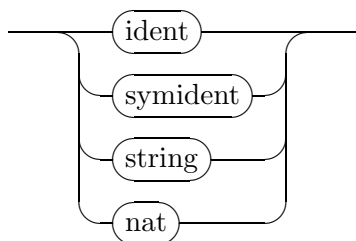
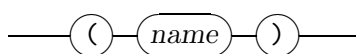
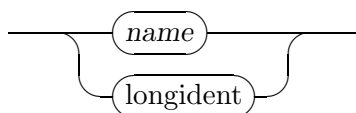
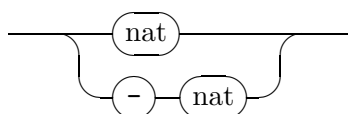
2.2 Common syntax entities

Subsequently, we introduce several basic syntactic entities, such as names, terms, and theorem specifications, which have been factored out of the actual Isar language elements to be described later.

Note that some of the basic syntactic entities introduced below (e.g. *name*) act much like tokens rather than plain nonterminals (e.g. *sort*), especially for the sake of error messages. E.g. syntax elements like **consts** referring to *name* or *type* would really report a missing name or type rather than any of the constituent primitive tokens such as *ident* or *string*.

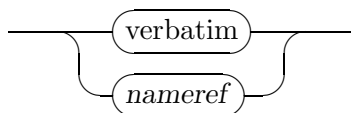
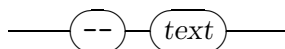
2.2.1 Names

Entity *name* usually refers to any name of types, constants, theorems etc. that are to be *declared* or *defined* (so qualified identifiers are excluded here). Quoted strings provide an escape for non-identifier names or those ruled out by outer syntax keywords (e.g. “`let`”). Already existing objects are usually referenced by *nameref*.

name*parname**nameref**int*

2.2.2 Comments

Large chunks of plain *text* are usually given verbatim, i.e. enclosed in `{* ... *}`. For convenience, any of the smaller text units conforming to *nameref* are admitted as well. A marginal *comment* is of the form `-- text`. Any number of these may occur within Isabelle/Isar commands.

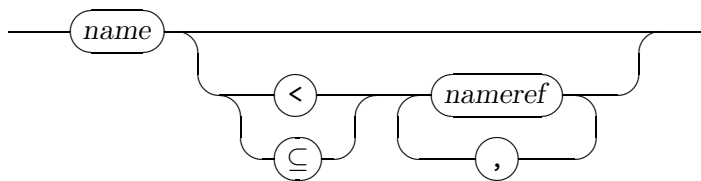
text*comment*

2.2.3 Type classes, sorts and arities

Classes are specified by plain names. Sorts have a very simple inner syntax, which is either a single class name c or a list $\{c_1, \dots, c_n\}$ referring to the

intersection of these classes. The syntax of type arities is given directly at the outer level.

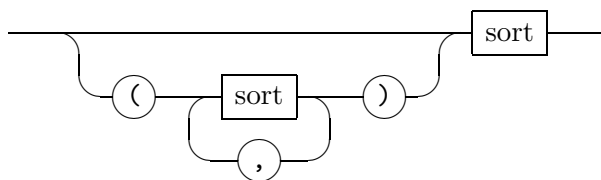
classdecl



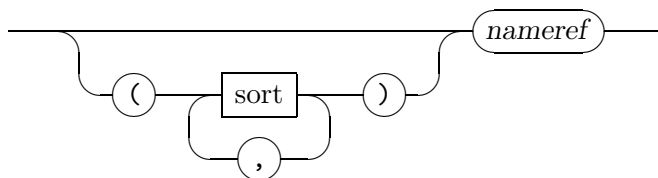
sort



arity

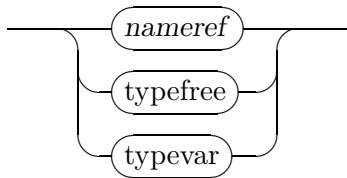
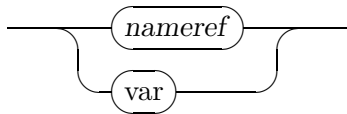
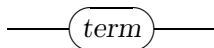


simplearity

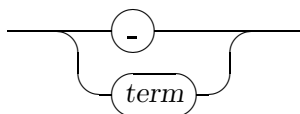
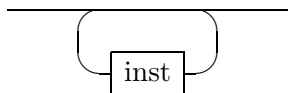


2.2.4 Types and terms

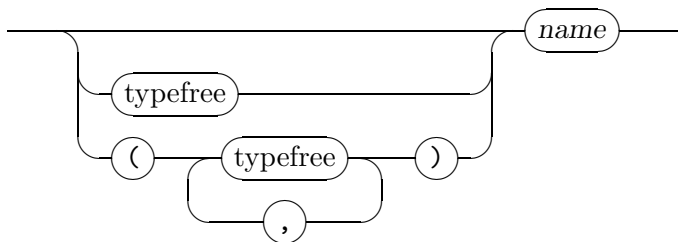
The actual inner Isabelle syntax, that of types and terms of the logic, is far too sophisticated in order to be modelled explicitly at the outer theory level. Basically, any such entity has to be quoted to turn it into a single token (the parsing and type-checking is performed internally later). For convenience, a slightly more liberal convention is adopted: quotes may be omitted for any type or term that is already atomic at the outer level. For example, one may just write *x* instead of "*x*". Note that symbolic identifiers (e.g. $++$ or \forall) are available as well, provided these have not been superseded by commands or other keywords already (e.g. $=$ or $+$).

type*term**prop*

Positional instantiations are indicated by giving a sequence of terms, or the placeholder “_” (underscore), which means to skip a position.

inst*insts*

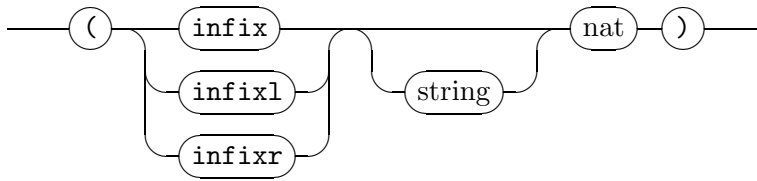
Type declarations and definitions usually refer to *typespec* on the left-hand side. This models basic type constructor application at the outer syntax level. Note that only plain postfix notation is available here, but no infixes.

typespec

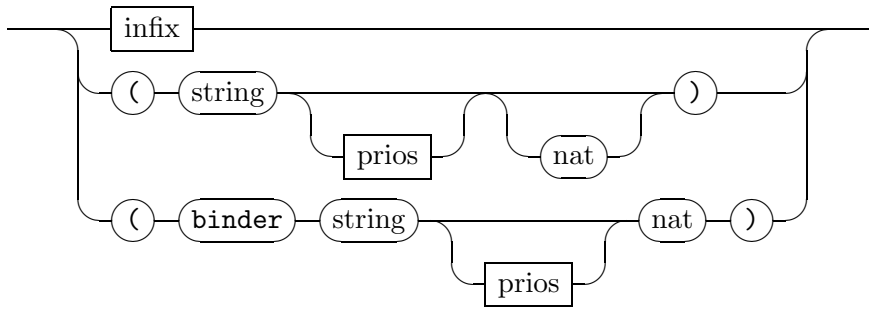
2.2.5 Mixfix annotations

Mixfix annotations specify concrete *inner* syntax of Isabelle types and terms. Some commands such as **types** (see §3.1.4) admit infixes only, while **consts** (see §3.1.5) and **syntax** (see §3.1.6) support the full range of general mixfixes and binders.

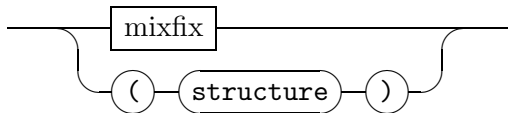
infix



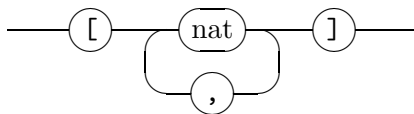
mixfix



structmixfix



prios

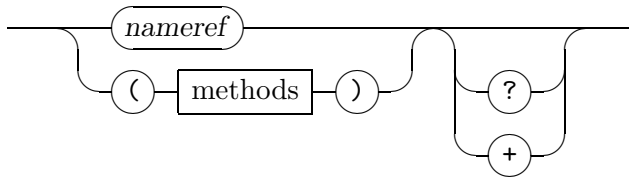


Here the string specifications refer to the actual mixfix template (see also [10]), which may include literal text, spacing, blocks, and arguments (denoted by “_”); the special symbol `\<index>` (printed as “i”) represents an index argument that specifies an implicit structure reference (see also §4.1.2). Infix and binder declarations provide common abbreviations for particular mixfix declarations. So in practice, mixfix templates mostly degenerate to literal text for concrete syntax, such as “++” for an infix symbol, or “++i” for an infix of an implicit structure.

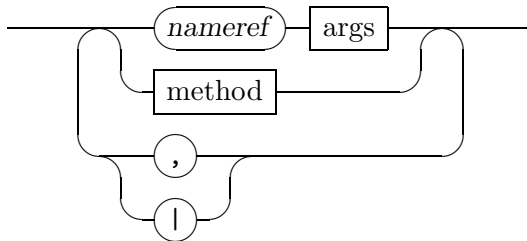
2.2.6 Proof methods

Proof methods are either basic ones, or expressions composed of methods via “,” (sequential composition), “|” (alternative choices), “?” (try), “+” (repeat at least once). In practice, proof methods are usually just a comma separated list of *nameref args* specifications. Note that parentheses may be dropped for single method specifications (with no arguments).

method

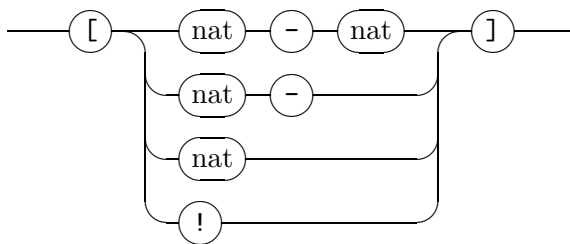


methods



Proper use of Isar proof methods does *not* involve goal addressing. Nevertheless, specifying goal ranges may occasionally come in handy in emulating tactic scripts. Note that $[n-]$ refers to all goals, starting from n . All goals may be specified by $[!]$, which is the same as $[1-]$.

goalspec

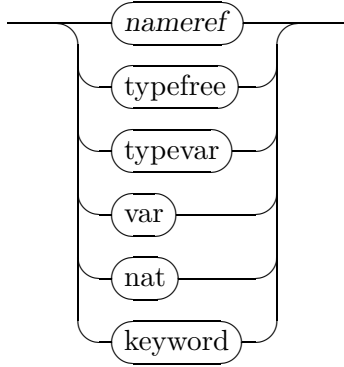


2.2.7 Attributes and theorems

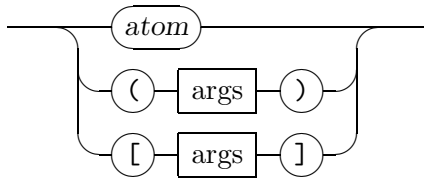
Attributes (and proof methods, see §2.2.6) have their own “semi-inner” syntax, in the sense that input conforming to *args* below is parsed by the attribute a second time. The attribute argument specifications may be any

sequence of atomic entities (identifiers, strings etc.), or properly bracketed argument lists. Below *atom* refers to any atomic entity, including any keyword conforming to *symident*.

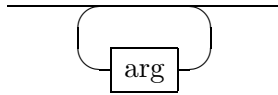
atom



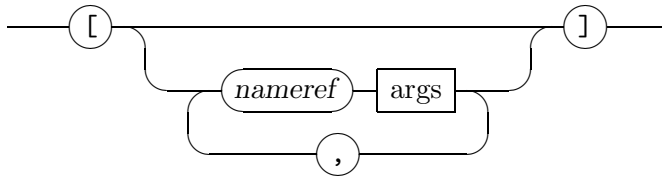
arg



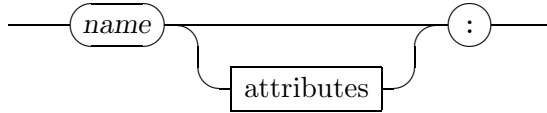
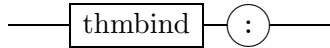
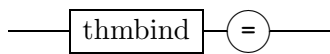
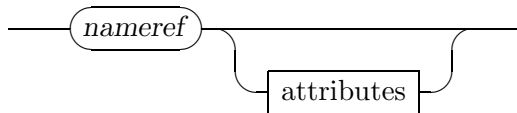
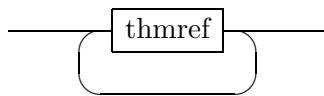
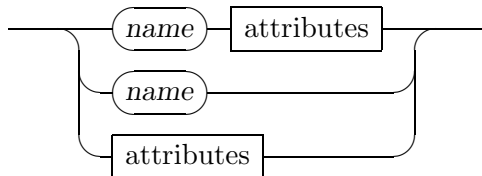
args



attributes

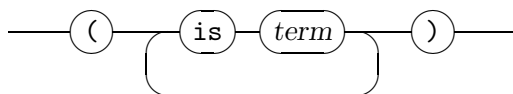


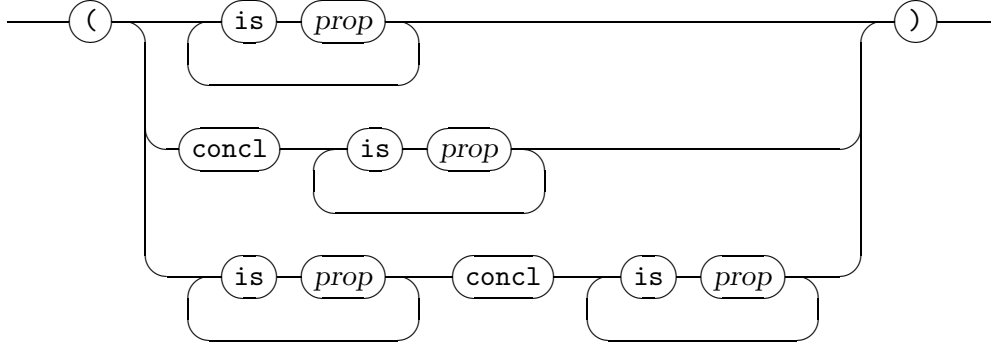
Theorem specifications come in several flavors: *axmdecl* and *thmdecl* usually refer to axioms, assumptions or results of goal statements, while *thmdef* collects lists of existing theorems. Existing theorems are given by *thmref* and *thmrefs*, the former requires an actual singleton result. Any of these theorem specifications may include lists of attributes both on the left and right hand sides; attributes are applied to any immediately preceding fact. If names are omitted, the theorems are not stored within the theorem database of the theory or proof context; any given attributes are still applied, though.

axmdecl*thmdecl**thmdef**thmref**thmrefs**thmbind*

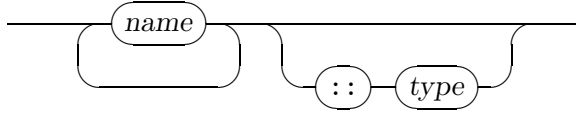
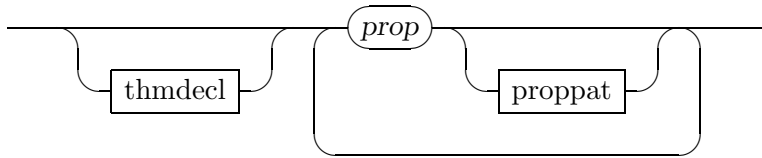
2.2.8 Term patterns and declarations

Wherever explicit propositions (or term fragments) occur in a proof text, casual binding of schematic term variables may be given specified via patterns of the form “(is $p_1 \dots$ is p_n)”. There are separate versions available for *terms* and *props*. The latter provides a **concl** part with patterns referring the (atomic) conclusion of a rule.

termpat

proppat

Declarations of local variables $x :: \tau$ and logical propositions $a : \varphi$ represent different views on the same principle of introducing a local scope. In practice, one may usually omit the typing of *vars* (due to type-inference), and the naming of propositions (due to implicit references of current facts). In any case, Isar proof elements usually admit to introduce multiple such items simultaneously.

vars*props*

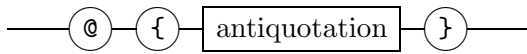
The treatment of multiple declarations corresponds to the complementary focus of *vars* versus *props*: in “ $x_1 \dots x_n :: \tau$ ” the typing refers to all variables, while in “ $a : \varphi_1 \dots \varphi_n$ ” the naming refers to all propositions collectively. Isar language elements that refer to *vars* or *props* typically admit separate typings or namings via another level of iteration, with explicit **and** separators; e.g. see **fix** and **assume** in §3.2.2.

2.2.9 Antiquotations

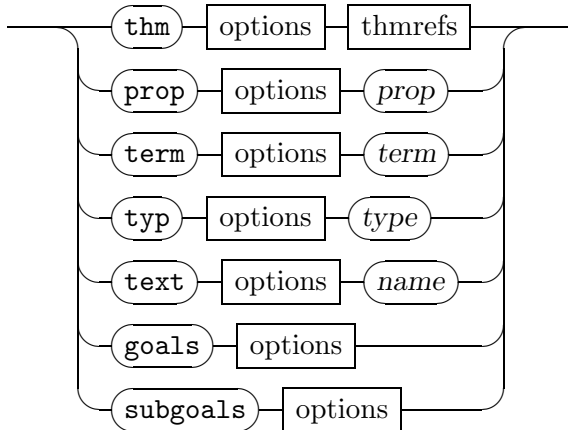
thm : antiquotation
 $prop$: antiquotation
 $term$: antiquotation
 typ : antiquotation
 $text$: antiquotation
 $goals$: antiquotation
 $subgoals$: antiquotation

The text body of formal comments (see also §2.2.2) may contain antiquotations of logical entities, such as theorems, terms and types, which are to be presented in the final output produced by the Isabelle document preparation system (see also §1.3.1).

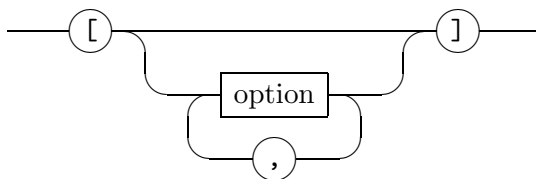
Thus embedding of “@{term [show_types] “f(x) = a + x”}” within a text block would cause $(f::'a \Rightarrow 'a) (x::'a) = (a::'a) + x$ to appear in the final L^AT_EX document. Also note that theorem antiquotations may involve attributes as well. For example, @{thm sym [no_vars]} would print the statement where all schematic variables have been replaced by fixed ones, which are easier to read.



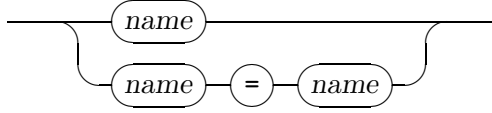
antiquotation



options



option



Note that the syntax of antiquotations may *not* include source comments (`(* ... *)`) or verbatim text (`{* ... *}`).

`@{thm \bar{a} }` prints theorems \bar{a} . Note that attribute specifications may be included as well (see also §2.2.7); the `no_vars` operation (see §4.3.1) would be particularly useful to suppress printing of schematic variables.

`@{prop φ }` prints a well-typed proposition φ .

`@{term t }` prints a well-typed term t .

`@{typ τ }` prints a well-formed type τ .

`@{text s }` prints uninterpreted source text s . This is particularly useful to print portions of text according to the Isabelle L^AT_EX output style, without demanding well-formedness (e.g. small pieces of terms that should not be parsed or type-checked yet).

`@{goals}` prints the current *dynamic* goal state. This is mainly for support of tactic-emulation scripts within Isar — presentation of goal states does not conform to actual human-readable proof documents. Please do not include goal states into document output unless you really know what you are doing!

`@{subgoals}` behaves almost like *goals*, except that it does not print the main goal.

The following options are available to tune the output. Note that most of these coincide with ML flags of the same names (see also [10]).

`show_types = bool` and `show_sorts = bool` control printing of explicit type and sort constraints.

`long_names = bool` forces names of types and constants etc. to be printed in their fully qualified internal form.

`eta_contract = bool` prints terms in η -contracted form.

display = *bool* indicates if the text is to be output as multi-line “display material”, rather than a small piece of text without line breaks (which is the default).

quotes = *bool* indicates if the output should be enclosed in double quotes.

mode = *name* adds *name* to the print mode to be used for presentation (see also [10]). Note that the standard setup for L^AT_EX output is already present by default, including the modes “*latex*”, “*xsymbols*”, “*symbols*”.

margin = *nat* and *indent* = *nat* change the margin or indentation for pretty printing of display material.

source = *bool* prints the source text of the antiquotation arguments, rather than the actual value. Note that this does not affect well-formedness checks of *thm*, *term*, etc. (only the *text* antiquotation admits arbitrary output).

goals_limit = *nat* determines the maximum number of goals to be printed.

For boolean flags, “*name* = *true*” may be abbreviated as “*name*”. All of the above flags are disabled by default, unless changed from ML.

Note that antiquotations do not only spare the author from tedious typing of logical entities, but also achieve some degree of consistency-checking of informal explanations with formal developments: well-formedness of terms and types with respect to the current theory or proof context is ensured here.

Basic language elements

Subsequently, we introduce the main part of Pure theory and proof commands, together with fundamental proof methods and attributes. Chapter 4 describes further Isar elements provided by generic tools and packages (such as the Simplifier) that are either part of Pure Isabelle or pre-installed in most object logics. Chapter 5 refers to object-logic specific elements (mainly for HOL and ZF).

Isar commands may be either *proper* document constructors, or *improper commands*. Some proof methods and attributes introduced later are classified as improper as well. Improper Isar language elements, which are subsequently marked by “*”, are often helpful when developing proof documents, while their use is discouraged for the final human-readable outcome. Typical examples are diagnostic commands that print terms or theorems according to the current context; other commands emulate old-style tactical theorem proving.

3.1 Theory commands

3.1.1 Defining theories

```

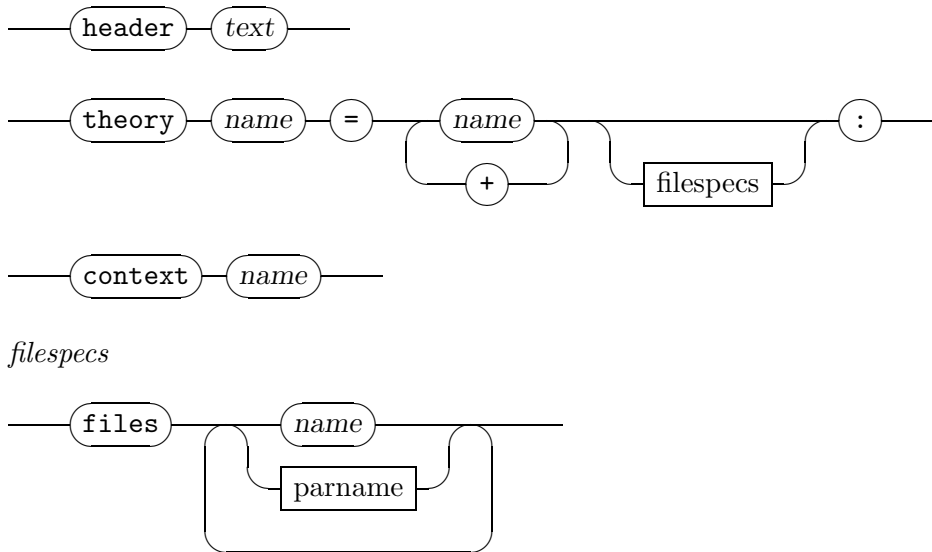
header  : toplevel → toplevel
theory  : toplevel → theory
context* : toplevel → theory
end     : theory → toplevel

```

Isabelle/Isar “new-style” theories are either defined via theory files or interactively. Both theory-level specifications and proofs are handled uniformly — occasionally definitional mechanisms even require some explicit proof as well. In contrast, “old-style” Isabelle theories support batch processing only, with the proof scripts collected in separate ML files.

The first “real” command of any theory has to be **theory**, which starts a new theory based on the merge of existing ones. Just preceding **theory**,

there may be an optional **header** declaration, which is relevant to document preparation only; it acts very much like a special pre-theory markup command (cf. §3.1.2 and §3.1.2). The **end** command concludes a theory development; it has to be the very last command of any theory file loaded in batch-mode. The theory context may be also changed interactively by **context** without creating a new theory.



header *text* provides plain text markup just preceding the formal beginning of a theory. In actual document preparation the corresponding \LaTeX macro `\isamarkupheader` may be redefined to produce chapter or section headings. See also §3.1.2 and §3.2.1 for further markup commands.

theory $A = B_1 + \dots + B_n$: starts a new theory A based on the merge of existing theories B_1, \dots, B_n .

Due to inclusion of several ancestors, the overall theory structure emerging in an Isabelle session forms a directed acyclic graph (DAG). Isabelle's theory loader ensures that the sources contributing to the development graph are always up-to-date. Changed files are automatically reloaded when processing theory headers interactively; batch-mode explicitly distinguishes `update_thy` from `use_thy`, see also [10].

The optional **files** specification declares additional dependencies on ML files. Files will be loaded immediately, unless the name is put in parentheses, which merely documents the dependency to be resolved later in the text (typically via explicit **use** in the body text, see §3.1.9). In reminiscence of the old-style theory system of Isabelle, $A.\text{thy}$ may be

also accompanied by an additional file *A.ML* consisting of ML code that is executed in the context of the *finished* theory *A*. That file should not be included in the **files** dependency declaration, though.

context *B* enters an existing theory context, basically in read-only mode, so only a limited set of commands may be performed without destroying the theory. Just as for **theory**, the theory loader ensures that *B* is loaded and up-to-date.

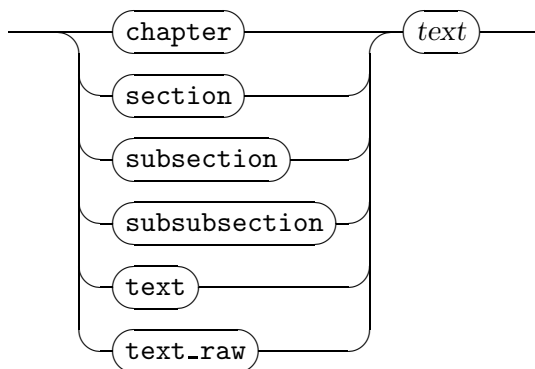
This command is occasionally useful for quick interactive experiments; normally one should always commence a new context via **theory**.

end concludes the current theory definition or context switch. Note that this command cannot be undone, but the whole theory definition has to be retracted.

3.1.2 Markup commands

chapter : *theory* \rightarrow *theory*
section : *theory* \rightarrow *theory*
subsection : *theory* \rightarrow *theory*
subsubsection : *theory* \rightarrow *theory*
text : *theory* \rightarrow *theory*
text_raw : *theory* \rightarrow *theory*

Apart from formal comments (see §2.2.2), markup commands provide a structured way to insert text into the document generated from a theory (see [18] for more information on Isabelle’s document preparation tools).



chapter, **section**, **subsection**, and **subsubsection** mark chapter and section headings.

text specifies paragraphs of plain text, including references to formal entities (see also §2.2.9 on “antiquotations”).

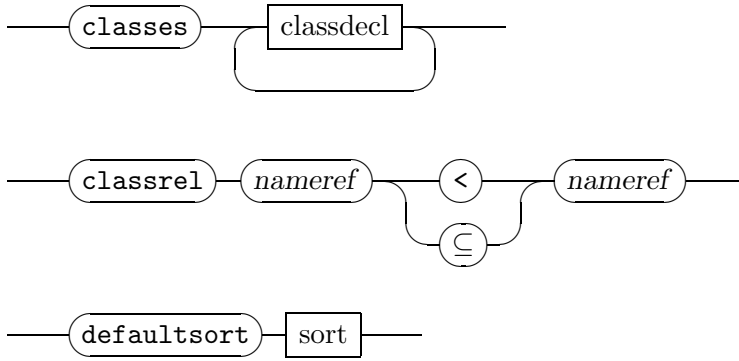
text_raw inserts \LaTeX source into the output, without additional markup. Thus the full range of document manipulations becomes available.

Any of these markup elements corresponds to a \LaTeX command with the name prefixed by `\isamarkup`. For the sectioning commands this is a plain macro with a single argument, e.g. `\isamarkupchapter{...}` for **chapter**. The **text** markup results in a \LaTeX environment `\begin{isamarkuptext} ... \end{isamarkuptext}`, while **text_raw** causes the text to be inserted directly into the \LaTeX source.

Additional markup commands are available for proofs (see §3.2.1). Also note that the **header** declaration (see §3.1.1) admits to insert section markup just preceding the actual theory definition.

3.1.3 Type classes and sorts

classes : $theory \rightarrow theory$
classrel : $theory \rightarrow theory$ (*axiomatic!*)
defaultsort : $theory \rightarrow theory$



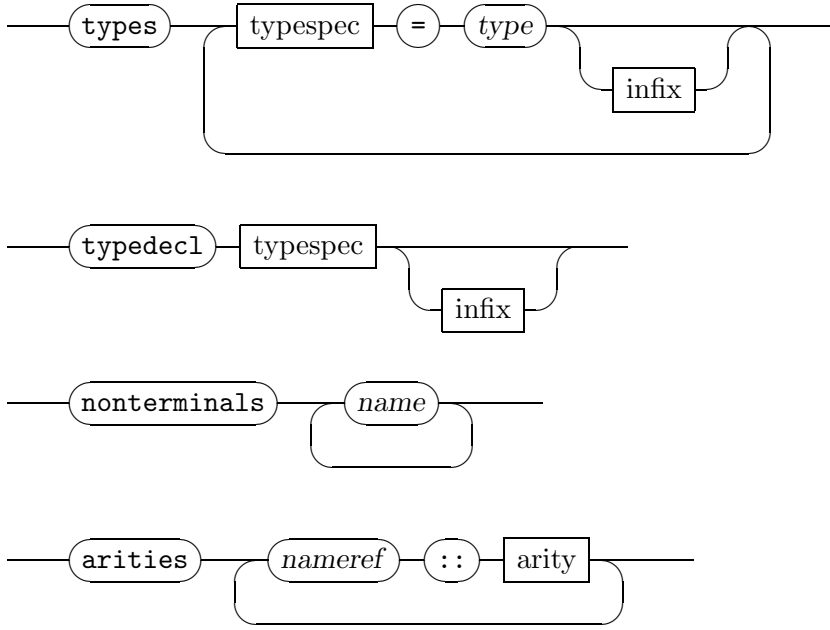
classes $c \subseteq \bar{c}$ declares class c to be a subclass of existing classes \bar{c} . Cyclic class structures are ruled out.

classrel $c_1 \subseteq c_2$ states a subclass relation between existing classes c_1 and c_2 . This is done axiomatically! The **instance** command (see §4.1.1) provides a way to introduce proven class relations.

defaultsort s makes sort s the new default sort for any type variables given without sort constraints. Usually, the default sort would be only changed when defining a new object-logic.

3.1.4 Primitive types and type abbreviations

$\mathbf{types} : theory \rightarrow theory$
 $\mathbf{typedecl} : theory \rightarrow theory$
 $\mathbf{nonterminals} : theory \rightarrow theory$
 $\mathbf{arities} : theory \rightarrow theory \text{ (axiomatic!)}$



types $(\bar{\alpha})t = \tau$ introduces *type synonym* $(\bar{\alpha})t$ for existing type τ . Unlike actual type definitions, as are available in Isabelle/HOL for example, type synonyms are just purely syntactic abbreviations without any logical significance. Internally, type synonyms are fully expanded.

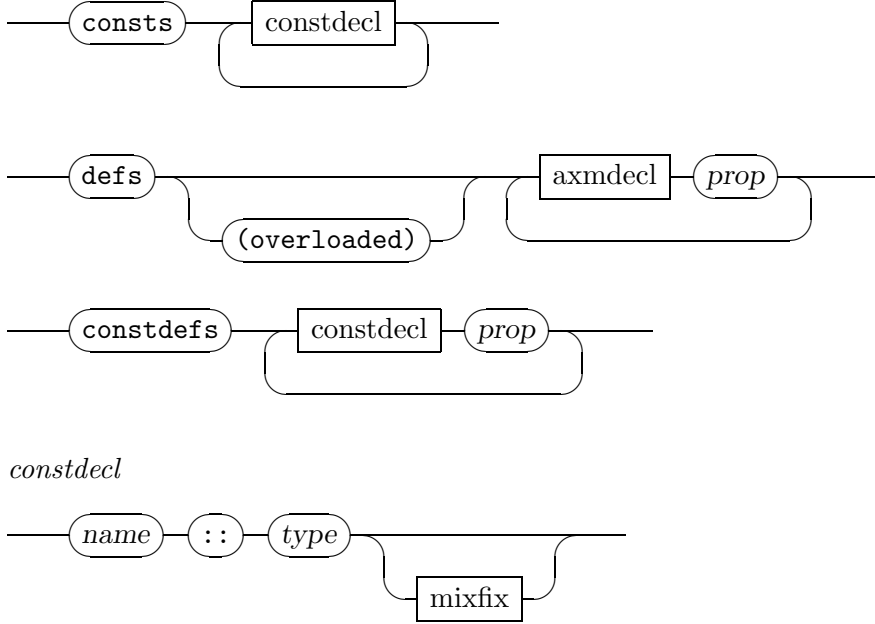
typedecl $(\bar{\alpha})t$ declares a new type constructor t , intended as an actual logical type. Note that the Isabelle/HOL object-logic overrides **typedecl** by its own version (§5.2.1).

nonterminals $\bar{\tau}$ declares 0-ary type constructors $\bar{\tau}$ to act as purely syntactic types, i.e. nonterminal symbols of Isabelle's inner syntax of terms or types.

arities $t :: (\bar{s})s$ augments Isabelle's order-sorted signature of types by new type constructor arities. This is done axiomatically! The **instance** command (see §4.1.1) provides a way to introduce proven type arities.

3.1.5 Constants and simple definitions

$\mathbf{consts} : theory \rightarrow theory$
 $\mathbf{defs} : theory \rightarrow theory$
 $\mathbf{constdefs} : theory \rightarrow theory$



consts $c :: \sigma$ declares constant c to have any instance of type scheme σ . The optional *mixfix* annotations may attach concrete syntax to the constants declared.

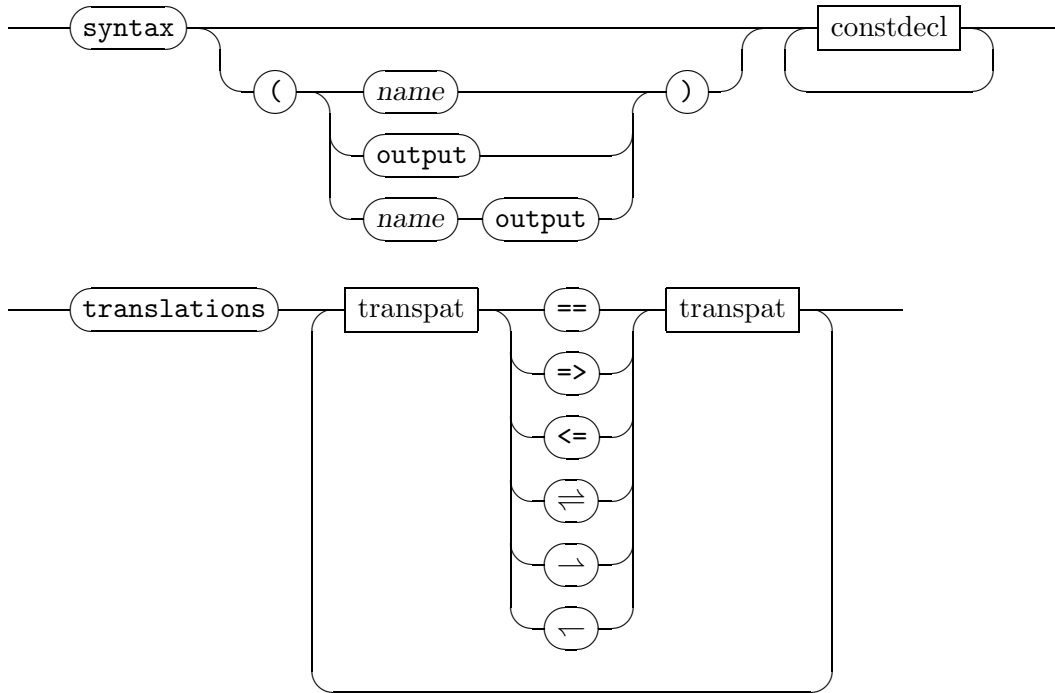
defs $name : eqn$ introduces eqn as a definitional axiom for some existing constant. See [10, §6] for more details on the form of equations admitted as constant definitions.

The *overloaded* option declares definitions to be potentially overloaded. Unless this option is given, a warning message would be issued for any definitional equation with a more special type than that of the corresponding constant declaration.

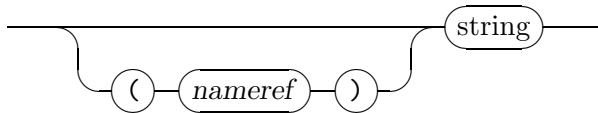
constdefs $c :: \sigma eqn$ combines declarations and definitions of constants, using the canonical name c_def for the definitional axiom.

3.1.6 Syntax and translations

$\mathbf{syntax} : theory \rightarrow theory$
 $\mathbf{translations} : theory \rightarrow theory$



transpat

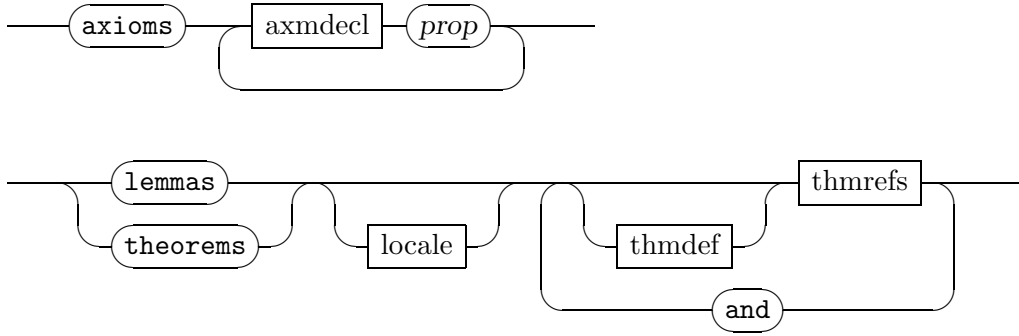


syntax (*mode*) *decls* is similar to **consts** *decls*, except that the actual logical signature extension is omitted. Thus the context free grammar of Isabelle's inner syntax may be augmented in arbitrary ways, independently of the logic. The *mode* argument refers to the print mode that the grammar rules belong; unless the **output** indicator is given, all productions are added both to the input and output grammar.

translations *rules* specifies syntactic translation rules (i.e. macros): parse / print rules (\rightleftharpoons), parse rules (\rightarrow), or print rules (\leftarrow). Translation patterns may be prefixed by the syntactic category to be used for parsing; the default is *logic*.

3.1.7 Axioms and theorems

axioms : *theory* \rightarrow *theory* (*axiomatic!*)
lemmas : *theory* \rightarrow *theory*
theorems : *theory* \rightarrow *theory*



axioms $a : \varphi$ introduces arbitrary statements as axioms of the meta-logic. In fact, axioms are “axiomatic theorems”, and may be referred later just as any other theorem.

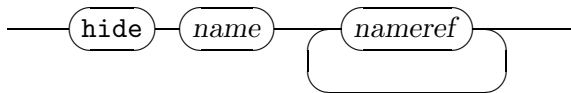
Axioms are usually only introduced when declaring new logical systems. Everyday work is typically done the hard way, with proper definitions and proven theorems.

lemmas $a = \bar{b}$ retrieves and stores existing facts in the theory context, or the specified locale (see also §4.1.2). Typical applications would also involve attributes, to declare Simplifier rules, for example.

theorems is essentially the same as **lemmas**, but marks the result as a different kind of facts.

3.1.8 Name spaces

global : $theory \rightarrow theory$
local : $theory \rightarrow theory$
hide : $theory \rightarrow theory$



Isabelle organizes any kind of name declarations (of types, constants, theorems etc.) by separate hierarchically structured name spaces. Normally the user does not have to control the behavior of name spaces by hand, yet the following commands provide some way to do so.

global and **local** change the current name declaration mode. Initially, theories start in **local** mode, causing all names to be automatically qualified

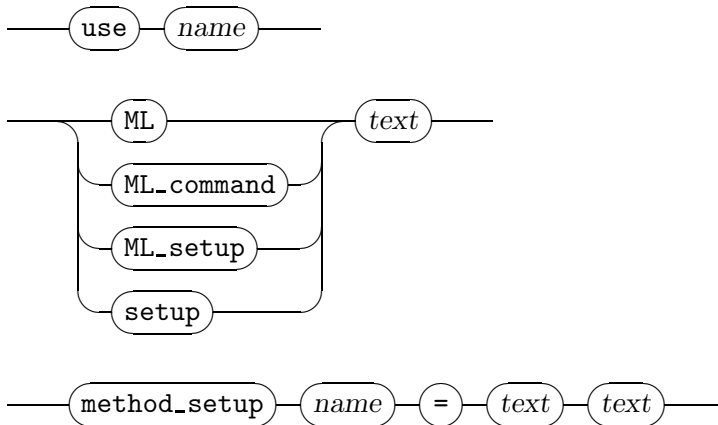
by the theory name. Changing this to **global** causes all names to be declared without the theory prefix, until **local** is declared again.

Note that global names are prone to get hidden accidentally later, when qualified names of the same base name are introduced.

hide *space names* removes declarations from a given name space (which may be *class*, *type*, or *const*). Hidden objects remain valid within the logic, but are inaccessible from user input. In output, the special qualifier “??” is prefixed to the full internal name. Unqualified (global) names may not be hidden.

3.1.9 Incorporating ML code

use : $\cdot \rightarrow \cdot$
ML : $\cdot \rightarrow \cdot$
ML_command : $\cdot \rightarrow \cdot$
ML_setup : *theory* \rightarrow *theory*
setup : *theory* \rightarrow *theory*
method_setup : *theory* \rightarrow *theory*



use *file* reads and executes ML commands from *file*. The current theory context (if present) is passed down to the ML session, but may not be modified. Furthermore, the file name is checked with the **files** dependency declaration given in the theory header (see also §3.1.1).

ML *text* and **ML_command** *text* execute ML commands from *text*. The theory context is passed in the same way as for **use**, but may not be changed. Note that the output of **ML_command** is less verbose than plain **ML**.

ML_setup *text* executes ML commands from *text*. The theory context is passed down to the ML session, and fetched back afterwards. Thus *text* may actually change the theory as a side effect.

setup *text* changes the current theory context by applying *text*, which refers to an ML expression of type `(theory -> theory) list`. The **setup** command is the canonical way to initialize any object-logic specific tools and packages written in ML.

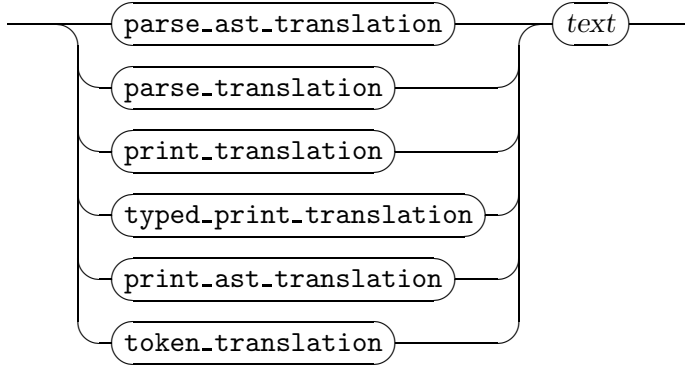
method_setup *name = text description* defines a proof method in the current theory. The given *text* has to be an ML expression of type `Args.src -> Proof.context -> Proof.method`. Parsing concrete method syntax from `Args.src` input can be quite tedious in general. The following simple examples are for methods without any explicit arguments, or a list of theorems, respectively.

```
Method.no_args (Method.METHOD (fn facts => foobar_tac))
Method.thms_args (fn thms => Method.METHOD (fn facts => foobar_tac))
Method.ctxt_args (fn ctxt => Method.METHOD (fn facts => foobar_tac))
Method.thms_ctxt_args (fn thms => fn ctxt =>
  Method.METHOD (fn facts => foobar_tac))
```

Note that mere tactic emulations may ignore the `facts` parameter above. Proper proof methods would do something appropriate with the list of current facts, though. Single-rule methods usually do strict forward-chaining (e.g. by using `Method.multi_resolves`), while automatic ones just insert the facts using `Method.insert_tac` before applying the main tactic.

3.1.10 Syntax translation functions

```
parse_ast_translation : theory -> theory
  parse_translation   : theory -> theory
    print_translation  : theory -> theory
typed_print_translation : theory -> theory
  print_ast_translation : theory -> theory
    token_translation  : theory -> theory
```



Syntax translation functions written in ML admit almost arbitrary manipulations of Isabelle’s inner syntax. Any of the above commands have a single *text* argument that refers to an ML expression of appropriate type.

```

val parse_ast_translation  : (string * (ast list -> ast)) list
val parse_translation      : (string * (term list -> term)) list
val print_translation      : (string * (term list -> term)) list
val typed_print_translation :
  (string * (bool -> typ -> term list -> term)) list
val print_ast_translation  : (string * (ast list -> ast)) list
val token_translation      :
  (string * string * (string -> string * real)) list

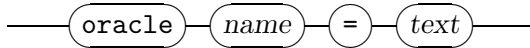
```

See [10, §8] for more information on syntax transformations.

3.1.11 Oracles

oracle : *theory* → *theory*

Oracles provide an interface to external reasoning systems, without giving up control completely — each theorem carries a derivation object recording any oracle invocation. See [10, §6] for more information.



oracle *name* = *text* declares oracle *name* to be ML function *text*, which has to be of type `Sign.sg * Object.T -> term`.

3.2 Proof commands

Proof commands perform transitions of Isar/VM machine configurations, which are block-structured, consisting of a stack of nodes with three main components: logical proof context, current facts, and open goals. Isar/VM transitions are *typed* according to the following three different modes of operation:

proof(prove) means that a new goal has just been stated that is now to be *proven*; the next command may refine it by some proof method, and enter a sub-proof to establish the actual result.

proof(state) is like a nested theory mode: the context may be augmented by *stating* additional assumptions, intermediate results etc.

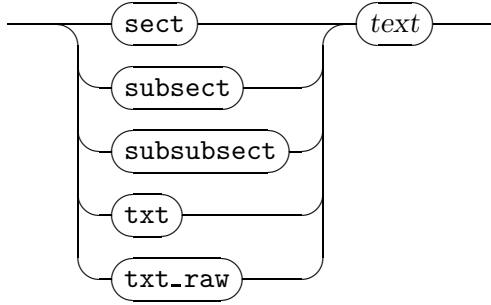
proof(chain) is intermediate between *proof(state)* and *proof(prove)*: existing facts (i.e. the contents of the special “*this*” register) have been just picked up in order to be used when refining the goal claimed next.

The proof mode indicator may be read as a verb telling the writer what kind of operation may be performed next. The corresponding typings of proof commands restricts the shape of well-formed proof texts to particular command sequences. So dynamic arrangements of commands eventually turn out as static texts of a certain structure. Appendix A gives a simplified grammar of the overall (extensible) language emerging that way.

3.2.1 Markup commands

sect	:	<i>proof</i>	\rightarrow	<i>proof</i>
subsect	:	<i>proof</i>	\rightarrow	<i>proof</i>
subsubsect	:	<i>proof</i>	\rightarrow	<i>proof</i>
txt	:	<i>proof</i>	\rightarrow	<i>proof</i>
txt_raw	:	<i>proof</i>	\rightarrow	<i>proof</i>

These markup commands for proof mode closely correspond to the ones of theory mode (see §3.1.2).



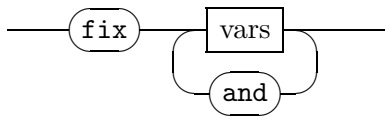
3.2.2 Context elements

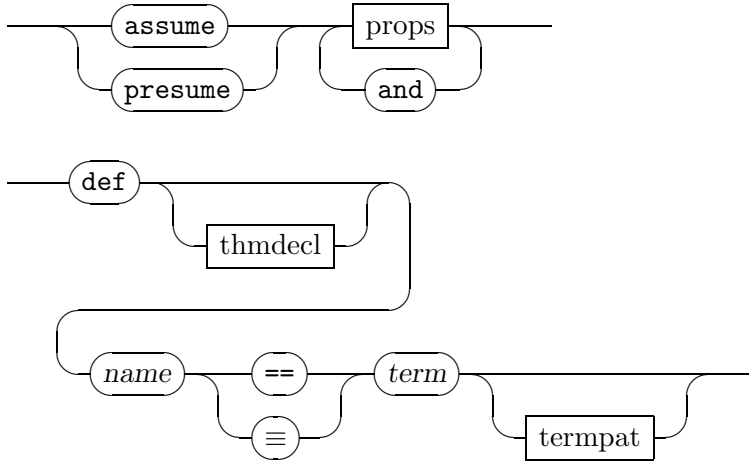
$\mathbf{fix} : proof(state) \rightarrow proof(state)$
 $\mathbf{assume} : proof(state) \rightarrow proof(state)$
 $\mathbf{presume} : proof(state) \rightarrow proof(state)$
 $\mathbf{def} : proof(state) \rightarrow proof(state)$

The logical proof context consists of fixed variables and assumptions. The former closely correspond to Skolem constants, or meta-level universal quantification as provided by the Isabelle/Pure logical framework. Introducing some *arbitrary, but fixed* variable via “**fix** x ” results in a local value that may be used in the subsequent proof as any other variable or constant. Furthermore, any result $\vdash \varphi[x]$ exported from the context will be universally closed wrt. x at the outermost level: $\vdash \bigwedge x. \varphi$ (this is expressed using Isabelle’s meta-variables).

Similarly, introducing some assumption χ has two effects. On the one hand, a local theorem is created that may be used as a fact in subsequent proof steps. On the other hand, any result $\chi \vdash \varphi$ exported from the context becomes conditional wrt. the assumption: $\vdash \chi \implies \varphi$. Thus, solving an enclosing goal using such a result would basically introduce a new subgoal stemming from the assumption. How this situation is handled depends on the actual version of assumption command used: while **assume** insists on solving the subgoal by unification with some premise of the goal, **presume** leaves the subgoal unchanged in order to be proved later by the user.

Local definitions, introduced by “**def** $x \equiv t$ ”, are achieved by combining “**fix** x ” with another version of assumption that causes any hypothetical equation $x \equiv t$ to be eliminated by the reflexivity rule. Thus, exporting some result $x \equiv t \vdash \varphi[x]$ yields $\vdash \varphi[t]$.





fix \bar{x} introduces local *arbitrary, but fixed* variables \bar{x} .

assume a : $\bar{\varphi}$ and **presume** a : $\bar{\varphi}$ introduce local theorems $\bar{\varphi}$ by assumption. Subsequent results applied to an enclosing goal (e.g. by **show**) are handled as follows: **assume** expects to be able to unify with existing premises in the goal, while **presume** leaves $\bar{\varphi}$ as new subgoals.

Several lists of assumptions may be given (separated by **and**); the resulting list of current facts consists of all of these concatenated.

def a : $x \equiv t$ introduces a local (non-polymorphic) definition. In results exported from the context, x is replaced by t . Basically, “**def** $x \equiv t$ ” abbreviates “**fix** x **assume** $x \equiv t$ ”, with the resulting hypothetical equation solved by reflexivity.

The default name for the definitional equation is x_def .

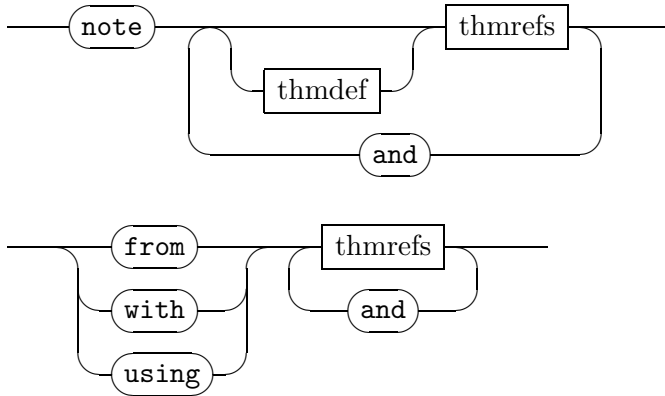
The special name *prems* refers to all assumptions of the current context as a list of theorems.

3.2.3 Facts and forward chaining

note : $proof(state) \rightarrow proof(state)$
then : $proof(state) \rightarrow proof(chain)$
from : $proof(state) \rightarrow proof(chain)$
with : $proof(state) \rightarrow proof(chain)$
using : $proof(prove) \rightarrow proof(prove)$

New facts are established either by assumption or proof of local statements. Any fact will usually be involved in further proofs, either as explicit

arguments of proof methods, or when forward chaining towards the next goal via **then** (and variants); **from** and **with** are composite forms involving **note**. The **using** element augments the collection of used facts *after* a goal has been stated. Note that the special theorem name *this* refers to the most recently established facts, but only *before* issuing a follow-up claim.



note $a = \bar{b}$ recalls existing facts \bar{b} , binding the result as a . Note that attributes may be involved as well, both on the left and right hand sides.

then indicates forward chaining by the current facts in order to establish the goal to be claimed next. The initial proof method invoked to refine that will be offered the facts to do “anything appropriate” (see also §3.2.5). For example, method *rule* (see §3.2.6) would typically do an elimination rather than an introduction. Automatic methods usually insert the facts into the goal state before operation. This provides a simple scheme to control relevance of facts in automated proof search.

from \bar{b} abbreviates “**note** \bar{b} **then**”; thus **then** is equivalent to “**from** *this*”.

with \bar{b} abbreviates “**from** \bar{b} **and** *this*”; thus the forward chaining is from earlier facts together with the current ones.

using \bar{b} augments the facts being currently indicated for use by a subsequent refinement step (such as **apply** or **proof**).

Forward chaining with an empty list of theorems is the same as not chaining at all. Thus “**from** *nothing*” has no effect apart from entering *prove(chain)* mode, since *nothing* is bound to the empty list of theorems.

Basic proof methods (such as *rule*) expect multiple facts to be given in their proper order, corresponding to a prefix of the premises of the rule involved. Note that positions may be easily skipped using something like

from $_ a b$, for example. This involves the trivial rule $\text{PROP } \psi \implies \text{PROP } \psi$, which happens to be bound in Isabelle/Pure as “ $_$ ” (underscore).

Automated methods (such as *simp* or *auto*) just insert any given facts before their usual operation. Depending on the kind of procedure involved, the order of facts is less significant here.

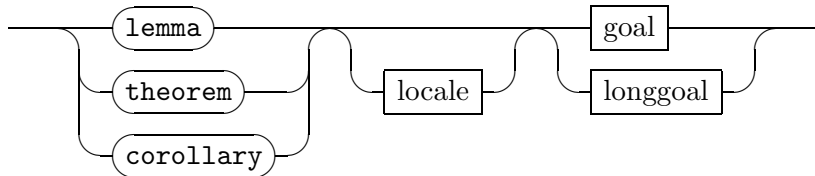
3.2.4 Goal statements

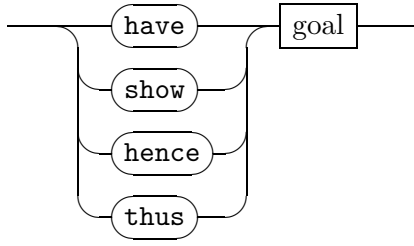
lemma : $theory \rightarrow proof(prove)$
theorem : $theory \rightarrow proof(prove)$
corollary : $theory \rightarrow proof(prove)$
have : $proof(state) \mid proof(chain) \rightarrow proof(prove)$
show : $proof(state) \mid proof(chain) \rightarrow proof(prove)$
hence : $proof(state) \rightarrow proof(prove)$
thus : $proof(state) \rightarrow proof(prove)$

From a theory context, proof mode is entered by an initial goal command such as **lemma**, **theorem**, or **corollary**. Within a proof, new claims may be introduced locally as well; four variants are available here to indicate whether forward chaining of facts should be performed initially (via **then**), and whether the final result is meant to solve some pending goal.

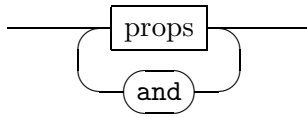
Goals may consist of multiple statements, resulting in a list of facts eventually. A pending multi-goal is internally represented as a meta-level conjunction (printed as $\&\&$), which is usually split into the corresponding number of sub-goals prior to an initial method application, via **proof** (§3.2.5) or **apply** (§3.2.9). The *induct* method covered in §4.3.5 acts on multiple claims simultaneously.

Claims at the theory level may be either in short or long form. A short goal merely consists of several simultaneous propositions (often just one). A long goal includes an explicit context specification for the subsequent conclusions, involving local parameters; here the role of each part of the statement is explicitly marked by separate keywords (see also §4.1.2).

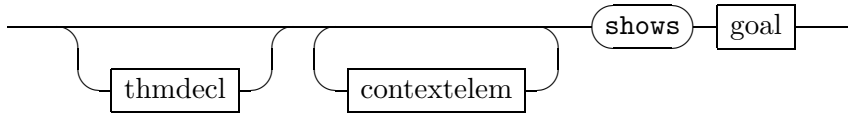




goal



longgoal



lemma a : φ enters proof mode with φ as main goal, eventually resulting in some fact $\vdash \varphi$ to be put back into the theory context, or into the specified locale (cf. §4.1.2). An additional *context* specification may build up an initial proof context for the subsequent claim; this includes local definitions and syntax as well, see the definition of *contextelem* in §4.1.2.

theorem a : φ and **corollary** a : φ are essentially the same as **lemma** a : φ , but the facts are internally marked as being of a different kind. This discrimination acts like a formal comment.

have a : φ claims a local goal, eventually resulting in a fact within the current logical context. This operation is completely independent of any pending sub-goals of an enclosing goal statements, so **have** may be freely used for experimental exploration of potential results within a proof body.

show a : φ is like **have** a : φ plus a second stage to refine some pending sub-goal for each one of the finished result, after having been exported into the corresponding context (at the head of the sub-proof of this **show** command).

To accommodate interactive debugging, resulting rules are printed before being applied internally. Even more, interactive execution of **show**

predicts potential failure and displays the resulting error as a warning beforehand. Watch out for the following message:

Problem! Local statement will fail to solve any pending goal

hence abbreviates “**then have**”, i.e. claims a local goal to be proven by forward chaining the current facts. Note that **hence** is also equivalent to “**from this have**”.

thus abbreviates “**then show**”. Note that **thus** is also equivalent to “**from this show**”.

Any goal statement causes some term abbreviations (such as *?thesis*) to be bound automatically, see also §3.2.7. Furthermore, the local context of a (non-atomic) goal is provided via the *rule_context* case.

! Isabelle/Isar suffers theory-level goal statements to contain *unbound schematic variables*, although this does not conform to the aim of human-readable proof documents! The main problem with schematic goals is that the actual outcome is usually hard to predict, depending on the behavior of the proof methods applied during the course of reasoning. Note that most semi-automated methods heavily depend on several kinds of implicit rule declarations within the current theory context. As this would also result in non-compositional checking of sub-proofs, *local goals* are not allowed to be schematic at all. Nevertheless, schematic goals do have their use in Prolog-style interactive synthesis of proven results, usually by stepwise refinement via emulation of traditional Isabelle tactic scripts (see also §3.2.9). In any case, users should know what they are doing.

3.2.5 Initial and terminal proof steps

proof : $proof(prove) \rightarrow proof(state)$
qed : $proof(state) \rightarrow proof(state) \mid theory$
by : $proof(prove) \rightarrow proof(state) \mid theory$
.. : $proof(prove) \rightarrow proof(state) \mid theory$
. : $proof(prove) \rightarrow proof(state) \mid theory$
sorry : $proof(prove) \rightarrow proof(state) \mid theory$

Arbitrary goal refinement via tactics is considered harmful. Properly, the Isar framework admits proof methods to be invoked in two places only.

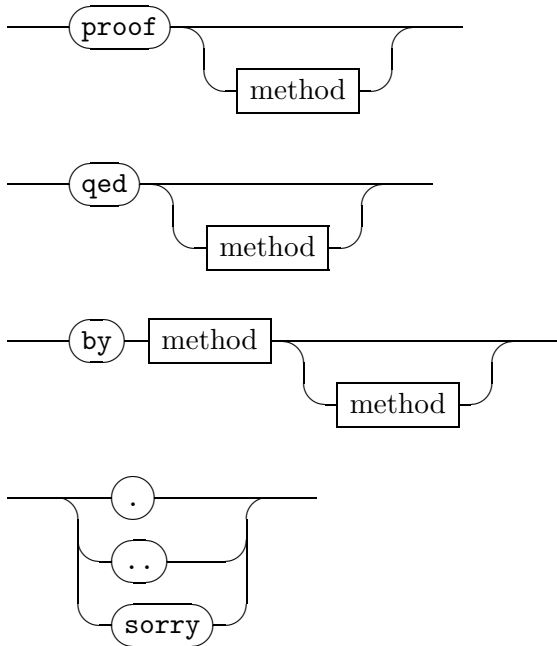
1. An *initial* refinement step **proof** m_1 reduces a newly stated goal to a number of sub-goals that are to be solved later. Facts are passed to m_1 for forward chaining, if so indicated by *proof(chain)* mode.

2. A *terminal* conclusion step **qed** m_2 is intended to solve remaining goals. No facts are passed to m_2 .

The only other (proper) way to affect pending goals in a proof body is by **show**, which involves an explicit statement of what is to be solved eventually. Thus we avoid the fundamental problem of unstructured tactic scripts that consist of numerous consecutive goal transformations, with invisible effects.

As a general rule of thumb for good proof style, initial proof methods should either solve the goal completely, or constitute some well-understood reduction to new sub-goals. Arbitrary automatic proof tools that are prone leave a large number of badly structured sub-goals are no help in continuing the proof document in an intelligible manner.

Unless given explicitly by the user, the default initial method is “*rule*”, which applies a single standard elimination or introduction rule according to the topmost symbol involved. There is no separate default terminal method. Any remaining goals are always solved by assumption in the very last step.



proof m_1 refines the goal by proof method m_1 ; facts for forward chaining are passed if so indicated by *proof(chain)* mode.

qed m_2 refines any remaining goals by proof method m_2 and concludes the sub-proof by assumption. If the goal had been **show** (or **thus**), some pending sub-goal is solved as well by the rule resulting from the result

exported into the enclosing goal context. Thus **qed** may fail for two reasons: either m_2 fails, or the resulting rule does not fit to any pending goal¹ of the enclosing context. Debugging such a situation might involve temporarily changing **show** into **have**, or weakening the local context by replacing occurrences of **assume** by **presume**.

by m_1 m_2 is a *terminal proof*; it abbreviates **proof** m_1 **qed** m_2 , but with backtracking across both methods. Debugging an unsuccessful **by** m_1 m_2 commands might be done by expanding its definition; in many cases **proof** m_1 (or even **apply** m_1) is already sufficient to see the problem.

“**..**” is a *default proof*; it abbreviates **by rule**.

“**.**” is a *trivial proof*; it abbreviates **by this**.

sorry is a *fake proof* pretending to solve the pending claim without further ado. This only works in interactive development, or if the **quick_and_dirty** flag is enabled. Facts emerging from fake proofs are not the real thing. Internally, each theorem container is tainted by an oracle invocation, which is indicated as “[!]” in the printed result.

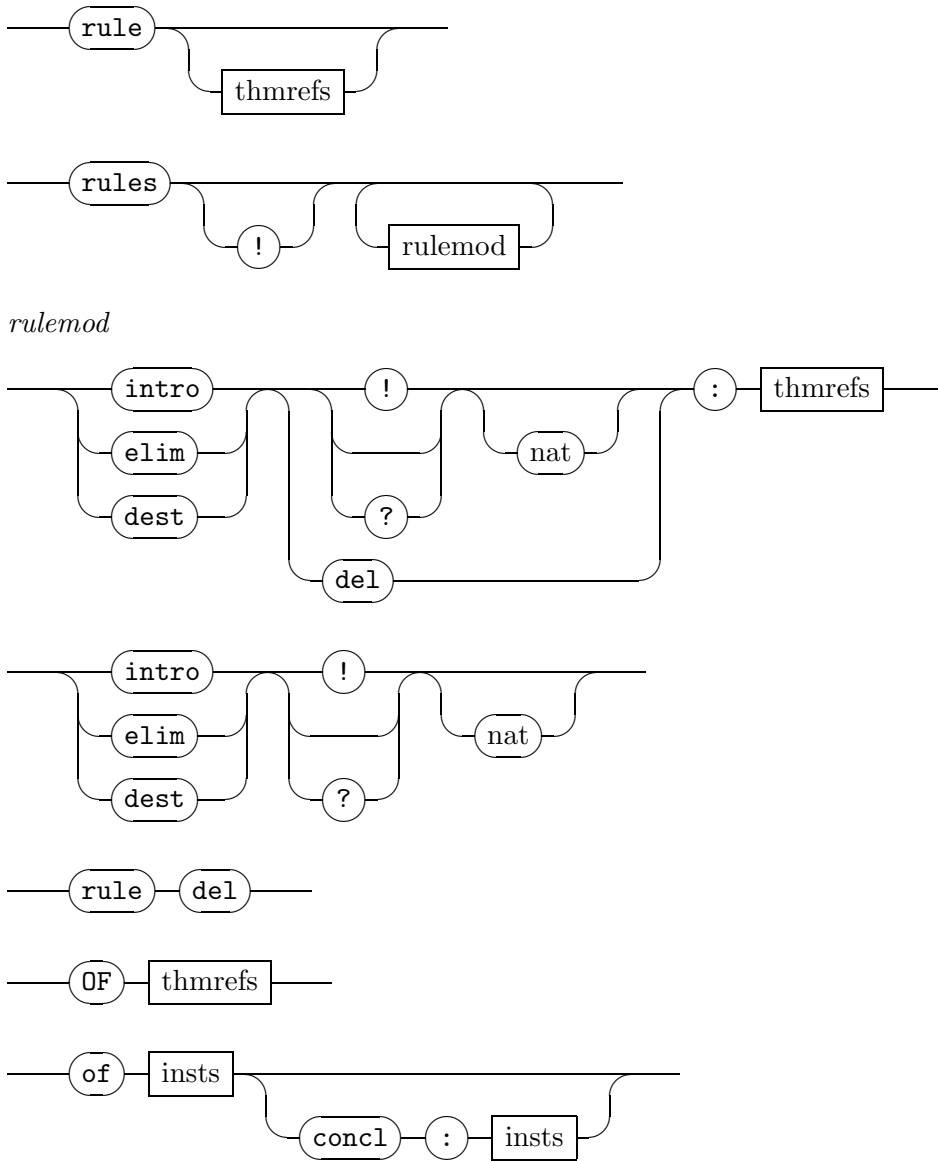
The most important application of **sorry** is to support experimentation and top-down proof development.

3.2.6 Fundamental methods and attributes

The following proof methods and attributes refer to basic logical operations of Isar. Further methods and attributes are provided by several generic and object-logic specific tools and packages (see chapters 4 and 5).

—	: <i>method</i>
<i>assumption</i>	: <i>method</i>
<i>this</i>	: <i>method</i>
<i>rule</i>	: <i>method</i>
<i>rules</i>	: <i>method</i>
<i>intro</i>	: <i>attribute</i>
<i>elim</i>	: <i>attribute</i>
<i>dest</i>	: <i>attribute</i>
<i>rule</i>	: <i>attribute</i>
<i>OF</i>	: <i>attribute</i>
<i>of</i>	: <i>attribute</i>

¹This includes any additional “strong” assumptions as introduced by **assume**.



“`—`” does nothing but insert the forward chaining facts as premises into the goal. Note that command **proof** without any method actually performs a single reduction step using the *rule* method; thus a plain *do-nothing* proof step would be “**proof** `—`” rather than **proof** alone.

assumption solves some goal by a single assumption step. All given facts are guaranteed to participate in the refinement; this means there may be only 0 or 1 in the first place. Recall that **qed** (see §3.2.5) already concludes any remaining sub-goals by assumption, so structured proofs usually need not quote the *assumption* method at all.

this applies all of the current facts directly as rules. Recall that “.” (dot) abbreviates “**by this**”.

rule \bar{a} applies some rule given as argument in backward manner; facts are used to reduce the rule before applying it to the goal. Thus *rule* without facts is plain introduction, while with facts it becomes elimination.

When no arguments are given, the *rule* method tries to pick appropriate rules automatically, as declared in the current context using the *intro*, *elim*, *dest* attributes (see below). This is the default behavior of **proof** and “.” (double-dot) steps (see §3.2.5).

rules performs intuitionistic proof search, depending on specifically declared rules from the context, or given as explicit arguments. Chained facts are inserted into the goal before commencing proof search; “*rules!*” means to include the current *prems* as well.

Rules need to be classified as *intro*, *elim*, or *dest*; here the “!” indicator refers to “safe” rules, which may be applied aggressively (without considering back-tracking later). Rules declared with “?” are ignored in proof search (the single-step *rule* method still observes these). An explicit weight annotation may be given as well; otherwise the number of rule premises will be taken into account here.

intro, *elim*, and *dest* declare introduction, elimination, and destruct rules, to be used with the *rule* and *rules* methods. Note that the latter will ignore rules declared with “?”, while “!” are used most aggressively.

The classical reasoner (see §4.3.4) introduces its own variants of these attributes; use qualified names to access the present versions of Isabelle/Pure, i.e. *Pure.intro* or *CPure.intro*.

rule del undeclares introduction, elimination, or destruct rules.

OF \bar{a} applies some theorem to given rules \bar{a} (in parallel). This corresponds to the MRS operator in ML [10, §5], but note the reversed order. Positions may be effectively skipped by including “_” (underscore) as argument.

of \bar{t} performs positional instantiation. The terms \bar{t} are substituted for any schematic variables occurring in a theorem from left to right; “_” (underscore) indicates to skip a position. Arguments following a “*concl:*” specification refer to positions of the conclusion of a rule.

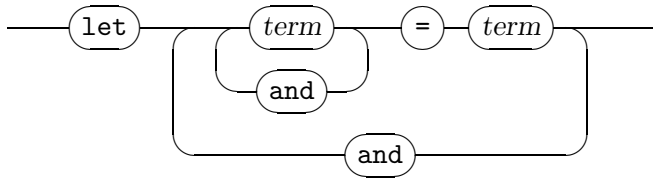
3.2.7 Term abbreviations

let : $proof(state) \rightarrow proof(state)$
is : *syntax*

Abbreviations may be either bound by explicit **let** $p \equiv t$ statements, or by annotating assumptions or goal statements with a list of patterns “(**is** $p_1 \dots$ **is** p_n)”. In both cases, higher-order matching is invoked to bind extra-logical term variables, which may be either named schematic variables of the form $?x$, or nameless dummies “_” (underscore). Note that in the **let** form the patterns occur on the left-hand side, while the **is** patterns are in postfix position.

Polymorphism of term bindings is handled in Hindley-Milner style, similar to ML. Type variables referring to local assumptions or open goal statements are *fixed*, while those of finished results or bound by **let** may occur in *arbitrary* instances later. Even though actual polymorphism should be rarely used in practice, this mechanism is essential to achieve proper incremental type-inference, as the user proceeds to build up the Isar proof text from left to right.

Term abbreviations are quite different from local definitions as introduced via **def** (see §3.2.2). The latter are visible within the logic as actual equations, while abbreviations disappear during the input process just after type checking. Also note that **def** does not support polymorphism.



The syntax of **is** patterns follows *termpat* or *proppat* (see §2.2.8).

let $\bar{p} = \bar{t}$ binds any text variables in patterns \bar{p} by simultaneous higher-order matching against terms \bar{t} .

(**is** \bar{p}) resembles **let**, but matches \bar{p} against the preceding statement. Also note that **is** is not a separate command, but part of others (such as **assume**, **have** etc.).

Some *automatic* term abbreviations for goals and facts are available as well. For any open goal, *?thesis* refers to its object-level statement, abstracted over any meta-level parameters (if present). Likewise, *?this* is bound for fact

statements resulting from assumptions or finished goals. In case *?this* refers to an object-logic statement that is an application $f(t)$, then t is bound to the special text variable “...” (three dots). The canonical application of the latter are calculational proofs (see §4.2.2).

3.2.8 Block structure

```

next  :  $proof(state) \rightarrow proof(state)$ 
    {   :  $proof(state) \rightarrow proof(state)$ 
    }   :  $proof(state) \rightarrow proof(state)$ 

```

While Isar is inherently block-structured, opening and closing blocks is mostly handled rather casually, with little explicit user-intervention. Any local goal statement automatically opens *two* blocks, which are closed again when concluding the sub-proof (by **qed** etc.). Sections of different context within a sub-proof may be switched via **next**, which is just a single block-close followed by block-open again. The effect of **next** is to reset the local proof context; there is no goal focus involved here!

For slightly more advanced applications, there are explicit block parentheses as well. These typically achieve a stronger forward style of reasoning.

next switches to a fresh block within a sub-proof, resetting the local context to the initial one.

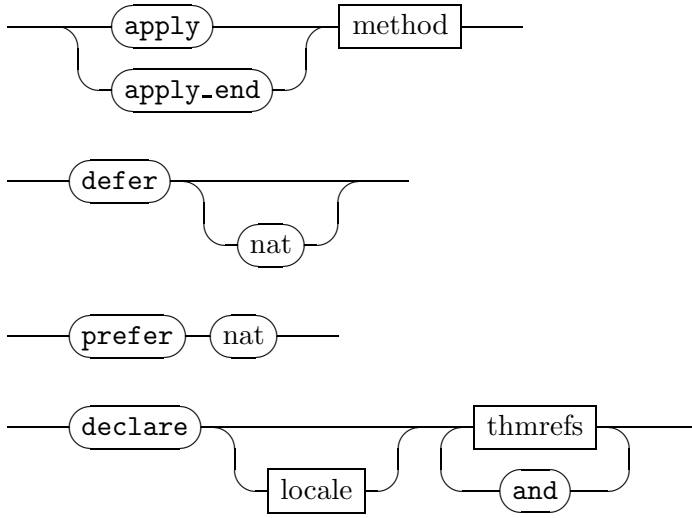
{ and } explicitly open and close blocks. Any current facts pass through “{” unchanged, while “}” causes any result to be *exported* into the enclosing context. Thus fixed variables are generalized, assumptions discharged, and local definitions unfolded (cf. §3.2.2). There is no difference of **assume** and **presume** in this mode of forward reasoning — in contrast to plain backward reasoning with the result exported at **show** time.

3.2.9 Emulating tactic scripts

The Isar provides separate commands to accommodate tactic-style proof scripts within the same system. While being outside the orthodox Isar proof language, these might come in handy for interactive exploration and debugging, or even actual tactical proof within new-style theories (to benefit from document preparation, for example). See also §4.3.2 for actual tactics, that have been encapsulated as proof methods. Proper proof methods may be

used in scripts, too.

$\mathbf{apply}^* : proof(prove) \rightarrow proof(prove)$
 $\mathbf{apply_end}^* : proof(state) \rightarrow proof(state)$
 $\mathbf{done}^* : proof(prove) \rightarrow proof(state)$
 $\mathbf{defer}^* : proof \rightarrow proof$
 $\mathbf{prefer}^* : proof \rightarrow proof$
 $\mathbf{back}^* : proof \rightarrow proof$
 $\mathbf{declare}^* : theory \rightarrow theory$



apply m applies proof method m in initial position, but unlike **proof** it retains “*proof(prove)*” mode. Thus consecutive method applications may be given just as in tactic scripts.

Facts are passed to m as indicated by the goal’s forward-chain mode, and are *consumed* afterwards. Thus any further **apply** command would always work in a purely backward manner.

apply_end (m) applies proof method m as if in terminal position. Basically, this simulates a multi-step tactic script for **qed**, but may be given anywhere within the proof body.

No facts are passed to m . Furthermore, the static context is that of the enclosing goal (as for actual **qed**). Thus the proof method may not refer to any assumptions introduced in the current body, for example.

done completes a proof script, provided that the current goal state is solved completely. Note that actual structured proof commands (e.g. “.” or **sorry**) may be used to conclude proof scripts as well.

defer n and **prefer** n shuffle the list of pending goals: *defer* puts off goal n to the end of the list ($n = 1$ by default), while *prefer* brings goal n to the top.

back does back-tracking over the result sequence of the latest proof command. Basically, any proof command may return multiple results.

declare *thms* declares theorems to the current theory context (or the specified locale, see also §4.1.2). No theorem binding is involved here, unlike **theorems** or **lemmas** (cf. §3.1.7), so **declare** only has the effect of applying attributes as included in the theorem specification.

Any proper Isar proof method may be used with tactic script commands such as **apply**. A few additional emulations of actual tactics are provided as well; these would be never used in actual structured proofs, of course.

3.2.10 Meta-linguistic features

oops : $proof \rightarrow theory$

The **oops** command discontinues the current proof attempt, while considering the partial proof text as properly processed. This is conceptually quite different from “faking” actual proofs via **sorry** (see §3.2.5): **oops** does not observe the proof structure at all, but goes back right to the theory level. Furthermore, **oops** does not produce any result theorem — there is no intended claim to be able to complete the proof anyhow.

A typical application of **oops** is to explain Isar proofs *within* the system itself, in conjunction with the document preparation tools of Isabelle described in [18]. Thus partial or even wrong proof attempts can be discussed in a logically sound manner. Note that the Isabelle L^AT_EX macros can be easily adapted to print something like “...” instead of an “**oops**” keyword.

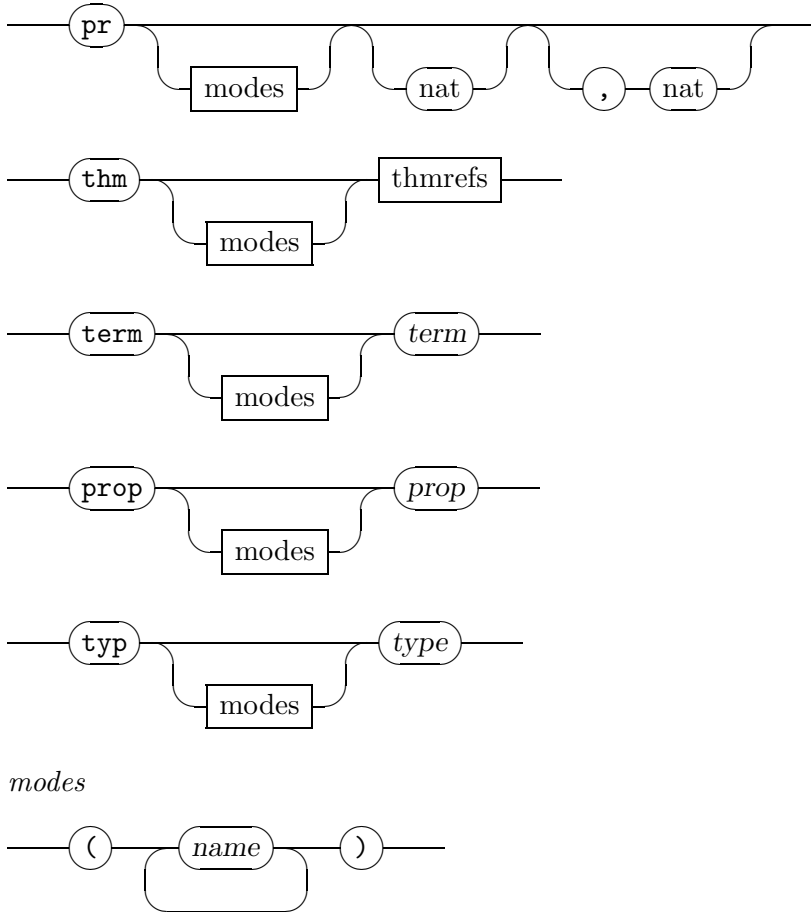
The **oops** command is undo-able, unlike **kill** (see §3.3.3). The effect is to get back to the theory just before the opening of the proof.

3.3 Other commands

3.3.1 Diagnostics

\mathbf{pr}^* : $\cdot \rightarrow \cdot$
 \mathbf{thm}^* : $theory \mid proof \rightarrow theory \mid proof$
 \mathbf{term}^* : $theory \mid proof \rightarrow theory \mid proof$
 \mathbf{prop}^* : $theory \mid proof \rightarrow theory \mid proof$
 \mathbf{typ}^* : $theory \mid proof \rightarrow theory \mid proof$

These diagnostic commands assist interactive development. Note that *undo* does not apply here, the theory or proof configuration is not changed.



pr goals, prems prints the current proof state (if present), including the proof context, current facts and goals. The optional limit arguments affect the number of goals and premises to be displayed, which is initially 10 for both. Omitting limit values leaves the current setting unchanged.

thm \bar{a} retrieves theorems from the current theory or proof context. Note that any attributes included in the theorem specifications are applied to a temporary context derived from the current theory or proof; the result is discarded, i.e. attributes involved in \bar{a} do not have any permanent effect.

term t and **prop** φ read, type-check and print terms or propositions according to the current theory or proof context; the inferred type of t is output as well. Note that these commands are also useful in inspecting the current environment of term abbreviations.

typ τ reads and prints types of the meta-logic according to the current theory or proof context.

All of the diagnostic commands above admit a list of *modes* to be specified, which is appended to the current print mode (see also [10]). Thus the output behavior may be modified according particular print mode features. For example, **pr** (*latex xsymbols symbols*) would print the current proof state with mathematical symbols and special characters represented in L^AT_EX source, according to the Isabelle style [18].

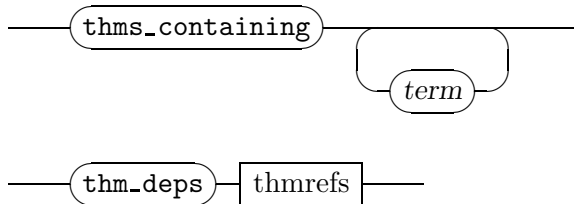
Note that antiquotations (cf. §2.2.9) provide a more systematic way to include formal items into the printed text document.

3.3.2 Inspecting the context

```

print_commands* : . → .
print_syntax*   : theory | proof → theory | proof
print_methods* : theory | proof → theory | proof
print_attributes* : theory | proof → theory | proof
print_theorems* : theory | proof → theory | proof
thms_containing* : theory | proof → theory | proof
thms_deps*      : theory | proof → theory | proof
print_facts*    : proof → proof
print_binds*    : proof → proof

```



These commands print certain parts of the theory and proof context. Note that there are some further ones available, such as for the set of rules declared for simplifications.

print_commands prints Isabelle’s outer theory syntax, including keywords and command.

print_syntax prints the inner syntax of types and terms, depending on the current context. The output can be very verbose, including grammar tables and syntax translation rules. See [10, §7, §8] for further information on Isabelle’s inner syntax.

print_methods prints all proof methods available in the current theory context.

print_attributes prints all attributes available in the current theory context.

print_theorems prints theorems available in the current theory context.

In interactive mode this actually refers to the theorems left by the last transaction; this allows to inspect the result of advanced definitional packages, such as **datatype**.

thms_containing \bar{t} retrieves theorems from the theory context containing all of the constants occurring in the terms \bar{t} . Note that giving the empty list yields *all* theorems of the current theory.

thm_deps \bar{a} visualizes dependencies of facts, using Isabelle’s graph browser tool (see also [18]).

print_facts prints any named facts of the current context, including assumptions and local results.

print_binds prints all term abbreviations present in the context.

3.3.3 History commands

```
undo**  :  . → .
redo**  :  . → .
kill**  :  . → .
```

The Isabelle/Isar top-level maintains a two-stage history, for theory and proof state transformation. Basically, any command can be undone using

undo, excluding mere diagnostic elements. Its effect may be revoked via **redo**, unless the corresponding **undo** step has crossed the beginning of a proof or theory. The **kill** command aborts the current history node altogether, discontinuing a proof or even the whole theory. This operation is *not* undo-able.

! History commands should never be used with user interfaces such as Proof General [1, 2], which takes care of stepping forth and back itself. Interfering by manual **undo**, **redo**, or even **kill** commands would quickly result in utter confusion.

3.3.4 System operations

```

      cd*      :  . → .
    pwd*      :  . → .
  use_thy*    :  . → .
use_thy_only* :  . → .
  update_thy* :  . → .
update_thy_only* : . → .

```

cd *name* changes the current directory of the Isabelle process.

pwd prints the current working directory.

use_thy, **use_thy_only**, **update_thy**, **update_thy_only** load some theory given as *name* argument. These commands are basically the same as the corresponding ML functions² (see also [10, §1,§6]). Note that both the ML and Isar versions may load new- and old-style theories alike.

These system commands are scarcely used when working with the Proof General interface, since loading of theories is done transparently.

²The ML versions also change the implicit theory context to that of the theory loaded.

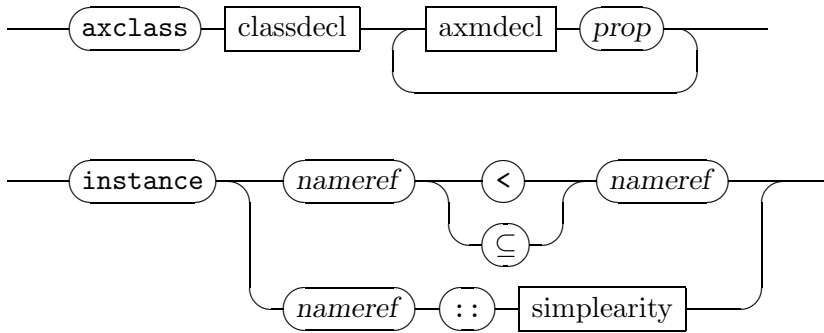
Generic tools and packages

4.1 Theory specification commands

4.1.1 Axiomatic type classes

`axclass` : $theory \rightarrow theory$
`instance` : $theory \rightarrow proof(prove)$
`intro_classes` : $method$

Axiomatic type classes are provided by Isabelle/Pure as a *definitional* interface to type classes (cf. §3.1.3). Thus any object logic may make use of this light-weight mechanism of abstract theories [14]. There is also a tutorial on using axiomatic type classes in Isabelle [16] that is part of the standard Isabelle documentation.



axclass $c \subseteq \bar{c}$ *axms* defines an axiomatic type class as the intersection of existing classes, with additional axioms holding. Class axioms (with implicit sort constraints added) are bound to the given names. Furthermore a class introduction rule is generated (being bound as $c.intro$); this rule is employed by method *intro_classes* to support instantiation proofs of this class.

The “axioms” are stored as theorems according to the given name specifications, adding the class name c as name space prefix; the same facts are also stored collectively as $c.axioms$.

instance $c_1 \subseteq c_2$ and **instance** $t :: (\overline{s})c$ setup a goal stating a class relation or type arity. The proof would usually proceed by *intro_classes*, and then establish the characteristic theorems of the type classes involved. After finishing the proof, the theory will be augmented by a type signature declaration corresponding to the resulting theorem.

intro_classes repeatedly expands all class introduction rules of this theory. Note that this method usually needs not be named explicitly, as it is already included in the default proof step (of **proof** etc.). In particular, instantiation of trivial (syntactic) classes may be performed by a single “..” proof step.

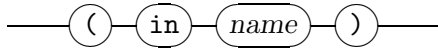
4.1.2 Locales and local contexts

Locales are named local contexts, consisting of a list of declaration elements that are modeled after the Isar proof context commands (cf. §3.2.2).

Localized commands

Existing locales may be augmented later on by adding new facts. Note that the actual context definition may not be changed! Several theory commands that produce facts in some way are available in “localized” versions, referring to a named locale instead of the global theory context.

locale

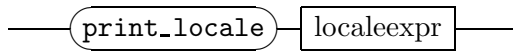
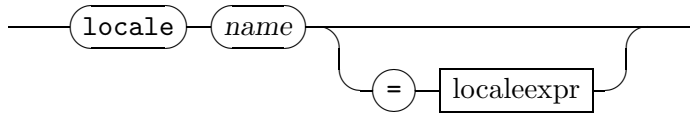


Emerging facts of localized commands are stored in two versions, both in the target locale and the theory (after export). The latter view produces a qualified binding, using the locale name as a name space prefix.

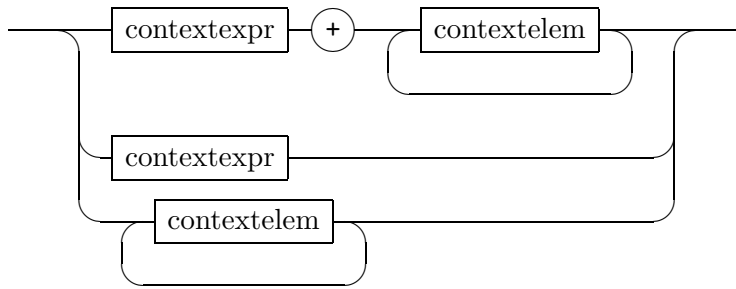
For example, “**lemmas** (**in** *loc*) $a = \overline{b}$ ” retrieves facts \overline{b} from the locale context of *loc* and augments its body by an appropriate “**notes**” element (see below). The exported view of a , after discharging the locale context, is stored as *loc.a* within the global theory. A localized goal “**lemma** (**in** *loc*) $a : \varphi$ ” works similarly, only that the fact emerges through the subsequent proof, which may refer to the full infrastructure of the locale context (covering local parameters with typing and concrete syntax, assumptions, definitions etc.). Most notably, fact declarations of the locale are active during the proof as well (e.g. local *simp* rules).

Locale specifications

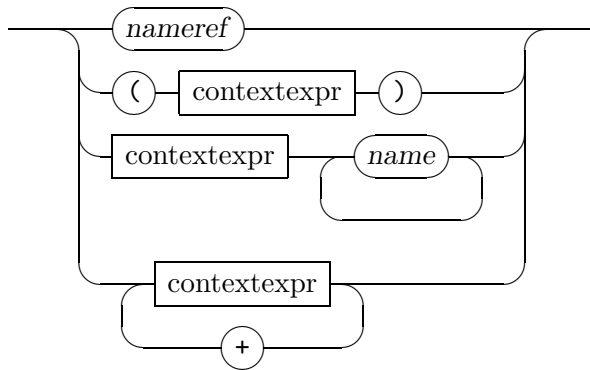
`locale` : $theory \rightarrow theory$
`print_locale*` : $theory \mid proof \rightarrow theory \mid proof$
`print_locales*` : $theory \mid proof \rightarrow theory \mid proof$

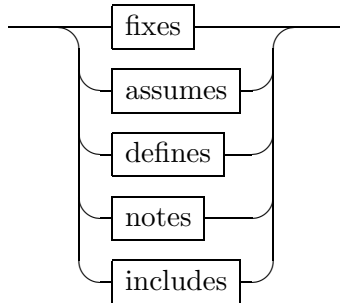
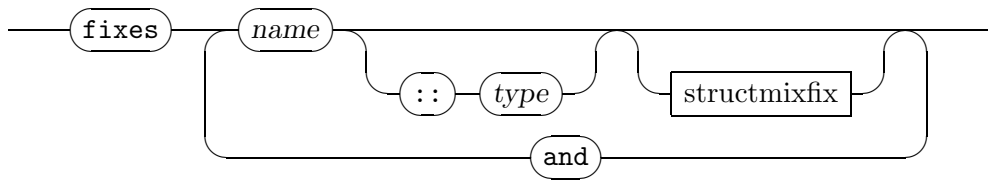
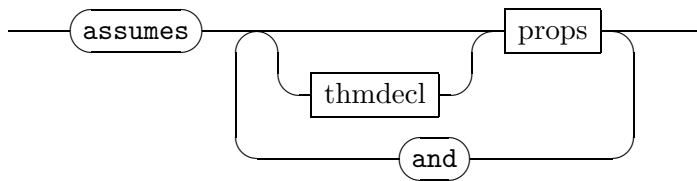
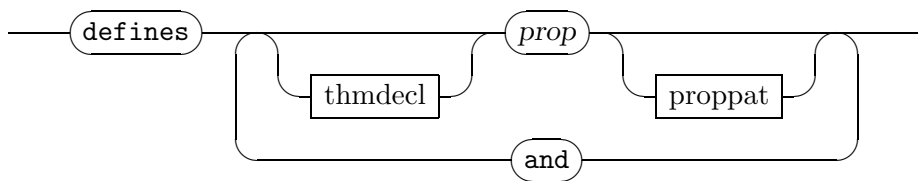
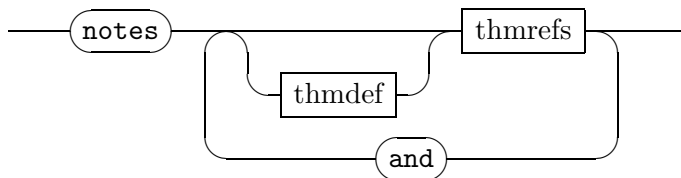
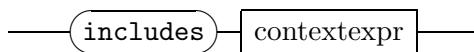


localeexpr



contextexpr



contextelem*fixes**assumes**defines**notes**includes*

locale $loc = import + body$ defines new locale loc as a context consisting of a certain view of existing locales (*import*) plus some additional elements (*body*). Both *import* and *body* are optional; the degenerate form **locale** loc defines an empty locale, which may still be useful to collect declarations of facts later on. Type-inference on locale expressions automatically takes care of the most general typing that the combined context elements may acquire.

The *import* consists of a structured context expression, consisting of references to existing locales, renamed contexts, or merged contexts. Renaming uses positional notation: $c \bar{x}$ means that (a prefix) the fixed parameters of context c are named according to \bar{x} ; a “ $_$ ” (underscore) means to skip that position. Also note that concrete syntax only works with the original name. Merging proceeds from left-to-right, suppressing any duplicates stemming from different paths through the import hierarchy.

The *body* consists of basic context elements, further context expressions may be included as well.

fixes $x :: \tau (mx)$ declares a local parameter of type τ and mixfix annotation mx (both are optional). The special syntax declaration “(*structure*)” means that x may be referenced implicitly in this context.

assumes $a: \bar{\varphi}$ introduces local premises, similar to **assume** within a proof (cf. §3.2.2).

defines $a: x \equiv t$ defines a previously declared parameter. This is close to **def** within a proof (cf. §3.2.2), but **defines** takes an equational proposition instead of variable-term pair. The left-hand side of the equation may have additional arguments, e.g. “**defines** $f \bar{x} \equiv t$ ”.

notes $a = \bar{b}$ reconsiders facts within a local context. Most notably, this may include arbitrary declarations in any attribute specifications included here, e.g. a local *simp* rule.

includes c copies the specified context in a statically scoped manner.

In contrast, the initial *import* specification of a locale expression maintains a dynamic relation to the locales being referenced (benefiting from any later fact declarations in the obvious manner).

Note that “(is p)” patterns given in the syntax of **assumes** and **defines** above are actually illegal in locale definitions. In the long goal format of §3.2.4, term bindings may be included as expected, though.

print_locale *import* + *body* prints the specified locale expression in a flattened form. The notable special case **print_locale** *loc* just prints the contents of the named locale, but keep in mind that type-inference will normalize type variables according to the usual alphabetical order.

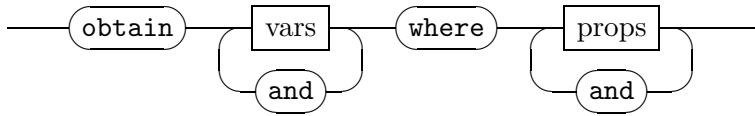
print_locales prints the names of all locales of the current theory.

4.2 Derived proof schemes

4.2.1 Generalized elimination

obtain : $proof(state) \rightarrow proof(prove)$

Generalized elimination means that additional elements with certain properties may be introduced in the current context, by virtue of a locally proven “soundness statement”. Technically speaking, the **obtain** language element is like a declaration of **fix** and **assume** (see also see §3.2.2), together with a soundness proof of its additional claim. According to the nature of existential reasoning, assumptions get eliminated from any result exported from the context later, provided that the corresponding parameters do *not* occur in the conclusion.



obtain is defined as a derived Isar command as follows, where \bar{b} shall refer to (optional) facts indicated for forward chaining.

```

<facts  $\bar{b}$ >
obtain  $\bar{x}$  where  $a: \bar{\varphi}$  <proof>  $\equiv$ 
  have  $\wedge thesis . (\wedge \bar{x} . \bar{\varphi} \implies thesis) \implies thesis$ 
  proof succeed
    fix thesis
    assume that [intro?]:  $\wedge \bar{x} . \bar{\varphi} \implies thesis$ 
    thus thesis
    apply —
    using  $\bar{b}$  <proof>
  qed
fix  $\bar{x}$  assume*  $a: \bar{\varphi}$ 

```

Typically, the soundness proof is relatively straight-forward, often just by canonical automated tools such as “**by simp**” or “**by blast**”. Accordingly, the “*that*” reduction above is declared as simplification and introduction rule.

In a sense, **obtain** represents at the level of Isar proofs what would be meta-logical existential quantifiers and conjunctions. This concept has a broad range of useful applications, ranging from plain elimination (or introduction) of object-level existentials and conjunctions, to elimination over results of symbolic evaluation of recursive definitions, for example. Also note that **obtain** without parameters acts much like **have**, where the result is treated as a genuine assumption.

4.2.2 Calculational reasoning

also	:	$proof(state) \rightarrow proof(state)$
finally	:	$proof(state) \rightarrow proof(chain)$
moreover	:	$proof(state) \rightarrow proof(state)$
ultimately	:	$proof(state) \rightarrow proof(chain)$
print_trans_rules*	:	$theory \mid proof \rightarrow theory \mid proof$
<i>trans</i>	:	<i>attribute</i>
<i>sym</i>	:	<i>attribute</i>
<i>symmetric</i>	:	<i>attribute</i>

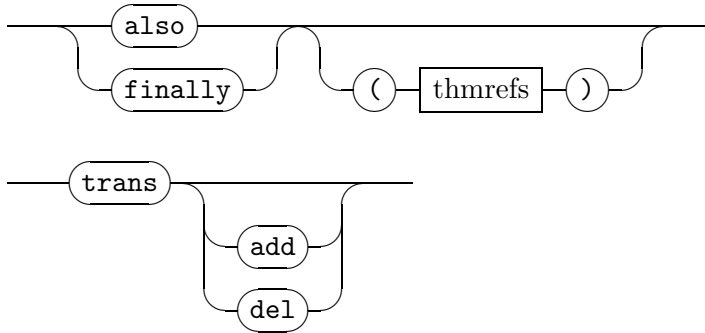
Calculational proof is forward reasoning with implicit application of transitivity rules (such those of $=$, \leq , $<$). Isabelle/Isar maintains an auxiliary register *calculation* for accumulating results obtained by transitivity composed with the current result. Command **also** updates *calculation* involving *this*, while **finally** exhibits the final *calculation* by forward chaining towards the next goal statement. Both commands require valid current facts, i.e. may occur only after commands that produce theorems such as **assume**, **note**, or some finished proof of **have**, **show** etc. The **moreover** and **ultimately** commands are similar to **also** and **finally**, but only collect further results in *calculation* without applying any rules yet.

Also note that the implicit term abbreviation “ \dots ” has its canonical application with calculational proofs. It refers to the argument of the preceding statement. (The argument of a curried infix expression happens to be its right-hand side.)

Isabelle/Isar calculations are implicitly subject to block structure in the sense that new threads of calculational reasoning are commenced for any new block (as opened by a local goal, for example). This means that, apart from being able to nest calculations, there is no separate *begin-calculation* command required.

The Isar calculation proof commands may be defined as follows:¹

$\mathbf{also}_0 \equiv \mathbf{note} \text{ calculation} = \text{this}$
 $\mathbf{also}_{n+1} \equiv \mathbf{note} \text{ calculation} = \text{trans } [OF \text{ calculation this}]$
 $\mathbf{finally} \equiv \mathbf{also from} \text{ calculation}$
 $\mathbf{moreover} \equiv \mathbf{note} \text{ calculation} = \text{calculation this}$
 $\mathbf{ultimately} \equiv \mathbf{moreover from} \text{ calculation}$



also (\bar{a}) maintains the auxiliary *calculation* register as follows. The first occurrence of **also** in some calculational thread initializes *calculation* by *this*. Any subsequent **also** on the same level of block-structure updates *calculation* by some transitivity rule applied to *calculation* and *this* (in that order). Transitivity rules are picked from the current context, unless alternative rules are given as explicit arguments.

finally (\bar{a}) maintaining *calculation* in the same way as **also**, and concludes the current calculational thread. The final result is exhibited as fact for forward chaining towards the next goal. Basically, **finally** just abbreviates **also from** *calculation*. Note that “**finally show** *?thesis* .” and “**finally have** φ .” are typical idioms for concluding calculational proofs.

moreover and **ultimately** are analogous to **also** and **finally**, but collect results only, without applying rules.

print_trans_rules prints the list of transitivity rules (for calculational commands **also** and **finally**) and symmetry rules (for the *symmetric* operation and single step elimination patterns) of the current context.

trans declares theorems as transitivity rules.

¹We suppress internal bookkeeping such as proper handling of block-structure.

sym declares symmetry rules.

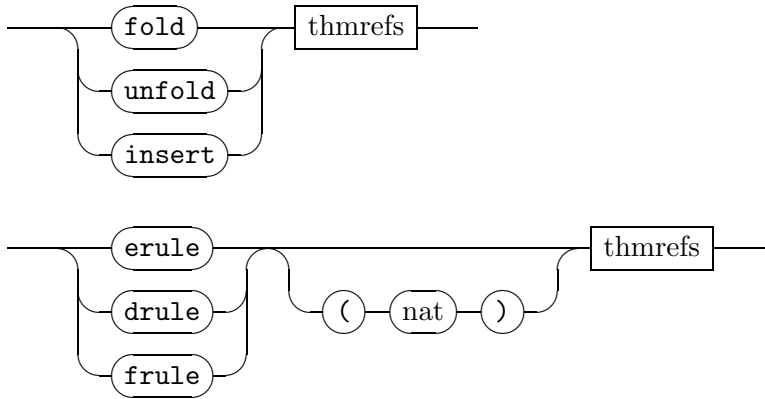
symmetric resolves a theorem with some rule declared as *sym* in the current context. For example, “**assume** [*symmetric*]: $x = y$ ” produces a swapped fact derived from that assumption.

In structured proof texts it is often more appropriate to use an explicit single-step elimination proof, such as “**assume** $x = y$ **hence** $y = x$. .”. The very same rules known to *symmetric* are declared as *elim?* as well.

4.3 Proof tools

4.3.1 Miscellaneous methods and attributes

unfold : method
fold : method
insert : method
*erule** : method
*drule** : method
*frule** : method
succeed : method
fail : method



unfold \bar{a} and *fold* \bar{a} expand (or fold back again) the given meta-level definitions throughout all goals; any chained facts provided are inserted into the goal and subject to rewriting as well.

insert \bar{a} inserts theorems as facts into all goals of the proof state. Note that current facts indicated for forward chaining are ignored.

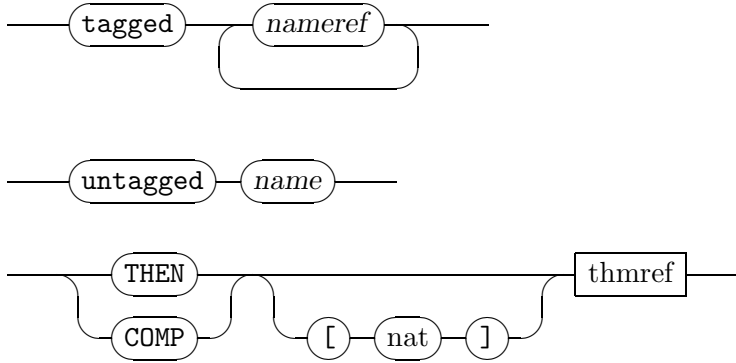
erule \bar{a} , *drule* \bar{a} , and *frule* \bar{a} are similar to the basic *rule* method (see §3.2.6), but apply rules by elim-resolution, destruct-resolution, and forward-resolution, respectively [10]. The optional natural number argument (default 0) specifies additional assumption steps to be performed here.

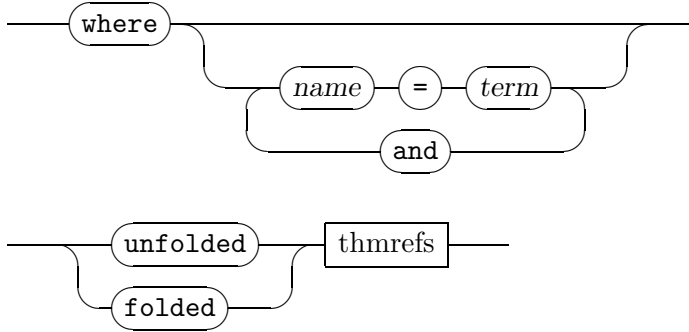
Note that these methods are improper ones, mainly serving for experimentation and tactic script emulation. Different modes of basic rule application are usually expressed in Isar at the proof language level, rather than via implicit proof state manipulations. For example, a proper single-step elimination would be done using the plain *rule* method, with forward chaining of current facts.

succeed yields a single (unchanged) result; it is the identity of the “,” method combinator (cf. §2.2.6).

fail yields an empty result sequence; it is the identity of the “|” method combinator (cf. §2.2.6).

tagged : attribute
untagged : attribute
THEN : attribute
COMP : attribute
where : attribute
unfolded : attribute
folded : attribute
elim_format : attribute
*standard** : attribute
*no_vars** : attribute





tagged name args and *untagged name* add and remove *tags* of some theorem. Tags may be any list of strings that serve as comment for some tools (e.g. **lemma** causes the tag “*lemma*” to be added to the result). The first string is considered the tag name, the rest its arguments. Note that *untag* removes any tags of the same name.

THEN a and *COMP a* compose rules by resolution. *THEN* resolves with the first premise of *a* (an alternative position may be also specified); the *COMP* version skips the automatic lifting process that is normally intended (cf. **RS** and **COMP** in [10, §5]).

where $\bar{x} = \bar{t}$ perform named instantiation of schematic variables occurring in a theorem. Unlike instantiation tactics such as *rule_tac* (see §3.2.9), actual schematic variables have to be specified on the left-hand side (e.g. $?x_3$).

unfolded \bar{a} and *folded \bar{a}* expand and fold back again the given meta-level definitions throughout a rule.

elim_format turns a destruction rule into elimination rule format, by resolving with the rule $\text{PROP } A \implies (\text{PROP } A \implies \text{PROP } B) \implies \text{PROP } B$.

Note that the Classical Reasoner (§4.3.4) provides its own version of this operation.

standard puts a theorem into the standard form of object-rules at the outermost theory level. Note that this operation violates the local proof context (including active locales).

no_vars replaces schematic variables by free ones; this is mainly for tuning output of pretty printed theorems.

4.3.2 Further tactic emulations

The following improper proof methods emulate traditional tactics. These admit direct access to the goal state, which is normally considered harmful! In particular, this may involve both numbered goal addressing (default 1), and dynamic instantiation within the scope of some subgoal.

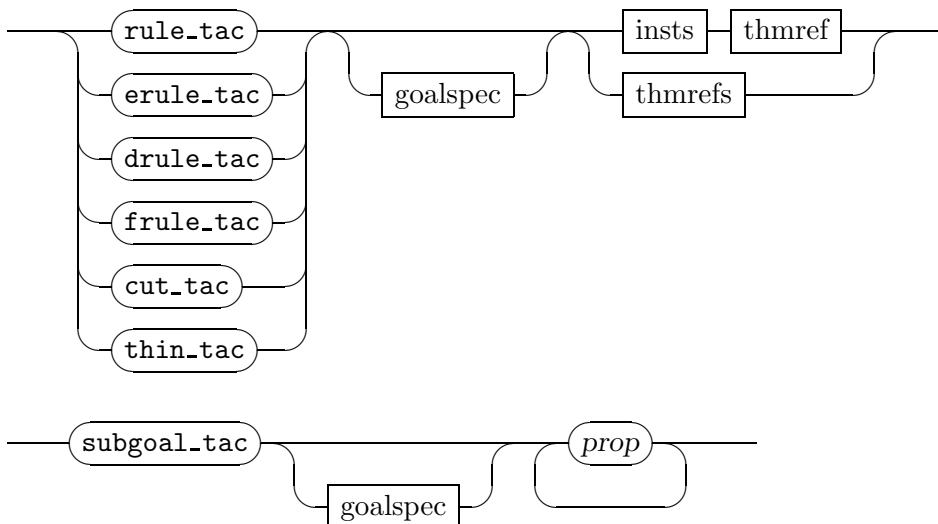
! Dynamic instantiations are read and type-checked according to a subgoal of the current dynamic goal state, rather than the static proof context! In particular, locally fixed variables and term abbreviations may not be included in the term specifications. Thus schematic variables are left to be solved by unification with certain parts of the subgoal involved.

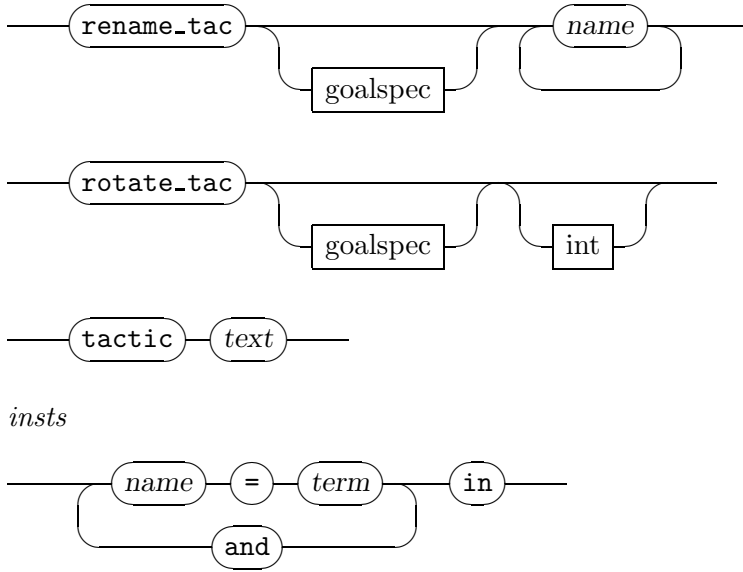
Note that the tactic emulation proof methods in Isabelle/Isar are consistently named *foo_tac*.

```

    rule_tac*   : method
    erule_tac*  : method
    drule_tac*  : method
    frule_tac*  : method
    cut_tac*    : method
    thin_tac*   : method
    subgoal_tac* : method
    rename_tac* : method
    rotate_tac* : method
    tactic*     : method

```





`rule_tac` etc. do resolution of rules with explicit instantiation. This works the same way as the ML tactics `res_inst_tac` etc. (see [10, §3]).

Multiple rules may be only given if there is no instantiation; then `rule_tac` is the same as `resolve_tac` in ML (see [10, §3]).

`cut_tac` inserts facts into the proof state as assumption of a subgoal, see also `cut_facts_tac` in [10, §3]. Note that the scope of schematic variables is spread over the main goal statement. Instantiations may be given as well, see also ML tactic `cut_inst_tac` in [10, §3].

`thin_tac` φ deletes the specified assumption from a subgoal; note that φ may contain schematic variables. See also `thin_tac` in [10, §3].

`subgoal_tac` φ adds φ as an assumption to a subgoal. See also `subgoal_tac` and `subgoals_tac` in [10, §3].

`rename_tac` \bar{x} renames parameters of a goal according to the list \bar{x} , which refers to the *suffix* of variables.

`rotate_tac` n rotates the assumptions of a goal by n positions: from right to left if n is positive, and from left to right if n is negative; the default value is 1. See also `rotate_tac` in [10, §3].

`tactic text` produces a proof method from any ML text of type `tactic`. Apart from the usual ML environment and the current implicit theory context, the ML code may refer to the following locally bound values:

```

val ctxt  : Proof.context
val facts : thm list
val thm   : string -> thm
val thms  : string -> thm list

```

Here `ctxt` refers to the current proof context, `facts` indicates any current facts for forward-chaining, and `thm` / `thms` retrieve named facts (including global theorems) from the context.

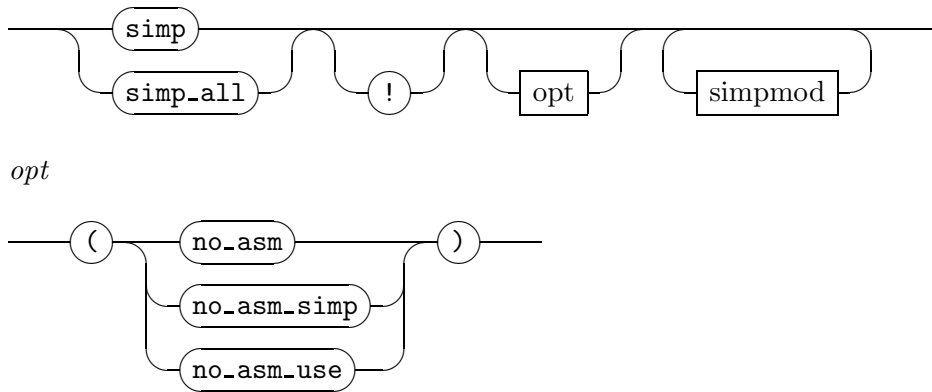
4.3.3 The Simplifier

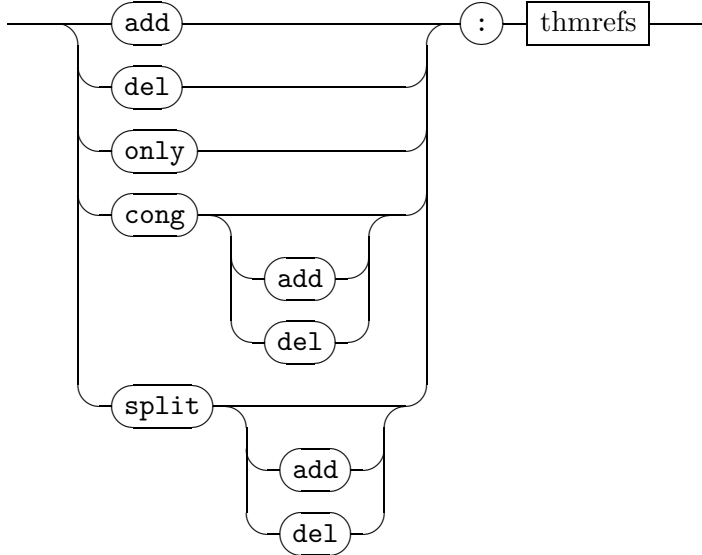
Simplification methods

```

simp      : method
simp_all  : method

```



simpmod

simp invokes Isabelle’s simplifier, after declaring additional rules according to the arguments given. Note that the **only** modifier first removes all other rewrite rules, congruences, and looper tactics (including splits), and then behaves like **add**.

The **cong** modifiers add or delete Simplifier congruence rules (see also [10]), the default is to add.

The **split** modifiers add or delete rules for the Splitter (see also [10]), the default is to add. This works only if the Simplifier method has been properly setup to include the Splitter (all major object logics such HOL, HOLCF, FOL, ZF do this already).

simp_all is similar to *simp*, but acts on all goals (backwards from the last to the first one).

By default the Simplifier methods take local assumptions fully into account, using equational assumptions in the subsequent normalization process, or simplifying assumptions themselves (cf. `asm_full_simp_tac` in [10, §10]). In structured proofs this is usually quite well behaved in practice: just the local premises of the actual goal are involved, additional facts may be inserted via explicit forward-chaining (using **then**, **from** etc.). The full context of assumptions is only included if the “!” (bang) argument is given, which should be used with some care, though.

Additional Simplifier options may be specified to tune the behavior further (mostly for unstructured scripts with many accidental local facts): “(*no_asm*)” means assumptions are ignored completely (cf. `simp_tac`), “(*no_asm_simp*)” means assumptions are used in the simplification of the conclusion but are not themselves simplified (cf. `asm_simp_tac`), and “(*no_asm_use*)” means assumptions are simplified but are not used in the simplification of each other or the conclusion (cf. `full_simp_tac`).

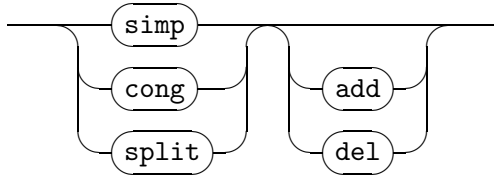
The Splitter package is usually configured to work as part of the Simplifier. The effect of repeatedly applying `split_tac` can be simulated by “(*simp only: split: \bar{a}*)”. There is also a separate *split* method available for single-step case splitting.

Declaring rules

```

print_simpset* : theory | proof → theory | proof
      simp : attribute
      cong : attribute
      split : attribute

```



print_simpset prints the collection of rules declared to the Simplifier, which is also known as “simpset” internally [10]. This is a diagnostic command; *undo* does not apply.

simp declares simplification rules.

cong declares congruence rules.

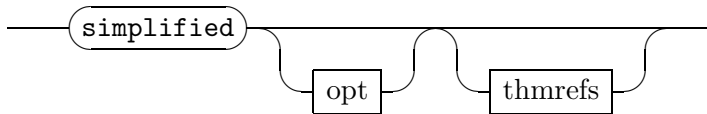
split declares case split rules.

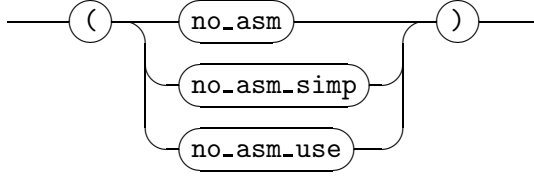
Forward simplification

```

simplified : attribute

```



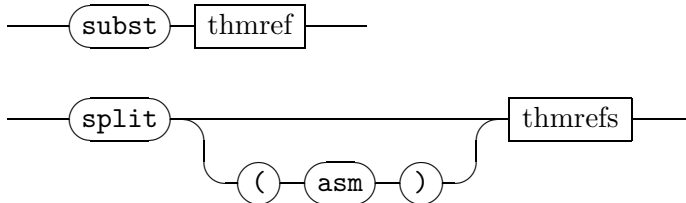
opt

simplified \bar{a} causes a theorem to be simplified, either by exactly the specified rules \bar{a} , or the implicit Simplifier context if no arguments are given. The result is fully simplified by default, including assumptions and conclusion; the options *no_asm* etc. tune the Simplifier in the same way as the for the *simp* method.

Note that forward simplification restricts the simplifier to its most basic operation of term rewriting; solver and loopier tactics [10] are *not* involved here. The *simplified* attribute should be only rarely required under normal circumstances.

Low-level equational reasoning

*subst** : method
*hypsubst** : method
*split** : method



These methods provide low-level facilities for equational reasoning that are intended for specialized applications only. Normally, single step calculations would be performed in a structured text (see also §4.2.2), while the Simplifier methods provide the canonical way for automated normalization (see §4.3.3).

subst *a* performs a single substitution step using rule *a*, which may be either a meta or object equality.

hypsubst performs substitution using some assumption; this only works for equations of the form $x = t$ where x is a free or bound variable.

split \bar{a} performs single-step case splitting using rules *thms*. By default, splitting is performed in the conclusion of a goal; the *asm* option indicates to operate on assumptions instead.

Note that the *simp* method already involves repeated application of split rules as declared in the current context.

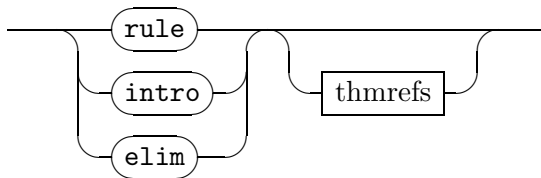
4.3.4 The Classical Reasoner

Basic methods

```

      rule  : method
contradiction : method
      intro : method
      elim  : method

```



rule as offered by the classical reasoner is a refinement over the primitive one (see §3.2.6). Both versions essentially work the same, but the classical version observes the classical rule context in addition to that of Isabelle/Pure.

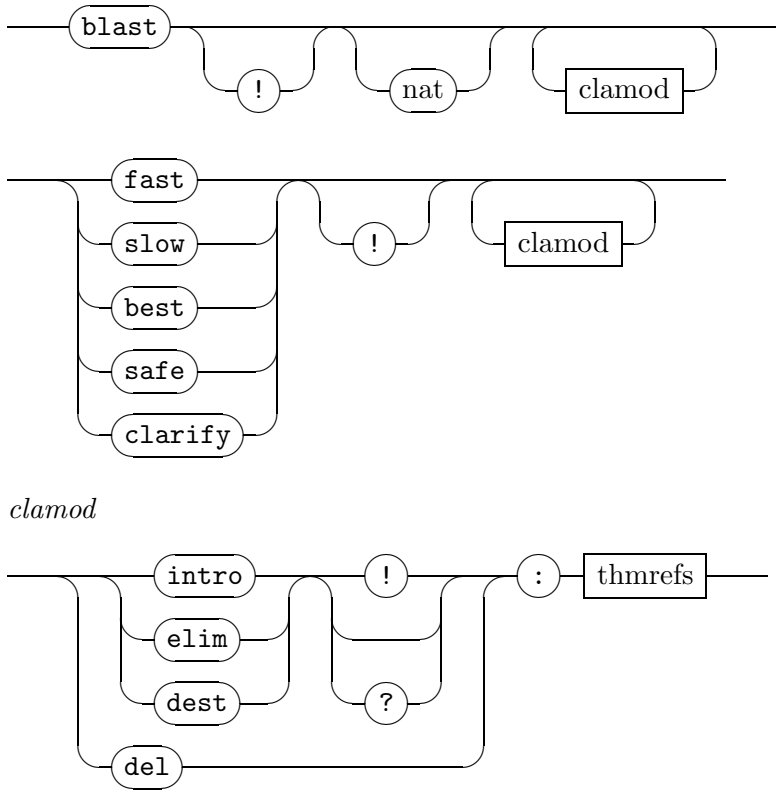
Common object logics (HOL, ZF, etc.) declare a rich collection of classical rules (even if these would qualify as intuitionistic ones), but only few declarations to the rule context of Isabelle/Pure (§3.2.6).

contradiction solves some goal by contradiction, deriving any result from both $\neg A$ and A . Chained facts, which are guaranteed to participate, may appear in either order.

intro and *elim* repeatedly refine some goal by intro- or elim-resolution, after having inserted any chained facts. Exactly the rules given as arguments are taken into account; this allows fine-tuned decomposition of a proof problem, in contrast to common automated tools.

Automated methods

blast : method
fast : method
slow : method
best : method
safe : method
clarify : method



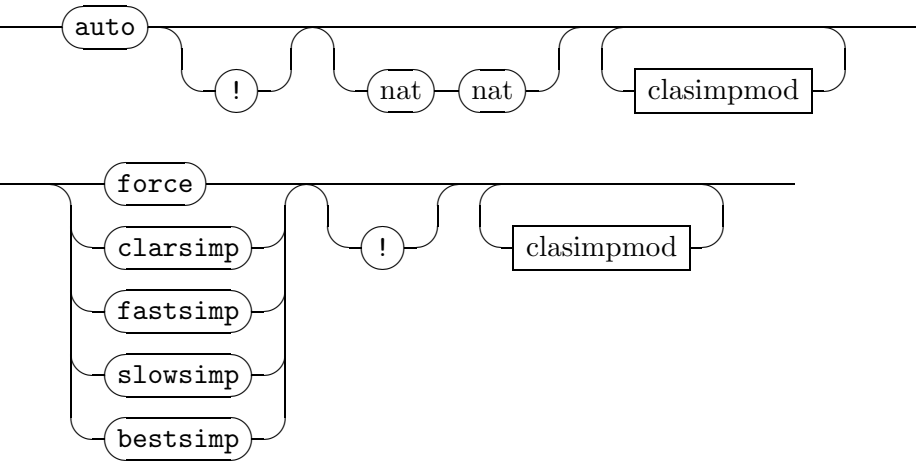
blast refers to the classical tableau prover (see *blast_tac* in [10, §11]). The optional argument specifies a user-supplied search bound (default 20).

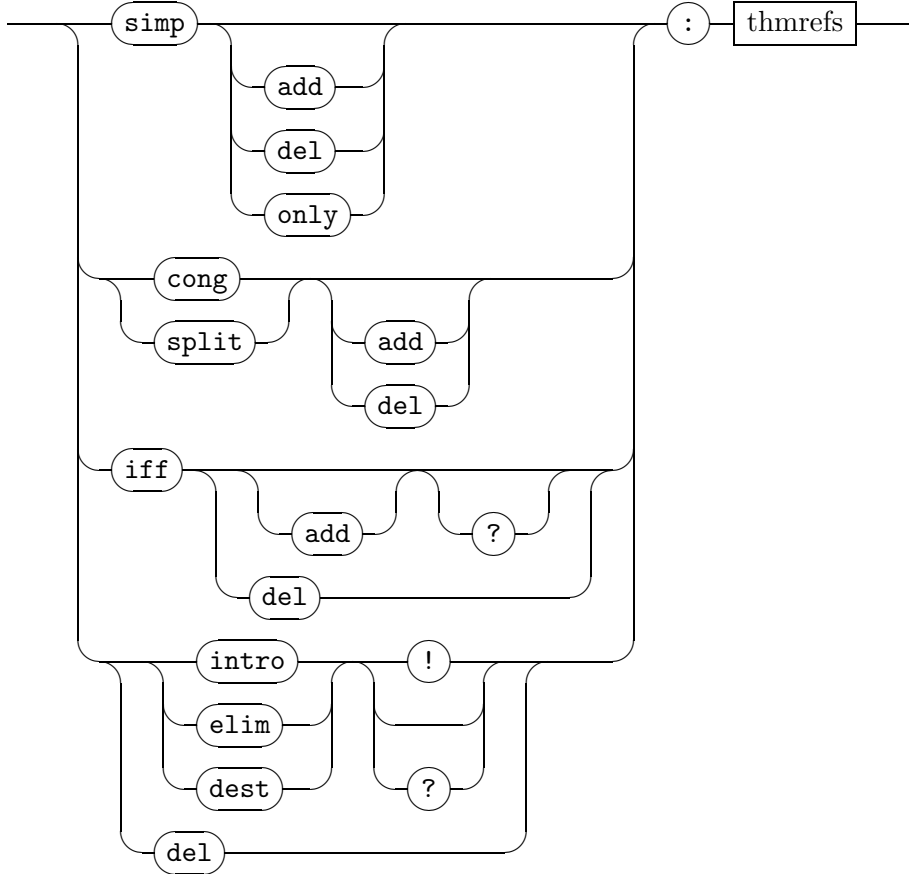
fast, *slow*, *best*, *safe*, and *clarify* refer to the generic classical reasoner. See *fast_tac*, *slow_tac*, *best_tac*, *safe_tac*, and *clarify_tac* in [10, §11] for more information.

Any of the above methods support additional modifiers of the context of classical rules. Their semantics is analogous to the attributes given before. Facts provided by forward chaining are inserted into the goal before commencing proof search. The “!” argument causes the full context of assumptions to be included as well.

Combined automated methods

auto : *method*
force : *method*
clarsimp : *method*
fastsimp : *method*
slowsimp : *method*
bestsimp : *method*



clasimpmod

auto, *force*, *clarsimp*, *fastsimp*, *slowsimp*, and *bestsimp* provide access to Isabelle’s combined simplification and classical reasoning tactics. These correspond to *auto_tac*, *force_tac*, *clarsimp_tac*, and Classical Reasoner tactics with the Simplifier added as wrapper, see [10, §11] for more information. The modifier arguments correspond to those given in §4.3.3 and §4.3.4. Just note that the ones related to the Simplifier are prefixed by *simp* here.

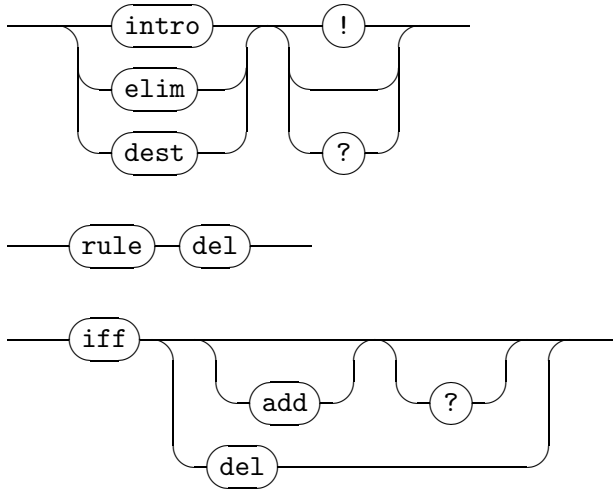
Facts provided by forward chaining are inserted into the goal before doing the search. The “!” argument causes the full context of assumptions to be included as well.

Declaring rules

```

print_claset* : theory | proof  $\rightarrow$  theory | proof
      intro : attribute
      elim  : attribute
      dest  : attribute
      rule  : attribute
      iff   : attribute

```



print_claset prints the collection of rules declared to the Classical Reasoner, which is also known as “simpset” internally [10]. This is a diagnostic command; *undo* does not apply.

intro, *elim*, and *dest* declare introduction, elimination, and destruction rules, respectively. By default, rules are considered as *unsafe* (i.e. not applied blindly without backtracking), while a single “!” classifies as *safe*. Rule declarations marked by “?” coincide with those of Isabelle/Pure, cf. §3.2.6 (i.e. are only applied in single steps of the *rule* method).

rule del deletes introduction, elimination, or destruction rules from the context.

iff declares logical equivalences to the Simplifier and the Classical reasoner at the same time. Non-conditional rules result in a “safe” introduction and elimination pair; conditional ones are considered “unsafe”. Rules with negative conclusion are automatically inverted (using \neg elimination internally).

The “?” version of *iff* declares rules to the Isabelle/Pure context only, and omits the Simplifier declaration.

Classical operations

elim_format : *attribute*
swapped : *attribute*

elim_format turns a destruction rule into elimination rule format; this operation is similar to the intuitionistic version (§4.3.1), but each premise of the resulting rule acquires an additional local fact of the negated main thesis; according to the classical principle $(\neg A \implies A) \implies A$.

swapped turns an introduction rule into an elimination, by resolving with the classical swap principle $(\neg B \implies A) \implies (\neg A \implies B)$.

4.3.5 Proof by cases and induction

Rule contexts

case : *proof*(*state*) \rightarrow *proof*(*state*)
print_cases* : *proof* \rightarrow *proof*
case_names : *attribute*
params : *attribute*
consumes : *attribute*

Basically, Isar proof contexts are built up explicitly using commands like **fix**, **assume** etc. (see §3.2.2). In typical verification tasks this can become hard to manage, though. In particular, a large number of local contexts may emerge from case analysis or induction over inductive sets and types.

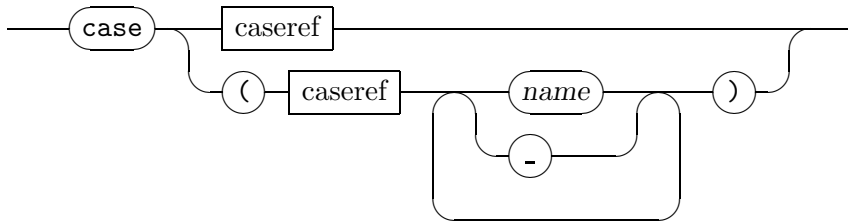
The **case** command provides a shorthand to refer to certain parts of logical context symbolically. Proof methods may provide an environment of named “cases” of the form $c:\bar{x},\bar{\varphi}$. Then the effect of “**case** *c*” is that of “**fix** \bar{x} **assume** $c:\bar{\varphi}$ ”. Term bindings may be covered as well, such as *?case* for the intended conclusion.

Normally the “terminology” of a case value (i.e. the parameters \bar{x}) are marked as hidden. Using the explicit form “**case** (*c* \bar{x})” enables proof writers to choose their own names for the subsequent proof text.

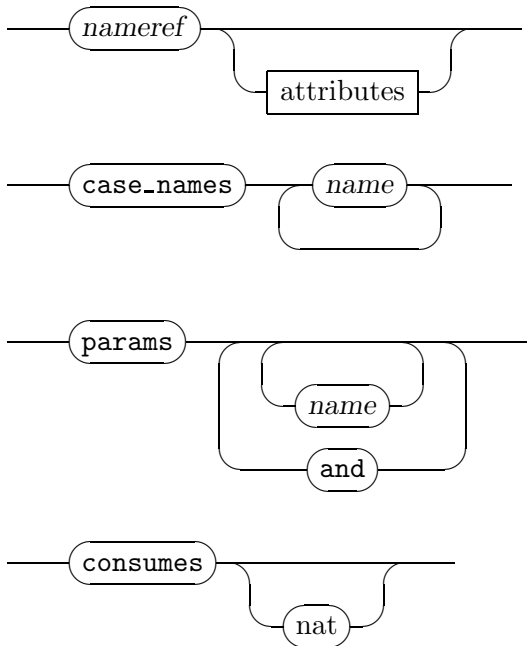
It is important to note that **case** does *not* provide direct means to peek at the current goal state, which is generally considered non-observable in Isar. The text of the cases basically emerge from standard elimination or

induction rules, which in turn are derived from previous theory specifications in a canonical way (say from **inductive** definitions).

Named cases may be exhibited in the current proof context only if both the proof method and the rules involved support this. Case names and parameters of basic rules may be declared by hand as well, by using appropriate attributes. Thus variant versions of rules that have been derived manually may be used in advanced case analysis later.



caseref



case ($c \ \bar{x}$) invokes a named local context $c: \bar{x}, \bar{\varphi}$, as provided by an appropriate proof method (such as *cases* and *induct*, see §4.3.5). The command “**case** ($c \ \bar{x}$)” abbreviates “**fix** \bar{x} **assume** $c: \bar{\varphi}$ ”.

print_cases prints all local contexts of the current state, using Isar proof language notation. This is a diagnostic command; *undo* does not apply.

case_names \bar{c} declares names for the local contexts of premises of some theorem; \bar{c} refers to the *suffix* of the list of premises.

params $\bar{p}_1 \dots \bar{p}_n$ renames the innermost parameters of premises $1, \dots, n$ of some theorem. An empty list of names may be given to skip positions, leaving the present parameters unchanged.

Note that the default usage of case rules does *not* directly expose parameters to the proof context (see also §4.3.5).

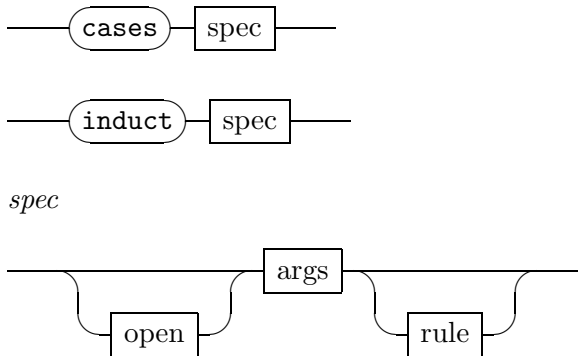
consumes n declares the number of “major premises” of a rule, i.e. the number of facts to be consumed when it is applied by an appropriate proof method (cf. §4.3.5). The default value of *consumes* is $n = 1$, which is appropriate for the usual kind of cases and induction rules for inductive sets (cf. §5.2.6). Rules without any *consumes* declaration given are treated as if *consumes* 0 had been specified.

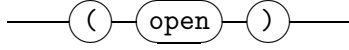
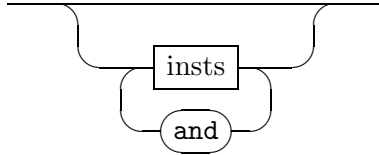
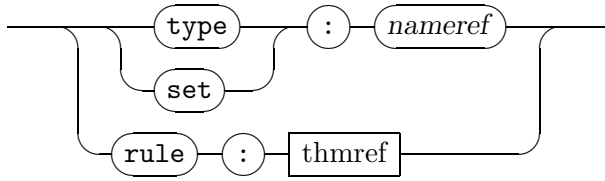
Note that explicit *consumes* declarations are only rarely needed; this is already taken care of automatically by the higher-level *cases* and *induct* declarations, see also §4.3.5.

Proof methods

cases : *method*
induct : *method*

The *cases* and *induct* methods provide a uniform interface to case analysis and induction over datatypes, inductive sets, and recursive functions. The corresponding rules may be specified and instantiated in a casual manner. Furthermore, these methods provide named local contexts that may be invoked via the **case** proof command within the subsequent proof text. This accommodates compact proof texts even when reasoning about large specifications.



open*args**rule*

cases insts R applies method *rule* with an appropriate case distinction theorem, instantiated to the subjects *insts*. Symbolic case names are bound according to the rule's local contexts.

The rule is determined as follows, according to the facts and arguments passed to the *cases* method:

facts	arguments	rule
	<i>cases</i>	classical case split
	<i>cases t</i>	datatype exhaustion (type of <i>t</i>)
$\vdash a \in A$	<i>cases ...</i>	inductive set elimination (of <i>A</i>)
...	<i>cases ... R</i>	explicit rule <i>R</i>

Several instantiations may be given, referring to the *suffix* of premises of the case rule; within each premise, the *prefix* of variables is instantiated. In most situations, only a single term needs to be specified; this refers to the first variable of the last premise (it is usually the same for all cases).

The “(*open*)” option causes the parameters of the new local contexts to be exposed to the current proof context. Thus local variables stemming from distant parts of the theory development may be introduced in an implicit manner, which can be quite confusing to the reader. Furthermore, this option may cause unwanted hiding of existing local variables, resulting in less robust proof texts.

induct insts R is analogous to the *cases* method, but refers to induction rules, which are determined as follows:

facts		arguments	rule
	<i>induct</i>	$P\ x\ \dots$	datatype induction (type of x)
$\vdash x \in A$	<i>induct</i>	\dots	set induction (of A)
\dots	<i>induct</i>	$\dots\ R$	explicit rule R

Several instantiations may be given, each referring to some part of a mutual inductive definition or datatype — only related partial induction rules may be used together, though. Any of the lists of terms P, x, \dots refers to the *suffix* of variables present in the induction rule. This enables the writer to specify only induction variables, or both predicates and variables, for example.

The “(*open*)” option works the same way as for *cases*.

Above methods produce named local contexts, as determined by the instantiated rule as specified in the text. Beyond that, the *induct* method guesses further instantiations from the goal specification itself. Any persisting unresolved schematic variables of the resulting rule will render the corresponding case invalid. The term binding *?case* for the conclusion will be provided with each case, provided that term is fully specified.

The **print-cases** command prints all named cases present in the current proof state.

It is important to note that there is a fundamental difference of the *cases* and *induct* methods in handling of non-atomic goal statements: *cases* just applies a certain rule in backward fashion, splitting the result into new goals with the local contexts being augmented in a purely monotonic manner.

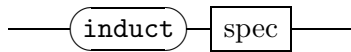
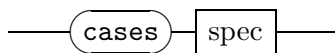
In contrast, *induct* passes the full goal statement through the “recursive” course involved in the induction. Thus the original statement is basically replaced by separate copies, corresponding to the induction hypotheses and conclusion; the original goal context is no longer available. This behavior allows *strengthened induction predicates* to be expressed concisely as meta-level rule statements, i.e. $\bigwedge \bar{x}. \bar{\varphi} \implies \psi$ to indicate “variable” parameters \bar{x} and “recursive” assumptions $\bar{\varphi}$. Also note that local definitions may be expressed as $\bigwedge \bar{x}. n \equiv t[\bar{x}] \implies \varphi[n]$, with induction over n .

Facts presented to either method are consumed according to the number of “major premises” of the rule involved (see also §4.3.5), which is usually 0 for plain cases and induction rules of datatypes etc. and 1 for rules of inductive sets and the like. The remaining facts are inserted into the goal

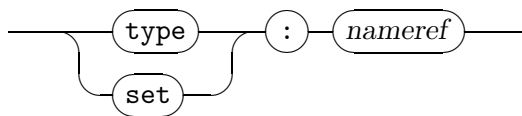
verbatim before the actual *cases* or *induct* rule is applied (thus facts may be even passed through an induction).

Declaring rules

```
print_induct_rules* : theory | proof → theory | proof
      cases : attribute
      induct : attribute
```



spec



print_induct_rules prints cases and induct rules for sets and types of the current context.

cases and *induct* (as attributes) augment the corresponding context of rules for reasoning about inductive sets and types, using the corresponding methods of the same name. Certain definitional packages of object-logics usually declare emerging cases and induction rules as expected, so users rarely need to intervene.

Manual rule declarations usually include the *case_names* and *ps* attributes to adjust names of cases and parameters of a rule (see §4.3.5); the *consumes* declaration is taken care of automatically: *consumes* 0 is specified for “type” rules and *consumes* 1 for “set” rules.

Object-logic specific elements

5.1 General logic setup

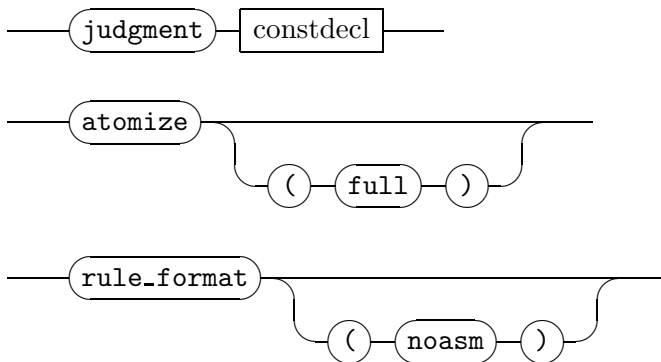
judgment : *theory* \rightarrow *theory*
atomize : *method*
atomize : *attribute*
rule_format : *attribute*
rulify : *attribute*

The very starting point for any Isabelle object-logic is a “truth judgment” that links object-level statements to the meta-logic (with its minimal language of *prop* that covers universal quantification \wedge and implication \implies).

Common object-logics are sufficiently expressive to internalize rule statements over \wedge and \implies within their own language. This is useful in certain situations where a rule needs to be viewed as an atomic statement from the meta-level perspective, e.g. $\wedge x . x \in A \implies P(x)$ versus $\forall x \in A . P(x)$.

From the following language elements, only the *atomize* method and *rule_format* attribute are occasionally required by end-users, the rest is for those who need to setup their own object-logic. In the latter case existing formulations of Isabelle/FOL or Isabelle/HOL may be taken as realistic examples.

Generic tools may refer to the information provided by object-logic declarations internally.



judgment $c :: \sigma \ (mx)$ declares constant c as the truth judgment of the current object-logic. Its type σ should specify a coercion of the category of object-level propositions to *prop* of the Pure meta-logic; the mixfix annotation (mx) would typically just link the object language (internally of syntactic category *logic*) with that of *prop*. Only one **judgment** declaration may be given in any theory development.

atomize (as a method) rewrites any non-atomic premises of a sub-goal, using the meta-level equations declared via *atomize* (as an attribute) beforehand. As a result, heavily nested goals become amenable to fundamental operations such as resolution (cf. the *rule* method) and proof-by-assumption (cf. *assumption*). Giving the “(full)” option here means to turn the whole subgoal into an object-statement (if possible), including the outermost parameters and assumptions as well.

A typical collection of *atomize* rules for a particular object-logic would provide an internalization for each of the connectives of \wedge , \implies , and \equiv . Meta-level conjunction expressed in the manner of minimal higher-order logic as $\wedge \text{PROP } C . (A \implies B \implies \text{PROP } C) \implies \text{PROP } C$ should be covered as well (this is particularly important for locales, see §4.1.2).

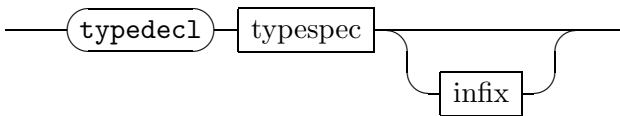
rule_format rewrites a theorem by the equalities declared as *rulify* rules in the current object-logic. By default, the result is fully normalized, including assumptions and conclusions at any depth. The *no_asm* option restricts the transformation to the conclusion of a rule.

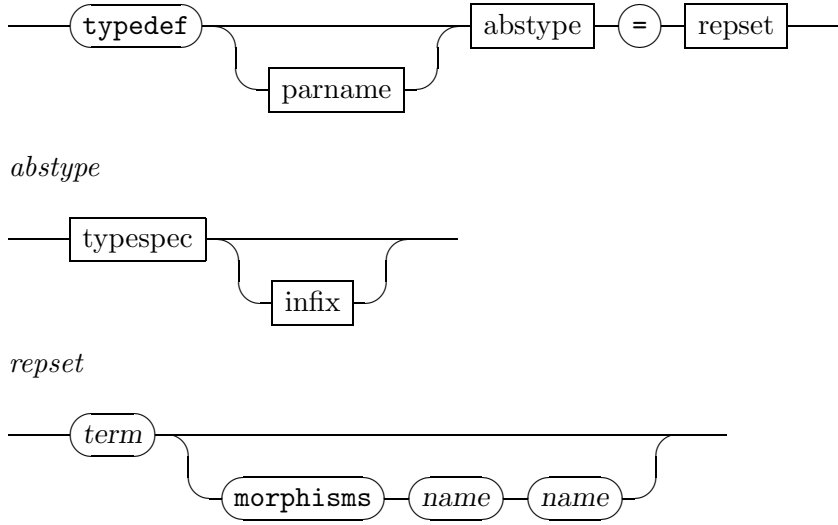
In common object-logics (HOL, FOL, ZF), the effect of *rule_format* is to replace (bounded) universal quantification (\forall) and implication (\rightarrow) by the corresponding rule statements over \wedge and \implies .

5.2 HOL

5.2.1 Primitive types

typedec1 : *theory* \rightarrow *theory*
typedef : *theory* \rightarrow *proof*(*prove*)





typedef $(\bar{\alpha})t$ is similar to the original **typedef** of Isabelle/Pure (see §3.1.4), but also declares type arity $t :: (type, \dots, type)type$, making t an actual HOL type constructor.

typedef $(\bar{\alpha})t = A$ sets up a goal stating non-emptiness of the set A . After finishing the proof, the theory will be augmented by a Gordon/HOL-style type definition, which establishes a bijection between the representing set A and the new type t .

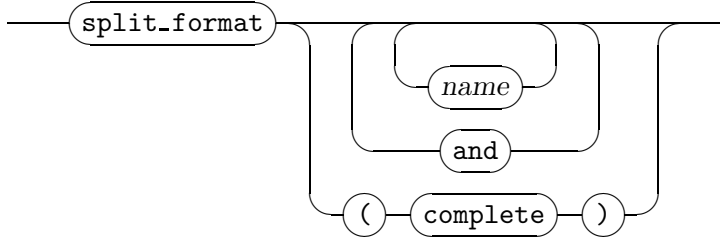
Technically, **typedef** defines both a type t and a set (term constant) of the same name (an alternative base name may be given in parentheses). The injection from type to set is called *Rep- t* , its inverse *Abs- t* (this may be changed via an explicit **morphisms** declaration).

Theorems *Rep- t* , *Rep- t -inverse*, and *Abs- t -inverse* provide the most basic characterization as a corresponding injection/surjection pair (in both directions). Rules *Rep- t -inject* and *Abs- t -inject* provide a slightly more convenient view on the injectivity part, suitable for automated proof tools (e.g. in *simp* or *iff* declarations). Rules *Rep- t -cases*/*Rep- t -induct*, and *Abs- t -cases*/*Abs- t -induct* provide alternative views on surjectivity; these are already declared as set or type rules for the generic *cases* and *induct* methods.

Note that raw type declarations are rarely used in practice; the main application is with experimental (or even axiomatic!) theory fragments. Instead of primitive HOL type definitions, user-level theories usually refer to higher-level packages such as **record** (see §5.2.3) or **datatype** (see §5.2.4).

5.2.2 Adhoc tuples

*split_format** : *attribute*



split_format $\bar{p}_1 \dots \bar{p}_n$ puts expressions of low-level tuple types into canonical form as specified by the arguments given; \bar{p}_i refers to occurrences in premise i of the rule. The “(*complete*)” option causes *all* arguments in function applications to be represented canonically according to their tuple type structure.

Note that these operations tend to invent funny names for new local parameters to be introduced.

5.2.3 Records

In principle, records merely generalize the concept of tuples, where components may be addressed by labels instead of just position. The logical infrastructure of records in Isabelle/HOL is slightly more advanced, though, supporting truly extensible record schemes. This admits operations that are polymorphic with respect to record extension, yielding “object-oriented” effects like (single) inheritance. See also [6] for more details on object-oriented verification and record subtyping in HOL.

Basic concepts

Isabelle/HOL supports both *fixed* and *schematic* records at the level of terms and types. The notation is as follows:

	record terms	record types
fixed	$\langle x = a, y = b \rangle$	$\langle x :: A, y :: B \rangle$
schematic	$\langle x = a, y = b, \dots = m \rangle$	$\langle x :: A, y :: B, \dots :: M \rangle$

The ASCII representation of $\langle x = a \rangle$ is $(\mid \mathbf{x} = \mathbf{a} \mid)$.

A fixed record $\langle x = a, y = b \rangle$ has field x of value a and field y of value b . The corresponding type is $\langle x :: A, y :: B \rangle$, assuming that $a :: A$ and $b :: B$.

A record scheme like $\langle x = a, y = b, \dots = m \rangle$ contains fields x and y as before, but also possibly further fields as indicated by the “...” notation (which is actually part of the syntax). The improper field “...” of a record scheme is called the *more part*. Logically it is just a free variable, which is occasionally referred to as “row variable” in the literature. The more part of a record scheme may be instantiated by zero or more further components. For example, the previous scheme may get instantiated to $\langle x = a, y = b, z = c, \dots = m' \rangle$, where m' refers to a different more part. Fixed records are special instances of record schemes, where “...” is properly terminated by the $() :: \text{unit}$ element. Actually, $\langle x = a, y = b \rangle$ is just an abbreviation for $\langle x = a, y = b, \dots = () \rangle$.

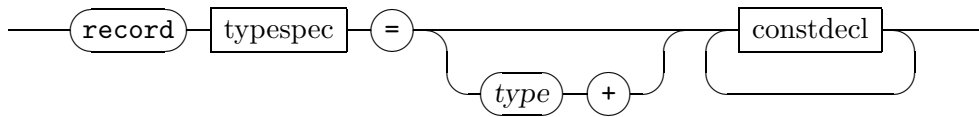
Two key observations make extensible records in a simply typed language like HOL feasible:

1. the more part is internalized, as a free term or type variable,
2. field names are externalized, they cannot be accessed within the logic as first-class values.

In Isabelle/HOL record types have to be defined explicitly, fixing their field names and types, and their (optional) parent record. Afterwards, records may be formed using above syntax, while obeying the canonical order of fields as given by their declaration. The record package provides several standard operations like selectors and updates. The common setup for various generic proof tools enable succinct reasoning patterns. See also the Isabelle/HOL tutorial [8] for further instructions on using records in practice.

Record specifications

record : *theory* \rightarrow *theory*



record $(\bar{\alpha})t = \tau + \bar{\tau} :: \bar{\sigma}$ defines extensible record type $(\bar{\alpha})t$, derived from the optional parent record τ by adding new field components $\bar{\tau} :: \bar{\sigma}$.

The type variables of τ and $\bar{\sigma}$ need to be covered by the (distinct) parameters $\bar{\alpha}$. Type constructor t has to be new, while τ needs to specify an instance of an existing record type. At least one new field $\bar{\tau}$

has to be specified. Basically, field names need to belong to a unique record. This is not a real restriction in practice, since fields are qualified by the record name internally.

The parent record specification τ is optional; if omitted t becomes a root record. The hierarchy of all records declared within a theory context forms a forest structure, i.e. a set of trees starting with a root record each. There is no way to merge multiple parent records!

For convenience, $(\bar{\alpha}) t$ is made a type abbreviation for the fixed record type $(\bar{c} :: \bar{\sigma})$, likewise is $(\bar{\alpha}, \zeta) t_scheme$ made an abbreviation for $(\bar{c} :: \bar{\sigma}, \dots :: \zeta)$.

Record operations

Any record definition of the form presented above produces certain standard operations. Selectors and updates are provided for any field, including the improper one “*more*”. There are also cumulative record constructor functions. To simplify the presentation below, we assume for now that $(\bar{\alpha}) t$ is a root record with fields $\bar{c} :: \bar{\sigma}$.

Selectors and **updates** are available for any field (including “*more*”):

$$\begin{aligned} c_i &:: (\bar{c} :: \bar{\sigma}, \dots :: \zeta) \Rightarrow \sigma_i \\ c_i_update &:: \sigma_i \Rightarrow (\bar{c} :: \bar{\sigma}, \dots :: \zeta) \Rightarrow (\bar{c} :: \bar{\sigma}, \dots :: \zeta) \end{aligned}$$

There is special syntax for application of updates: $r (x := a)$ abbreviates term $x_update a r$. 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 because of postfix notation the order of fields shown here is reverse than in the actual term. Since repeated updates are just function applications, fields may be freely permuted in $(x := a, y := b, z := c)$, as far as logical equality is concerned. Thus commutativity of independent updates can be proven within the logic for any two fields, but not as a general theorem.

The **make** operation provides a cumulative record constructor function:

$$t.make :: \bar{\sigma} \Rightarrow (\bar{c} :: \bar{\sigma})$$

We now reconsider the case of non-root records, which are derived of some parent. In general, the latter may depend on another parent as well, resulting in a list of *ancestor records*. Appending the lists of fields of all ancestors results in a certain field prefix. The record package automatically takes care of this by lifting operations over this context of ancestor fields.

Assuming that $(\overline{\alpha}) t$ has ancestor fields $\overline{b} :: \overline{\rho}$, the above record operations will get the following types:

$$\begin{aligned} c_i &:: (\overline{b} :: \overline{\rho}, \overline{c} :: \overline{\sigma}, \dots :: \zeta) \Rightarrow \sigma_i \\ c_i\text{-update} &:: \sigma_i \Rightarrow (\overline{b} :: \overline{\rho}, \overline{c} :: \overline{\sigma}, \dots :: \zeta) \Rightarrow (\overline{b} :: \overline{\rho}, \overline{c} :: \overline{\sigma}, \dots :: \zeta) \\ t.\text{make} &:: \overline{\rho} \Rightarrow \overline{\sigma} \Rightarrow (\overline{b} :: \overline{\rho}, \overline{c} :: \overline{\sigma}) \end{aligned}$$

Some further operations address the extension aspect of a derived record scheme specifically: *fields* produces a record fragment consisting of exactly the new fields introduced here (the result may serve as a more part elsewhere); *extend* takes a fixed record and adds a given more part; *truncate* restricts a record scheme to a fixed record.

$$\begin{aligned} t.\text{fields} &:: \overline{\sigma} \Rightarrow (\overline{c} :: \overline{\sigma}) \\ t.\text{extend} &:: (\overline{d} :: \overline{\rho}, \overline{c} :: \overline{\sigma}) \Rightarrow \zeta \Rightarrow (\overline{d} :: \overline{\rho}, \overline{c} :: \overline{\sigma}, \dots :: \zeta) \\ t.\text{truncate} &:: (\overline{d} :: \overline{\rho}, \overline{c} :: \overline{\sigma}, \dots :: \zeta) \Rightarrow (\overline{d} :: \overline{\rho}, \overline{c} :: \overline{\sigma}) \end{aligned}$$

Note that *t.make* and *t.fields* actually coincide for root records.

Derived rules and proof tools

The record package proves several results internally, declaring these facts to appropriate proof tools. This enables users to reason about record structures quite conveniently. Assume that *t* is a record type as specified above.

1. Standard conversions for selectors or updates applied to record constructor terms are made part of the default Simplifier context; thus proofs by reduction of basic operations merely require the *simp* method without further arguments. These rules are available as *t.simps*, too.
2. Selectors applied to updated records are automatically reduced by an internal simplification procedure, which is also part of the standard Simplifier setup.
3. Inject equations of a form analogous to $((x, y) = (x', y')) \equiv x = x' \wedge y = y'$ are declared to the Simplifier and Classical Reasoner as *iff* rules. These rules are available as *t.iffs*.
4. The introduction rule for record equality analogous to $x \ r = x \ r' \implies y \ r = y \ r' \implies \dots \implies r = r'$ is declared to the Simplifier, and as the basic rule context as “*intro?*”. The rule is called *t.equality*.

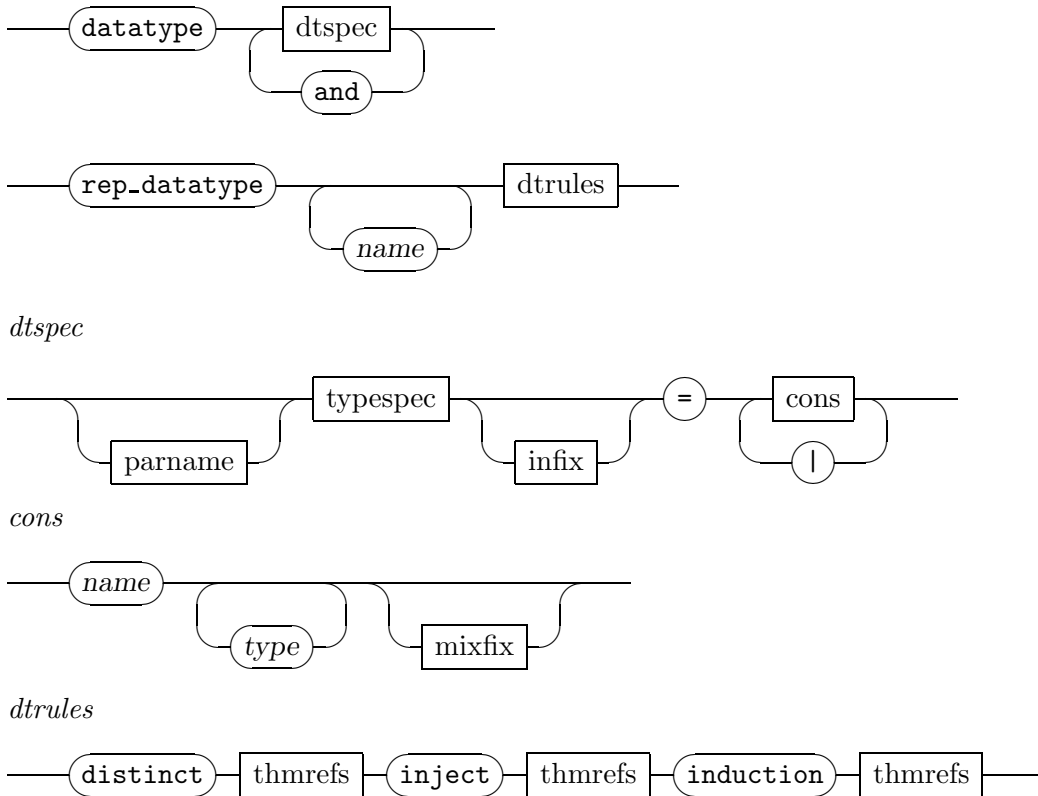
5. Representations of arbitrary record expressions as canonical constructor terms are provided both in *cases* and *induct* format (cf. the generic proof methods of the same name, §4.3.5). Several variations are available, for fixed records, record schemes, more parts etc.

The generic proof methods are sufficiently smart to pick the most sensible rule according to the type of the indicated record expression: users just need to apply something like “(*cases* *r*)” to a certain proof problem.

6. The derived record operations *t.make*, *t.fields*, *t.extend*, *t.truncate* are *not* treated automatically, but usually need to be expanded by hand, using the collective fact *t.defs*.

5.2.4 Datatypes

datatype : *theory* → *theory*
rep_datatype : *theory* → *theory*



datatype defines inductive datatypes in HOL.

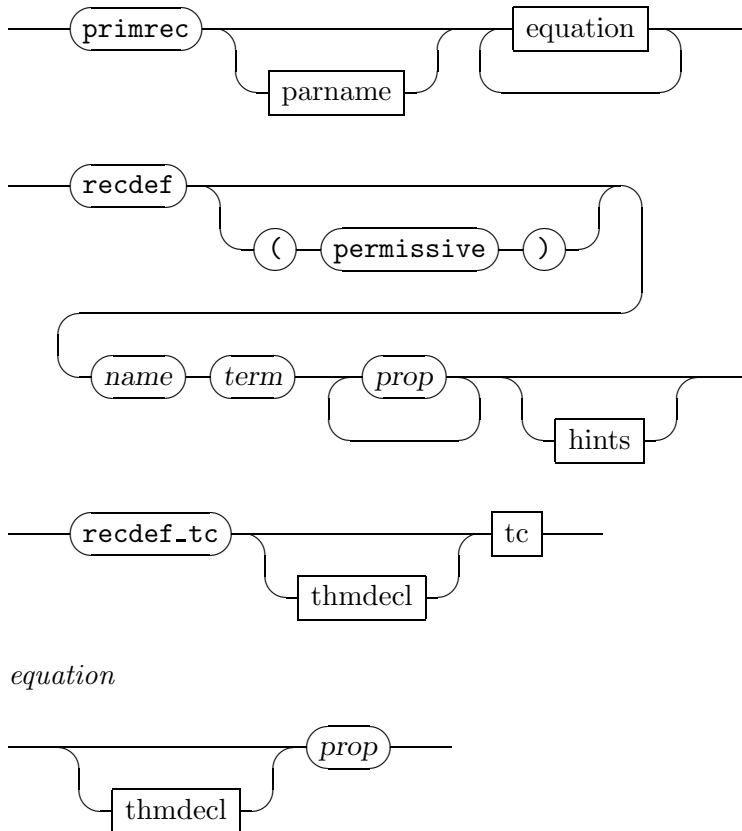
rep_datatype represents existing types as inductive ones, generating the standard infrastructure of derived concepts (primitive recursion etc.).

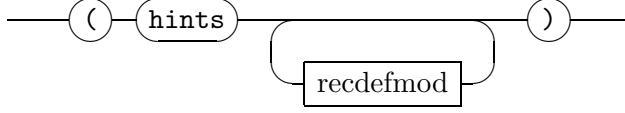
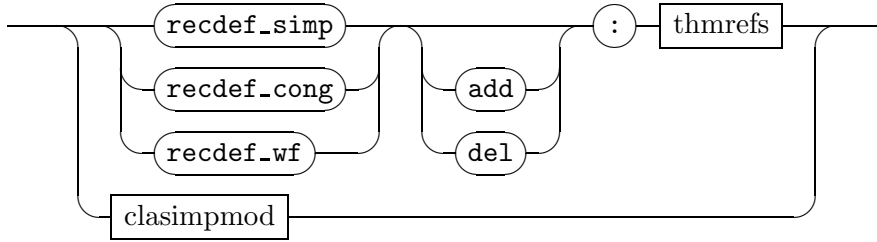
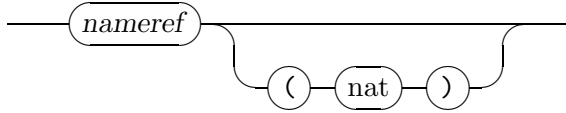
The induction and exhaustion theorems generated provide case names according to the constructors involved, while parameters are named after the types (see also §4.3.5).

See [7] for more details on datatypes, but beware of the old-style theory syntax being used there! Apart from proper proof methods for case-analysis and induction, there are also emulations of ML tactics **case_tac** and **induct_tac** available, see §5.2.8; these admit to refer directly to the internal structure of subgoals (including internally bound parameters).

5.2.5 Recursive functions

primrec : *theory* → *theory*
recdef : *theory* → *theory*
recdef_tc* : *theory* → *proof*(*prove*)



hints*recdefmod**tc*

primrec defines primitive recursive functions over datatypes, see also [7].

recdef defines general well-founded recursive functions (using the TFL package), see also [7]. The “(*permissive*)” option tells TFL to recover from failed proof attempts, returning unfinished results. The *recdef_simp*, *recdef_cong*, and *recdef_wf* hints refer to auxiliary rules to be used in the internal automated proof process of TFL. Additional *clasimpmod* declarations (cf. §4.3.4) may be given to tune the context of the Simplifier (cf. §4.3.3) and Classical reasoner (cf. §4.3.4).

recdef_tc *c* (*i*) recommences the proof for leftover termination condition number *i* (default 1) as generated by a **recdef** definition of constant *c*.

Note that in most cases, **recdef** is able to finish its internal proofs without manual intervention.

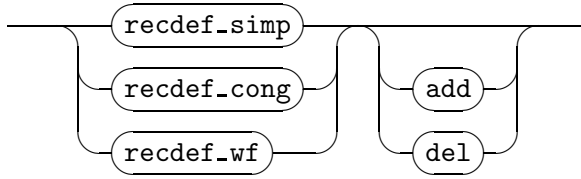
Both kinds of recursive definitions accommodate reasoning by induction (cf. §4.3.5): rule *c.induct* (where *c* is the name of the function definition) refers to a specific induction rule, with parameters named according to the user-specified equations. Case names of **primrec** are that of the datatypes involved, while those of **recdef** are numbered (starting from 1).

The equations provided by these packages may be referred later as theorem list *f.simps*, where *f* is the (collective) name of the functions defined.

Individual equations may be named explicitly as well; note that for **recdef** each specification given by the user may result in several theorems.

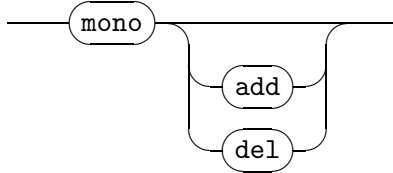
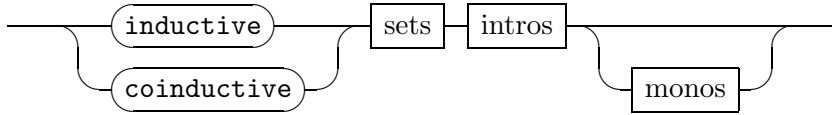
Hints for **recdef** may be also declared globally, using the following attributes.

recdef_simp : attribute
recdef_cong : attribute
recdef_wf : attribute

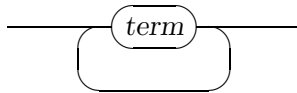


5.2.6 (Co)Inductive sets

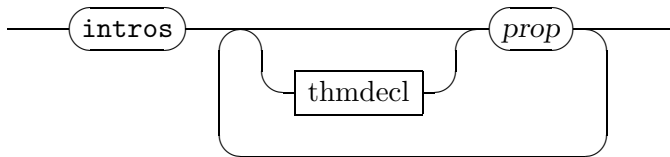
inductive : *theory* \rightarrow *theory*
coinductive : *theory* \rightarrow *theory*
mono : attribute



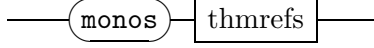
sets



intros



monos



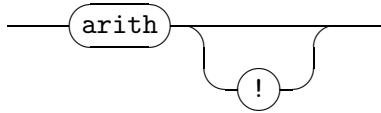
inductive and **coinductive** define (co)inductive sets from the given introduction rules.

mono declares monotonicity rules. These rule are involved in the automated monotonicity proof of **inductive**.

See [7] for further information on inductive definitions in HOL, but note that this covers the old-style theory format.

5.2.7 Arithmetic proof support

arith : *method*
arith_split : *attribute*



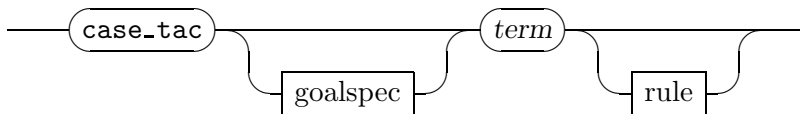
The *arith* method decides linear arithmetic problems (on types *nat*, *int*, *real*). Any current facts are inserted into the goal before running the procedure. The “!” argument causes the full context of assumptions to be included. The *arith_split* attribute declares case split rules to be expanded before the arithmetic procedure is invoked.

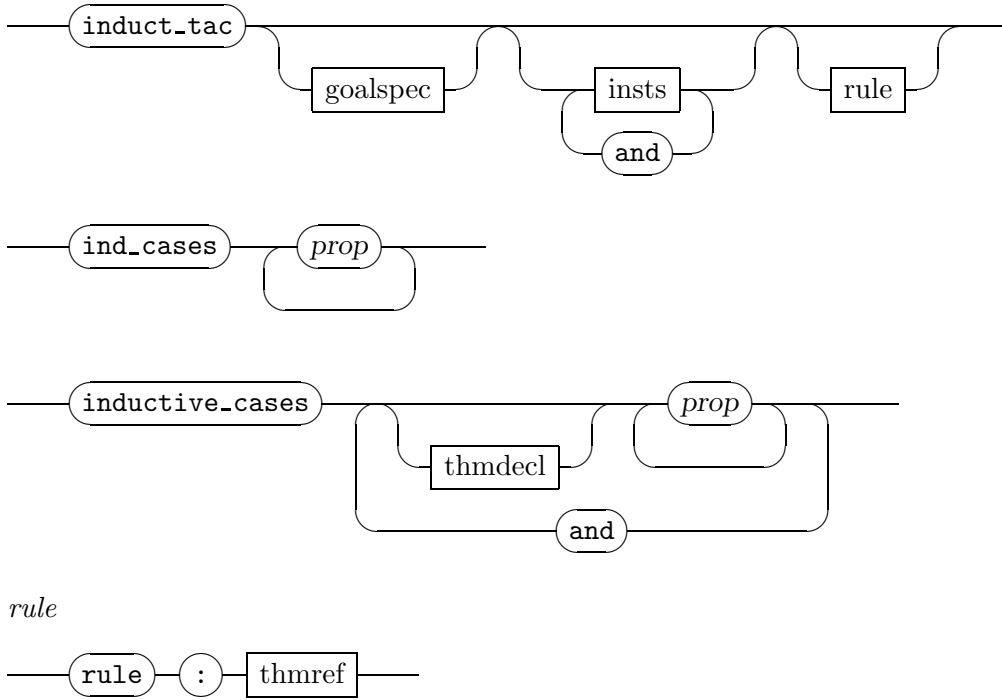
Note that a simpler (but faster) version of arithmetic reasoning is already performed by the Simplifier.

5.2.8 Cases and induction: emulating tactic scripts

The following important tactical tools of Isabelle/HOL have been ported to Isar. These should be never used in proper proof texts!

*case_tac** : *method*
*induct_tac** : *method*
*ind_cases** : *method*
inductive_cases : *theory* \rightarrow *theory*





`case_tac` and `induct_tac` admit to reason about inductive datatypes only (unless an alternative rule is given explicitly). Furthermore, `case_tac` does a classical case split on booleans; `induct_tac` allows only variables to be given as instantiation. These tactic emulations feature both goal addressing and dynamic instantiation. Note that named rule cases are *not* provided as would be by the proper `induct` and `cases` proof methods (see §4.3.5).

`ind_cases` and **`inductive_cases`** provide an interface to the internal `mk_cases` operation. Rules are simplified in an unrestricted forward manner.

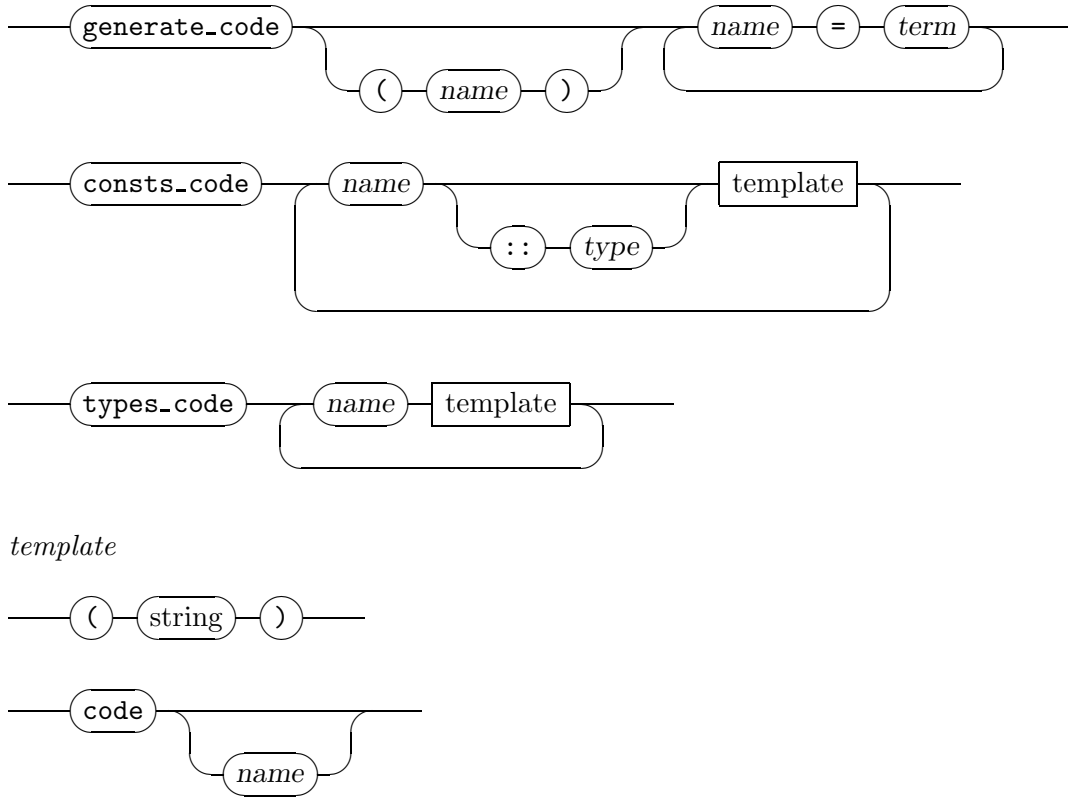
While `ind_cases` is a proof method to apply the result immediately as elimination rules, **`inductive_cases`** provides case split theorems at the theory level for later use,

5.2.9 Executable code

Isabelle/Pure provides a generic infrastructure to support code generation from executable specifications, both functional and relational programs. Isabelle/HOL instantiates these mechanisms in a way that is amenable to end-

user applications. See [7] for further information (this actually covers the new-style theory format as well).

generate_code : *theory* \rightarrow *theory*
consts_code : *theory* \rightarrow *theory*
types_code : *theory* \rightarrow *theory*
code : *attribute*



5.3 HOLCF

5.3.1 Mixfix syntax for continuous operations

consts : *theory* \rightarrow *theory*
constdefs : *theory* \rightarrow *theory*

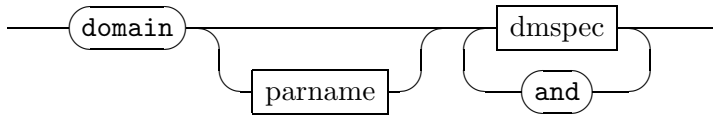
HOLCF provides a separate type for continuous functions $\alpha \rightarrow \beta$, with an explicit application operator $f \cdot x$. Isabelle mixfix syntax normally refers directly to the pure meta-level function type $\alpha \Rightarrow \beta$, with application $f x$.

The HOLCF variants of **consts** and **constdefs** have the same outer syntax as the pure versions (cf. §3.1.5). Internally, declarations involving continuous function types are treated specifically, transforming the syntax template accordingly and generating syntax translation rules for the abstract and concrete representation of application.

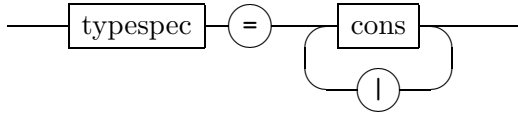
The behavior for plain meta-level function types is unchanged. Mixed continuous and meta-level application is *not* supported.

5.3.2 Recursive domains

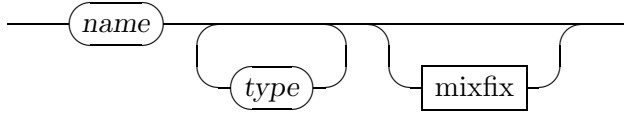
domain : *theory* → *theory*



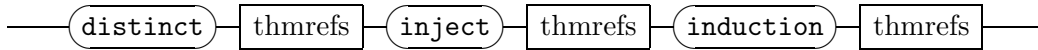
dmspec



cons



dtrules



Recursive domains in HOLCF are analogous to datatypes in classical HOL (cf. §5.2.4). Mutual recursion is supported, but no nesting nor arbitrary branching. Domain constructors may be strict (default) or lazy, the latter admits to introduce infinitary objects in the typical LCF manner (e.g. lazy lists). See also [5] for a general discussion of HOLCF domains.

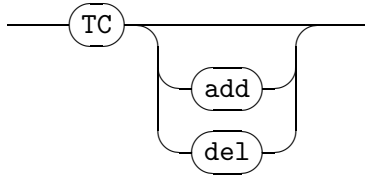
5.4 ZF

5.4.1 Type checking

The ZF logic is essentially untyped, so the concept of “type checking” is performed as logical reasoning about set-membership statements. A special

method assists users in this task; a version of this is already declared as a “solver” in the standard Simplifier setup.

print_tcset* : $theory \mid proof \rightarrow theory \mid proof$
typecheck : *method*
TC : *attribute*



print_tcset prints the collection of typechecking rules of the current context.

Note that the component built into the Simplifier only knows about those rules being declared globally in the theory!

typecheck attempts to solve any pending type-checking problems in subgoals.

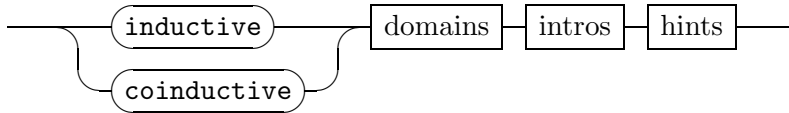
TC adds or deletes type-checking rules from the context.

5.4.2 (Co)Inductive sets and datatypes

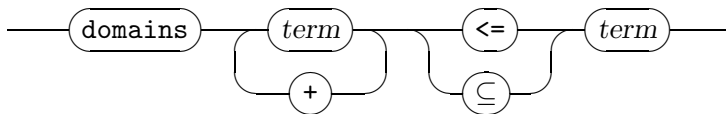
Set definitions

In ZF everything is a set. The generic inductive package also provides a specific view for “datatype” specifications. Coinductive definitions are available in both cases, too.

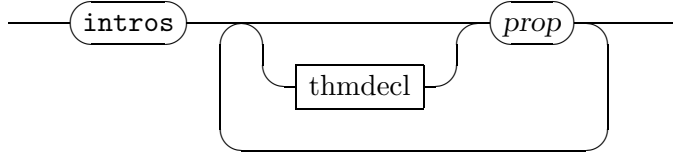
inductive : $theory \rightarrow theory$
coinductive : $theory \rightarrow theory$
datatype : $theory \rightarrow theory$
codatatype : $theory \rightarrow theory$



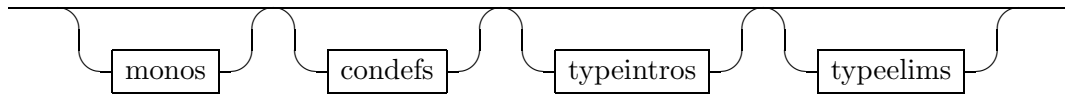
domains



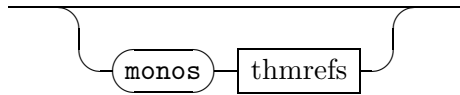
intros



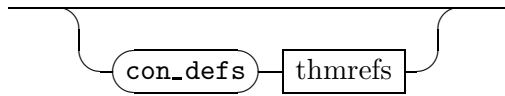
hints



monos



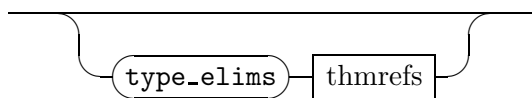
condefs



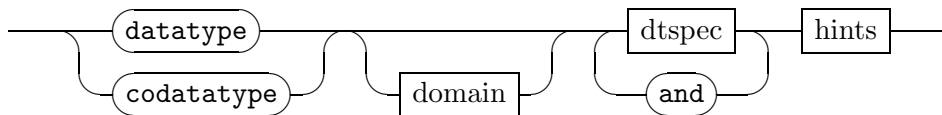
typeintros

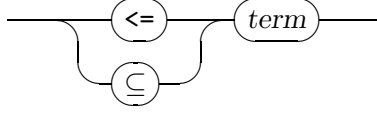
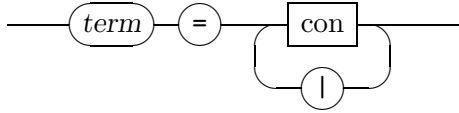
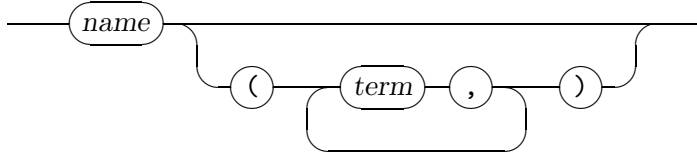
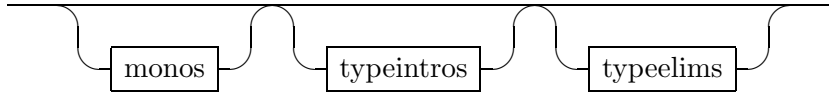


typeelims



In the following diagram *monos*, *typeintros*, and *typeelims* are the same as above.

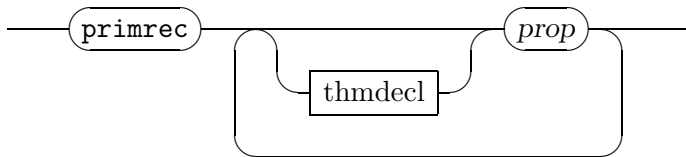


domain*dtspec**con**hints*

See [12] for further information on inductive definitions in HOL, but note that this covers the old-style theory format.

Primitive recursive functions

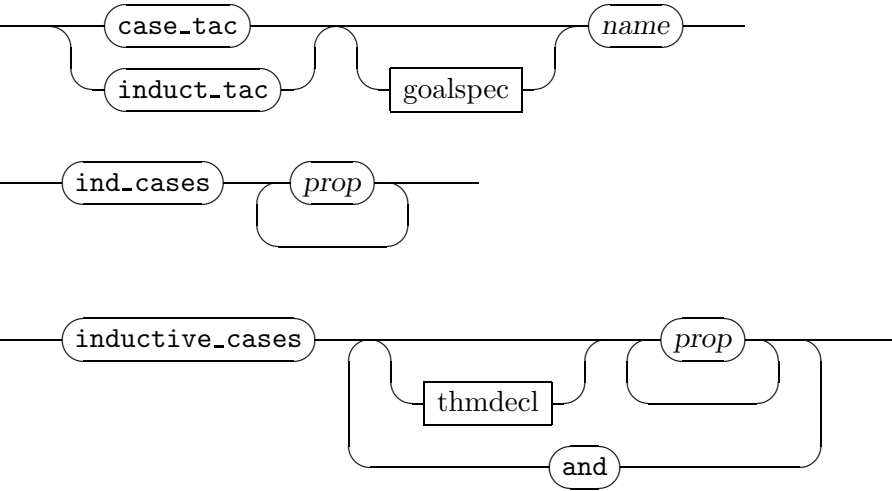
primrec : *theory* \rightarrow *theory*



Cases and induction: emulating tactic scripts

The following important tactical tools of Isabelle/ZF have been ported to Isar. These should be never used in proper proof texts!

*case_tac** : *method*
*induct_tac** : *method*
*ind_cases** : *method*
inductive_cases : *theory* \rightarrow *theory*



Isabelle/Isar quick reference

A.1 Proof commands

A.1.1 Primitives and basic syntax

fix \bar{x}	augment context by $\bigwedge \bar{x} . \square$
assume $a: \bar{\varphi}$	augment context by $\bar{\varphi} \implies \square$
then	indicate forward chaining of facts
have $a: \varphi$	prove local result
show $a: \varphi$	prove local result, establishing some goal
using \bar{a}	indicate use of additional facts
proof $m_1 \dots \mathbf{qed}$ m_2	apply proof methods
{ ... }	declare explicit blocks
next	switch implicit blocks
note $a = \bar{b}$	reconsider facts
let $p = t$	abbreviate terms by higher-order matching

$theory-stmt$	=	theorem $name: prop proof$
		lemma $name: prop proof$
		types ... consts ... defs
$proof$	=	$prfx^* \mathbf{proof} method stmt^* \mathbf{qed} method$
$prfx$	=	apply $method$
		using $name^+$
$stmt$	=	{ $stmt^*$ }
		next
		note $name = name^+$
		let $term = term$
		fix var^+
		assume $name: prop^+$
		then $goal-stmt$
		$goal$
$goal$	=	have $name: prop proof$
		show $name: prop proof$

A.1.2 Abbreviations and synonyms

by m_1 m_2 \equiv **proof** m_1 **qed** m_2
 \dots \equiv **by rule**
 \cdot \equiv **by this**
hence \equiv **then have**
thus \equiv **then show**
from \bar{a} \equiv **note** $this = \bar{a}$ **then**
with \bar{a} \equiv **from** \bar{a} **and** $this$

from $this$ \equiv **then**
from $this$ **have** \equiv **hence**
from $this$ **show** \equiv **thus**

A.1.3 Derived elements

also₀ \approx **note** $calculation = this$
also _{$n+1$} \approx **note** $calculation = trans [OF\ calculation\ this]$
finally \approx **also from** $calculation$
moreover \approx **note** $calculation = calculation\ this$
ultimately \approx **moreover from** $calculation$

presume $a: \bar{\varphi}$ \approx **assume** $a: \bar{\varphi}$
def $a: x \equiv t$ \approx **fix** x **assume** $a: x \equiv t$
obtain \bar{x} **where** $a: \bar{\varphi}$ \approx \dots **fix** \bar{x} **assume** $a: \bar{\varphi}$
case c \approx **fix** \bar{x} **assume** $c: \bar{\varphi}$
sorry \approx **by cheating**

A.1.4 Diagnostic commands

pr print current state
thm \bar{a} print theorems
term t print term
prop φ print meta-level proposition
typ τ print meta-level type

A.2 Proof methods

Single steps (forward-chaining facts)

<i>assumption</i>	apply some assumption
<i>this</i>	apply current facts
<i>rule</i> \bar{a}	apply some rule
<i>rule</i>	apply standard rule (default for proof)
<i>contradiction</i>	apply \neg elimination rule (any order)
<i>cases</i> t	case analysis (provides cases)
<i>induct</i> \bar{x}	proof by induction (provides cases)

Repeated steps (inserting facts)

—	no rules
<i>intro</i> \bar{a}	introduction rules
<i>intro_classes</i>	class introduction rules
<i>elim</i> \bar{a}	elimination rules
<i>unfold</i> \bar{a}	definitions

Automated proof tools (inserting facts, or even prems!)

<i>rules</i>	intuitionistic proof search
<i>simp</i> , <i>simp_all</i>	Simplifier (+ Splitter)
<i>blast</i> , <i>fast</i>	Classical Reasoner
<i>auto</i> , <i>force</i>	Simplifier + Classical Reasoner
<i>arith</i>	Arithmetic procedure

A.3 Attributes

Operations

<i>OF</i> \bar{a}	rule applied to facts (skipping “-”)
<i>of</i> \bar{t}	rule applied to terms (skipping “-”)
<i>symmetric</i>	resolution with symmetry rule
<i>THEN</i> b	resolution with another rule
<i>rule_format</i>	result put into standard rule format
<i>elim_format</i>	destruct rule turned into elimination rule format

Declarations

<i>simp</i>	Simplifier rule
<i>intro</i> , <i>elim</i> , <i>dest</i>	Pure or Classical Reasoner rule
<i>iff</i>	Simplifier + Classical Reasoner rule
<i>split</i>	case split rule
<i>trans</i>	transitivity rule
<i>sym</i>	symmetry rule

A.4 Emulating tactic scripts

A.4.1 Commands

apply m	apply proof method at initial position
apply_end (m)	apply proof method near terminal position
done	complete proof
defer n	move subgoal to end
prefer n	move subgoal to beginning
back	backtrack last command
declare	declare rules in current theory

A.4.2 Methods

<i>rule_tac insts</i>	resolution (with instantiation)
<i>erule_tac insts</i>	elim-resolution (with instantiation)
<i>drule_tac insts</i>	destruct-resolution (with instantiation)
<i>frule_tac insts</i>	forward-resolution (with instantiation)
<i>cut_tac insts</i>	insert facts (with instantiation)
<i>thin_tac φ</i>	delete assumptions
<i>subgoal_tac φ</i>	new claims
<i>rename_tac \bar{x}</i>	rename suffix of goal parameters
<i>rotate_tac n</i>	rotate assumptions of goal
<i>tactic text</i>	arbitrary ML tactic
<i>case_tac t</i>	exhaustion (datatypes)
<i>induct_tac \bar{x}</i>	induction (datatypes)
<i>ind_cases t</i>	exhaustion + simplification (inductive sets)

Isabelle/Isar conversion guide

Subsequently, we give a few practical hints on working in a mixed environment of old Isabelle ML proof scripts and new Isabelle/Isar theories. There are basically three ways to cope with this issue.

1. Do not convert old sources at all, but communicate directly at the level of *internal* theory and theorem values.
2. Port old-style theory files to new-style ones (very easy), and ML proof scripts to Isar tactic-emulation scripts (quite easy).
3. Actually redo ML proof scripts as human-readable Isar proof texts (probably hard, depending who wrote the original scripts).

B.1 No conversion

Internally, Isabelle is able to handle both old and new-style theories at the same time; the theory loader automatically detects the input format. In any case, the results are certain internal ML values of type `theory` and `thm`. These may be accessed from either classic Isabelle or Isabelle/Isar, provided that some minimal precautions are observed.

B.1.1 Referring to theorem and theory values

```
thm      : xstring -> thm
thms     : xstring -> thm list
the_context : unit -> theory
theory   : string -> theory
```

These functions provide general means to refer to logical objects from ML. Old-style theories used to emit many ML bindings of theorems and theories, but this is no longer done in new-style Isabelle/Isar theories.

`thm name` and `thms name` retrieve theorems stored in the current theory context, including any ancestor node.

The convention of old-style theories was to bind any theorem as an ML value as well. New-style theories no longer do this, so ML code may require `thm "foo"` rather than just `foo`.

`the_context()` refers to the current theory context.

Old-style theories often use the ML binding `thy`, which is dynamically created by the ML code generated from old theory source. This is no longer the recommended way in any case! Function `the_context` should be used for old scripts as well.

`theory name` retrieves the named theory from the global theory-loader database.

The ML code generated from old-style theories would include an ML binding `name.thy` as part of an ML structure.

B.1.2 Storing theorem values

```
qed      : string -> unit
bind_thm : string * thm -> unit
bind_thms : string * thm list -> unit
```

ML proof scripts have to be well-behaved by storing theorems properly within the current theory context, in order to enable new-style theories to retrieve these later.

`qed name` is the canonical way to conclude a proof script in ML. This already manages entry in the theorem database of the current theory context.

`bind_thm (name, thm)` and `bind_thms (name, thms)` store theorems that have been produced in ML in an ad-hoc manner.

Note that the original “LCF-system” approach of binding theorem values on the ML toplevel only has long been given up in Isabelle! Despite of this, old legacy proof scripts occasionally contain code such as `val foo = result();` which is ill-behaved in several respects. Apart from preventing access from Isar theories, it also omits the result from the WWW presentation, for example.

B.1.3 ML declarations in Isar

```
ML      :  $\cdot \rightarrow \cdot$ 
ML_setup : theory  $\rightarrow$  theory
```

Isabelle/Isar theories may contain ML declarations as well. For example, an old-style theorem binding may be mimicked as follows.

```
ML {* val foo = thm "foo" *}
```

Note that this command cannot be undone, so invalid theorem bindings in ML may persist. Also note that the current theory may not be modified; use **ML_setup** for declarations that act on the current context.

B.2 Porting theories and proof scripts

Porting legacy theory and ML files to proper Isabelle/Isar theories has several advantages. For example, the Proof General user interface [1] for Isabelle/Isar is more robust and more comfortable to use than the version for classic Isabelle. This is due to the fact that the generic ML toplevel has been replaced by a separate Isar interaction loop, with full control over input synchronization and error conditions.

Furthermore, the Isabelle document preparation system (see also [18]) only works properly with new-style theories. Output of old-style sources is at the level of individual characters (and symbols), without proper document markup as in Isabelle/Isar theories.

B.2.1 Theories

Basically, the Isabelle/Isar theory syntax is a proper superset of the classic one. Only a few quirks and legacy problems have been eliminated, resulting in simpler rules and less special cases. The main changes of theory syntax are as follows.

- Quoted strings may contain arbitrary white space, and span several lines without requiring `\... \` escapes.
- Names may always be quoted.

The old syntax would occasionally demand plain identifiers vs. quoted strings to accommodate certain syntactic features.

- Types and terms have to be *atomic* as far as the theory syntax is concerned; this typically requires quoting of input strings, e.g. `" $x + y$ "`. The old theory syntax used to fake part of the syntax of types in order to require less quoting in common cases; this was hard to predict, though. On the other hand, Isar does not require quotes for simple terms, such as plain identifiers x , numerals 1, or symbols \forall (input as `\<forall>`).

- Theorem declarations require an explicit colon to separate the name from the statement (the name is usually optional). Cf. the syntax of **defs** in §3.1.5, or **theorem** in §3.1.7.

Note that Isabelle/Isar error messages are usually quite explicit about the problem at hand. So in cases of doubt, input syntax may be just as well tried out interactively.

B.2.2 Goal statements

Simple goals

In ML the canonical a goal statement together with a complete proof script is as follows:

```
Goal " $\varphi$ ";
by  $tac_1$ ;
 $\vdots$ 
qed " $name$ ";
```

This form may be turned into an Isar tactic-emulation script like this:

```
lemma  $name$ : " $\varphi$ "
  apply  $meth_1$ 
   $\vdots$ 
done
```

Note that the main statement may be **theorem** or **corollary** as well. See §B.2.3 for further details on how to convert actual tactic expressions into proof methods.

Classic Isabelle provides many variant forms of goal commands, see also [10] for further details. The second most common one is **Goalw**, which expands definitions before commencing the actual proof script.

```
Goalw [ $def_1$ , ...] " $\varphi$ ";
```

This may be replaced by using the *unfold* proof method explicitly.

```
lemma  $name$ : " $\varphi$ "
  apply (unfold  $def_1$  ...)
```

Deriving rules

Deriving non-atomic meta-level propositions requires special precautions in classic Isabelle: the primitive `goal` command decomposes a statement into the atomic conclusion and a list of assumptions, which are exhibited as ML values of type `thm`. After the proof is finished, these premises are discharged again, resulting in the original rule statement. The “long format” of Isabelle/Isar goal statements admits to emulate this technique nicely. The general ML goal statement for derived rules looks like this:

```
val [prem1, ...] = goal "φ1 ⇒ ... ⇒ ψ";
by tac1;
  ⋮
qed "a"
```

This form may be turned into a tactic-emulation script as follows:

```
lemma a:
  assumes prem1: "φ1" and ...
  shows "ψ"
    apply meth1
      ⋮
    done
```

In practice, actual rules are often rather direct consequences of corresponding atomic statements, typically stemming from the definition of a new concept. In that case, the general scheme for deriving rules may be greatly simplified, using one of the standard automated proof tools, such as *simp*, *blast*, or *auto*. This could work as follows:

```
lemma "φ1 ⇒ ... ⇒ ψ"
  by (unfold defs) blast
```

Note that classic Isabelle would support this form only in the special case where φ_1, \dots are atomic statements (when using the standard `Goal` command). Otherwise the special treatment of rules would be applied, disturbing this simple setup.

Occasionally, derived rules would be established by first proving an appropriate atomic statement (using \forall and \longrightarrow of the object-logic), and putting the final result into “rule format”. In classic Isabelle this would usually proceed as follows:


```

Goal " $\varphi$ ";
by  $tac_1$ ;
 $\vdots$ 
qed_spec_mp " $name$ ";

```

The operation performed by `qed_spec_mp` is also performed by the Isar attribute “`rule_format`”, see also §5.1. Thus the corresponding Isar text may look like this:

```

lemma  $name$  [rule_format]: " $\varphi$ "
  apply  $meth_1$ 
   $\vdots$ 
done

```

Note plain “`rule_format`” actually performs a slightly different operation: it fully replaces object-level implication and universal quantification throughout the whole result statement. This is the right thing in most cases. For historical reasons, `qed_spec_mp` would only operate on the conclusion; one may get this exact behavior by using “`rule_format (no_asm)`” instead.

Actually “`rule_format`” is a bit unpleasant to work with, since the final result statement is not shown in the text. An alternative is to state the resulting rule in the intended form in the first place, and have the initial refinement step turn it into internal object-logic form using the `atomize` method indicated below. The remaining script is unchanged.

```

lemma  $name$ : " $\bigwedge \bar{x}. \bar{\varphi} \implies \psi$ "
  apply (atomize (full))
  apply  $meth_1$ 
   $\vdots$ 
done

```

In many situations the `atomize` step above is actually unnecessary, especially if the subsequent script mainly consists of automated tools.

B.2.3 Tactics

Isar Proof methods closely resemble traditional tactics, when used in unstructured sequences of **apply** commands (cf. §B.2.2). Isabelle/Isar provides emulations for all major ML tactics of classic Isabelle — mostly for the sake of easy porting of existing developments, as actual Isar proof texts would demand much less diversity of proof methods.

Unlike tactic expressions in ML, Isar proof methods provide proper concrete syntax for additional arguments, options, modifiers etc. Thus a typical method text is usually more concise than the corresponding ML tactic. Furthermore, the Isar versions of classic Isabelle tactics often cover several variant forms by a single method with separate options to tune the behavior. For example, method *simp* replaces all of *simp_tac* / *asm_simp_tac* / *full_simp_tac* / *asm_full_simp_tac*, there is also concrete syntax for augmenting the Simplifier context (the current “simpset”) in a convenient way.

Resolution tactics

Classic Isabelle provides several variant forms of tactics for single-step rule applications (based on higher-order resolution). The space of resolution tactics has the following main dimensions.

1. The “mode” of resolution: *intro*, *elim*, *destruct*, or *forward* (e.g. *resolve_tac*, *eresolve_tac*, *dresolve_tac*, *forward_tac*).
2. Optional explicit instantiation (e.g. *resolve_tac* vs. *res_inst_tac*).
3. Abbreviations for singleton arguments (e.g. *resolve_tac* vs. *rtac*).

Basically, the set of Isar tactic emulations *rule_tac*, *erule_tac*, *drule_tac*, *frule_tac* (see §4.3.2) would be sufficient to cover the four modes, either with or without instantiation, and either with single or multiple arguments. Although it is more convenient in most cases to use the plain *rule* method (see §3.2.6), or any of its “improper” variants *erule*, *drule*, *frule* (see §4.3.1). Note that explicit goal addressing is only supported by the actual *rule_tac* version.

With this in mind, plain resolution tactics may be ported as follows.

<i>rtac</i> <i>a</i> 1	<i>rule</i> <i>a</i>
<i>resolve_tac</i> [<i>a</i> ₁ ,...] 1	<i>rule</i> <i>a</i> ₁ ...
<i>res_inst_tac</i> [(<i>x</i> ₁ , <i>t</i> ₁),...] <i>a</i> 1	<i>rule_tac</i> <i>x</i> ₁ = <i>t</i> ₁ and ... in <i>a</i>
<i>rtac</i> <i>a</i> <i>i</i>	<i>rule_tac</i> [<i>i</i>] <i>a</i>
<i>resolve_tac</i> [<i>a</i> ₁ ,...] <i>i</i>	<i>rule_tac</i> [<i>i</i>] <i>a</i> ₁ ...
<i>res_inst_tac</i> [(<i>x</i> ₁ , <i>t</i> ₁),...] <i>a</i> <i>i</i>	<i>rule_tac</i> [<i>i</i>] <i>x</i> ₁ = <i>t</i> ₁ and ... in <i>a</i>

Note that explicit goal addressing may be usually avoided by changing the order of subgoals with **defer** or **prefer** (see §3.2.9).

Some further (less frequently used) combinations of basic resolution tactics may be expressed as follows.

<code>ares_tac [a₁, ...] 1</code>	<code>assumption rule a₁ ...</code>
<code>eatac a n 1</code>	<code>erule (n) a</code>
<code>datac a n 1</code>	<code>drule (n) a</code>
<code>fatac a n 1</code>	<code>frule (n) a</code>

Simplifier tactics

The main Simplifier tactics `Simp_tac`, `simp_tac` and variants (cf. [10]) are all covered by the `simp` and `simp_all` methods (see §4.3.3). Note that there is no individual goal addressing available, simplification acts either on the first goal (`simp`) or all goals (`simp_all`).

<code>Asm_full_simp_tac 1</code>	<code>simp</code>
<code>ALLGOALS Asm_full_simp_tac</code>	<code>simp_all</code>
<code>Simp_tac 1</code>	<code>simp (no_asm)</code>
<code>Asm_simp_tac 1</code>	<code>simp (no_asm_simp)</code>
<code>Full_simp_tac 1</code>	<code>simp (no_asm_use)</code>

Isar also provides separate method modifier syntax for augmenting the Simplifier context (see §4.3.3), which is known as the “simpset” in ML. A typical ML expression with simpset changes looks like this:

```
asm_full_simp_tac (simpset () addsimps [a1, ...] delsimps [b1, ...]) 1
```

The corresponding Isar text is as follows:

```
simp add : a1 ... del : b1 ...
```

Global declarations of Simplifier rules (e.g. `Addsimps`) are covered by application of attributes, see §B.2.4 for more information.

Classical Reasoner tactics

The Classical Reasoner provides a rather large number of variations of automated tactics, such as `Blast_tac`, `Fast_tac`, `Clarify_tac` etc. (see [10]). The corresponding Isar methods usually share the same base name, such as `blast`, `fast`, `clarify` etc. (see §4.3.4).

Similar to the Simplifier, there is separate method modifier syntax for augmenting the Classical Reasoner context, which is known as the “claset” in ML. A typical ML expression with claset changes looks like this:

```
blast_tac (claset () addIs [a1, ...] addSEs [b1, ...]) 1
```

The corresponding Isar text is as follows:

```
blast intro : a1 ... elim! : b1 ...
```

Global declarations of Classical Reasoner rules (e.g. `AddIs`) are covered by application of attributes, see §B.2.4 for more information.

Miscellaneous tactics

There are a few additional tactics defined in various theories of Isabelle/HOL, some of these also in Isabelle/FOL or Isabelle/ZF. The most common ones of these may be ported to Isar as follows.

<code>stac a 1</code>	<code>subst a</code>
<code>hyp_subst_tac 1</code>	<code>hypsubst</code>
<code>strip_tac 1</code>	\approx <code>intro strip</code>
<code>split_all_tac 1</code>	<code>simp (no_asm_simp) only : split_tupled_all</code>
	\approx <code>simp only : split_tupled_all</code>
	\ll <code>clarify</code>

Tacticals

Classic Isabelle provides a huge amount of tacticals for combination and modification of existing tactics. This has been greatly reduced in Isar, providing the bare minimum of combinators only: “,” (sequential composition), “|” (alternative choices), “?” (try), “+” (repeat at least once). These are usually sufficient in practice; if all fails, arbitrary ML tactic code may be invoked via the *tactic* method (see §4.3.2).

Common ML tacticals may be expressed directly in Isar as follows:

<code>tac₁ THEN tac₂</code>	<code>meth₁, meth₂</code>
<code>tac₁ ORELSE tac₂</code>	<code>meth₁ meth₂</code>
<code>TRY tac</code>	<code>meth?</code>
<code>REPEAT1 tac</code>	<code>meth+</code>
<code>REPEAT tac</code>	<code>(meth+)?</code>
<code>EVERY [tac₁, ...]</code>	<code>meth₁, ...</code>
<code>FIRST [tac₁, ...]</code>	<code>meth₁ ...</code>

`CHANGED` (see [10]) is usually not required in Isar, since most basic proof methods already fail unless there is an actual change in the goal state. Nevertheless, “?” (try) may be used to accept *unchanged* results as well.

`ALLGOALS`, `SOMEGOAL` etc. (see [10]) are not available in Isar, since there is no direct goal addressing. Nevertheless, some basic methods address all goals internally, notably `simp_all` (see §4.3.3). Also note that `ALLGOALS` may be often replaced by “+” (repeat at least once), although this usually has a different operational behavior, such as solving goals in a different order.

Iterated resolution, such as `REPEAT (FIRSTGOAL (resolve_tac ...))`, is usually better expressed using the *intro* and *elim* methods of Isar (see §4.3.4).

B.2.4 Declarations and ad-hoc operations

Apart from proof commands and tactic expressions, almost all of the remaining ML code occurring in legacy proof scripts are either global context declarations (such as `Addsimps`) or ad-hoc operations on theorems (such as `RS`). In Isar both of these are covered by theorem expressions with *attributes*.

Theorem operations may be attached as attributes in the very place where theorems are referenced, say within a method argument. The subsequent ML combinators may be expressed directly in Isar as follows.

<code>thm₁ RS thm₂</code>	<code>thm₁ [THEN thm₂]</code>
<code>thm₁ RSN (i, thm₂)</code>	<code>thm₁ [THEN [i] thm₂]</code>
<code>thm₁ COMP thm₂</code>	<code>thm₁ [COMP thm₂]</code>
<code>[thm₁, ...] MRS thm</code>	<code>thm [OF thm₁ ...]</code>
<code>read_instantiate [(x₁, t₁), ...] thm</code>	<code>thm [where x₁ = t₁ and ...]</code>
<code>make_elim thm</code>	<code>thm [elim_format]</code>
<code>standard thm</code>	<code>thm [standard]</code>

Note that *OF* is often more readable as *THEN*; likewise positional instantiation with *of* is often more appropriate than *where*.

The special ML command `qed_spec_mp` of Isabelle/HOL and FOL may be replaced by passing the result of a proof through *rule_format*.

Global ML declarations may be expressed using the **declare** command (see §3.2.9) together with appropriate attributes. The most common ones

are as follows.

<code>Addsimps [thm]</code>	<code>declare thm [simp]</code>
<code>Delsimps [thm]</code>	<code>declare thm [simp del]</code>
<code>Addsplits [thm]</code>	<code>declare thm [split]</code>
<code>Delsplits [thm]</code>	<code>declare thm [split del]</code>
<code>AddIs [thm]</code>	<code>declare thm [intro]</code>
<code>AddEs [thm]</code>	<code>declare thm [elim]</code>
<code>AddDs [thm]</code>	<code>declare thm [dest]</code>
<code>AddSIs [thm]</code>	<code>declare thm [intro!]</code>
<code>AddSEs [thm]</code>	<code>declare thm [elim!]</code>
<code>AddSDs [thm]</code>	<code>declare thm [dest!]</code>
<code>AddIffs [thm]</code>	<code>declare thm [iff]</code>

Note that explicit **declare** commands are rarely needed in practice; Isar admits to declare theorems on-the-fly wherever they emerge. Consider the following ML idiom:

```
Goal " $\varphi$ ";
:
qed "name";
Addsimps [name];
```

This may be expressed more succinctly in Isar like this:

```
lemma name [simp]:  $\varphi$ 
:
```

The *name* may be even omitted, although this would make it difficult to declare the theorem otherwise later (e.g. as `[simp del]`).

B.3 Writing actual Isar proof texts

Porting legacy ML proof scripts into Isar tactic emulation scripts (see §B.2) is mainly a technical issue, since the basic representation of formal “proof script” is preserved. In contrast, converting existing Isabelle developments into actual human-readably Isar proof texts is more involved, due to the fundamental change of the underlying paradigm.

This issue is comparable to that of converting programs written in a low-level programming languages (say Assembler) into higher-level ones (say Haskell). In order to accomplish this, one needs a working knowledge of the target language, as well an understanding of the *original* idea of the piece of code expressed in the low-level language.

As far as Isar proofs are concerned, it is usually much easier to re-use only definitions and the main statements, while following the arrangement of proof scripts only very loosely. Ideally, one would also have some *informal* proof outlines available for guidance as well. In the worst case, obscure proof scripts would have to be re-engineered by tracing forth and backwards, and by educated guessing!

This is a possible schedule to embark on actual conversion of legacy proof scripts into Isar proof texts.

1. Port ML scripts to Isar tactic emulation scripts (see §B.2).
2. Get sufficiently acquainted with Isabelle/Isar proof development.¹
3. Recover the proof structure of a few important theorems.
4. Rephrase the original intention of the course of reasoning in terms of Isar proof language elements.

Certainly, rewriting formal reasoning in Isar requires some additional effort. On the other hand, one gains a human-readable representation of machine-checked formal proof. Depending on the context of application, this might be even indispensable to start with!

¹As there is still no Isar tutorial around, it is best to look at existing Isar examples, see also §1.3.2.

Bibliography

- [1] David Aspinall. Proof General. <http://www.proofgeneral.org>.
- [2] David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer-Verlag, 2000.
- [3] Gertrud Bauer and Markus Wenzel. Computer-assisted mathematics at work — the Hahn-Banach theorem in Isabelle/Isar. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs: TYPES'99*, LNCS, 2000.
- [4] Gertrud Bauer and Markus Wenzel. Calculational reasoning revisited — an Isabelle/Isar experience. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
- [5] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
- [6] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Jim Grundy and Malcom Newey, editors, *Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
- [7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle's Logics: HOL*. <http://isabelle.in.tum.de/doc/logics-HOL.pdf>.
- [8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS 2283.
- [9] Lawrence C. Paulson. *Introduction to Isabelle*. <http://isabelle.in.tum.de/doc/intro.pdf>.
- [10] Lawrence C. Paulson. *The Isabelle Reference Manual*. <http://isabelle.in.tum.de/doc/ref.pdf>.
- [11] Lawrence C. Paulson. *Isabelle's Logics*. <http://isabelle.in.tum.de/doc/logics.pdf>.

- [12] Lawrence C. Paulson. *Isabelle's Logics: FOL and ZF*.
<http://isabelle.in.tum.de/doc/logics-ZF.pdf>.
- [13] Christoph Wedler. Emacs package “X-Symbol”.
<http://www.fmi.uni-passau.de/~wedler/x-symbol/>.
- [14] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
- [15] Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- [16] Markus Wenzel. *Using Axiomatic Type Classes in Isabelle*, 2000.
<http://isabelle.in.tum.de/doc/axclass.pdf>.
- [17] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.
<http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html>.
- [18] Markus Wenzel and Stefan Berghofer. *The Isabelle System Manual*.
<http://isabelle.in.tum.de/doc/system.pdf>.
- [19] Freek Wiedijk. The mathematical vernacular. Unpublished paper, 2000.
<http://www.cs.kun.nl/~freek/notes/mv.ps.gz>.

Index

- (method), [40](#)
- . (command), [38](#)
- .. (command), [38](#)
- ... (variable), [44](#)
- { (command), [44](#)
- } (command), [44](#)
- _ (theorem), [36](#), [55](#)
- _ (variable), [43](#)
- also (command), [57](#)
- Antiquotations
 - goals, [18](#)
 - prop, [18](#)
 - subgoals, [18](#)
 - term, [18](#)
 - text, [18](#)
 - thm, [18](#)
 - typ, [18](#)
- apply (command), [45](#)
- apply-end (command), [45](#)
- args (syntax), [15](#)
- arith (HOL method), [90](#)
- arith-split (HOL attribute), [90](#)
- arities (command), [25](#)
- arity (syntax), [11](#)
- assume (command), [33](#)
- assumption (method), [40](#)
- atom (syntax), [15](#)
- atomize (attribute), [79](#)
- atomize (method), [79](#)
- Attributes
 - arith-split (HOL), [90](#)
 - atomize, [79](#)
 - case-names, [73](#)
 - cases, [78](#)
 - code, [92](#)
 - COMP, [60](#)
 - cong, [66](#)
 - consumes, [73](#)
 - dest, [72](#)
 - dest (Pure), [40](#)
 - elim, [72](#)
 - elim (Pure), [40](#)
 - elim-format, [73](#)
 - elim-format (Pure), [60](#)
 - folded, [60](#)
 - iff, [72](#)
 - induct, [78](#)
 - intro, [72](#)
 - intro (Pure), [40](#)
 - mono (HOL), [89](#)
 - no-vars, [60](#)
 - OF, [40](#)
 - of, [40](#)
 - params, [73](#)
 - recdef-cong (HOL), [89](#)
 - recdef-simp (HOL), [89](#)
 - recdef-wf (HOL), [89](#)
 - rule, [72](#)
 - rule (Pure), [40](#)
 - rule-format, [79](#)
 - rulify, [79](#)
 - simp, [66](#)
 - simplified, [66](#)
 - split, [66](#)
 - split-format (HOL), [82](#)
 - standard, [60](#)
 - swapped, [73](#)
 - sym, [57](#)

- symmetric, [57](#)
- tagged, [60](#)
- TC, [94](#)
- THEN, [60](#)
- trans, [57](#)
- typecheck, [94](#)
- unfolded, [60](#)
- untagged, [60](#)
- where, [60](#)
- attributes (syntax), [15](#)
- auto (method), [70](#)
- axclass (command), [51](#)
- axioms (command), [27](#)
- axmdecl (syntax), [15](#)
- back (command), [45](#)
- best (method), [69](#)
- bestsimp (method), [70](#)
- blast (method), [69](#)
- by (command), [38](#)
- calculation (theorem), [57](#)
- case (command), [73](#)
- case (variable), [77](#)
- case-names (attribute), [73](#)
- case-tac (HOL method), [90](#)
- case-tac (ZF method), [96](#)
- Cases
 - rule-context, [38](#)
- cases (attribute), [78](#)
- cases (method), [75](#)
- cd (command), [50](#)
- chapter (command), [23](#)
- clamod (syntax), [69](#)
- clarify (method), [69](#)
- clarsimp (method), [70](#)
- clasimpmod (syntax), [70](#)
- classdecl (syntax), [11](#)
- classes (command), [24](#)
- classrel (command), [24](#)
- codatatype (ZF command), [94](#)
- code (attribute), [92](#)
- coinductive (HOL command), [89](#)
- coinductive (ZF command), [94](#)
- Commands
 - ., [38](#)
 - .., [38](#)
 - {, [44](#)
 - }, [44](#)
 - also, [57](#)
 - apply, [45](#)
 - apply-end, [45](#)
 - arities, [25](#)
 - assume, [33](#)
 - axclass, [51](#)
 - axioms, [27](#)
 - back, [45](#)
 - by, [38](#)
 - case, [73](#)
 - cd, [50](#)
 - chapter, [23](#)
 - classes, [24](#)
 - classrel, [24](#)
 - codatatype (ZF), [94](#)
 - coinductive (HOL), [89](#)
 - coinductive (ZF), [94](#)
 - constdefs, [26](#)
 - constdefs (HOLCF), [92](#)
 - consts, [26](#)
 - consts (HOLCF), [92](#)
 - consts-code, [92](#)
 - context, [21](#)
 - corollary, [36](#)
 - datatype (HOL), [86](#)
 - datatype (ZF), [94](#)
 - declare, [45](#)
 - def, [33](#)
 - defaultsort, [24](#)
 - defer, [45](#)
 - defs, [26](#)
 - domain (HOLCF), [93](#)
 - done, [45](#)

- end, [21](#)
- finally, [57](#)
- fix, [33](#)
- from, [34](#)
- generate-code, [92](#)
- global, [28](#)
- have, [36](#)
- header, [21](#)
- hence, [36](#)
- hide, [28](#)
- inductive (HOL), [89](#)
- inductive (ZF), [94](#)
- inductive-cases (HOL), [90](#)
- inductive-cases (ZF), [96](#)
- instance, [51](#)
- judgment, [79](#)
- kill, [49](#)
- lemma, [36](#)
- lemmas, [27](#)
- let, [43](#)
- local, [28](#)
- locale, [53](#)
- method-setup, [29](#)
- ML, [29](#)
- ML-command, [29](#)
- ML-setup, [29](#)
- moreover, [57](#)
- next, [44](#)
- nonterminals, [25](#)
- note, [34](#)
- obtain, [56](#)
- oops, [46](#)
- oracle, [31](#)
- parse-ast-translation, [30](#)
- parse-translation, [30](#)
- pr, [47](#)
- prefer, [45](#)
- presume, [33](#)
- primrec (HOL), [87](#)
- primrec (ZF), [96](#)
- print-ast-translation, [30](#)
- print-attributes, [48](#)
- print-binds, [48](#)
- print-cases, [73](#)
- print-claset, [72](#)
- print-commands, [48](#)
- print-facts, [48](#)
- print-induct-rules, [78](#)
- print-locale, [53](#)
- print-locales, [53](#)
- print-methods, [48](#)
- print-simpset, [66](#)
- print-syntax, [48](#)
- print-tcset, [94](#)
- print-theorems, [48](#)
- print-trans-rules, [57](#)
- print-translation, [30](#)
- proof, [38](#)
- prop, [47](#)
- pwd, [50](#)
- qed, [38](#)
- recdef (HOL), [87](#)
- recdef-tc (HOL), [87](#)
- record (HOL), [83](#)
- redo, [49](#)
- rep-datatype (HOL), [86](#)
- sect, [32](#)
- section, [23](#)
- setup, [29](#)
- show, [36](#)
- sorry, [38](#)
- subsect, [32](#)
- subsection, [23](#)
- subsubsect, [32](#)
- subsubsection, [23](#)
- syntax, [26](#)
- term, [47](#)
- text, [23](#)
- text-raw, [23](#)
- then, [34](#)
- theorem, [36](#)
- theorems, [27](#)

- theory, [21](#)
- thm, [47](#)
- thm-deps, [48](#)
- thms-containing, [48](#)
- thus, [36](#)
- token-translation, [30](#)
- translations, [26](#)
- txt, [32](#)
- txt-raw, [32](#)
- typ, [47](#)
- typed-print-translation, [30](#)
- typeddecl, [25](#)
- typeddecl (HOL), [80](#)
- typedef (HOL), [80](#)
- types, [25](#)
- types-code, [92](#)
- ultimately, [57](#)
- undo, [49](#)
- update-thy, [50](#)
- update-thy-only, [50](#)
- use, [29](#)
- use-thy, [50](#)
- use-thy-only, [50](#)
- using, [34](#)
- with, [34](#)
- comment (syntax), [10](#)
- COMP (attribute), [60](#)
- cong (attribute), [66](#)
- constdecl (syntax), [26](#)
- constdefs (command), [26](#)
- constdefs (HOLCF command), [92](#)
- consts (command), [26](#)
- consts (HOLCF command), [92](#)
- consts-code (command), [92](#)
- consumes (attribute), [73](#)
- context (command), [21](#)
- contextelem (syntax), [53](#)
- contextexpr (syntax), [53](#)
- contradiction (method), [68](#)
- corollary (command), [36](#)
- cut-tac (method), [62](#)
- datatype (HOL command), [86](#)
- datatype (ZF command), [94](#)
- declare (command), [45](#)
- def (command), [33](#)
- default (method), [68](#)
- defaultsort (command), [24](#)
- defer (command), [45](#)
- defs (command), [26](#)
- dest (attribute), [72](#)
- dest (Pure attribute), [40](#)
- domain (HOLCF command), [93](#)
- done (command), [45](#)
- drule (method), [59](#)
- drule-tac (method), [62](#)
- elim (attribute), [72](#)
- elim (method), [68](#)
- elim (Pure attribute), [40](#)
- elim-format (attribute), [73](#)
- elim-format (Pure attribute), [60](#)
- end (command), [21](#)
- erule (method), [59](#)
- erule-tac (method), [62](#)
- fail (method), [59](#)
- fast (method), [69](#)
- fastsimp (method), [70](#)
- finally (command), [57](#)
- fix (command), [33](#)
- fold (method), [59](#)
- folded (attribute), [60](#)
- force (method), [70](#)
- from (command), [34](#)
- frule (method), [59](#)
- frule-tac (method), [62](#)
- generate-code (command), [92](#)
- global (command), [28](#)
- goals (antiquotation), [18](#)
- goalspec (syntax), [14](#)
- have (command), [36](#)

- header (command), [21](#)
- hence (command), [36](#)
- hide (command), [28](#)
- hypsubst (method), [67](#)

- ident (syntax), [8](#)
- iff (attribute), [72](#)
- ind-cases (HOL method), [90](#)
- ind-cases (ZF method), [96](#)
- induct (attribute), [78](#)
- induct (method), [75](#)
- induct-tac (HOL method), [90](#)
- induct-tac (ZF method), [96](#)
- inductive (HOL command), [89](#)
- inductive (ZF command), [94](#)
- inductive-cases (HOL command), [90](#)
- inductive-cases (ZF command), [96](#)
- infix (syntax), [13](#)
- insert (method), [59](#)
- inst (syntax), [12](#)
- instance (command), [51](#)
- insts (syntax), [12](#)
- int (syntax), [9](#)
- intro (attribute), [72](#)
- intro (method), [68](#)
- intro (Pure attribute), [40](#)
- intro-classes (method), [51](#)

- judgment (command), [79](#)

- kill (command), [49](#)

- lemma (command), [36](#)
- lemmas (command), [27](#)
- let (command), [43](#)
- local (command), [28](#)
- locale (command), [53](#)
- locale (syntax), [52](#)
- longident (syntax), [8](#)

- method (syntax), [14](#)

- method-setup (command), [29](#)
- Methods
 - , [40](#)
 - arith (HOL), [90](#)
 - assumption, [40](#)
 - atomize, [79](#)
 - auto, [70](#)
 - best, [69](#)
 - bestsimp, [70](#)
 - blast, [69](#)
 - case-tac (HOL), [90](#)
 - case-tac (ZF), [96](#)
 - cases, [75](#)
 - clarify, [69](#)
 - clarsimp, [70](#)
 - contradiction, [68](#)
 - cut-tac, [62](#)
 - default, [68](#)
 - drule, [59](#)
 - drule-tac, [62](#)
 - elim, [68](#)
 - erule, [59](#)
 - erule-tac, [62](#)
 - fail, [59](#)
 - fast, [69](#)
 - fastsimp, [70](#)
 - fold, [59](#)
 - force, [70](#)
 - frule, [59](#)
 - frule-tac, [62](#)
 - hypsubst, [67](#)
 - ind-cases (HOL), [90](#)
 - ind-cases (ZF), [96](#)
 - induct, [75](#)
 - induct-tac (HOL), [90](#)
 - induct-tac (ZF), [96](#)
 - insert, [59](#)
 - intro, [68](#)
 - intro-classes, [51](#)
 - rename-tac, [62](#)
 - rotate-tac, [62](#)

- rule, [40](#), [68](#)
- rule-tac, [62](#)
- rules, [40](#)
- safe, [69](#)
- simp, [64](#)
- simp-all, [64](#)
- slow, [69](#)
- slowsimp, [70](#)
- split, [67](#)
- subgoal-tac, [62](#)
- subst, [67](#)
- succeed, [59](#)
- tactic, [62](#)
- thin-tac, [62](#)
- this, [40](#)
- unfold, [59](#)
- mixfix (syntax), [13](#)
- ML (command), [29](#)
- ML-command (command), [29](#)
- ML-setup (command), [29](#)
- mono (HOL attribute), [89](#)
- moreover (command), [57](#)
- name (syntax), [9](#)
- nameref (syntax), [9](#)
- nat (syntax), [8](#)
- next (command), [44](#)
- no-vars (attribute), [60](#)
- nonterminals (command), [25](#)
- note (command), [34](#)
- nothing (theorem), [35](#)
- obtain (command), [56](#)
- OF (attribute), [40](#)
- of (attribute), [40](#)
- oops (command), [46](#)
- oracle (command), [31](#)
- params (attribute), [73](#)
- parname (syntax), [9](#)
- parse-ast-translation (command), [30](#)
- parse-translation (command), [30](#)
- pr (command), [47](#)
- prefer (command), [45](#)
- prems (theorem), [34](#)
- presume (command), [33](#)
- primrec (HOL command), [87](#)
- primrec (ZF command), [96](#)
- print-ast-translation (command), [30](#)
- print-attributes (command), [48](#)
- print-binds (command), [48](#)
- print-cases (command), [73](#)
- print-claset (command), [72](#)
- print-commands (command), [48](#)
- print-facts (command), [48](#)
- print-induct-rules (command), [78](#)
- print-locale (command), [53](#)
- print-locales (command), [53](#)
- print-methods (command), [48](#)
- print-simpset (command), [66](#)
- print-syntax (command), [48](#)
- print-tcset (command), [94](#)
- print-theorems (command), [48](#)
- print-trans-rules (command), [57](#)
- print-translation (command), [30](#)
- proof
 - default, [40](#)
 - fake, [40](#)
 - terminal, [40](#)
 - trivial, [40](#)
- proof (command), [38](#)
- prop (antiquotation), [18](#)
- prop (command), [47](#)
- prop (syntax), [11](#)
- proppat (syntax), [16](#)
- props (syntax), [17](#)
- pwd (command), [50](#)
- qed (command), [38](#)
- recdef (HOL command), [87](#)

- recdef-cong (HOL attribute), [89](#)
- recdef-simp (HOL attribute), [89](#)
- recdef-tc (HOL command), [87](#)
- recdef-wf (HOL attribute), [89](#)
- record (HOL command), [83](#)
- redo (command), [49](#)
- rename-tac (method), [62](#)
- rep-datatype (HOL command), [86](#)
- rotate-tac (method), [62](#)
- rule (attribute), [72](#)
- rule (method), [40](#), [68](#)
- rule (Pure attribute), [40](#)
- rule-context (case), [38](#)
- rule-format (attribute), [79](#)
- rule-tac (method), [62](#)
- rules (method), [40](#)
- rulify (attribute), [79](#)
- safe (method), [69](#)
- sect (command), [32](#)
- section (command), [23](#)
- setup (command), [29](#)
- show (command), [36](#)
- simp (attribute), [66](#)
- simp (method), [64](#)
- simp-all (method), [64](#)
- simplearity (syntax), [11](#)
- simplified (attribute), [66](#)
- simpmod (syntax), [64](#)
- slow (method), [69](#)
- slowsimp (method), [70](#)
- sorry (command), [38](#)
- sort (syntax), [11](#)
- split (attribute), [66](#)
- split (method), [67](#)
- split-format (HOL attribute), [82](#)
- standard (attribute), [60](#)
- string (syntax), [8](#)
- structmixfix (syntax), [13](#)
- subgoal-tac (method), [62](#)
- subgoals (antiquotation), [18](#)
- subsect (command), [32](#)
- subsection (command), [23](#)
- subst (method), [67](#)
- subsubsect (command), [32](#)
- subsubsection (command), [23](#)
- succeed (method), [59](#)
- swapped (attribute), [73](#)
- sym (attribute), [57](#)
- symident (syntax), [8](#)
- symmetric (attribute), [57](#)
- Syntax
 - args, [15](#)
 - arity, [11](#)
 - atom, [15](#)
 - attributes, [15](#)
 - axmdecl, [15](#)
 - clamod, [69](#)
 - clasimpmod, [70](#)
 - classdecl, [11](#)
 - comment, [10](#)
 - constdecl, [26](#)
 - contextelem, [53](#)
 - contextexpr, [53](#)
 - goalspec, [14](#)
 - ident, [8](#)
 - infix, [13](#)
 - inst, [12](#)
 - insts, [12](#)
 - int, [9](#)
 - locale, [52](#)
 - longident, [8](#)
 - method, [14](#)
 - mixfix, [13](#)
 - name, [9](#)
 - nameref, [9](#)
 - nat, [8](#)
 - parname, [9](#)
 - prop, [11](#)
 - proppat, [16](#)
 - props, [17](#)
 - simplearity, [11](#)

- simpmod, [64](#)
- sort, [11](#)
- string, [8](#)
- structmixfix, [13](#)
- symident, [8](#)
- term, [11](#)
- termpat, [16](#)
- text, [10](#)
- thmdecl, [15](#)
- thmdef, [15](#)
- thmrefs, [15](#)
- type, [11](#)
- typefree, [8](#)
- typespec, [12](#)
- typevar, [8](#)
- var, [8](#)
- vars, [17](#)
- verbatim, [8](#)
- syntax (command), [26](#)
- tactic (method), [62](#)
- tagged (attribute), [60](#)
- TC (attribute), [94](#)
- term (antiquotation), [18](#)
- term (command), [47](#)
- term (syntax), [11](#)
- term abbreviations, [43](#)
- termpat (syntax), [16](#)
- text (antiquotation), [18](#)
- text (command), [23](#)
- text (syntax), [10](#)
- text-row (command), [23](#)
- THEN (attribute), [60](#)
- then (command), [34](#)
- theorem (command), [36](#)
- Theorems
 - _, [36](#), [55](#)
 - calculation, [57](#)
 - nothing, [35](#)
 - prems, [34](#)
 - this, [35](#)
 - theorems (command), [27](#)
 - theory (command), [21](#)
 - thesis (variable), [43](#)
 - thin-tac (method), [62](#)
 - this (method), [40](#)
 - this (theorem), [35](#)
 - this (variable), [43](#)
 - thm (antiquotation), [18](#)
 - thm (command), [47](#)
 - thm-deps (command), [48](#)
 - thmdecl (syntax), [15](#)
 - thmdef (syntax), [15](#)
 - thmrefs (syntax), [15](#)
 - thms-containing (command), [48](#)
 - thus (command), [36](#)
 - token-translation (command), [30](#)
 - trans (attribute), [57](#)
 - translations (command), [26](#)
 - txt (command), [32](#)
 - txt-row (command), [32](#)
 - typ (antiquotation), [18](#)
 - typ (command), [47](#)
 - type (syntax), [11](#)
 - typecheck (attribute), [94](#)
 - typed-print-translation (command), [30](#)
 - typeddecl (command), [25](#)
 - typeddecl (HOL command), [80](#)
 - typedef (HOL command), [80](#)
 - typefree (syntax), [8](#)
 - types (command), [25](#)
 - types-code (command), [92](#)
 - typespec (syntax), [12](#)
 - typevar (syntax), [8](#)
 - ultimately (command), [57](#)
 - undo (command), [49](#)
 - unfold (method), [59](#)
 - unfolded (attribute), [60](#)
 - untagged (attribute), [60](#)
 - update-thy (command), [50](#)

update-thy-only (command), **50**
use (command), **29**
use-thy (command), **50**
use-thy-only (command), **50**
using (command), **34**

var (syntax), **8**
Variables
 ..., **44**
 _, **43**
 case, **77**
 thesis, **43**
 this, **43**
vars (syntax), **17**
verbatim (syntax), **8**

where (attribute), **60**
with (command), **34**