The Isabelle/Isar Reference Manual

Makarius Wenzel

With Contributions by Clemens Ballarin, Stefan Berghofer,
Lucas Dixon, Florian Haftmann, Gerwin Klein,
Alexander Krauss, Tobias Nipkow, David von Oheimb,
Larry Paulson, and Sebastian Skalberg

8 June 2008

Contents

Chapter 1

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 [14, 15].

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 [22, 23]. "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 mechanic 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 [20] is generic and should work reasonably well for any Isabelle object-logic that conforms to the natural deduction view of the Isabelle/Pure framework. Specific language elements introduced by the major object-logics are described in chapter 8 (Isabelle/HOL), chapter 9 (Isabelle/HOLCF), and chapter 10 (Isabelle/ZF). The main language elements are already provided by the Isabelle/Pure framework. Nevertheless, examples given in the generic parts will usually refer to Isabelle/HOL as well.

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 marked by "*" in the subsequent chapters; they are often helpful when developing proof documents, but 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.

1.2 User interfaces

1.2.1 Terminal sessions

The Isabelle `tty` tool provides a very interface for running the Isar interaction loop, with some support for command line editing. For example:

```
isatool tty

  Welcome to Isabelle/HOL (Isabelle2008)

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

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 Emacs 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. 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 [24]):

```
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-mule"
```

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`.

The X-Symbol package

Proof General incorporates a version of the Emacs X-Symbol package [19], 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. The main challenge of getting X-Symbol to work properly is the underlying (semi-automated) X11 font setup.

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, $\Longrightarrow$ would appear as `\<Longrightarrow>` according the default set of Isabelle symbols. Nevertheless, the Isabelle document preparation system (see chapter 5) 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 *theory format* that integrates specifications and proofs, 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.

1.4 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. Especially note that appendix A provides a quick reference card of the most common Isabelle/Isar language elements.

Further issues concerning the Isar concepts are covered in the literature [22, 25, 3, 4]. The author's PhD thesis [23] 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/). The "Archive of Formal Proofs" http://afp.sourceforge.net/ also provides plenty of examples, both in proper Isar proof style and unstructured tactic scripts.

Chapter 2

Outer syntax

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). Subsequently 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 (specifications and 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` without quotes is not.

Printed theory documents usually omit quotes to gain readability (this is a matter of LaTeX macro setup, say via `\isabellestyle`, see also [24]). 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.

More 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). Note that option `-L` does both of this at the same time.

2.1 Lexical matters

The Isabelle/Isar outer syntax provides token classes as presented below; most of these coincide with the inner lexical syntax as presented in [15].

ident = *letter quasiletter**
longident = *ident*(`.`*ident*)$^+$
symident = *sym*$^+$ | `\`<*ident*>
nat = *digit*$^+$
var = *ident* | `?`*ident* | `?`*ident*`.`*nat*
typefree = `'`*ident*
typevar = *typefree* | `?`*typefree* | `?`*typefree*`.`*nat*
string = `"` ... `"`
altstring = `` ` `` ... `` ` ``
verbatim = `{*` ... `*}`

letter = *latin* | `\`<*latin*> | `\`<*latin latin*> | *greek* |
`\`<^isub> | `\`<^isup>
quasiletter = *letter* | *digit* | `_` | `'`
latin = a | ... | z | A | ... | Z
digit = 0 | ... | 9
sym = ! | # | $ | % | & | * | + | - | / |
< | = | > | ? | @ | ^ | _ | | | ~
greek = \<alpha> | \<beta> | \<gamma> | \<delta> |
\<epsilon> | \<zeta> | \<eta> | \<theta> |
\<iota> | \<kappa> | \<mu> | \<nu> |
\<xi> | \<pi> | \<rho> | \<sigma> | \<tau> |
\<upsilon> | \<phi> | \<chi> | \<psi> |
\<omega> | \<Gamma> | \<Delta> | \<Theta> |
\<Lambda> | \<Xi> | \<Pi> | \<Sigma> |
\<Upsilon> | \<Phi> | \<Psi> | \<Omega>

The syntax of *string* admits any characters, including newlines; "`"`" (double-quote) and "`\`" (backslash) need to be escaped by a backslash; arbitrary character codes may be specified as "`\`*ddd*", with three decimal digits.

Alternative strings according to *altstring* are analogous, using single back-quotes instead. The body of *verbatim* may consist of any text not containing "*}"; this allows convenient inclusion of quotes without further escapes. The greek letters do *not* include \<lambda>, which is already used differently in the meta-logic.

Common mathematical symbols such as $\forall$ are represented in Isabelle as \<forall>. There are infinitely many Isabelle symbols like this, although proper presentation is left to front-end tools such as LaTeX or Proof General with the X-Symbol package. A list of standard Isabelle symbols that work well with these tools is given in [24, appendix A].

Source comments take the form (* ... *) and may be nested, although user-interface tools might prevent this. Note that this form indicates 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).

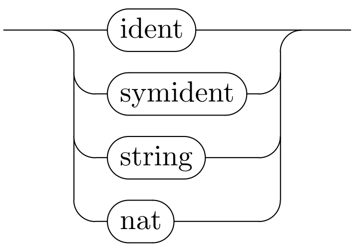
2.2 Common syntax entities

We now introduce several basic syntactic entities, such as names, terms, and theorem specifications, which are factored out of the actual Isar language elements to be described later.

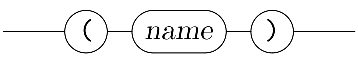
2.2.1 Names

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. quoted "let"). Already existing objects are usually referenced by *nameref*.

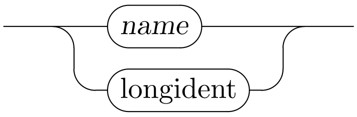
name



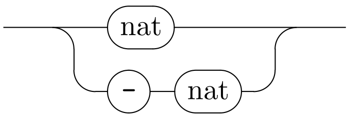
parname



nameref



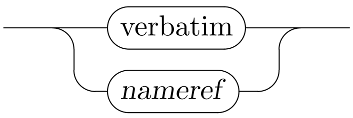
int



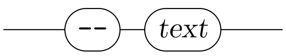
2.2.2 Comments

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



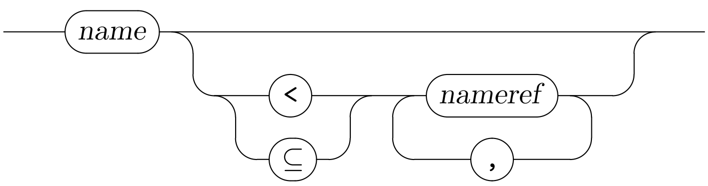
comment



2.2.3 Type classes, sorts and arities

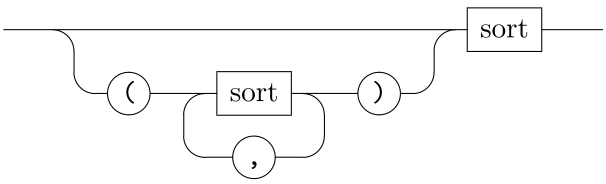
Classes are specified by plain names. Sorts have a very simple inner syntax, which is either a single class name c or a list $\{c_1, \ldots, c_n\}$ referring to the intersection of these classes. The syntax of type arities is given directly at the outer level.

classdecl



sort

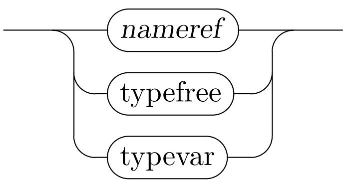
arity



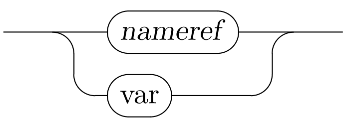
2.2.4 Types and terms

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 quoted `"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 (such as `=` or `+`).

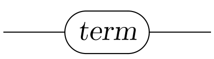
type



term

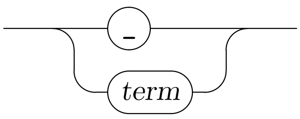


prop

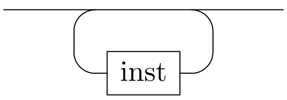


Positional instantiations are indicated by giving a sequence of terms, or the placeholder "_" (underscore), which means to skip a position.

inst

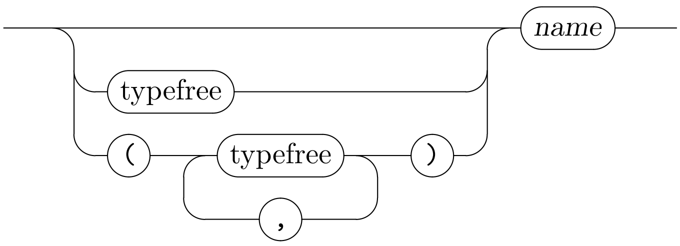


insts



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.

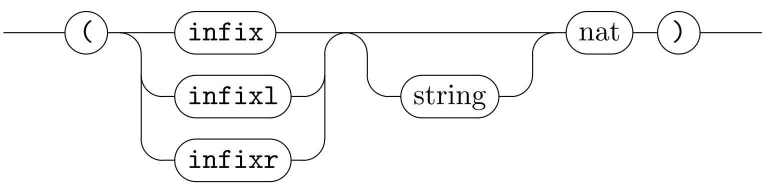
typespec



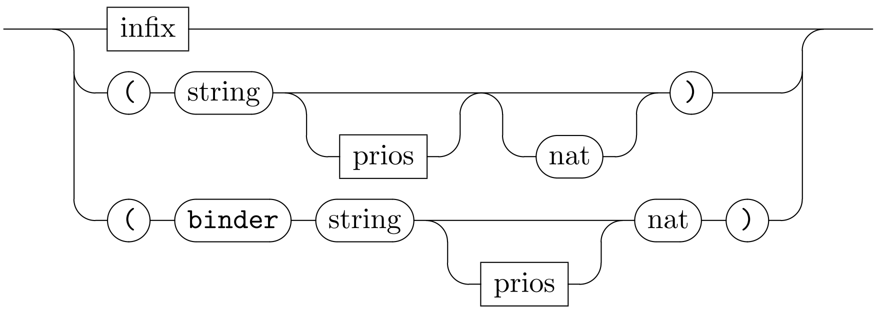
2.2.5 Mixfix annotations

Mixfix annotations specify concrete *inner* syntax of Isabelle types and terms. Some commands such as **types** (see §3.9.2) admit infixes only, while **consts** (see §3.9.3) and **syntax** (see §3.13) support the full range of general mixfixes and binders.

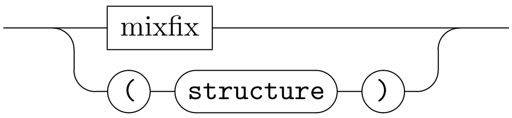
infix



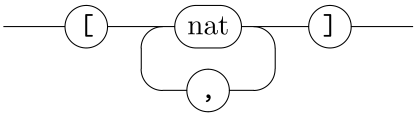
mixfix



structmixfix



prios

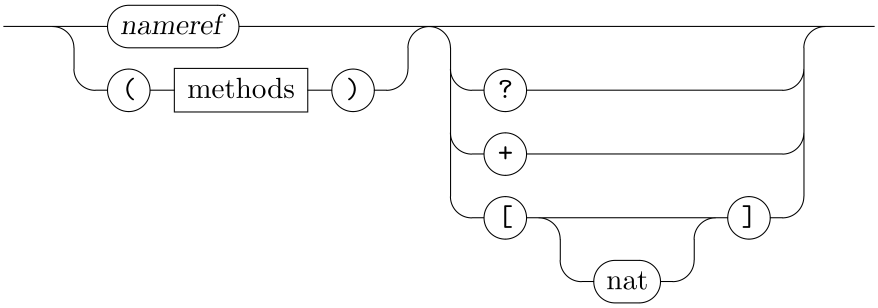


Here the string specifications refer to the actual mixfix template (see also [15]), which may include literal text, spacing, blocks, and arguments (denoted by “_”); the special symbol “`\<index>`” (printed as “ı”) represents an index argument that specifies an implicit structure reference (see also §3.5). 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 “`++`ı” for an infix of an implicit structure.

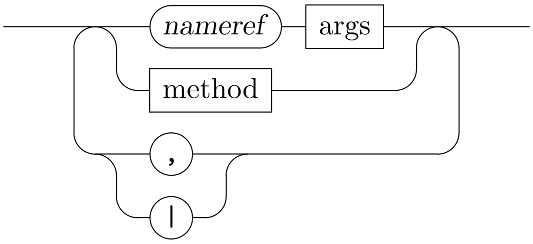
2.2.6 Proof methods

Proof methods are either basic ones, or expressions composed of methods via “`,`” (sequential composition), “`|`” (alternative choices), “`?`” (try), “`+`” (repeat at least once), “`[`n`]`” (restriction to first n sub-goals, with default $n = 1$). 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



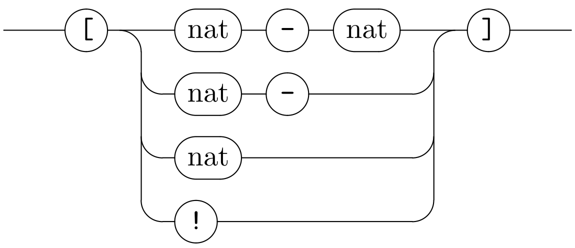
methods



Proper Isar proof methods do *not* admit arbitrary goal addressing, but refer either to the first sub-goal or all sub-goals uniformly. The goal restriction operator "[n]" evaluates a method expression within a sandbox consisting of the first n sub-goals (which need to exist). For example, the method "*simp_all*[3]" simplifies the first three sub-goals, while "(*rule foo*, *simp_all*)[]" simplifies all new goals that emerge from applying rule *foo* to the originally first one.

Improper methods, notably tactic emulations, offer a separate low-level goal addressing scheme as explicit argument to the individual tactic being involved. Here "[!]" refers to all goals, and "[$n-$]" to all goals starting from n.

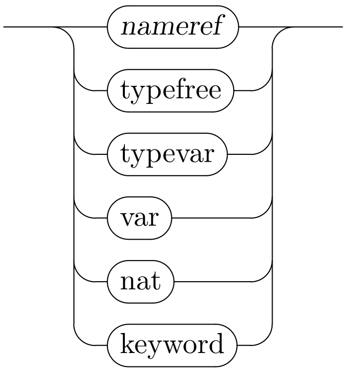
goalspec



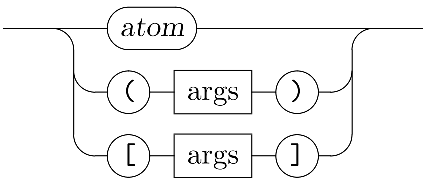
2.2.7 Attributes and theorems

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.

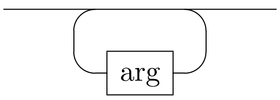
atom



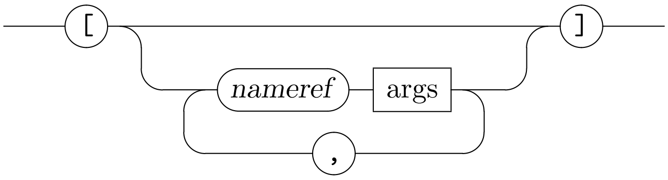
arg



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.

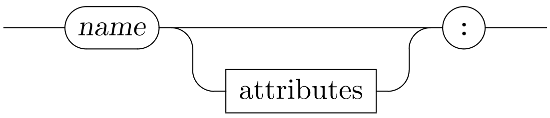
There are three forms of theorem references:

1. named facts a,
2. selections from named facts $a(i)$ or $a(j - k)$,
3. literal fact propositions using *altstring* syntax 'φ' (see also method *fact* in §4.5).

Any kind of theorem specification 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, but any given attributes are applied nonetheless.

An extra pair of brackets around attributes (like "[[*simproc a*]]") abbreviates a theorem reference involving an internal dummy fact, which will be ignored later on. So only the effect of the attribute on the background context will persist. This form of in-place declarations is particularly useful with commands like **declare** and **using**.

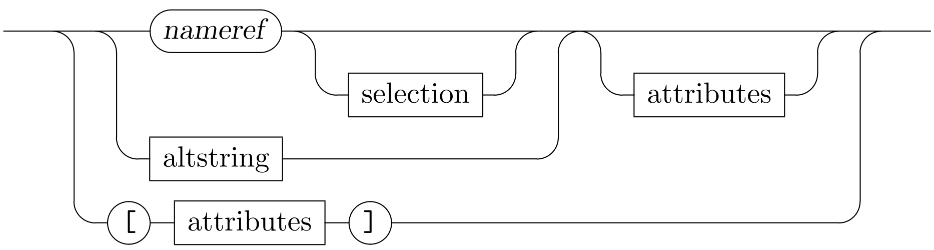
axmdecl



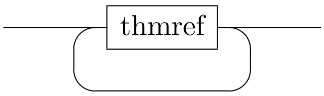
thmdecl

thmdef

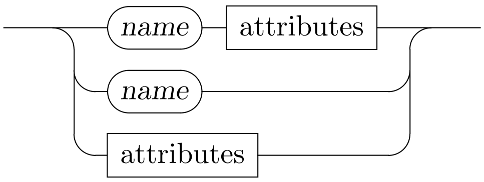
thmref



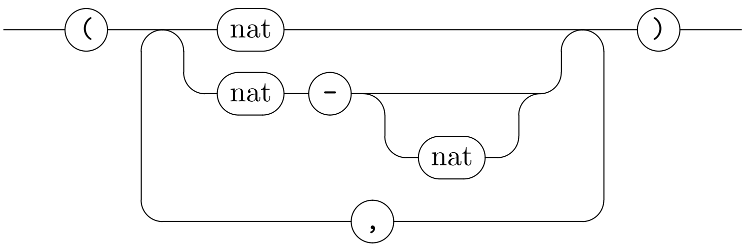
thmrefs



thmbind



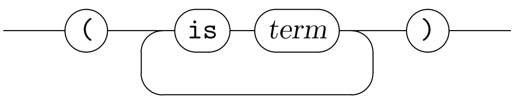
selection



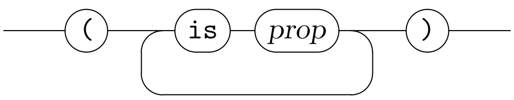
2.2.8 Term patterns and declarations

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** $p_1 \ldots p_n$)". This works both for *term* and *prop*.

termpat

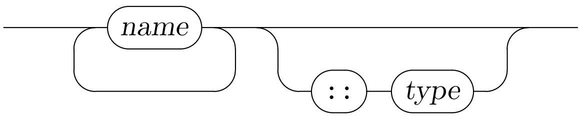


proppat

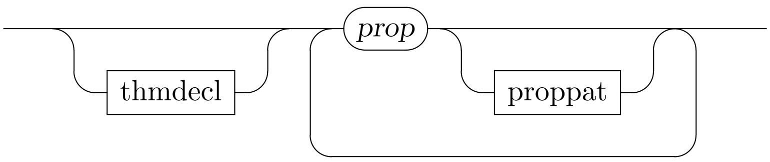


Declarations of local variables $x :: \tau$ and logical propositions $a : \varphi$ represent different views on the same principle of introducing a local scope. In practice, one may usually omit the typing of *vars* (due to type-inference), and the naming of propositions (due to implicit references of 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 "$x_1 \ldots x_n :: \tau$" the typing refers to all variables, while in a: $\varphi_1 \ldots \varphi_n$ 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 §4.1.

Chapter 3

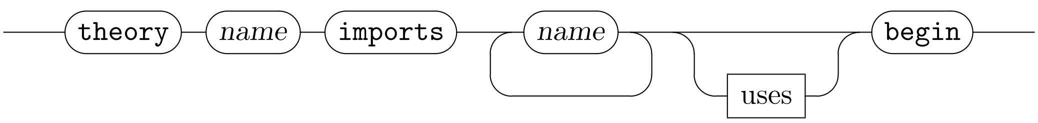
Theory specifications

3.1 Defining theories

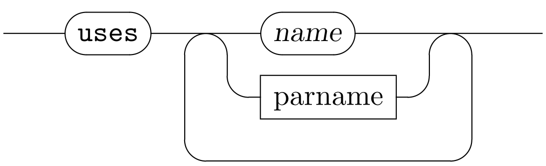
theory	:	*toplevel* → *theory*
end	:	*theory* → *toplevel*

Isabelle/Isar theories are defined via theory file, which contain both specifications and proofs; occasionally definitional mechanisms also require some explicit proof. The theory body may be sub-structered by means of *local theory* target mechanisms, notably **locale** and **class**.

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 the **theory** keyword, 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. §5.1). The **end** command concludes a theory development; it has to be the very last command of any theory file loaded in batch-mode.



uses



theory A **imports** $B_1 \ldots B_n$ **begin** starts a new theory A based on the merge of existing theories $B_1 \ldots B_n$.

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.

The optional **uses** specification declares additional dependencies on extra files (usually ML sources). Files will be loaded immediately (as ML), 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 §3.8).

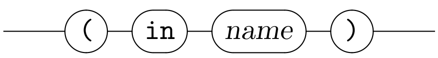
end concludes the current theory definition.

3.2 Local theory targets

A local theory target is a context managed separately within the enclosing theory. Contexts may introduce parameters (fixed variables) and assumptions (hypotheses). Definitions and theorems depending on the context may be added incrementally later on. Named contexts refer to locales (cf. §3.5) or type classes (cf. §3.6); the name "−" signifies the global theory context.

context	:	*theory* $\rightarrow$ *local-theory*
end	:	*local-theory* $\rightarrow$ *theory*

target



context c **begin** recommences an existing locale or class context c. Note that locale and class definitions allow to include the **begin** keyword as well, in order to continue the local theory immediately after the initial specification.

end concludes the current local theory and continues the enclosing global theory. Note that a global **end** has a different meaning: it concludes the theory itself (§3.1).

(**in** c) given after any local theory command specifies an immediate target, e.g. "**definition** (**in** c) ..." or "**theorem** (**in** c) ...". This works both in a local or global theory context; the current target context will be suspended for this command only. Note that "(**in** $-$)" will always produce a global result independently of the current target context.

The exact meaning of results produced within a local theory context depends on the underlying target infrastructure (locale, type class etc.). The general idea is as follows, considering a context named c with parameter x and assumption $A[x]$.

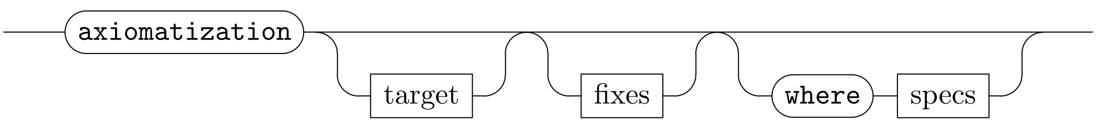
Definitions are exported by introducing a global version with additional arguments; a syntactic abbreviation links the long form with the abstract version of the target context. For example, $a \equiv t[x]$ becomes $c.a\ ?x \equiv t[?x]$ at the theory level (for arbitrary $?x$), together with a local abbreviation $c \equiv c.a\ x$ in the target context (for the fixed parameter x).

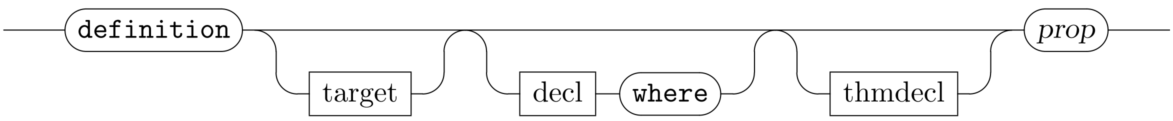
Theorems are exported by discharging the assumptions and generalizing the parameters of the context. For example, a: $B[x]$ becomes $c.a$: $A[?x] \Longrightarrow B[?x]$, again for arbitrary $?x$.

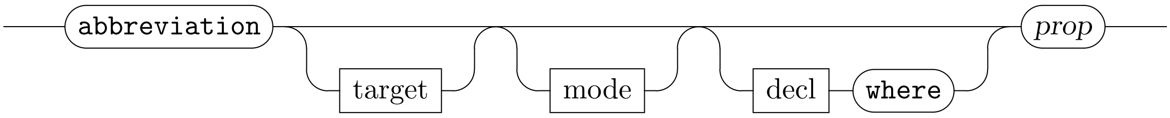
3.3 Basic specification elements

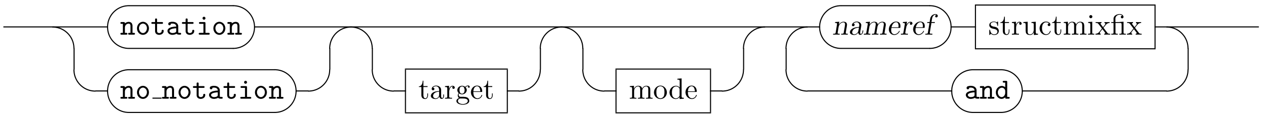
axiomatization	:	*local-theory* → *local-theory*	(*axiomatic*!)
definition	:	*local-theory* → *local-theory*	
defn	:	*attribute*	
abbreviation	:	*local-theory* → *local-theory*	
print_abbrevs*	:	*theory* | *proof* → *theory* | *proof*	
notation	:	*local-theory* → *local-theory*	
no_notation	:	*local-theory* → *local-theory*	

These specification mechanisms provide a slightly more abstract view than the underlying primitives of **consts**, **defs** (see §3.9.3), and **axioms** (see §3.10). In particular, type-inference is commonly available, and result names need not be given.

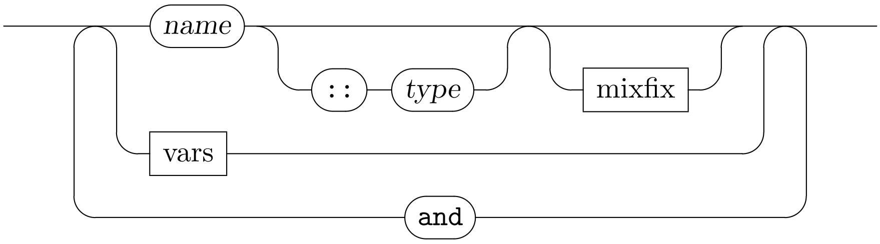




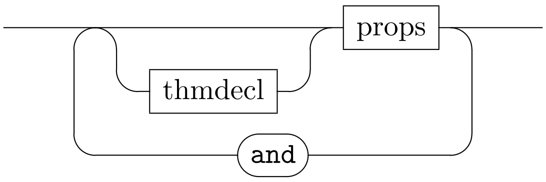




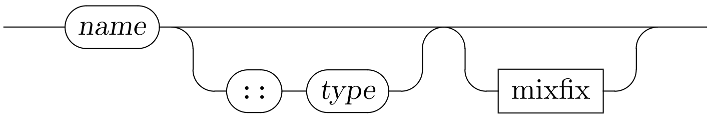
fixes



specs



decl



axiomatization $c_1 \dots c_m$ **where** $\varphi_1 \dots \varphi_n$ introduces several constants simultaneously and states axiomatic properties for these. The constants are marked as being specified once and for all, which prevents additional specifications being issued later on.

Note that axiomatic specifications are only appropriate when declaring a new logical system. Normal applications should only use definitional mechanisms!

definition c **where** *eq* produces an internal definition $c \equiv t$ according to the specification given as *eq*, which is then turned into a proven fact. The given proposition may deviate from internal meta-level equality according to the rewrite rules declared as *defn* by the object-logic.

This usually covers object-level equality $x = y$ and equivalence $A \leftrightarrow B$. End-users normally need not change the *defn* setup.

Definitions may be presented with explicit arguments on the LHS, as well as additional conditions, e.g. $f\ x\ y = t$ instead of $f \equiv \lambda x\ y.\ t$ and $y \neq 0 \Longrightarrow g\ x\ y = u$ instead of an unrestricted $g \equiv \lambda x\ y.\ u$.

abbreviation c **where** *eq* introduces a syntactic constant which is associated with a certain term according to the meta-level equality *eq*.

Abbreviations participate in the usual type-inference process, but are expanded before the logic ever sees them. Pretty printing of terms involves higher-order rewriting with rules stemming from reverted abbreviations. This needs some care to avoid overlapping or looping syntactic replacements!

The optional *mode* specification restricts output to a particular print mode; using "*input*" here achieves the effect of one-way abbreviations. The mode may also include an "**output**" qualifier that affects the concrete syntax declared for abbreviations, cf. **syntax** in §3.13.

print_abbrevs prints all constant abbreviations of the current context.

notation c (mx) associates mixfix syntax with an existing constant or fixed variable. This is a robust interface to the underlying **syntax** primitive (§3.13). Type declaration and internal syntactic representation of the given entity is retrieved from the context.

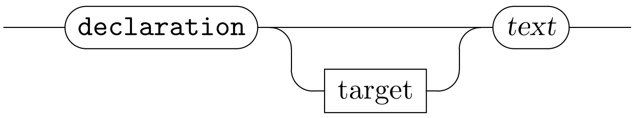
no_notation is similar to **notation**, but removes the specified syntax annotation from the present context.

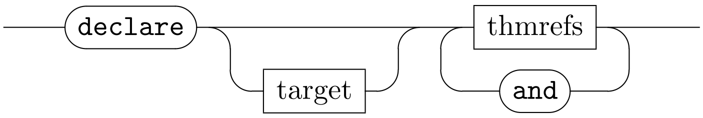
All of these specifications support local theory targets (cf. §3.2).

3.4 Generic declarations

Arbitrary operations on the background context may be wrapped-up as generic declaration elements. Since the underlying concept of local theories may be subject to later re-interpretation, there is an additional dependency on a morphism that tells the difference of the original declaration context wrt. the application context encountered later on. A fact declaration is an important special case: it consists of a theorem which is applied to the context by means of an attribute.

declaration	:	*local-theory* → *local-theory*
declare	:	*local-theory* → *local-theory*





declaration *d* adds the declaration function *d* of ML type `declaration`, to the current local theory under construction. In later application contexts, the function is transformed according to the morphisms being involved in the interpretation hierarchy.

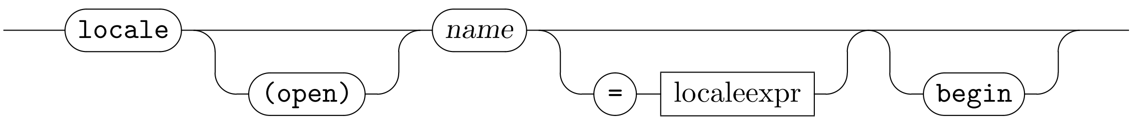
declare *thms* declares theorems to the current local theory context. No theorem binding is involved here, unlike **theorems** or **lemmas** (cf. §3.10), so **declare** only has the effect of applying attributes as included in the theorem specification.

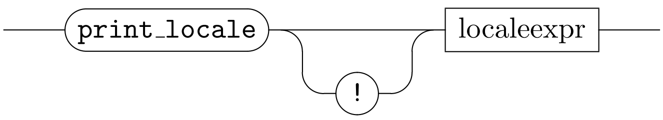
3.5 Locales

Locales are named local contexts, consisting of a list of declaration elements that are modeled after the Isar proof context commands (cf. §4.1).

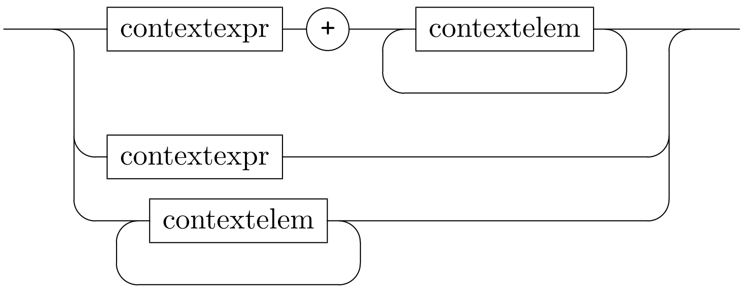
3.5.1 Locale specifications

locale	:	*theory* → *local-theory*
print_locale*	:	*theory* | *proof* → *theory* | *proof*
print_locales*	:	*theory* | *proof* → *theory* | *proof*
intro_locales	:	*method*
unfold_locales	:	*method*

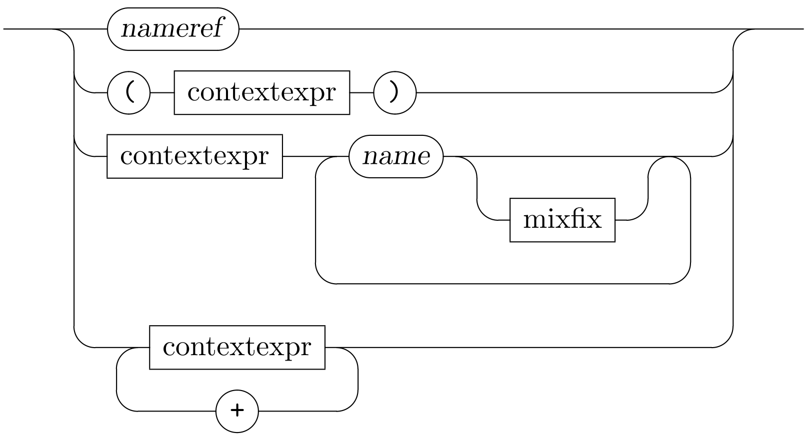




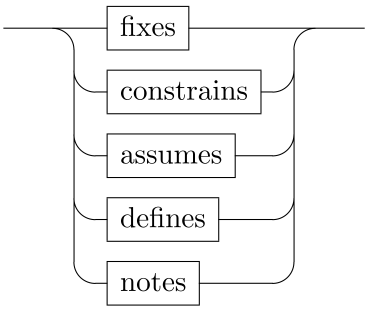
localeexpr



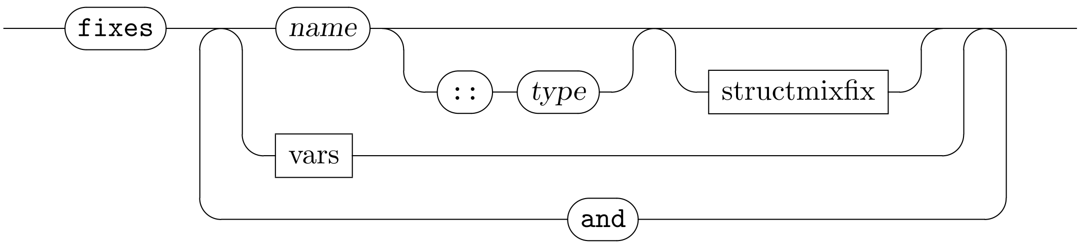
contextexpr



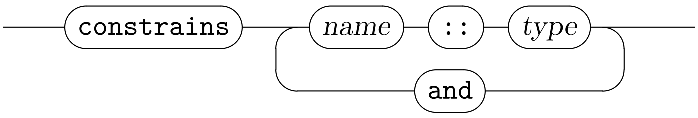
contextelem



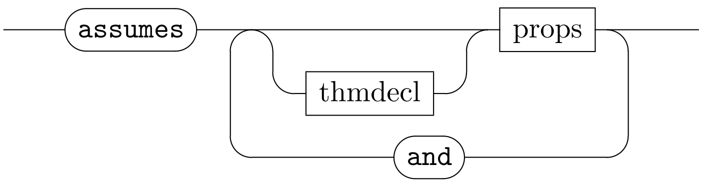
fixes



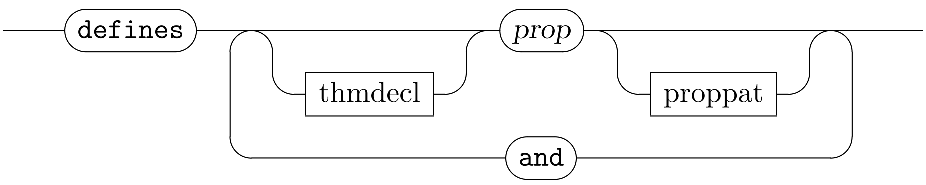
constrains



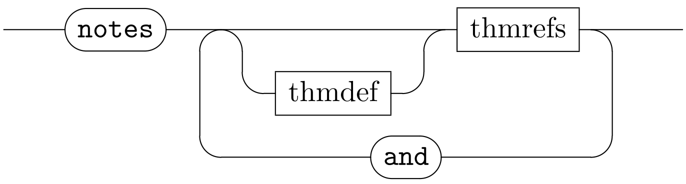
assumes



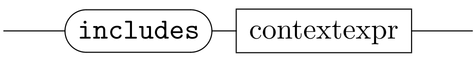
defines



notes



includes



locale *loc* = *import* + *body* defines a 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: $c\ x_1\ \ldots\ x_n$ means that (a prefix of) the fixed parameters of context c are named $x_1, \ldots, x_n$; a "_" (underscore) means to skip that position. Renaming by default deletes concrete syntax, but new syntax may by specified with a mixfix annotation. An exeption of this rule is the special syntax declared with "(**structure**)" (see below), which is neither deleted nor can it be changed. 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.

constrains $x :: \tau$ introduces a type constraint τ on the local parameter x.

assumes a: $\varphi_1 \ldots \varphi_n$ introduces local premises, similar to **assume** within a proof (cf. §4.1).

defines a: $x \equiv t$ defines a previously declared parameter. This is similar to **def** within a proof (cf. §4.1), 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** $f\, x_1 \ldots x_n \equiv t$".

notes $a = b_1 \ldots b_n$ 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. Only available in the long goal format of §4.3.

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** $p_1 \ldots p_n$)" patterns given in the syntax of **assumes** and **defines** above are illegal in locale definitions. In the long goal format of §4.3, term bindings may be included as expected, though.

By default, locale specifications are "closed up" by turning the given text into a predicate definition *loc_axioms* and deriving the original assumptions as local lemmas (modulo local definitions). The predicate statement covers only the newly specified assumptions, omitting the content of included locale expressions. The full cumulative view is only provided on export, involving another predicate *loc* that refers to the complete specification text.

In any case, the predicate arguments are those locale parameters that actually occur in the respective piece of text. Also note that these predicates operate at the meta-level in theory, but the locale packages attempts to internalize statements according to the object-logic setup (e.g. replacing $\bigwedge$ by $\forall$, and $\Longrightarrow$ by $\longrightarrow$ in HOL; see also §7.5). Separate introduction rules *loc_axioms.intro* and *loc.intro* are provided as well.

The (*open*) option of a locale specification prevents both the current *loc_axioms* and cumulative *loc* predicate constructions. Predicates are also omitted for empty specification texts.

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. The command omits **notes** elements by default. Use **print_locale**! to get them included.

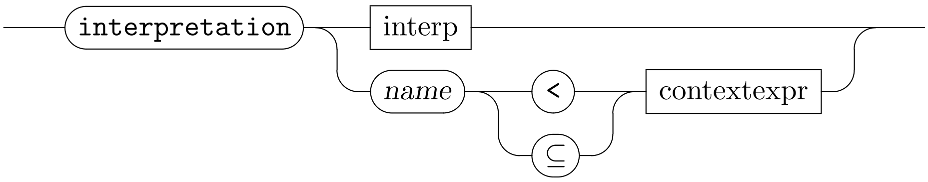
print_locales prints the names of all locales of the current theory.

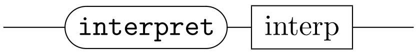
intro_locales and *unfold_locales* repeatedly expand all introduction rules of locale predicates of the theory. While *intro_locales* only applies the *loc.intro* introduction rules and therefore does not decend to assumptions, *unfold_locales* is more aggressive and applies *loc_axioms.intro* as well. Both methods are aware of locale specifications entailed by the context, both from target and **includes** statements, and from interpretations (see below). New goals that are entailed by the current context are discharged automatically.

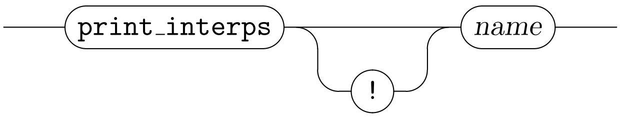
3.5.2 Interpretation of locales

Locale expressions (more precisely, *context expressions*) may be instantiated, and the instantiated facts added to the current context. This requires a proof of the instantiated specification and is called *locale interpretation*. Interpretation is possible in theories and locales (command **interpretation**) and also within a proof body (command **interpret**).

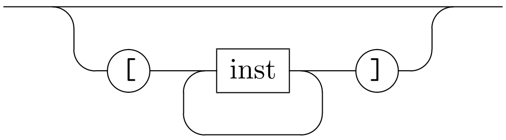
interpretation	:	*theory* → *proof*(*prove*)
interpret	:	*proof*(*state*) | *proof*(*chain*) → *proof*(*prove*)
print_interps*	:	*theory* | *proof* → *theory* | *proof*



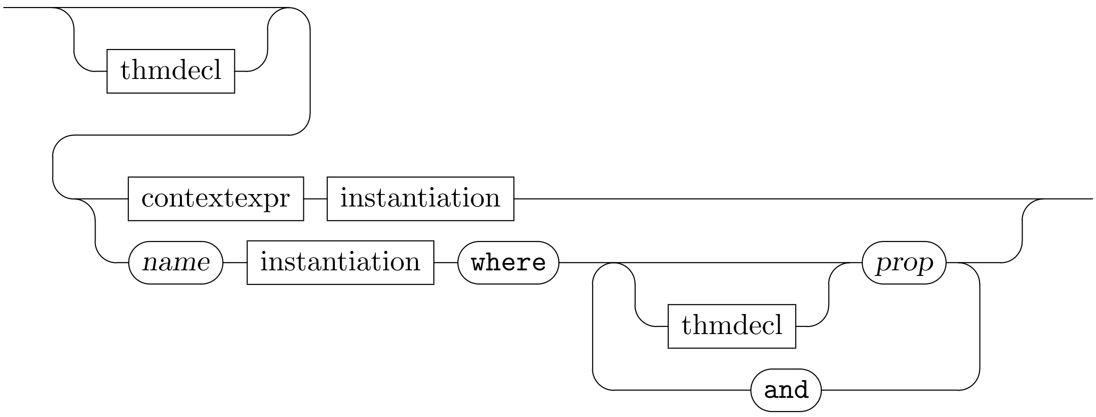




instantiation



interp



interpretation *expr insts* **where** *eqns* The first form of **interpretation** interprets *expr* in the theory. The instantiation is given as a list of terms *insts* and is positional. All parameters must receive an instantiation term — with the exception of defined parameters. These are, if omitted, derived from the defining equation and other instantiations. Use "_" to omit an instantiation term.

The command generates proof obligations for the instantiated specifications (assumes and defines elements). Once these are discharged by the user, instantiated facts are added to the theory in a post-processing phase.

Additional equations, which are unfolded in facts during post-processing, may be given after the keyword **where**. This is useful for interpreting concepts introduced through definition specification elements. The equations must be proved. Note that if equations are present, the context expression is restricted to a locale name.

The command is aware of interpretations already active in the theory. No proof obligations are generated for those, neither is post-processing applied to their facts. This avoids duplication of interpreted facts, in particular. Note that, in the case of a locale with import, parts of the interpretation may already be active. The command will only generate proof obligations and process facts for new parts.

The context expression may be preceded by a name and/or attributes. These take effect in the post-processing of facts. The name is used to prefix fact names, for example to avoid accidental hiding of other facts. Attributes are applied after attributes of the interpreted facts.

Adding facts to locales has the effect of adding interpreted facts to the theory for all active interpretations also. That is, interpretations dynamically participate in any facts added to locales.

interpretation *name* $\subseteq$ *expr* This form of the command interprets *expr* in the locale *name*. It requires a proof that the specification of *name* implies the specification of *expr*. As in the localized version of the theorem command, the proof is in the context of *name*. After the proof obligation has been dischared, the facts of *expr* become part of locale *name* as *derived* context elements and are available when the context *name* is subsequently entered. Note that, like import, this is dynamic: facts added to a locale part of *expr* after interpretation become also available in *name*. Like facts of renamed context elements, facts obtained by interpretation may be accessed by prefixing with the parameter renaming (where the parameters are separated by "_").

Unlike interpretation in theories, instantiation is confined to the renaming of parameters, which may be specified as part of the context expression *expr*. Using defined parameters in *name* one may achieve an effect similar to instantiation, though.

Only specification fragments of *expr* that are not already part of *name* (be it imported, derived or a derived fragment of the import) are considered by interpretation. This enables circular interpretations.

If interpretations of *name* exist in the current theory, the command adds interpretations for *expr* as well, with the same prefix and attributes, although only for fragments of *expr* that are not interpreted in the theory already.

interpret *expr insts* **where** *eqns* interprets *expr* in the proof context and is otherwise similar to interpretation in theories.

print_interps *loc* prints the interpretations of a particular locale *loc* that are active in the current context, either theory or proof context. The exclamation point argument triggers printing of *witness* theorems justifying interpretations. These are normally omitted from the output.

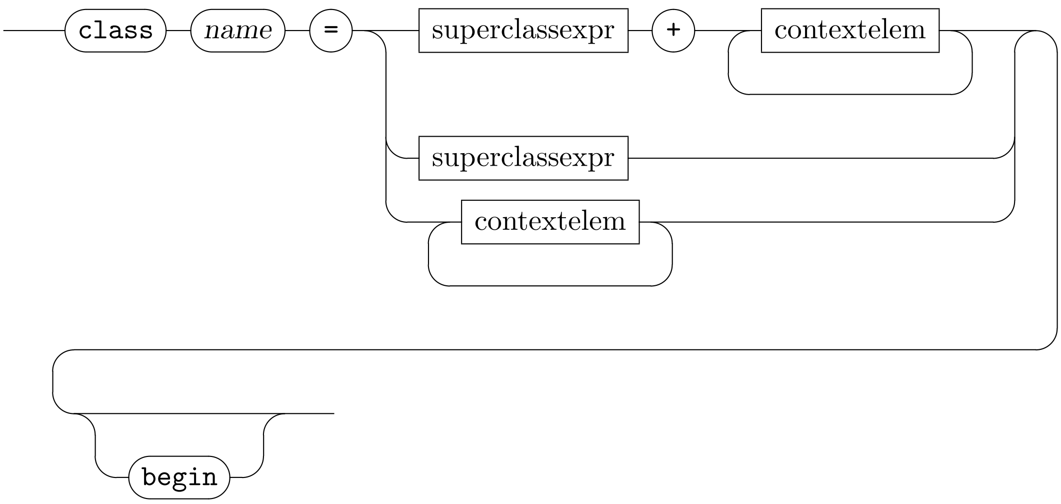
! Since attributes are applied to interpreted theorems, interpretation may modify the context of common proof tools, e.g. the Simplifier or Classical Reasoner. Since the behavior of such automated reasoning tools is *not* stable under interpretation morphisms, manual declarations might have to be issued.

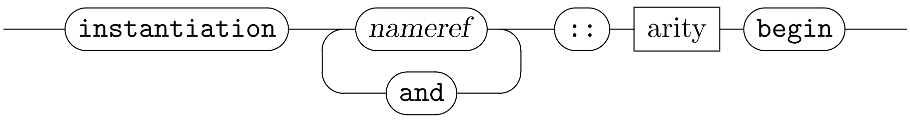
! An interpretation in a theory may subsume previous interpretations. This happens if the same specification fragment is interpreted twice and the instantiation of the second interpretation is more general than the interpretation of the first. A warning is issued, since it is likely that these could have been generalized in the first place. The locale package does not attempt to remove subsumed interpretations.

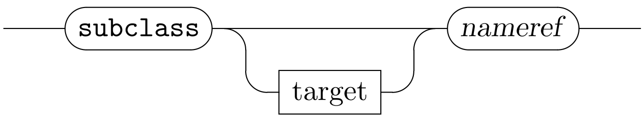
3.6 Classes

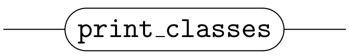
A class is a particular locale with *exactly one* type variable α. Beyond the underlying locale, a corresponding type class is established which is interpreted logically as axiomatic type class [21] whose logical content are the assumptions of the locale. Thus, classes provide the full generality of locales combined with the commodity of type classes (notably type-inference). See [6] for a short tutorial.

class	:	*theory* → *local-theory*
instantiation	:	*theory* → *local-theory*
instance	:	*local-theory* → *local-theory*
subclass	:	*local-theory* → *local-theory*
print_classes*	:	*theory* | *proof* → *theory* | *proof*
intro_classes	:	*method*

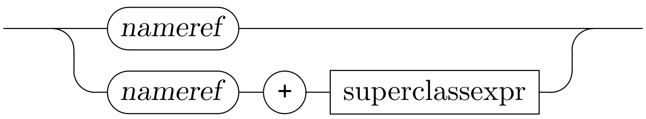








superclassexpr



class c = *superclasses* + *body* defines a new class c, inheriting from *superclasses*. This introduces a locale c with import of all locales *superclasses*.

Any **fixes** in *body* are lifted to the global theory level (*class operations* $f_1, \ldots, f_n$ of class c), mapping the local type parameter α to a schematic type variable $?\alpha :: c$.

Likewise, **assumes** in *body* are also lifted, mapping each local parameter $f :: \tau[\alpha]$ to its corresponding global constant $f :: \tau[?\alpha :: c]$. The corresponding introduction rule is provided as *c_class_axioms.intro*. This rule should be rarely needed directly — the *intro_classes* method takes care of the details of class membership proofs.

instantiation t :: $(s_1, \ldots, s_n)$ s **begin** opens a theory target (cf. §3.2) which allows to specify class operations $f_1, \ldots, f_n$ corresponding to sort s at the particular type instance $(\alpha_1 :: s_1, \ldots, \alpha_n :: s_n)$ t. A plain **instance** command in the target body poses a goal stating these type arities. The target is concluded by an **end** command.

Note that a list of simultaneous type constructors may be given; this corresponds nicely to mutual recursive type definitions, e.g. in Isabelle/HOL.

instance in an instantiation target body sets up a goal stating the type arities claimed at the opening **instantiation**. The proof would usually proceed by *intro_classes*, and then establish the characteristic theorems of the type classes involved. After finishing the proof, the background theory will be augmented by the proven type arities.

subclass c in a class context for class d sets up a goal stating that class c is logically contained in class d. After finishing the proof, class d is proven to be subclass c and the locale c is interpreted into d simultaneously.

print_classes prints all classes in the current theory.

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 (e.g. of **proof**). In particular, instantiation of trivial (syntactic) classes may be performed by a single "**..**" proof step.

3.6.1 The class target

A named context may refer to a locale (cf. §3.2). If this locale is also a class c, apart from the common locale target behaviour the following happens.

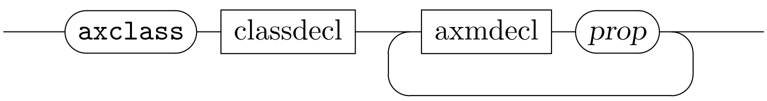
- Local constant declarations $g[\alpha]$ referring to the local type parameter α and local parameters $f[\alpha]$ are accompanied by theory-level constants $g[?\alpha :: c]$ referring to theory-level class operations $f[?\alpha :: c]$.
- Local theorem bindings are lifted as are assumptions.

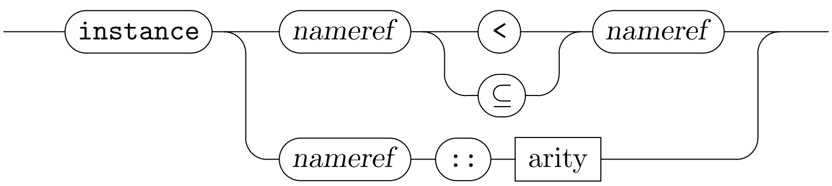
- Local syntax refers to local operations $g[\alpha]$ and global operations $g[?\alpha :: c]$ uniformly. Type inference resolves ambiguities. In rare cases, manual type annotations are needed.

3.6.2 Old-style axiomatic type classes

axclass : *theory* → *theory*
instance : *theory* → *proof*(*prove*)

Axiomatic type classes are Isabelle/Pure's primitive *definitional* interface to type classes. For practical applications, you should consider using classes (cf. §3.9.1) which provide high level interface.





axclass $c \subseteq c_1, \ldots, c_n$ *axms* defines an axiomatic type class as the intersection of existing classes, with additional axioms holding. Class axioms may not contain more than one type variable. The class axioms (with implicit sort constraints added) are bound to the given names. Furthermore a class introduction rule is generated (being bound as *c_class.intro*); this rule is employed by method *intro_classes* to support instantiation proofs of this class.

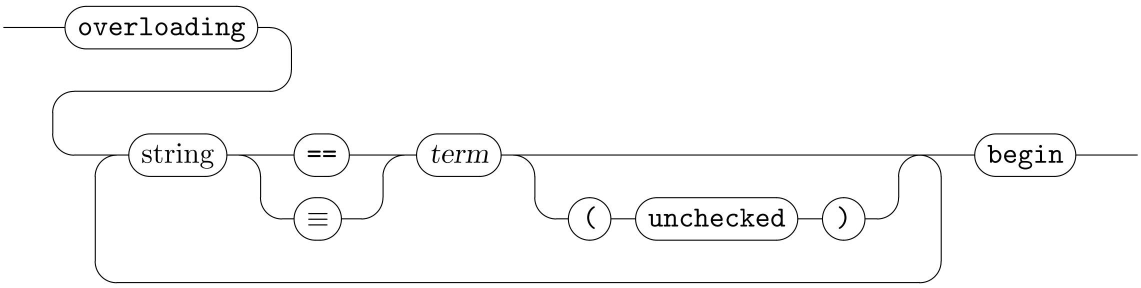
The "class axioms" are stored as theorems according to the given name specifications, adding *c_class* as name space prefix; the same facts are also stored collectively as *c_class.axioms*.

instance $c_1 \subseteq c_2$ and **instance** $t :: (s_1, \ldots, s_n)\ s$ 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.

3.7 Unrestricted overloading

Isabelle/Pure's definitional schemes support certain forms of overloading (see §3.9.3). At most occassions overloading will be used in a Haskell-like fashion together with type classes by means of **instantiation** (see §3.6). Sometimes low-level overloading is desirable. The **overloading** target provides a convenient view for end-users.

overloading : *theory* → *local-theory*

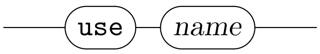


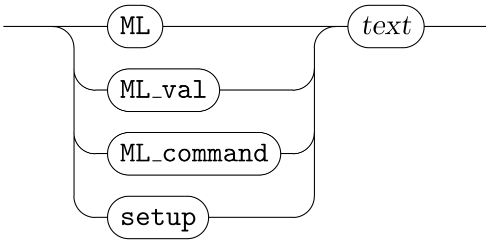
overloading $x_1 \equiv c_1 :: \tau_1$ **and** $\ldots$ $x_n \equiv c_n :: \tau_n$ **begin** opens a theory target (cf. §3.2) which allows to specify constants with overloaded definitions. These are identified by an explicitly given mapping from variable names x_i to constants c_i at particular type instances. The definitions themselves are established using common specification tools, using the names x_i as reference to the corresponding constants. The target is concluded by **end**.

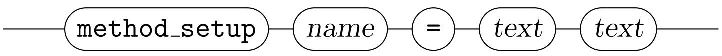
A (*unchecked*) option disables global dependency checks for the corresponding definition, which is occasionally useful for exotic overloading. It is at the discretion of the user to avoid malformed theory specifications!

3.8 Incorporating ML code

use	:	*theory* | *local-theory* → *theory* | *local-theory*
ML	:	*theory* | *local-theory* → *theory* | *local-theory*
ML_val	:	$\cdot \to \cdot$
ML_command	:	$\cdot \to \cdot$
setup	:	*theory* → *theory*
method_setup	:	*theory* → *theory*







use *file* reads and executes ML commands from *file*. The current theory context is passed down to the ML toplevel and may be modified, using `"Context.>>"` or derived ML commands. The file name is checked with the **uses** dependency declaration given in the theory header (see also §3.1).

ML *text* is similar to **use**, but executes ML commands directly from the given *text*.

ML_val and **ML_command** are diagnostic versions of **ML**, which means that the context may not be updated. **ML_val** echos the bindings produced at the ML toplevel, but **ML_command** is silent.

setup *text* changes the current theory context by applying *text*, which refers to an ML expression of type `"theory -> theory"`. This enables to initialize any object-logic specific tools and packages written in ML, for example.

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.

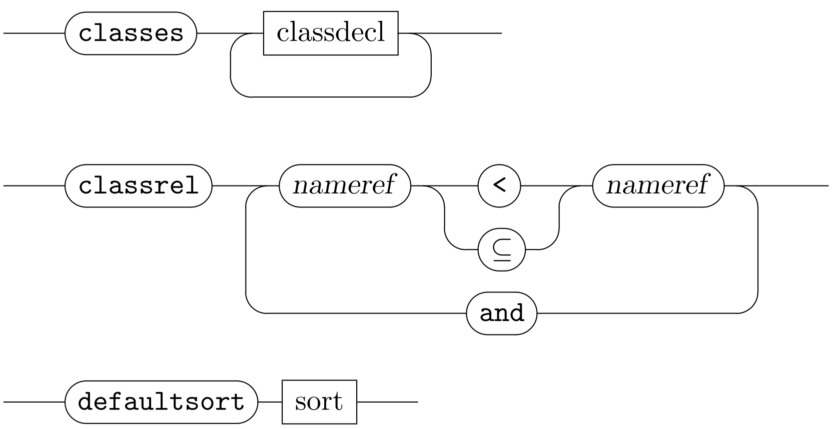
```
Method.no_args (Method.METHOD (fn facts => foobar_tac))
Method.thms_args (fn thms => Method.METHOD (fn facts => foobar_tac))
Method.ctxt_args (fn ctxt => Method.METHOD (fn facts => foobar_tac))
Method.thms_ctxt_args (fn thms => fn ctxt =>
   Method.METHOD (fn facts => foobar_tac))
```

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 `Drule.multi_resolves`), while automatic ones just insert the facts using `Method.insert_tac` before applying the main tactic.

3.9 Primitive specification elements

3.9.1 Type classes and sorts

classes	:	*theory* → *theory*	
classrel	:	*theory* → *theory*	(*axiomatic*!)
defaultsort	:	*theory* → *theory*	
class_deps*	:	*theory* | *proof* → *theory* | *proof*	



classes $c \subseteq c_1, \ldots, c_n$ declares class c to be a subclass of existing classes $c_1, \ldots, c_n$. Cyclic class structures are not permitted.

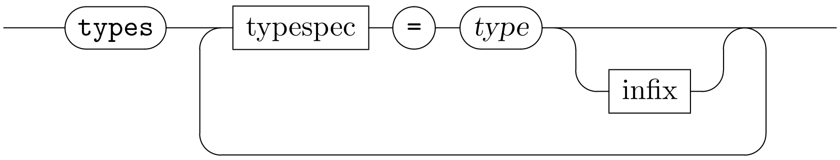
classrel $c_1 \subseteq c_2$ states subclass relations between existing classes c_1 and c_2. This is done axiomatically! The **instance** command (see §3.6.2) provides a way to introduce proven class relations.

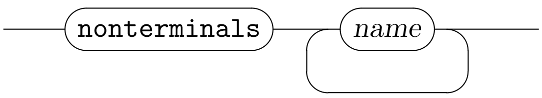
defaultsort s makes sort s the new default sort for any type variables given without sort constraints. Usually, the default sort would be only changed when defining a new object-logic.

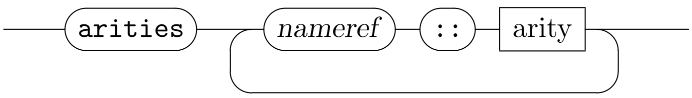
class_deps visualizes the subclass relation, using Isabelle's graph browser tool (see also [24]).

3.9.2 Types and type abbreviations

types	:	*theory* → *theory*
typedecl	:	*theory* → *theory*
nonterminals	:	*theory* → *theory*
arities	:	*theory* → *theory* (*axiomatic*!)







types $(\alpha_1, \ldots, \alpha_n)\ t = \tau$ introduces *type synonym* $(\alpha_1, \ldots, \alpha_n)\ t$ 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 $(\alpha_1, \ldots, \alpha_n)\ t$ declares a new type constructor t, intended as an actual logical type (of the object-logic, if available).

nonterminals c declares type constructors c (without arguments) to act as purely syntactic types, i.e. nonterminal symbols of Isabelle's inner syntax of terms or types.

arities $t :: (s_1, \ldots, s_n)\ s$ augments Isabelle's order-sorted signature of types by new type constructor arities. This is done axiomatically! The **instance** command (see §3.6.2) provides a way to introduce proven type arities.

3.9.3 Constants and definitions

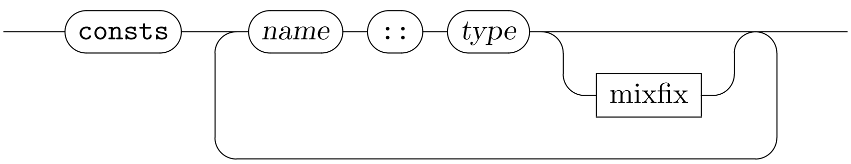
Definitions essentially express abbreviations within the logic. The simplest form of a definition is $c :: \sigma \equiv t$, where c is a newly declared constant. Isabelle also allows derived forms where the arguments of c appear on the left, abbreviating a prefix of λ-abstractions, e.g. $c \equiv \lambda x\ y.\ t$ may be written more conveniently as $c\ x\ y \equiv t$. Moreover, definitions may be weakened by adding arbitrary pre-conditions: $A \Longrightarrow c\ x\ y \equiv t$.

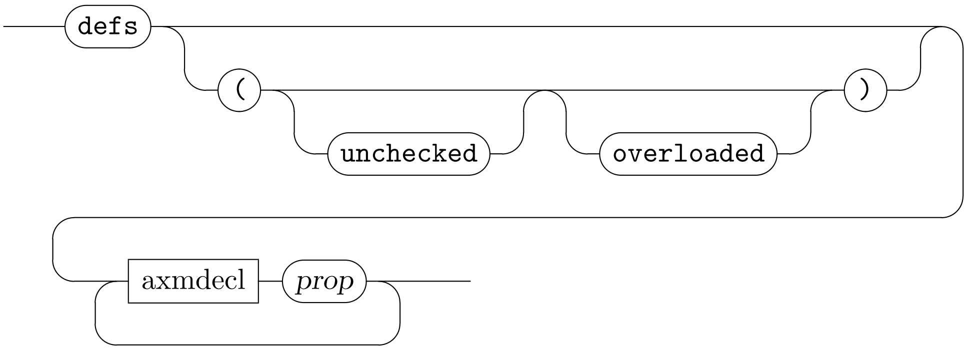
The built-in well-formedness conditions for definitional specifications are:

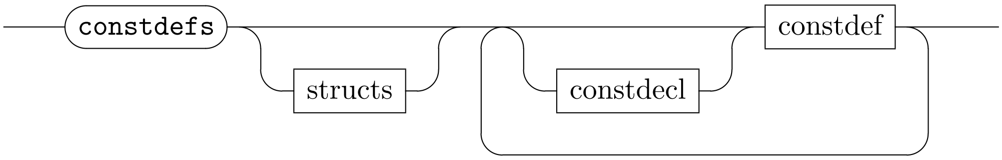
- Arguments (on the left-hand side) must be distinct variables.
- All variables on the right-hand side must also appear on the left-hand side.
- All type variables on the right-hand side must also appear on the left-hand side; this prohibits $0 :: nat \equiv length\ ([] :: \alpha\ list)$ for example.
- The definition must not be recursive. Most object-logics provide definitional principles that can be used to express recursion safely.

Overloading means that a constant being declared as $c :: \alpha\ decl$ may be defined separately on type instances $c :: (\beta_1,\ \ldots,\ \beta_n)\ t\ decl$ for each type constructor t. The right-hand side may mention overloaded constants recursively at type instances corresponding to the immediate argument types $\beta_1, \ldots, \beta_n$. Incomplete specification patterns impose global constraints on all occurrences, e.g. $d :: \alpha \times \alpha$ on the left-hand side means that all corresponding occurrences on some right-hand side need to be an instance of this, general $d :: \alpha \times \beta$ will be disallowed.

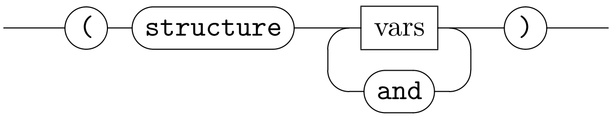
$$
\begin{array}{rcl}
\textbf{consts} & : & \mathit{theory} \to \mathit{theory} \\
\textbf{defs} & : & \mathit{theory} \to \mathit{theory} \\
\textbf{constdefs} & : & \mathit{theory} \to \mathit{theory}
\end{array}
$$



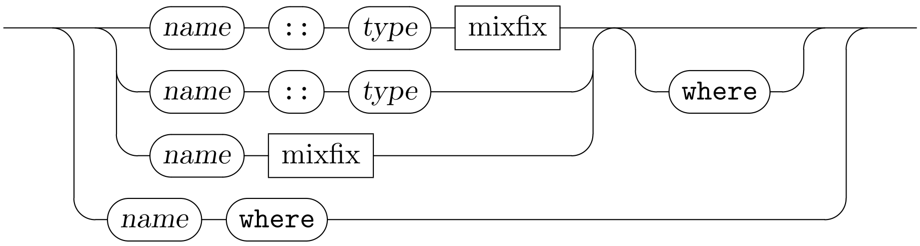




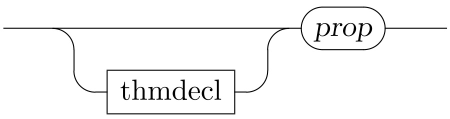
structs



constdecl



constdef



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.

The (*unchecked*) option disables global dependency checks for this definition, which is occasionally useful for exotic overloading. It is at the discretion of the user to avoid malformed theory specifications!

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.

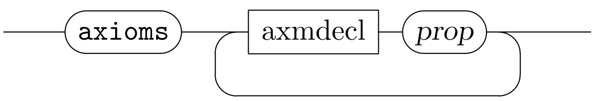
constdefs provides a streamlined combination of constants declarations and definitions: type-inference takes care of the most general typing of the given specification (the optional type constraint may refer to type-inference dummies "_" as usual). The resulting type declaration needs to agree with that of the specification; overloading is *not* supported here!

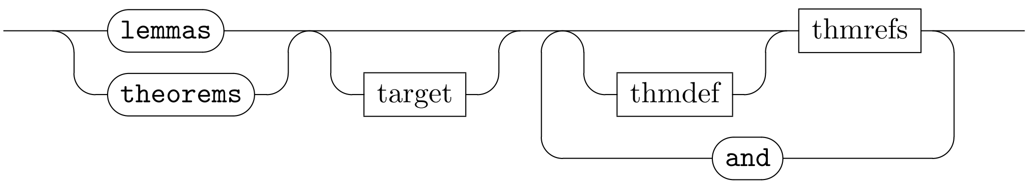
The constant name may be omitted altogether, if neither type nor syntax declarations are given. The canonical name of the definitional axiom for constant c will be c_def, unless specified otherwise. Also note that the given list of specifications is processed in a strictly sequential manner, with type-checking being performed independently.

An optional initial context of (*structure*) declarations admits use of indexed syntax, using the special symbol `\<index>` (printed as "ı"). The latter concept is particularly useful with locales (see also §3.5).

3.10 Axioms and theorems

axioms	:	*theory* → *theory*	(*axiomatic*!)
lemmas	:	*local-theory* → *local-theory*	
theorems	:	*local-theory* → *local-theory*	





axioms a: φ 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.

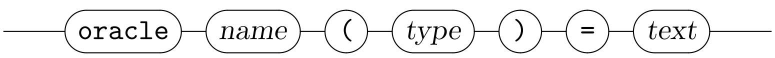
lemmas $a = b_1 \dots b_n$ retrieves and stores existing facts in the theory context, or the specified target context (see also §3.2). Typical applications would also involve attributes, to declare Simplifier rules, for example.

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

3.11 Oracles

oracle : *theory* → *theory*

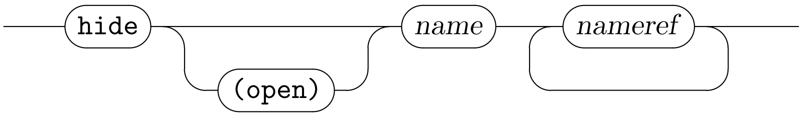
The oracle interface promotes a given ML function `theory -> T -> term` to `theory -> T -> thm`, for some type `T` given by the user. This acts like an infinitary specification of axioms – there is no internal check of the correctness of the results! The inference kernel records oracle invocations within the internal derivation object of theorems, and the pretty printer attaches "[!]" to indicate results that are not fully checked by Isabelle inferences.



oracle *name* (*type*) = *text* turns the given ML expression *text* of type `theory ->` *type* `-> term` into an ML function of type `theory ->` *type* `-> thm`, which is bound to the global identifier `name`.

3.12 Name spaces

global : *theory* → *theory*
local : *theory* → *theory*
hide : *theory* → *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.

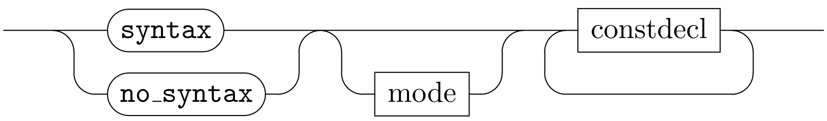
Note that global names are prone to get hidden accidently later, when qualified names of the same base name are introduced.

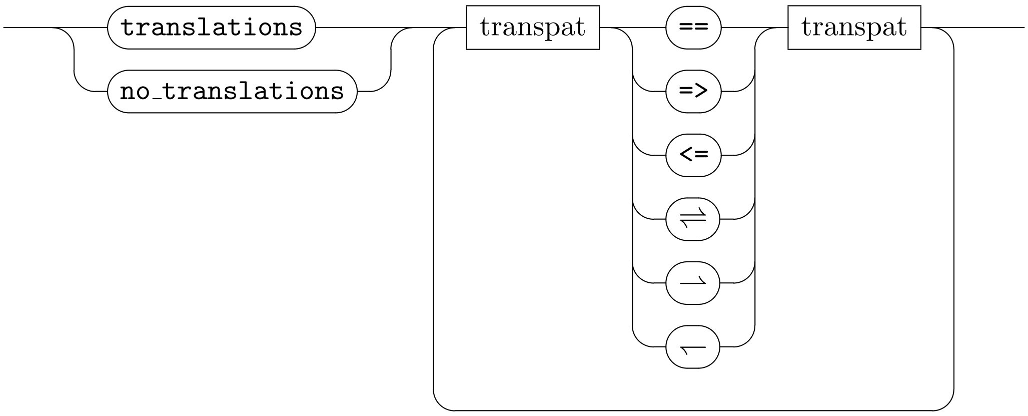
hide *space names* fully removes declarations from a given name space (which may be *class*, *type*, *const*, or *fact*); with the (*open*) option, only the base name is hidden. Global (unqualified) names may never be hidden.

Note that hiding name space accesses has no impact on logical declarations – they remain valid internally. Entities that are no longer accessible to the user are printed with the special qualifier "*??*" prefixed to the full internal name.

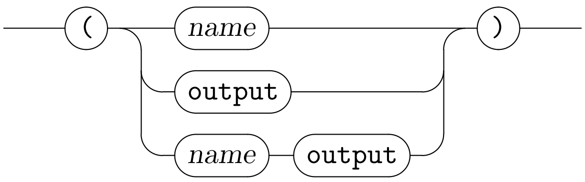
3.13 Syntax and translations

syntax	:	*theory* → *theory*
no_syntax	:	*theory* → *theory*
translations	:	*theory* → *theory*
no_translations	:	*theory* → *theory*

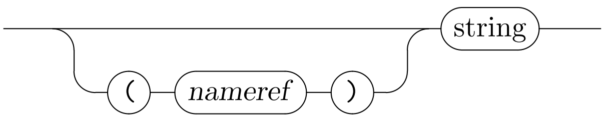




mode



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.

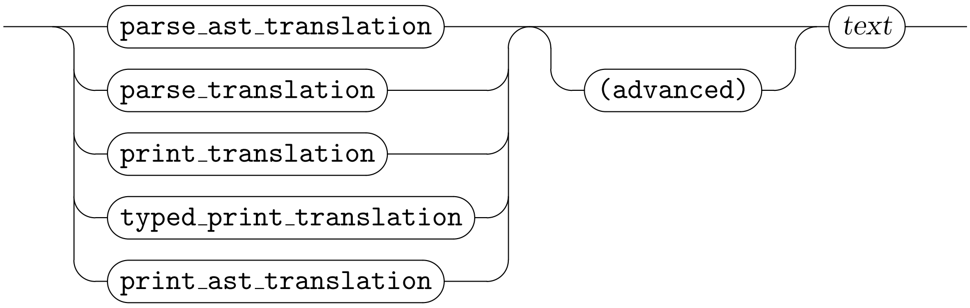
no_syntax (*mode*) *decls* removes grammar declarations (and translations) resulting from *decls*, which are interpreted in the same manner as for **syntax** above.

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

no_translations *rules* removes syntactic translation rules, which are interpreted in the same manner as for **translations** above.

3.14 Syntax translation functions

$$
\begin{array}{rcl}
\textbf{parse\_ast\_translation} & : & \mathit{theory} \to \mathit{theory} \\
\textbf{parse\_translation} & : & \mathit{theory} \to \mathit{theory} \\
\textbf{print\_translation} & : & \mathit{theory} \to \mathit{theory} \\
\textbf{typed\_print\_translation} & : & \mathit{theory} \to \mathit{theory} \\
\textbf{print\_ast\_translation} & : & \mathit{theory} \to \mathit{theory} \\
\textbf{token\_translation} & : & \mathit{theory} \to \mathit{theory}
\end{array}
$$



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, which are as follows by default:

```
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
```

If the (*advanced*) option is given, the corresponding translation functions may depend on the current theory or proof context. This allows to implement advanced syntax mechanisms, as translations functions may refer to specific theory declarations or auxiliary proof data.

See also [15, §8] for more information on the general concept of syntax transformations in Isabelle.

```
val parse_ast_translation:
  (string * (Proof.context -> ast list -> ast)) list
val parse_translation:
  (string * (Proof.context -> term list -> term)) list
val print_translation:
  (string * (Proof.context -> term list -> term)) list
val typed_print_translation:
  (string * (Proof.context -> bool -> typ -> term list -> term)) list
val print_ast_translation:
  (string * (Proof.context -> ast list -> ast)) list
```

Chapter 4

Proofs

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:

$\mathit{proof}(\mathit{prove})$ 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.

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

$\mathit{proof}(\mathit{chain})$ is intermediate between $\mathit{proof}(\mathit{state})$ and $\mathit{proof}(\mathit{prove})$: 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.

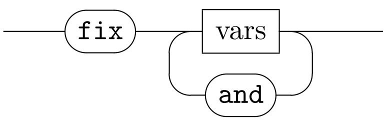
4.1 Context elements

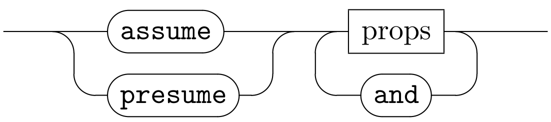
$$\begin{array}{rcl} \textbf{fix} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\ \textbf{assume} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\ \textbf{presume} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\ \textbf{def} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \end{array}$$

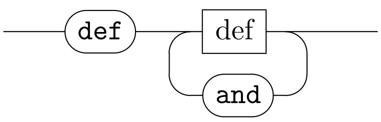
The logical proof context consists of fixed variables and assumptions. The former closely correspond to Skolem constants, or meta-level universal quantification as provided by the Isabelle/Pure logical framework. Introducing some *arbitrary, but fixed* variable via "**fix** x" results in a local value 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: $\vdash \bigwedge x.\ \varphi[x]$ (this is expressed in normal form 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: $\vdash \chi \Longrightarrow \varphi$. 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 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.

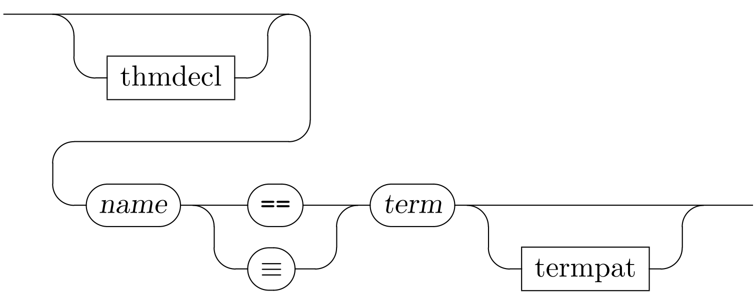
Local definitions, introduced by "**def** $x \equiv t$", are achieved by combining "**fix** x" with another version of assumption that causes any hypothetical equation $x \equiv t$ to be eliminated by the reflexivity rule. Thus, exporting some result $x \equiv t \vdash \varphi[x]$ yields $\vdash \varphi[t]$.







def



fix x introduces a local variable x that is *arbitrary, but fixed.*

assume a: φ and **presume** a: φ introduce a local fact $\varphi \vdash \varphi$ 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 φ 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 $x \equiv t$ 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.

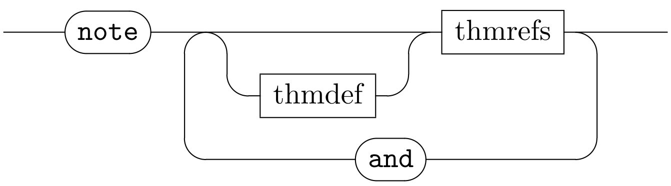
The default name for the definitional equation is *x_def*. Several simultaneous definitions may be given at the same time.

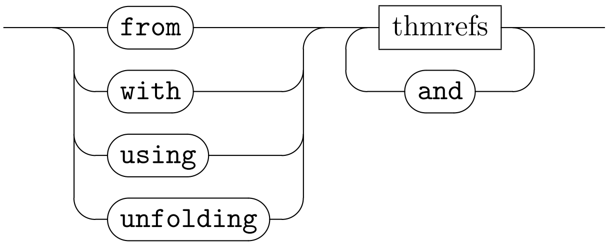
The special name *prems* refers to all assumptions of the current context as a list of theorems. This feature should be used with great care! It is better avoided in final proof texts.

4.2 Facts and forward chaining

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

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** elements 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 $a = b_1 \ldots b_n$ recalls existing facts $b_1, \ldots, b_n$, 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 §4.4). For example, method *rule* (see §4.5) 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 b abbreviates "**note** b **then**"; thus **then** is equivalent to "**from** *this*".

with $b_1 \ldots b_n$ abbreviates "**from** $b_1 \ldots b_n$ **and** *this*"; thus the forward chaining is from earlier facts together with the current ones.

using $b_1 \ldots b_n$ augments the facts being currently indicated for use by a subsequent refinement step (such as **apply** or **proof**).

unfolding $b_1 \ldots b_n$ is structurally similar to **using**, but unfolds definitional equations $b_1, \ldots b_n$ throughout the goal state and facts.

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** _ **and** a **and** b, for example. This involves the trivial rule $PROP\ \psi \Longrightarrow PROP\ \psi$, which is 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.

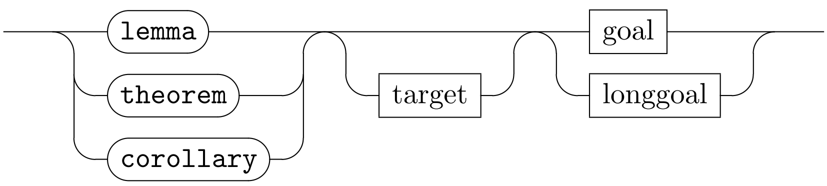
4.3 Goal statements

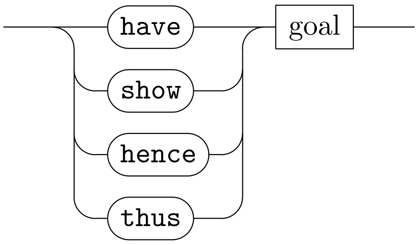
lemma	:	*local-theory* $\rightarrow$ *proof*(*prove*)
theorem	:	*local-theory* $\rightarrow$ *proof*(*prove*)
corollary	:	*local-theory* $\rightarrow$ *proof*(*prove*)
have	:	*proof*(*state*) $\mid$ *proof*(*chain*) $\rightarrow$ *proof*(*prove*)
show	:	*proof*(*state*) $\mid$ *proof*(*chain*) $\rightarrow$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

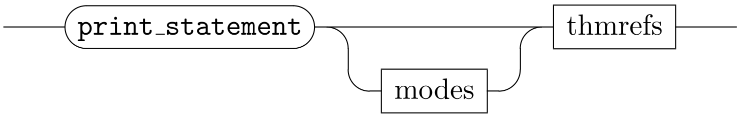
From a theory context, proof mode is entered by an initial goal command such as **lemma**, **theorem**, or **corollary**. Within a proof, new claims may be introduced locally as well; four variants are available here to indicate whether forward chaining of facts should be performed initially (via **then**), 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** (§4.4) or **apply** (§4.8). The *induct* method covered in §4.12 acts on multiple claims simultaneously.

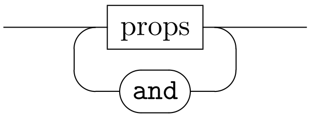
Claims at the theory level may be either in short or long form. A short goal merely consists of several simultaneous propositions (often just one). A long goal includes an explicit context specification for the subsequent conclusion, involving local parameters and assumptions. Here the role of each part of the statement is explicitly marked by separate keywords (see also §3.5); the local assumptions being introduced here are available as *assms* in the proof. Moreover, there are two kinds of conclusions: **shows** states several simultaneous propositions (essentially a big conjunction), while **obtains** claims several simultaneous simultaneous contexts of (essentially a big disjunction of eliminated parameters and assumptions, cf. §4.10).



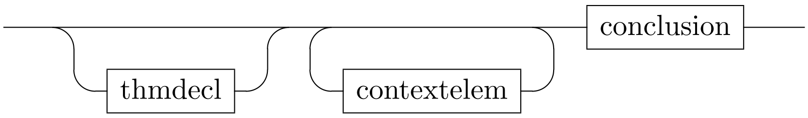




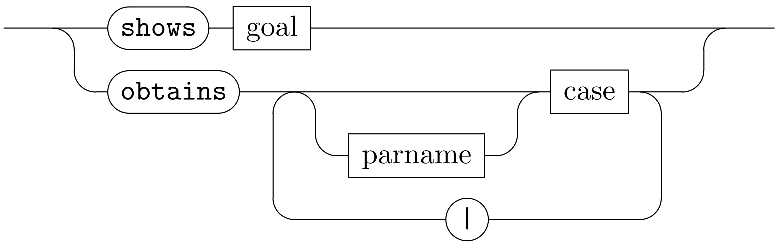
goal



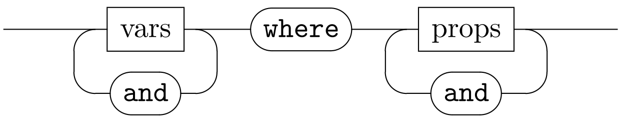
longgoal



conclusion



case



lemma a: φ enters proof mode with φ as main goal, eventually resulting in some fact $\vdash \varphi$ to be put back into the target context. 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 §3.5.

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**".

print_statement a prints facts from the current theory or proof context in long statement form, according to the syntax for **lemma** given above.

Any goal statement causes some term abbreviations (such as *?thesis*) to be bound automatically, see also §4.6.

The optional case names of **obtains** have a twofold meaning: (1) during the of this claim they refer to the the local context introductions, (2) the resulting rule is annotated accordingly to support symbolic case splits when used with the *cases* method (cf. §4.12).

! 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 §4.8). In any case, users should know what they are doing.

4.4 Initial and terminal proof steps

$$\begin{array}{rcl}
\textbf{proof} & : & \mathit{proof}(\mathit{prove}) \to \mathit{proof}(\mathit{state}) \\
\textbf{qed} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \mid \mathit{theory} \\
\textbf{by} & : & \mathit{proof}(\mathit{prove}) \to \mathit{proof}(\mathit{state}) \mid \mathit{theory} \\
\textbf{..} & : & \mathit{proof}(\mathit{prove}) \to \mathit{proof}(\mathit{state}) \mid \mathit{theory} \\
\textbf{.} & : & \mathit{proof}(\mathit{prove}) \to \mathit{proof}(\mathit{state}) \mid \mathit{theory} \\
\textbf{sorry} & : & \mathit{proof}(\mathit{prove}) \to \mathit{proof}(\mathit{state}) \mid \mathit{theory}
\end{array}$$

Arbitrary goal refinement via tactics is considered harmful. Structured proof composition in Isar admits proof methods to be invoked in two places only.

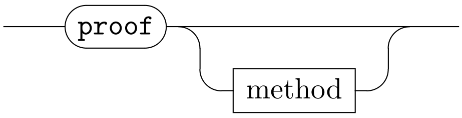
1. An *initial* refinement step **proof** m_1 reduces a newly stated goal to a number of sub-goals that are to be solved later. Facts are passed to m_1 for forward chaining, if so indicated by *proof*(*chain*) mode.

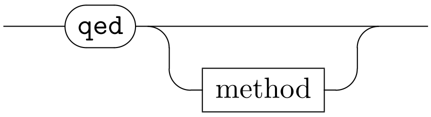
2. A *terminal* conclusion step **qed** m_2 is intended to solve remaining goals. No facts are passed to m_2.

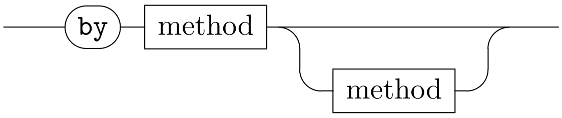
The only other (proper) way to affect pending goals in a proof body is by **show**, which involves an explicit statement of what is to be solved eventually. Thus we avoid the fundamental problem of unstructured tactic scripts that consist of numerous consecutive goal transformations, with invisible effects.

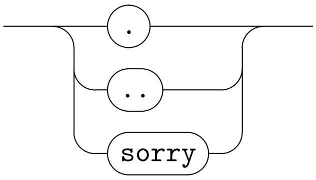
As a general rule of thumb for good proof style, initial proof methods should either solve the goal completely, or constitute some well-understood reduction to new sub-goals. Arbitrary automatic proof tools that are prone leave a large number of badly structured sub-goals are no help in continuing the 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 goal[1] 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 command can 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*.

[1]This includes any additional "strong" assumptions as introduced by **assume**.

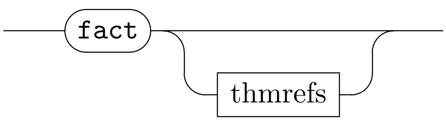
sorry is a *fake proof* pretending to solve the pending claim without further ado. This only works in interactive development, or if the `quick_and_dirty` flag is enabled (in ML). Facts emerging from fake proofs are not the real thing. Internally, each theorem container is tainted by an oracle invocation,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

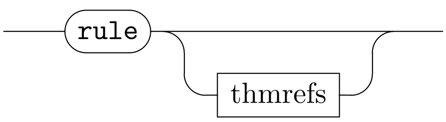
The most important application of **sorry** is to support experimentation and top-down proof development.

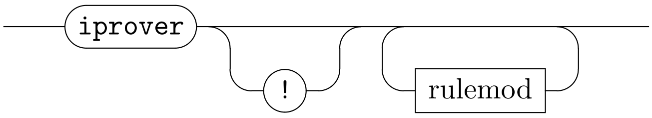
4.5 Fundamental methods and attributes

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 chapter 7 and chapter 8).

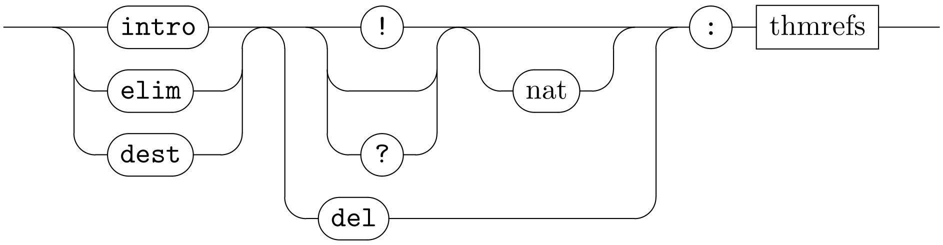
–	:	*method*
fact	:	*method*
assumption	:	*method*
this	:	*method*
rule	:	*method*
iprover	:	*method*
intro	:	*attribute*
elim	:	*attribute*
dest	:	*attribute*
rule	:	*attribute*
OF	:	*attribute*
of	:	*attribute*
where	:	*attribute*

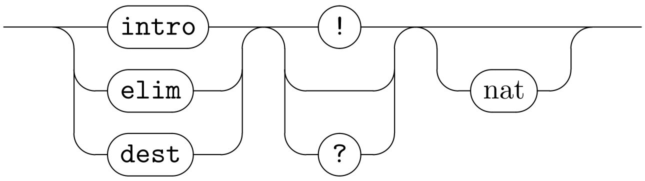


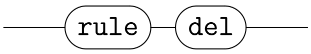


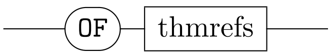


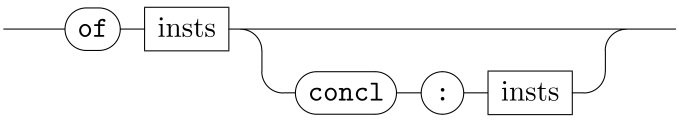
rulemod

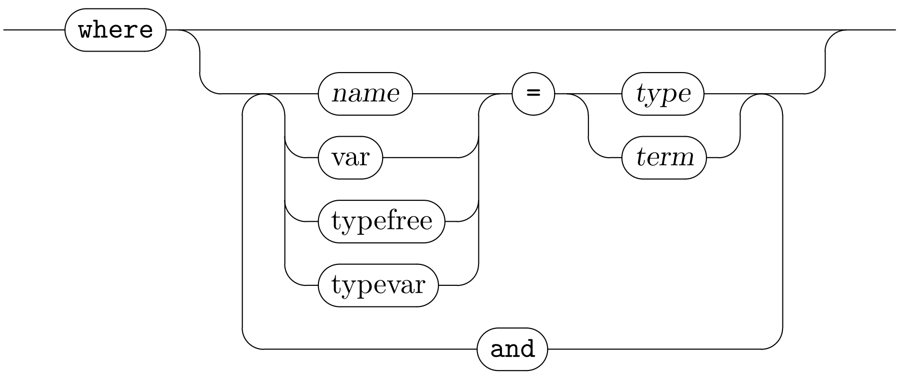












"–" (minus) does nothing but insert the forward chaining facts as premises into the goal. Note that command **proof** without any method actually performs a single reduction step using the *rule* method; thus a plain *do-nothing* proof step would be "**proof** $-$" rather than **proof** alone.

fact $a_1 \ldots a_n$ composes some fact from $a_1, \ldots, a_n$ (or implicitly from the current proof context) modulo unification of schematic type and term variables. The rule structure is not taken into account, i.e. meta-level implication is considered atomic. This is the same principle underlying literal facts (cf. §2.2.7): "**have** φ **by** *fact*" is equivalent to "**note** 'φ'" provided that $\vdash \varphi$ is an instance of some known $\vdash \varphi$ in the proof context.

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** (§4.4) 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 $a_1 \ldots a_n$ 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 §4.4).

iprover 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; "*iprover*!" 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 *iprover* methods. Note that the latter will ignore rules declared with "*?*", while "!" are used most aggressively.

The classical reasoner (see §7.4) introduces its own variants of these attributes; use qualified names to access the present versions of Isabelle/Pure, i.e. *Pure.intro*.

rule del undeclares introduction, elimination, or destruct rules.

OF $a_1 \ldots a_n$ applies some theorem to all of the given rules $a_1, \ldots, a_n$ (in parallel). This corresponds to the `"op MRS"` operation in ML, but note the reversed order. Positions may be effectively skipped by including "_" (underscore) as argument.

of $t_1 \ldots t_n$ performs positional instantiation of term variables. The terms $t_1, \ldots, t_n$ 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.

where $x_1 = t_1$ **and** $\ldots$ $x_n = t_n$ performs named instantiation of schematic type and term variables occurring in a theorem. Schematic variables have to be specified on the left-hand side (e.g. $?x1.3$). The question mark may be omitted if the variable name is a plain identifier without index. As type instantiations are inferred from term instantiations, explicit type instantiations are seldom necessary.

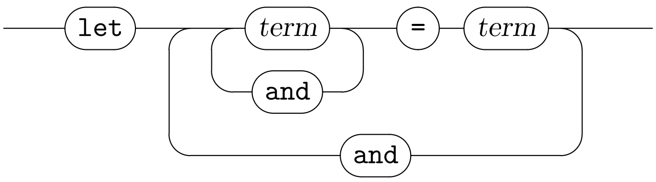
4.6 Term abbreviations

$$\begin{array}{rcl} \textbf{let} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\ \textbf{is} & : & \mathit{syntax} \end{array}$$

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** $p_1 \ldots p_n$)". 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 §4.1). 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 $p_1 = t_1$ **and** $\ldots$ $p_n = t_n$ binds any text variables in patterns $p_1, \ldots, p_n$ by simultaneous higher-order matching against terms $t_1, \ldots, t_n$.

(**is** $p_1 \ldots p_n$) resembles **let**, but matches $p_1, \ldots, p_n$ against the preceding statement. Also note that **is** is not a separate command, but part of others (such as **assume**, **have** etc.).

Some *implicit* 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 this convenience are calculational proofs (see §4.11).

4.7 Block structure

$$\begin{array}{rcl} \textbf{next} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\ \boldsymbol{\{} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\ \boldsymbol{\}} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \end{array}$$

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* internal 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.

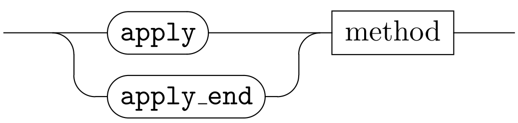
next switches to a fresh block within a sub-proof, resetting the local context to the initial one.

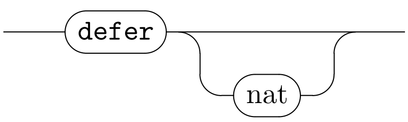
{ and **}** explicitly open and close blocks. Any current facts pass through "**{**" unchanged, while "**}**" causes any result to be *exported* into the enclosing context. Thus fixed variables are generalized, assumptions discharged, and local definitions unfolded (cf. §4.1). There is no difference of **assume** and **presume** in this mode of forward reasoning — in contrast to plain backward reasoning with the result exported at **show** time.

4.8 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 §7.2.3 for actual tactics, that have been encapsulated as proof methods. Proper proof methods may be used in scripts, too.

apply*	:	*proof*(*prove*) → *proof*(*prove*)
apply_end*	:	*proof*(*state*) → *proof*(*state*)
done*	:	*proof*(*prove*) → *proof*(*state*)
defer*	:	*proof* → *proof*
prefer*	:	*proof* → *proof*
back*	:	*proof* → *proof*





`prefer` nat

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 here. 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 sub-goal n to the end of the list ($n = 1$ by default), while **prefer** brings sub-goal n to the front.

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

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.

4.9 Omitting proofs

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 §4.4): **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.

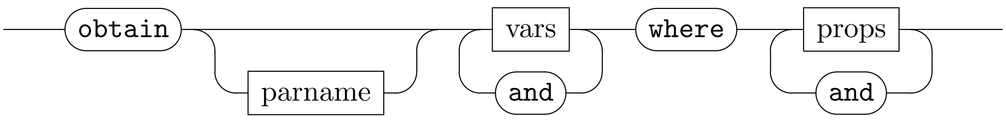
A typical application of **oops** is to explain Isar proofs *within* the system itself, in conjunction with the document preparation tools of Isabelle described in [24]. Thus partial or even wrong proof attempts can be discussed in a logically sound manner. Note that the Isabelle LaTeX macros can be easily adapted to print something like "..." instead of the keyword "**oops**".

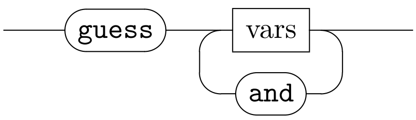
The **oops** command is undo-able, unlike **kill** (see §6.3). The effect is to get back to the theory just before the opening of the proof.

4.10 Generalized elimination

obtain	:	$\mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{prove})$
guess*	:	$\mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{prove})$

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 §4.1), 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.





The derived Isar command **obtain** is defined as follows (where $b_1, \ldots, b_k$

shall refer to (optional) facts indicated for forward chaining).

$\langle using\ b_1 \ldots b_k \rangle$ **obtain** $x_1 \ldots x_m$ **where** a: $\varphi_1 \ldots \varphi_n$ $\langle proof \rangle \equiv$

have $\bigwedge thesis.\ (\bigwedge x_1 \ldots x_m.\ \varphi_1 \Longrightarrow \ldots \varphi_n \Longrightarrow thesis) \Longrightarrow thesis$
proof *succeed*
  fix *thesis*
  assume *that* [*Pure.intro?*]: $\bigwedge x_1 \ldots x_m.\ \varphi_1 \Longrightarrow \ldots \varphi_n \Longrightarrow thesis$
  then show *thesis*
    apply $-$
    using $b_1 \ldots b_k$ $\langle proof \rangle$
qed
fix $x_1 \ldots x_m$ **assume*** a: $\varphi_1 \ldots \varphi_n$

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 existential 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.

An alternative name to be used instead of "*that*" above may be given in parentheses.

The improper variant **guess** is similar to **obtain**, but derives the obtained statement from the course of reasoning! The proof starts with a fixed goal *thesis*. The subsequent proof may refine this to anything of the form like $\bigwedge x_1 \ldots x_m.\ \varphi_1 \Longrightarrow \ldots \varphi_n \Longrightarrow thesis$, but must not introduce new subgoals. The final goal state is then used as reduction rule for the obtain scheme described above. Obtained parameters $x_1, \ldots, x_m$ are marked as internal by default, which prevents the proof context from being polluted by ad-hoc variables. The variable names and type constraints given as arguments for **guess** specify a prefix of obtained parameters explicitly in the text.

It is important to note that the facts introduced by **obtain** and **guess** may not be polymorphic: any type-variables occurring here are fixed in the present context!

4.11 Calculational reasoning

$$
\begin{array}{rcl}
\textbf{also} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\
\textbf{finally} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{chain}) \\
\textbf{moreover} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\
\textbf{ultimately} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{chain}) \\
\textbf{print\_trans\_rules}^* & : & \mathit{theory} \mid \mathit{proof} \to \mathit{theory} \mid \mathit{proof} \\
\mathit{trans} & : & \mathit{attribute} \\
\mathit{sym} & : & \mathit{attribute} \\
\mathit{symmetric} & : & \mathit{attribute}
\end{array}
$$

Calculational proof is forward reasoning with implicit application of transitivity rules (such those of $=$, $\leq$, $<$). Isabelle/Isar maintains an auxiliary fact 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.

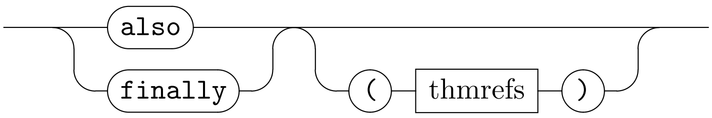
Also note that the implicit term abbreviation "..." has its canonical application with calculational proofs. It refers to the argument of the preceding statement. (The argument of a curried infix expression happens to be its right-hand side.)

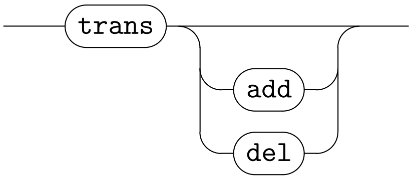
Isabelle/Isar calculations are implicitly subject to block structure in the sense that new threads of calculational reasoning are commenced for any new block (as opened by a local goal, for example). This means that, apart from being able to nest calculations, there is no separate *begin-calculation*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

The Isar calculation proof commands may be defined as follows:[2]

$$
\begin{array}{rcl}
\textbf{also}_0 & \equiv & \textbf{note}\ \mathit{calculation} = \mathit{this} \\
\textbf{also}_{n+1} & \equiv & \textbf{note}\ \mathit{calculation} = \mathit{trans}\ [\mathit{OF}\ \mathit{calculation}\ \mathit{this}] \\
\textbf{finally} & \equiv & \textbf{also}\ \textbf{from}\ \mathit{calculation} \\
\textbf{moreover} & \equiv & \textbf{note}\ \mathit{calculation} = \mathit{calculation}\ \mathit{this} \\
\textbf{ultimately} & \equiv & \textbf{moreover}\ \textbf{from}\ \mathit{calculation}
\end{array}
$$

[2]We suppress internal bookkeeping such as proper handling of block-structure.





also $(a_1 \ldots a_n)$ 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 $(a_1 \ldots a_n)$ 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*. Typical idioms for concluding calculational proofs are "**finally show** *?thesis* **.**" and "**finally have** φ **.**".

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 patters) of the current context.

trans declares theorems as transitivity rules.

sym declares symmetry rules, as well as *Pure.elim?* 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** $x = y$ **then have** $y = x$ **..**".

4.12 Proof by cases and induction

4.12.1 Rule contexts

case	:	*proof*(*state*) → *proof*(*state*)
print_cases*	:	*proof* → *proof*
case_names	:	*attribute*
case_conclusion	:	*attribute*
params	:	*attribute*
consumes	:	*attribute*

The puristic way to build up Isar proof contexts is by explicit language elements like **fix**, **assume**, **let** (see §4.1). This is adequate for plain natural deduction, but easily becomes unwieldy in concrete verification tasks, which typically involve big induction rules with several cases.

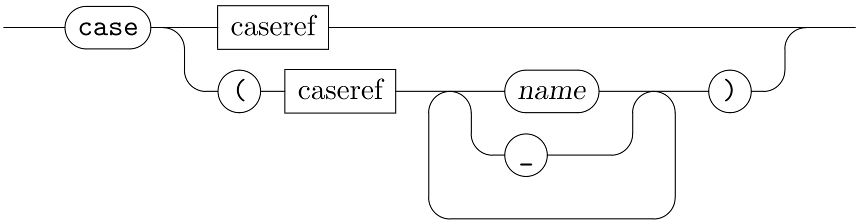
The **case** command provides a shorthand to refer to a local context symbolically: certain proof methods provide an environment of named "cases" of the form c: $x_1, \ldots, x_m, \varphi_1, \ldots, \varphi_n$; the effect of "**case** c" is then equivalent to "**fix** $x_1 \ldots x_m$ **assume** c: $\varphi_1 \ldots \varphi_n$". Term bindings may be covered as well, notably *?case* for the main conclusion.

By default, the "terminology" $x_1, \ldots, x_m$ of a case value is marked as hidden, i.e. there is no way to refer to such parameters in the subsequent proof text. After all, original rule parameters stem from somewhere outside of the current proof text. By using the explicit form "**case** $(c\ y_1 \ldots y_m)$" instead, the proof author is able to chose local names that fit nicely into the current context.

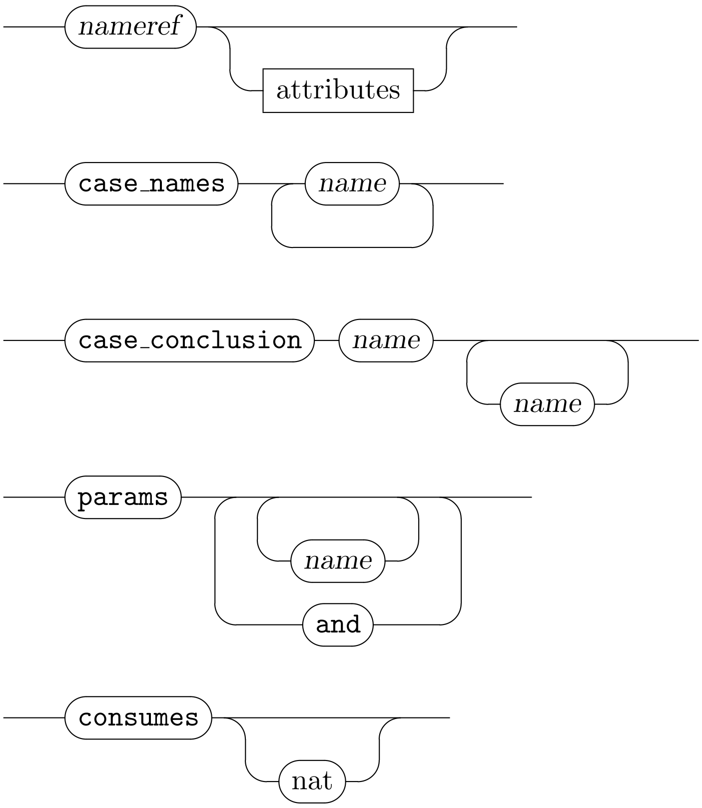
It is important to note that proper use of **case** does not provide means to peek at the current goal state, which is not directly observable in Isar! Nonetheless, goal refinement commands do provide named cases $goal_i$ for each subgoal $i = 1, \ldots, n$ of the resulting goal state. Using this extra feature requires great care, because some bits of the internal tactical machinery intrude the proof text. In particular, parameter names stemming from the left-over of automated reasoning tools are usually quite unpredictable.

Under normal circumstances, the text of cases emerge from standard elimination or induction rules, which in turn are derived from previous theory specifications in a canonical way (say from **inductive** definitions).

Proper cases are only available if both the proof method and the rules involved support this. By using appropriate attributes, case names, conclusions, and parameters may be also declared by hand. Thus variant versions of rules that have been derived manually become ready to use in advanced case analysis later.



caseref



case $(c\ x_1\ \dots\ x_m)$ invokes a named local context c: $x_1, \dots, x_m, \varphi_1, \dots, \varphi_m$, as provided by an appropriate proof method (such as *cases* and *induct*). The command "**case** $(c\ x_1\ \dots\ x_m)$" abbreviates "**fix** $x_1\ \dots\ x_m$ **assume** c: $\varphi_1\ \dots\ \varphi_n$".

print_cases prints all local contexts of the current state, using Isar proof language notation.

case_names $c_1\ \dots\ c_k$ declares names for the local contexts of premises of a theorem; $c_1, \dots, c_k$ refers to the *suffix* of the list of premises.

case_conclusion c $d_1 \ldots d_k$ declares names for the conclusions of a named premise c; here $d_1, \ldots, d_k$ refers to the prefix of arguments of a logical formula built by nesting a binary connective (e.g. $\vee$).

Note that proof methods such as *induct* and *coinduct* already provide a default name for the conclusion as a whole. The need to name subformulas only arises with cases that split into several sub-cases, as in common co-induction rules.

params $p_1 \ldots p_m$ **and** $\ldots$ $q_1 \ldots q_n$ renames the innermost parameters of premises $1, \ldots, n$ 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.

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. The default value of *consumes* is $n = 1$, which is appropriate for the usual kind of cases and induction rules for inductive sets (cf. §8.6). Rules without any *consumes* declaration given are treated as if *consumes* 0 had been specified.

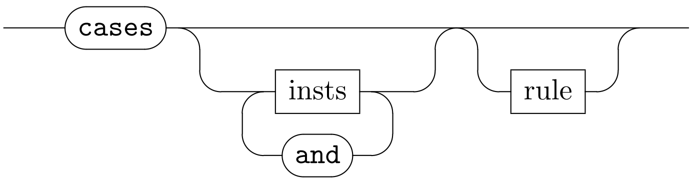
Note that explicit *consumes* declarations are only rarely needed; this is already taken care of automatically by the higher-level *cases*, *induct*, and *coinduct* declarations.

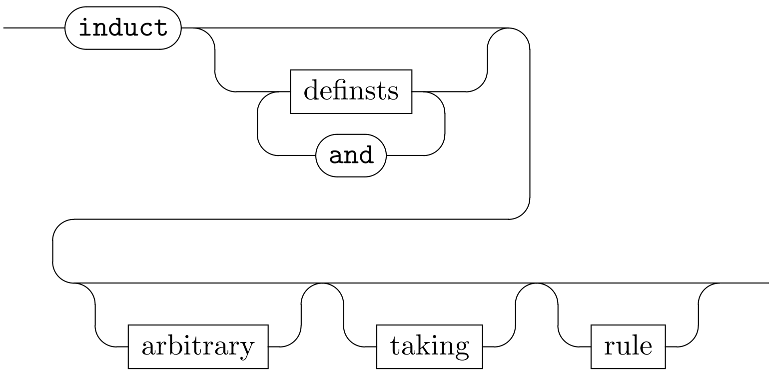
4.12.2 Proof methods

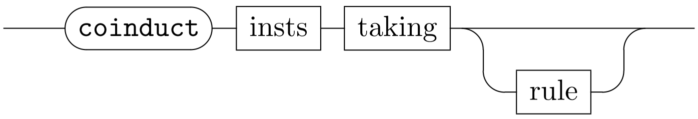
$$\begin{array}{rcl} \mathit{cases} & : & \mathit{method} \\ \mathit{induct} & : & \mathit{method} \\ \mathit{coinduct} & : & \mathit{method} \end{array}$$

The *cases*, *induct*, and *coinduct* methods provide a uniform interface to common proof techniques over datatypes, inductive predicates (or sets), recursive functions etc. 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.

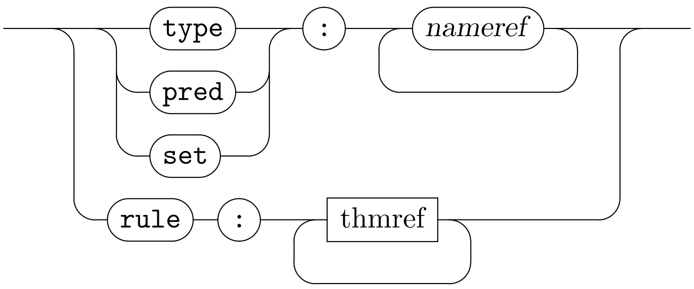
The *induct* method also provides some additional infrastructure in order to be applicable to structure statements (either using explicit meta-level connectives, or including facts and parameters separately). This avoids cumbersome encoding of "strengthened" inductive statements within the object-logic.



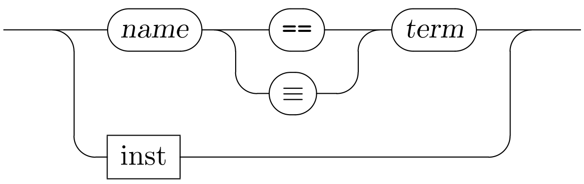




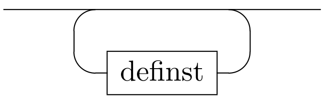
rule



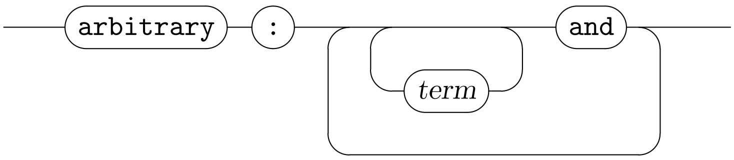
definst



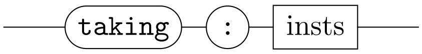
definsts



arbitrary



taking



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
	cases		classical case split
	cases	t	datatype exhaustion (type of t)
$\vdash A\ t$	*cases*	$\ldots$	inductive predicate/set elimination (of A)
$\ldots$	*cases*	$\ldots$ *rule*: R	explicit rule R

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).

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

facts		arguments	rule
	induct	$P\ x$	datatype induction (type of x)
$\vdash A\ x$	*induct*	$\ldots$	predicate/set induction (of A)
$\ldots$	*induct*	$\ldots$ *rule*: 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, ... 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.

Instantiations may be definitional: equations $x \equiv t$ introduce local definitions, which are inserted into the claim and discharged after applying the induction rule. Equalities reappear in the inductive cases, but have been transformed according to the induction principle being involved here. In order to achieve practically useful induction hypotheses, some variables occurring in t need to be fixed (see below).

The optional "*arbitrary*: $x_1 \ldots x_m$" specification generalizes variables $x_1, \ldots, x_m$ of the original goal before applying induction. Thus induction hypotheses may become sufficiently general to get the proof through. Together with definitional instantiations, one may effectively perform induction over expressions of a certain structure.

The optional "*taking*: $t_1 \ldots t_n$" specification provides additional instantiations of a prefix of pending variables in the rule. Such schematic induction rules rarely occur in practice, though.

coinduct inst R is analogous to the *induct* method, but refers to coinduction rules, which are determined as follows:

goal		arguments	rule
	coinduct	x	type coinduction (type of x)
$A\ x$	*coinduct*	...	predicate/set coinduction (of A)
...	*coinduct*	... *rule*: R	explicit rule R

Coinduction is the dual of induction. Induction essentially eliminates $A\ x$ towards a generic result $P\ x$, while coinduction introduces $A\ x$ starting with $B\ x$, for a suitable "bisimulation" B. The cases of a coinduct rule are typically named after the predicates or sets being covered, while the conclusions consist of several alternatives being named after the individual destructor patterns.

The given instantiation refers to the *suffix* of variables occurring in the rule's major premise, or conclusion if unavailable. An additional "*taking*: $t_1 \ldots t_n$" specification may be required in order to specify the bisimulation to be used in the coinduction step.

Above methods produce named local contexts, as determined by the instantiated rule as given in the text. Beyond that, the *induct* and *coinduct* methods guess further instantiations from the goal specification itself. Any persisting unresolved schematic variables of the resulting rule will render the 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.

Despite the additional infrastructure, both *cases* and *coinduct* merely apply a certain rule, after instantiation, while conforming due to the usual way of monotonic natural deduction: the context of a structured statement $\bigwedge x_1 \ldots x_m.\ \varphi_1 \Longrightarrow \ldots \varphi_n \Longrightarrow \ldots$ reappears unchanged after the case split.

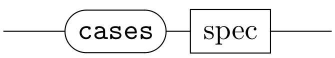
The *induct* method is fundamentally different in this respect: the meta-level structure is passed through the "recursive" course involved in the induction. Thus the original statement is basically replaced by separate copies, corresponding to the induction hypotheses and conclusion; the original goal context is no longer available. Thus local assumptions, fixed parameters and definitions effectively participate in the inductive rephrasing of the original statement.

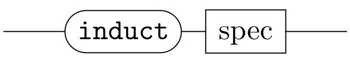
In induction proofs, local assumptions introduced by cases are split into two different kinds: *hyps* stemming from the rule and *prems* from the goal statement. This is reflected in the extracted cases accordingly, so invoking "**case** c" will provide separate facts *c.hyps* and *c.prems*, as well as fact c to hold the all-inclusive list.

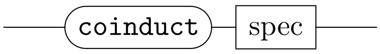
Facts presented to either method are consumed according to the number of "major premises" of the rule involved, which is usually 0 for plain cases and induction rules of datatypes etc. and 1 for rules of inductive predicates or sets and the like. The remaining facts are inserted into the goal verbatim before the actual *cases*, *induct*, or *coinduct* rule is applied.

4.12.3 Declaring rules

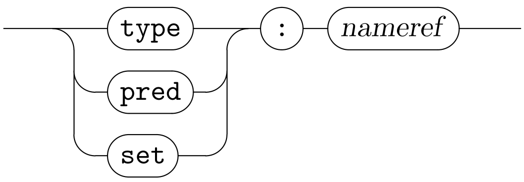
print_induct_rules*	:	*theory* | *proof* → *theory* | *proof*
cases	:	*attribute*
induct	:	*attribute*
coinduct	:	*attribute*







spec



print_induct_rules prints cases and induct rules for predicates (or sets) and types of the current context.

cases, *induct*, and *coinduct* (as attributes) augment the corresponding context of rules for reasoning about (co)inductive predicates (or 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 refer to the *case_names* and *params* attributes to adjust names of cases and parameters of a rule; the *consumes* declaration is taken care of automatically: *consumes* 0 is specified for "type" rules and *consumes* 1 for "predicate" / "set" rules.

Chapter 5

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) and bookmarks. 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 [24]). 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.

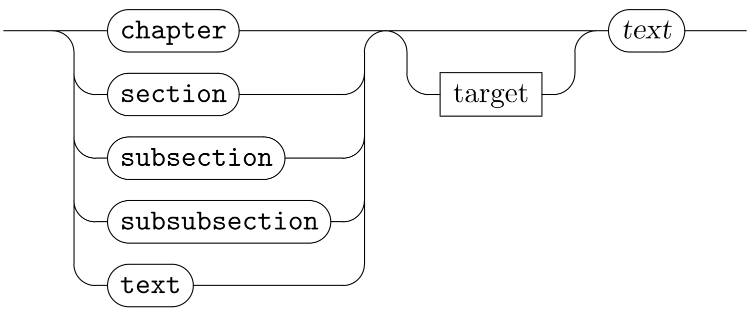
You may also consider to tune the `usedir` options in `IsaMakefile`, for example to change the output format from `pdf` to `dvi`, or activate the `-D` option to retain a second copy of the generated LaTeX sources.

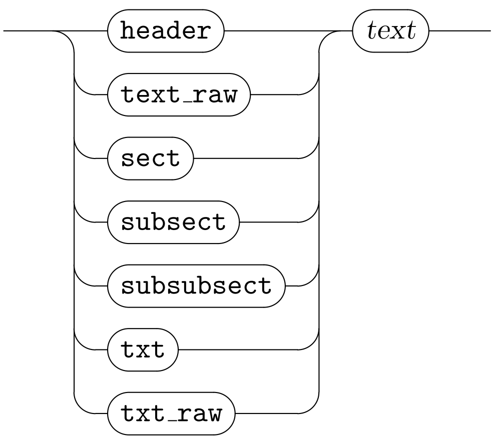
See *The Isabelle System Manual* [24] for further details on Isabelle logic sessions and theory presentation. The Isabelle/HOL tutorial [13] also covers theory presentation issues.

5.1 Markup commands

header	:	*toplevel* → *toplevel*
chapter	:	*local-theory* → *local-theory*
section	:	*local-theory* → *local-theory*
subsection	:	*local-theory* → *local-theory*
subsubsection	:	*local-theory* → *local-theory*
text	:	*local-theory* → *local-theory*
text_raw	:	*local-theory* → *local-theory*
sect	:	*proof* → *proof*
subsect	:	*proof* → *proof*
subsubsect	:	*proof* → *proof*
txt	:	*proof* → *proof*
txt_raw	:	*proof* → *proof*

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 [24] for more information on Isabelle's document preparation tools).





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.

chapter, **section**, **subsection**, and **subsubsection** mark chapter and section headings. The corresponding LaTeX macros are `\isamarkupchapter`, `\isamarkupsection` etc.

text and **txt** specify paragraphs of plain text.

text_raw and **txt_raw** insert LaTeX source into the output, without additional markup. Thus the full range of document manipulations becomes available.

The *text* argument of these markup commands (except for **text_raw**) may contain references to formal entities ("antiquotations", see also §5.2). These are interpreted in the present theory context, or the named *target*.

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.

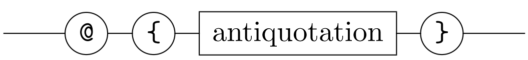
The proof markup commands closely resemble those for theory specifications, but have a different formal status and produce different LaTeX macros. Also note that the **header** declaration (see §3.1) admits to insert section markup just preceding the actual theory definition.

5.2 Antiquotations

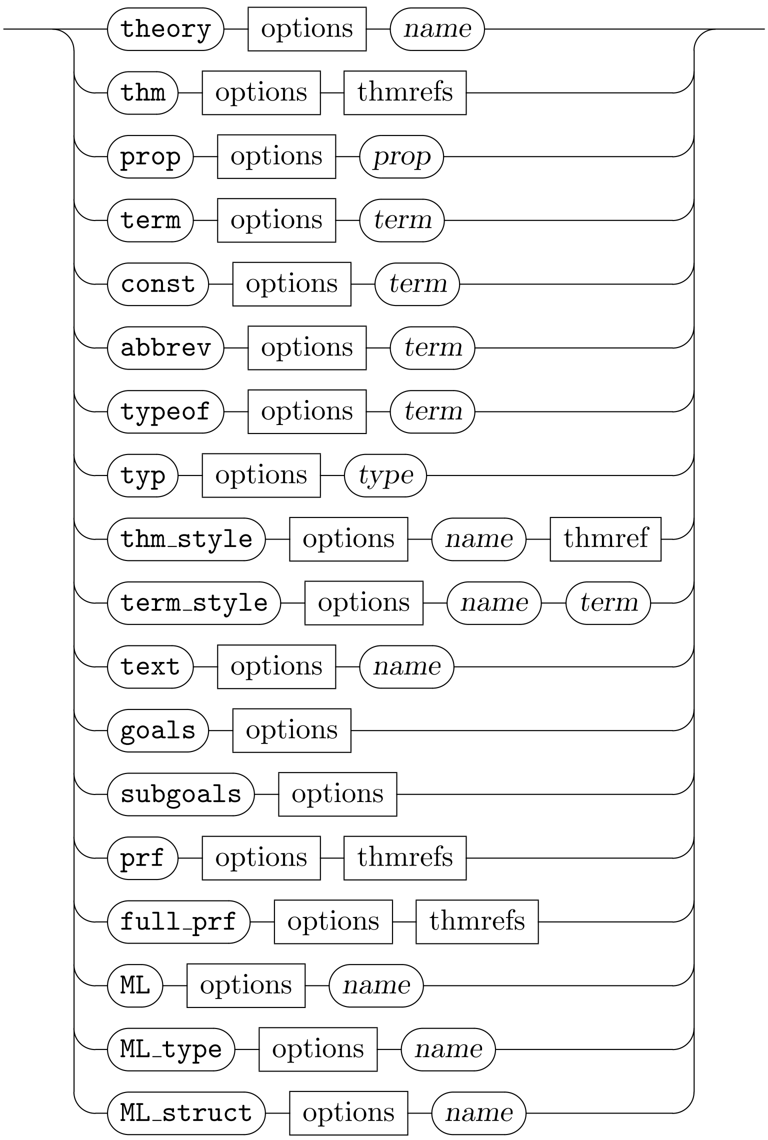
$$
\begin{array}{rcl}
\mathit{theory} & : & \mathit{antiquotation} \\
\mathit{thm} & : & \mathit{antiquotation} \\
\mathit{prop} & : & \mathit{antiquotation} \\
\mathit{term} & : & \mathit{antiquotation} \\
\mathit{const} & : & \mathit{antiquotation} \\
\mathit{abbrev} & : & \mathit{antiquotation} \\
\mathit{typeof} & : & \mathit{antiquotation} \\
\mathit{typ} & : & \mathit{antiquotation} \\
\mathit{thm\_style} & : & \mathit{antiquotation} \\
\mathit{term\_style} & : & \mathit{antiquotation} \\
\mathit{text} & : & \mathit{antiquotation} \\
\mathit{goals} & : & \mathit{antiquotation} \\
\mathit{subgoals} & : & \mathit{antiquotation} \\
\mathit{prf} & : & \mathit{antiquotation} \\
\mathit{full\_prf} & : & \mathit{antiquotation} \\
\mathit{ML} & : & \mathit{antiquotation} \\
\mathit{ML\_type} & : & \mathit{antiquotation} \\
\mathit{ML\_struct} & : & \mathit{antiquotation}
\end{array}
$$

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.

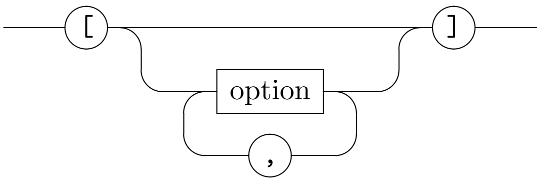
Thus embedding of "@{*term* [*show_types*] $f\ x = a + x$}" within a text block would cause $(f{::}'a \Rightarrow 'a)\ (x{::}'a) = (a{::}'a) + x$ to appear in the final LaTeX document. Also note that theorem antiquotations may involve attributes as well. For example, @{*thm sym* [*no_vars*]} would print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



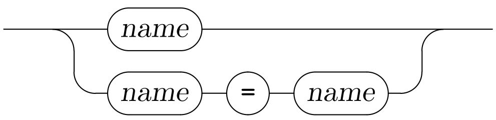
antiquotation



options



option



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

@{*theory* A} prints the name A, which is guaranteed to refer to a valid ancestor theory in the current context.

@{*thm* $a_1 \ldots a_n$} prints theorems $a_1 \ldots a_n$. Note that attribute specifications may be included as well (see also §2.2.7); the *no_vars* rule (see §7.2.1) 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.

@{*const* c} prints a logical or syntactic constant c.

@{*abbrev* $c\ x_1 \ldots x_n$} prints a constant abbreviation $c\ x_1 \ldots x_n \equiv \mathit{rhs}$ as defined in the current context.

@{*typeof* t} prints the type of a well-typed term t.

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

@{*thm_style* $s\ a$} prints theorem a, previously applying a style s to it (see below).

@{*term_style* $s\ t$} prints a well-typed term t after applying a style s to it (see below).

@{*text* s} prints uninterpreted source text s. This is particularly useful to print portions of text according to the Isabelle LaTeX 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*} is similar to @{*goals*}, but does not print the main goal.

@{*prf* $a_1 \ldots a_n$} prints the (compact) proof terms corresponding to the theorems $a_1 \ldots a_n$. Note that this requires proof terms to be switched on for the current object logic (see the "Proof terms" section of the Isabelle reference manual for information on how to do this).

@{*full_prf* $a_1 \ldots a_n$} is like @{*prf* $a_1 \ldots a_n$}, but displays the full proof terms, i.e. also displays information omitted in the compact proof term, which is denoted by "_" placeholders there.

@{*ML s*}, @{*ML_type s*}, and @{*ML_struct s*} check text *s* as ML value, type, and structure, respectively. The source is displayed verbatim.

The following standard styles for use with *thm_style* and *term_style* are available:

lhs extracts the first argument of any application form with at least two arguments – typically meta-level or object-level equality, or any other binary relation.

rhs is like *lhs*, but extracts the second argument.

concl extracts the conclusion C from a rule in Horn-clause normal form $A_1 \Longrightarrow \ldots A_n \Longrightarrow C$.

*prem*1, …, *prem*9 extract premise number 1, …, 9, respectively, from from a rule in Horn-clause normal form $A_1 \Longrightarrow \ldots A_n \Longrightarrow C$

The following options are available to tune the output. Note that most of these coincide with ML flags of the same names (see also [15]).

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

show_structs = *bool* controls printing of implicit structures.

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

short_names = *bool* forces names of types and constants etc. to be printed unqualified. Note that internalizing the output again in the current context may well yield a different result.

unique_names = *bool* determines whether the printed version of qualified names should be made sufficiently long to avoid overlap with names declared further back. Set to *false* for more concise output.

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).

break = *bool* controls line breaks in non-display material.

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 [15]). Note that the standard setup for LaTeX output is already present by default, including the modes *latex* and *xsymbols*.

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.

locale = *name* specifies an alternative locale context used for evaluating and printing the subsequent argument.

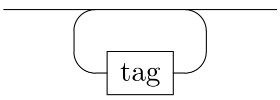
For boolean flags, "*name* = *true*" may be abbreviated as "*name*". All of the above flags are disabled by default, unless changed from ML.

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.

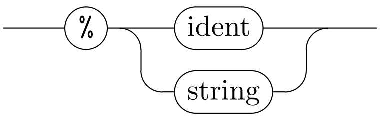
5.3 Tagged commands

Each Isabelle/Isar command may be decorated by presentation tags:

tags



tag



The tags *theory*, *proof*, *ML* are already pre-declared for certain classes of commands:

theory	theory begin/end
proof	all proof commands
ML	all commands involving ML code

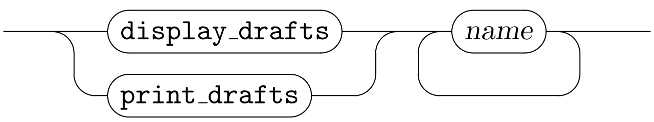
The Isabelle document preparation system (see also [24]) allows tagged command regions to be presented specifically, e.g. to fold proof texts, or drop parts of the text completely.

For example "**by** %*invisible auto*" would cause that piece of proof to be treated as *invisible* instead of *proof* (the default), which may be either show or hidden depending on the document setup. In contrast, "**by** %*visible auto*" would force this text to be shown invariably.

Explicit tag specifications within a proof apply to all subsequent commands of the same level of nesting. For example, "**proof** %*visible* ... **qed**" would force the whole sub-proof to be typeset as *visible* (unless some of its parts are tagged differently).

5.4 Draft presentation

$$\begin{array}{rcl} \textbf{display\_drafts}^* & : & \cdot \rightarrow \cdot \\ \textbf{print\_drafts}^* & : & \cdot \rightarrow \cdot \end{array}$$



display_drafts *paths* and **print_drafts** *paths* perform simple output of a given list of raw source files. Only those symbols that do not require additional LaTeX packages are displayed properly, everything else is left verbatim.

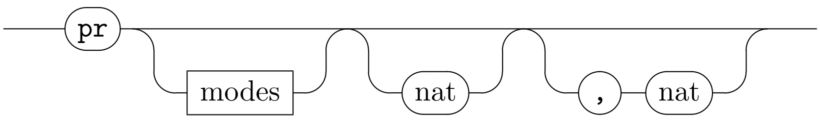
Chapter 6

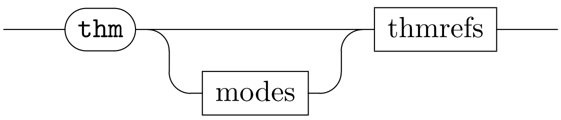
Other commands

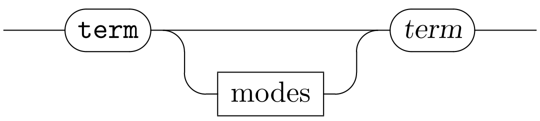
6.1 Diagnostics

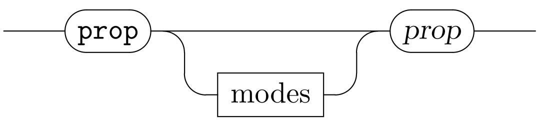
pr*	:	$\cdot \to \cdot$
thm*	:	*theory* | *proof* $\to$ *theory* | *proof*
term*	:	*theory* | *proof* $\to$ *theory* | *proof*
prop*	:	*theory* | *proof* $\to$ *theory* | *proof*
typ*	:	*theory* | *proof* $\to$ *theory* | *proof*
prf*	:	*theory* | *proof* $\to$ *theory* | *proof*
full_prf*	:	*theory* | *proof* $\to$ *theory* | *proof*

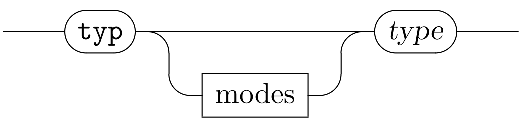
These diagnostic commands assist interactive development. Note that **undo** does not apply here, the theory or proof configuration is not changed.

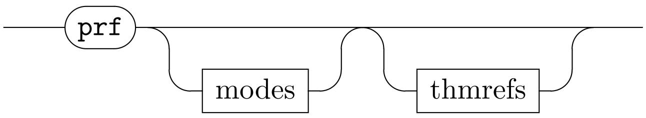


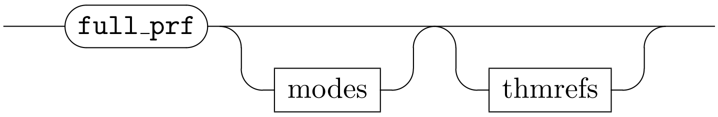




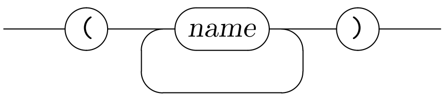








modes



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 $a_1 \ldots a_n$ 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 $a_1, \ldots, a_n$ 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.

prf displays the (compact) proof term of the current proof state (if present), or of the given theorems. Note that this requires proof terms to be switched on for the current object logic (see the "Proof terms" section of the Isabelle reference manual for information on how to do this).

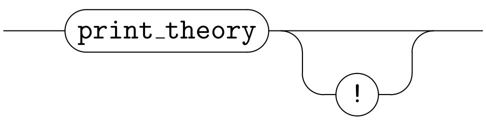
full_prf is like **prf**, but displays the full proof term, i.e. also displays information omitted in the compact proof term, which is denoted by "_" placeholders there.

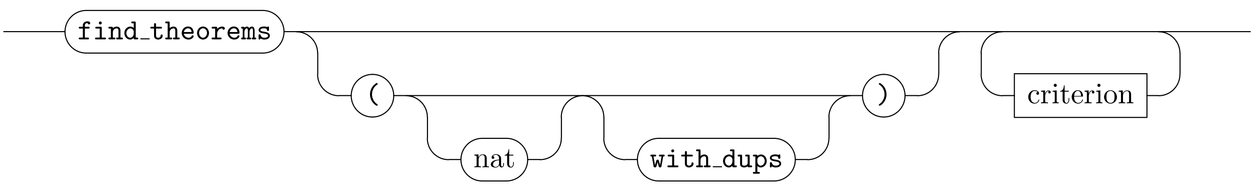
All of the diagnostic commands above admit a list of *modes* to be specified, which is appended to the current print mode (see also [15]). Thus the output behavior may be modified according particular print mode features. For example, **pr** (*latex xsymbols symbols*) would print the current proof state with mathematical symbols and special characters represented in LaTeX source, according to the Isabelle style [24].

Note that antiquotations (cf. §5.2) provide a more systematic way to include formal items into the printed text document.

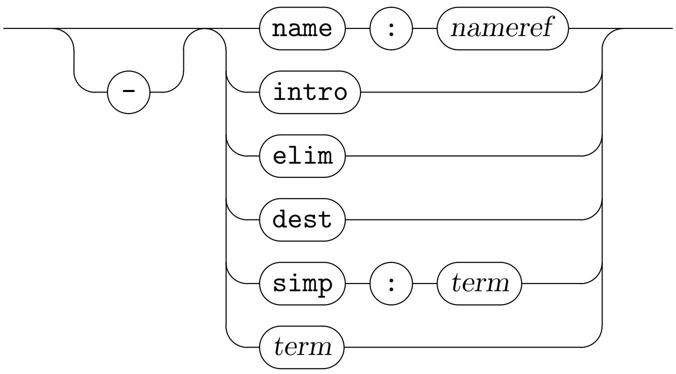
6.2 Inspecting the context

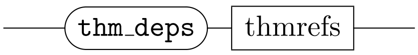
print_commands*	:	$\cdot \to \cdot$
print_theory*	:	*theory* | *proof* $\to$ *theory* | *proof*
print_syntax*	:	*theory* | *proof* $\to$ *theory* | *proof*
print_methods*	:	*theory* | *proof* $\to$ *theory* | *proof*
print_attributes*	:	*theory* | *proof* $\to$ *theory* | *proof*
print_theorems*	:	*theory* | *proof* $\to$ *theory* | *proof*
find_theorems*	:	*theory* | *proof* $\to$ *theory* | *proof*
thm_deps*	:	*theory* | *proof* $\to$ *theory* | *proof*
print_facts*	:	*proof* $\to$ *proof*
print_binds*	:	*proof* $\to$ *proof*





criterion





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_theory prints the main logical content of the theory context; the "!" option indicates extra verbosity.

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 [15, §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 resulting from the last command.

find_theorems *criteria* retrieves facts from the theory or proof context matching all of given search criteria. The criterion *name*: p selects all theorems whose fully qualified name matches pattern p, which may contain "$*$" wildcards. The criteria *intro*, *elim*, and *dest* select theorems that match the current goal as introduction, elimination or destruction rules, respectively. The criterion *simp*: t selects all rewrite rules whose left-hand side matches the given term. The criterion term t selects all theorems that contain the pattern t – as usual, patterns may contain occurrences of the dummy "_", schematic variables, and type constraints.

Criteria can be preceded by "−" to select theorems that do *not* match. Note that giving the empty list of criteria yields *all* currently known facts. An optional limit for the number of printed facts may be given; the default is 40. By default, duplicates are removed from the search result. Use *with_dups* to display duplicates.

thm_deps $a_1 \ldots a_n$ visualizes dependencies of facts, using Isabelle's graph browser tool (see also [24]).

print_facts prints all local facts of the current context, both named and unnamed ones.

print_binds prints all term abbreviations present in the context.

6.3 History commands

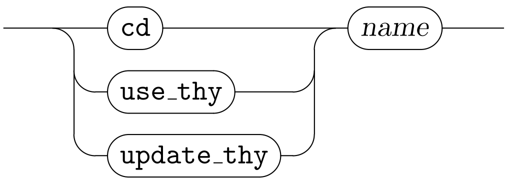
$$
\begin{array}{rcl}
\textbf{undo}^{**} & : & \cdot \to \cdot \\
\textbf{redo}^{**} & : & \cdot \to \cdot \\
\textbf{kill}^{**} & : & \cdot \to \cdot
\end{array}
$$

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.

6.4 System commands

$$
\begin{array}{rcl}
\textbf{cd}^* & : & \cdot \to \cdot \\
\textbf{pwd}^* & : & \cdot \to \cdot \\
\textbf{use\_thy}^* & : & \cdot \to \cdot
\end{array}
$$



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

pwd prints the current working directory.

use_thy *A* preload theory *A*. These system commands are scarcely used when working interactively, since loading of theories is done automatically as required.

Chapter 7

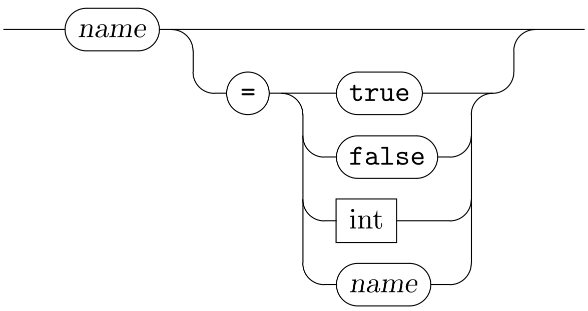
Generic tools and packages

7.1 Configuration options

Isabelle/Pure maintains a record of named configuration options within the theory or proof context, with values of type `bool`, `int`, or `string`. Tools may declare options in ML, and then refer to these values (relative to the context). Thus global reference variables are easily avoided. The user may change the value of a configuration option by means of an associated attribute of the same name. This form of context declaration works particularly well with commands such as **declare** or **using**.

For historical reasons, some tools cannot take the full proof context into account and merely refer to the background theory. This is accommodated by configuration options being declared as "global", which may not be changed within a local context.

print_configs : *theory* | *proof* → *theory* | *proof*



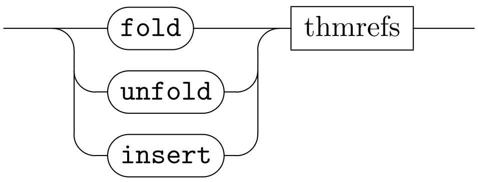
print_configs prints the available configuration options, with names, types, and current values.

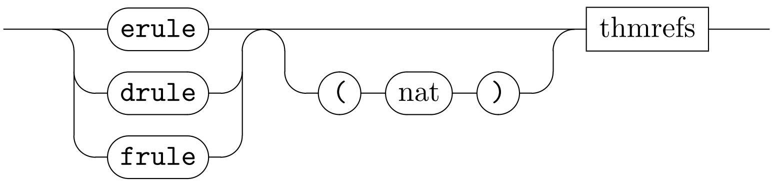
name = *value* as an attribute expression modifies the named option, with the syntax of the value depending on the option's type. For `bool` the default value is *true*. Any attempt to change a global option in a local context is ignored.

7.2 Basic proof tools

7.2.1 Miscellaneous methods and attributes

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





unfold $a_1 \ldots a_n$ and *fold* $a_1 \ldots a_n$ expand (or fold back) the given definitions throughout all goals; any chained facts provided are inserted into the goal and subject to rewriting as well.

insert $a_1 \ldots a_n$ inserts theorems as facts into all goals of the proof state. Note that current facts indicated for forward chaining are ignored.

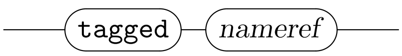
erule $a_1 \ldots a_n$, *drule* $a_1 \ldots a_n$, and *frule* $a_1 \ldots a_n$ are similar to the basic *rule* method (see §4.5), but apply rules by elim-resolution, destruct-resolution, and forward-resolution, respectively [15]. The optional natural number argument (default 0) specifies additional assumption steps to be performed here.

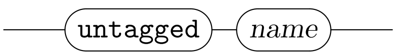
Note that these methods are improper ones, mainly serving for experimentation and tactic script emulation. Different modes of basic rule application are usually expressed in Isar at the proof language level, rather than via implicit proof state manipulations. For example, a proper single-step elimination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

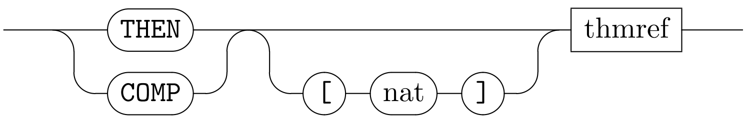
succeed yields a single (unchanged) result; it is the identity of the "," method combinator (cf. §2.2.6).

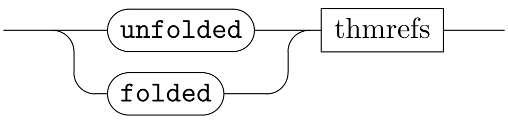
fail yields an empty result sequence; it is the identity of the "|" method combinator (cf. §2.2.6).

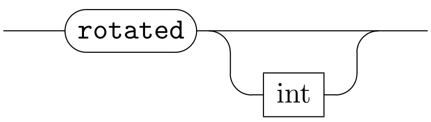
tagged	:	*attribute*
untagged	:	*attribute*
THEN	:	*attribute*
COMP	:	*attribute*
unfolded	:	*attribute*
folded	:	*attribute*
rotated	:	*attribute*
elim_format	:	*attribute*
*standard**	:	*attribute*
*no_vars**	:	*attribute*











tagged name arg and *untagged name* add and remove *tags* of some theorem. Tags may be any list of string pairs that serve as formal comment. The first string is considered the tag name, the second its argument. Note that *untagged* 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. `"op RS"` and `"op COMP"` in [15, §5]).

unfolded $a_1 \ldots a_n$ and *folded* $a_1 \ldots a_n$ expand and fold back again the given definitions throughout a rule.

rotated n rotate the premises of a theorem by n (default 1).

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

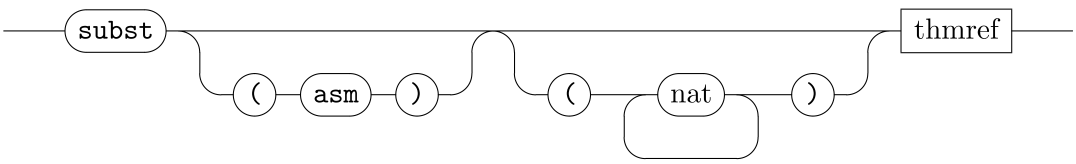
Note that the Classical Reasoner (§7.4) provides its own version of this operation.

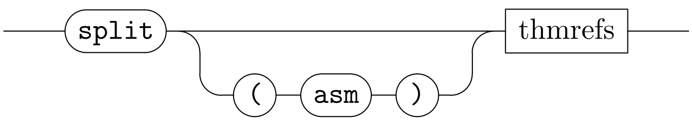
standard puts a theorem into the standard form of object-rules at the outermost theory level. Note that this operation violates the local proof context (including active locales).

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

7.2.2 Low-level equational reasoning

$$\begin{array}{rcl} subst & : & method \\ hypsubst & : & method \\ split & : & method \end{array}$$





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 §4.11), while the Simplifier methods provide the canonical way for automated normalization (see §7.3).

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

subst (*asm*) *eq* substitutes in an assumption.

subst (i ... j) *eq* performs several substitutions in the conclusion. The numbers i to j indicate the positions to substitute at. Positions are ordered from the top of the term tree moving down from left to right. For example, in $(a + b) + (c + d)$ there are three positions where commutativity of + is applicable: 1 refers to $a + b$, 2 to the whole term, and 3 to $c + d$.

If the positions in the list (i ... j) are non-overlapping (e.g. (2 3) in $(a + b) + (c + d)$) you may assume all substitutions are performed simultaneously. Otherwise the behaviour of *subst* is not specified.

subst (*asm*) (i ... j) *eq* performs the substitutions in the assumptions. The positions refer to the assumptions in order from left to right. For example, given in a goal of the form $P\ (a + b) \Longrightarrow P\ (c + d) \Longrightarrow \ldots$, position 1 of commutativity of + is the subterm $a + b$ and position 2 is the subterm $c + d$.

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 $a_1 \ldots a_n$ performs single-step case splitting using the given rules. 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.

7.2.3 Further tactic emulations

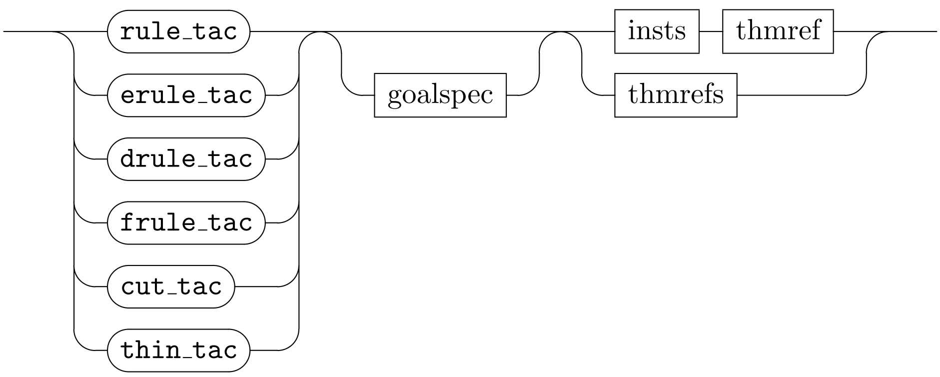
The following improper proof methods emulate traditional tactics. These admit direct access to the goal state, which is normally considered harmful! In particular, this may involve both numbered goal addressing (default 1), and dynamic instantiation within the scope of some subgoal.

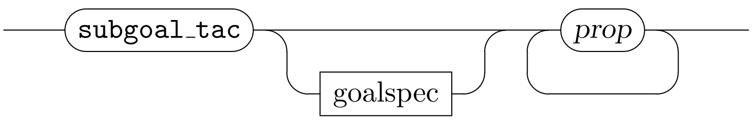
! Dynamic instantiations refer to universally quantified parameters of a subgoal (the dynamic context) rather than fixed variables and term abbreviations of a (static) Isar context.

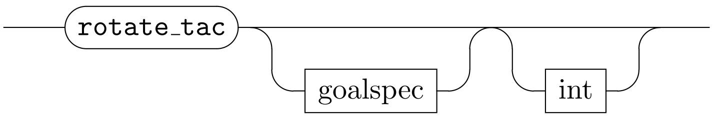
Tactic emulation methods, unlike their ML counterparts, admit simultaneous instantiation from both dynamic and static contexts. If names occur in both contexts goal parameters hide locally fixed variables. Likewise, schematic variables refer to term abbreviations, if present in the static context. Otherwise the schematic variable is interpreted as a schematic variable and left to be solved by unification with certain parts of the subgoal.

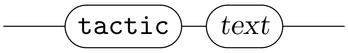
Note that the tactic emulation proof methods in Isabelle/Isar are consistently named *foo_tac*. Note also that variable names occurring on left hand sides of instantiations must be preceded by a question mark if they coincide with a keyword or contain dots. This is consistent with the attribute *where* (see §4.5).

*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*

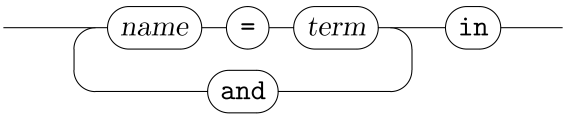








insts



rule_tac etc. do resolution of rules with explicit instantiation. This works the same way as the ML tactics `res_inst_tac` etc. (see [15, §3]).

Multiple rules may be only given if there is no instantiation; then *rule_tac* is the same as `resolve_tac` in ML (see [15, §3]).

cut_tac inserts facts into the proof state as assumption of a subgoal, see also `cut_facts_tac` in [15, §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 [15, §3].

thin_tac φ deletes the specified assumption from a subgoal; note that φ may contain schematic variables. See also `thin_tac` in [15, §3].

subgoal_tac φ adds φ as an assumption to a subgoal. See also `subgoal_tac` and `subgoals_tac` in [15, §3].

rename_tac $x_1 \dots x_n$ renames parameters of a goal according to the list x_1, ..., x_n, 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 [15, §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:

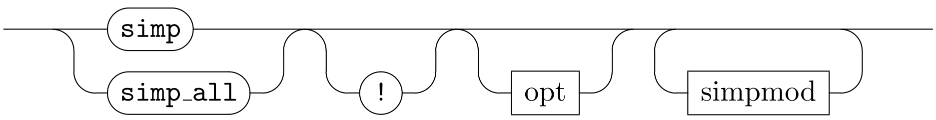
```
val ctxt  : Proof.context
val facts : thm list
val thm   : string -> thm
val thms  : string -> thm list
```

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.

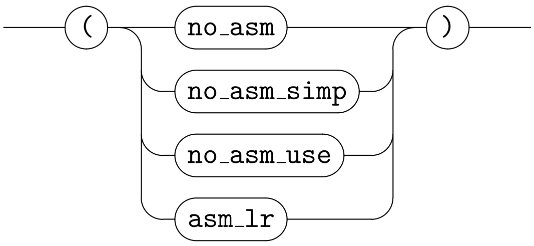
7.3 The Simplifier

7.3.1 Simplification methods

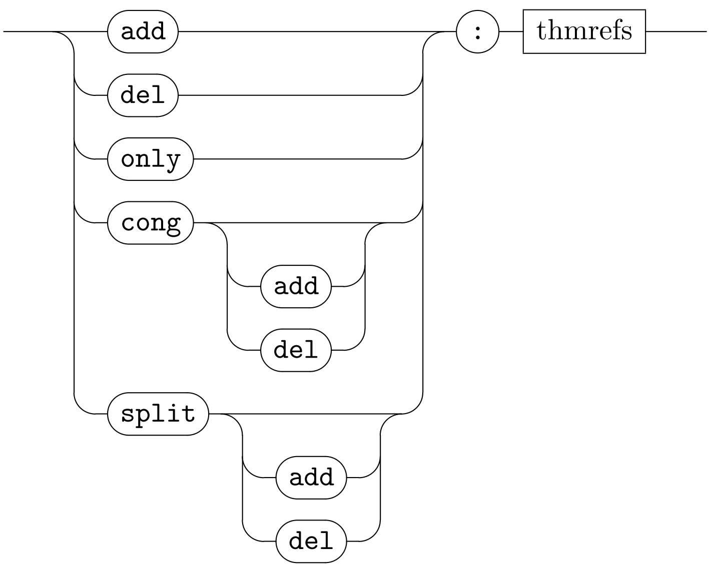
simp	:	*method*
simp_all	:	*method*



opt



simpmod



simp invokes the 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 [15]), the default is to add.

The `split` modifiers add or delete rules for the Splitter (see also [15]), 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 [15, §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 (via **then**, **from**, **using** etc.). The full context of premises is only included if the "!" (bang) argument is given, which should be used with some care, though.

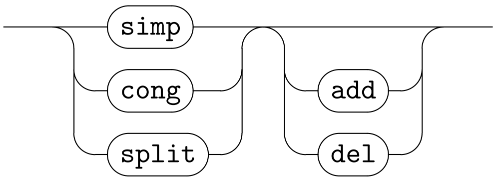
Additional Simplifier options may be specified to tune the behavior further (mostly for unstructured scripts with many accidental local facts): "(*no_asm*)" means assumptions are ignored completely (cf. `simp_tac`), "(*no_asm_simp*)" means assumptions are used in the simplification of the conclusion 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`). For compatibility reasons, there is also an option "(*asm_lr*)", which means that an assumption is only used for simplifying assumptions which are to the right of it (cf. `asm_lr_simp_tac`).

The configuration option *depth_limit* limits the number of recursive invocations of the simplifier during conditional rewriting.

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*: $a_1 \ldots a_n$)". There is also a separate *split* method available for single-step case splitting.

7.3.2 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 [15].

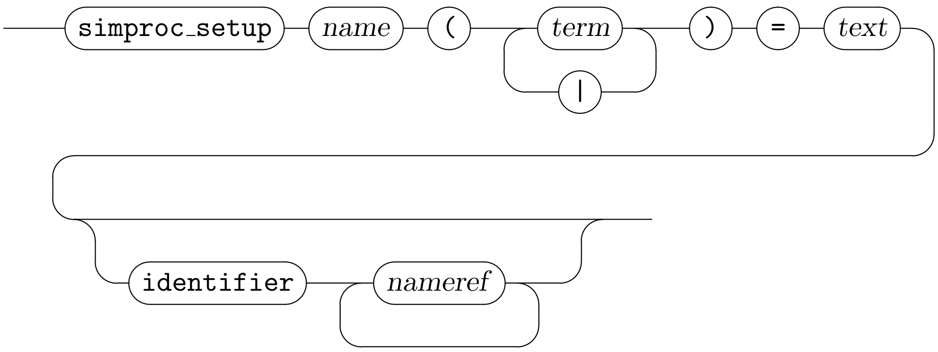
simp declares simplification rules.

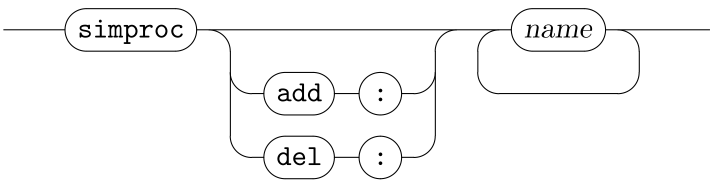
cong declares congruence rules.

split declares case split rules.

7.3.3 Simplification procedures

simproc_setup	:	*local-theory* → *local-theory*
simproc	:	*attribute*





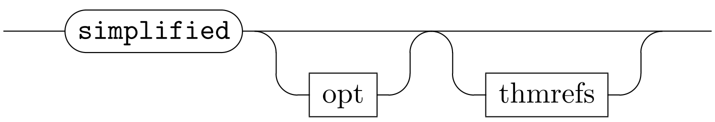
simproc_setup defines a named simplification procedure that is invoked by the Simplifier whenever any of the given term patterns match the current redex. The implementation, which is provided as ML source text, needs to be of type `"morphism -> simpset -> cterm -> thm option"`, where the `cterm` represents the current redex r and the result is supposed to be some proven rewrite rule $r \equiv r'$ (or a generalized version), or `NONE` to indicate failure. The `simpset` argument holds the full context of the current Simplifier invocation, including the actual Isar proof context. The `morphism` informs about the difference of the original compilation context wrt. the one of the actual application later on. The optional **identifier** specifies theorems that represent the logical content of the abstract theory of this simproc.

Morphisms and identifiers are only relevant for simprocs that are defined within a local target context, e.g. in a locale.

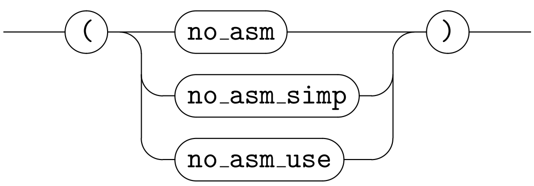
simproc add: *name* and *simproc del*: *name* add or delete named simprocs to the current Simplifier context. The default is to add a simproc. Note that **simproc_setup** already adds the new simproc to the subsequent context.

7.3.4 Forward simplification

simplified : *attribute*



opt



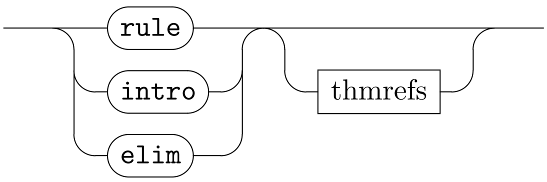
simplified $a_1 \dots a_n$ causes a theorem to be simplified, either by exactly the specified rules $a_1, \dots, a_n$, or the implicit Simplifier context if no arguments are given. The result is fully simplified by default, including assumptions and conclusion; the options *no_asm* etc. tune the Simplifier 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 looper tactics [15] are *not* involved here. The *simplified* attribute should be only rarely required under normal circumstances.

7.4 The Classical Reasoner

7.4.1 Basic methods

$$\begin{array}{rcl} \mathit{rule} & : & \mathit{method} \\ \mathit{contradiction} & : & \mathit{method} \\ \mathit{intro} & : & \mathit{method} \\ \mathit{elim} & : & \mathit{method} \end{array}$$



rule as offered by the Classical Reasoner is a refinement over the primitive one (see §4.5). Both versions essentially work the same, but the classical version observes the classical rule context in addition to that of Isabelle/Pure.

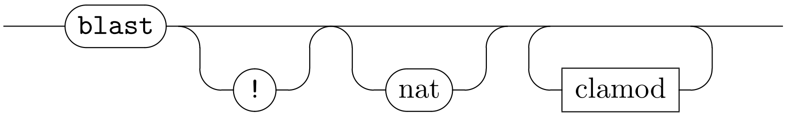
Common object logics (HOL, ZF, etc.) declare a rich collection of classical rules (even if these would qualify as intuitionistic ones), but only few declarations to the rule context of Isabelle/Pure (§4.5).

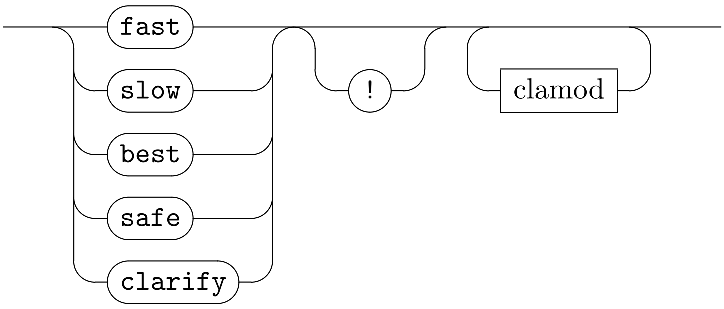
contradiction solves some goal by contradiction, deriving any result from both $\neg A$ and A. Chained facts, which are guaranteed to participate, may appear in either order.

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.

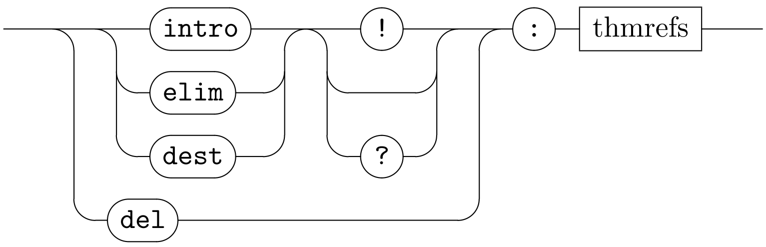
7.4.2 Automated methods

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





clamod



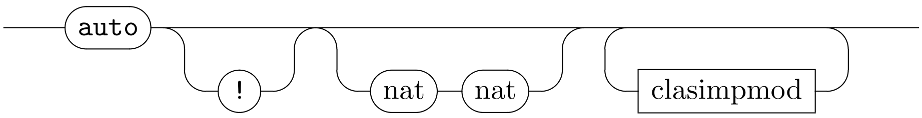
blast refers to the classical tableau prover (see `blast_tac` in [15, §11]). The optional argument specifies a user-supplied search bound (default 20).

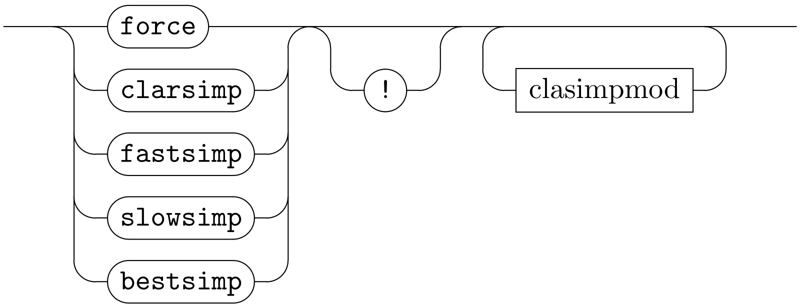
fast, *slow*, *best*, *safe*, and *clarify* refer to the generic classical reasoner. See `fast_tac`, `slow_tac`, `best_tac`, `safe_tac`, and `clarify_tac` in [15, §11] for more information.

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.

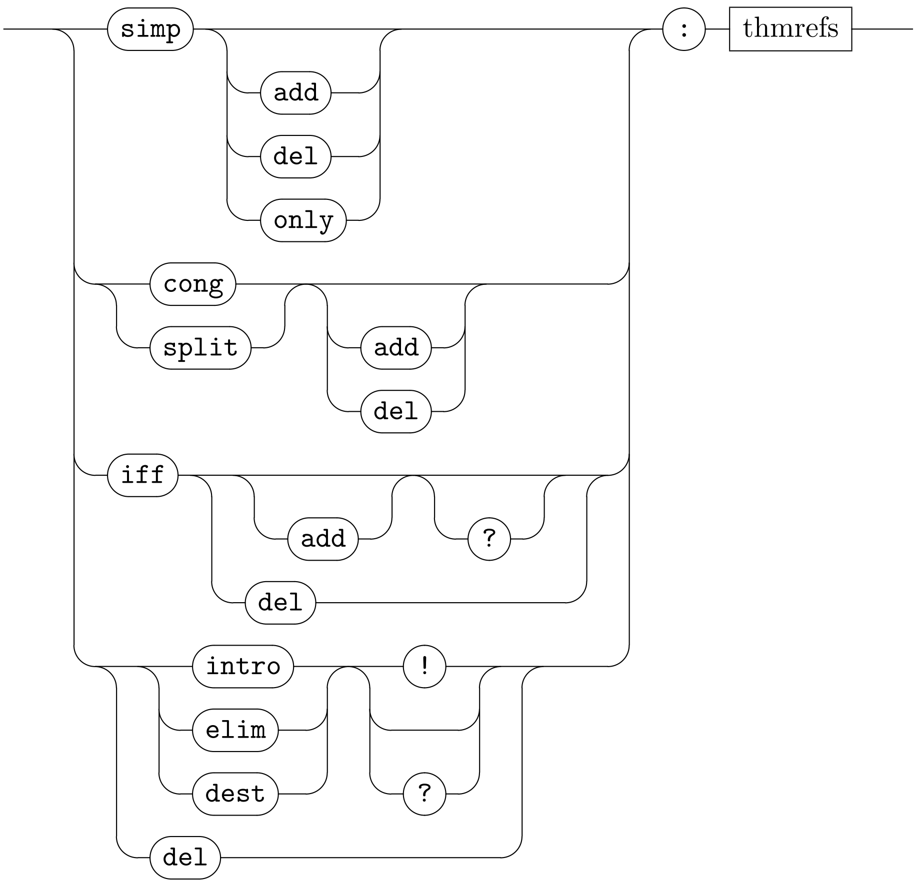
7.4.3 Combined automated methods

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





clasimpmod

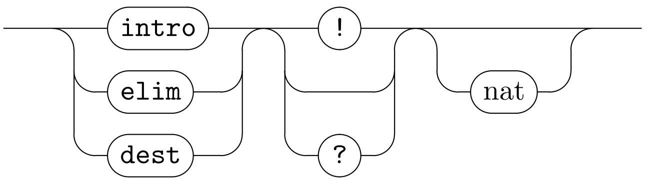


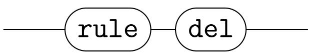
auto, *force*, *clarsimp*, *fastsimp*, *slowsimp*, and *bestsimp* provide access to Isabelle's combined simplification and classical reasoning tactics. These correspond to `auto_tac`, `force_tac`, `clarsimp_tac`, and Classical Reasoner tactics with the Simplifier added as wrapper, see [15, §11] for more information. The modifier arguments correspond to those given in §7.3 and §7.4. Just note that the ones related to the Simplifier are prefixed by `simp` here.

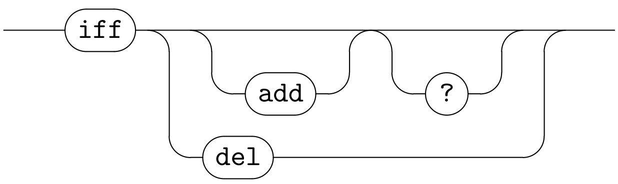
Facts provided by forward chaining are inserted into the goal before doing the search. The "!" argument causes the full context of assumptions to be included as well.

7.4.4 Declaring rules

print_claset*	:	*theory* | *proof* → *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 "claset" internally [15].

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 "!" classifies as *safe*. Rule declarations marked by "*?*" coincide with those of Isabelle/Pure, cf. §4.5 (i.e. are only applied in single steps of the *rule* method). The optional natural number specifies an explicit weight argument, which is ignored by automated tools, but determines the search order of single rule steps.

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 ¬-elimination internally).

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

7.4.5 Classical operations

swapped : *attribute*

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

7.5 Object-logic setup

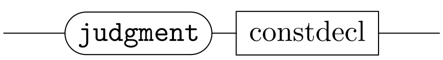
judgment	:	*theory* → *theory*
atomize	:	*method*
atomize	:	*attribute*
rule_format	:	*attribute*
rulify	:	*attribute*

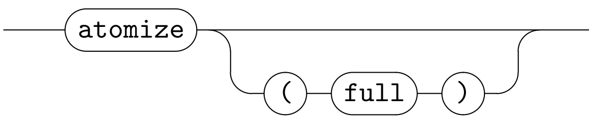
The very starting point for any Isabelle object-logic is a "truth judgment" that links object-level statements to the meta-logic (with its minimal language of *prop* that covers universal quantification $\bigwedge$ and implication $\Longrightarrow$).

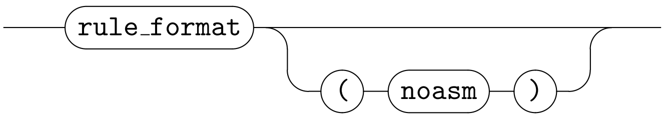
Common object-logics are sufficiently expressive to internalize rule statements over $\bigwedge$ and $\Longrightarrow$ within their own language. This is useful in certain situations where a rule needs to be viewed as an atomic statement from the meta-level perspective, e.g. $\bigwedge x.\ x \in A \Longrightarrow P\ x$ 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). 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 $\bigwedge$, $\Longrightarrow$, and $\equiv$. Meta-level conjunction should be covered as well (this is particularly important for locales, see §3.5).

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.

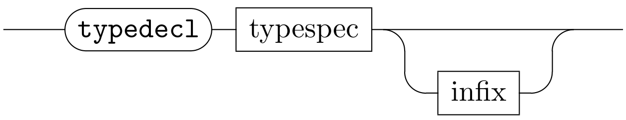
In common object-logics (HOL, FOL, ZF), the effect of *rule_format* is to replace (bounded) universal quantification ($\forall$) and implication ($\longrightarrow$) by the corresponding rule statements over $\bigwedge$ and $\Longrightarrow$.

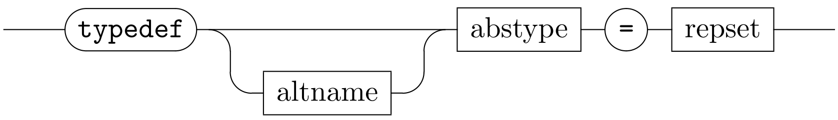
Chapter 8

Isabelle/HOL

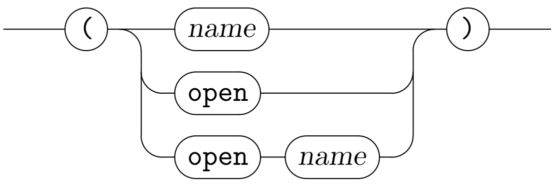
8.1 Primitive types

typedecl	:	*theory* → *theory*
typedef	:	*theory* → *proof*(*prove*)

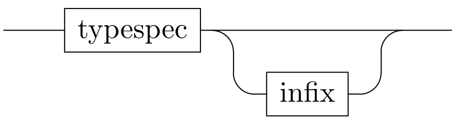




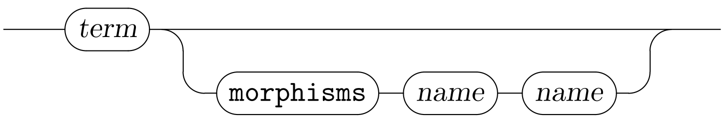
altname



abstype



repset



typedecl $(\alpha_1, \ldots, \alpha_n)\ t$ is similar to the original **typedecl** of Isabelle/Pure (see §3.9.2), but also declares type arity t :: (*type*, …, *type*) *type*, making t an actual HOL type constructor.

typedef $(\alpha_1, \ldots, \alpha_n)\ t = A$ 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).

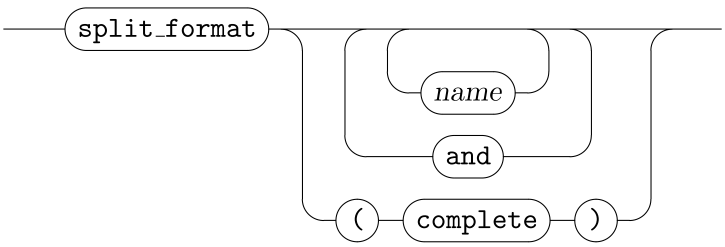
Theorems *Rep_t*, *Rep_t_inverse*, and *Abs_t_inverse* provide the most basic characterization as a corresponding injection/surjection pair (in both directions). Rules *Rep_t_inject* and *Abs_t_inject* provide a slightly more convenient view on the injectivity part, suitable for automated proof tools (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.

An alternative name may be specified in parentheses; the default is to use t as indicated before. The "(*open*)" declaration suppresses a separate constant definition for the representing set.

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 §8.3) or **datatype** (see §8.4).

8.2 Adhoc tuples

*split_format** : *attribute*



split_format $p_1 \ldots p_m$ **and** … **and** $q_1 \ldots q_n$ puts expressions of low-level tuple types into canonical form as specified by the arguments given;

the i-th collection of arguments 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.

8.3 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 [11] for more details on object-oriented verification and record subtyping in HOL.

8.3.1 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	$(\!\vert x = a,\ y = b \vert\!)$	$(\!\vert x :: A,\ y :: B \vert\!)$
schematic	$(\!\vert x = a,\ y = b,\ \ldots = m \vert\!)$	$(\!\vert x :: A,\ y :: B,\ \ldots :: M \vert\!)$

The ASCII representation of $(\!\vert x = a \vert\!)$ is `(| x = a |)`.

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

A record scheme like $(\!\vert x = a,\ y = b,\ \ldots = m \vert\!)$ 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 $(\!\vert x = a,\ y = b,\ z = c,\ \ldots = m' \vert\!)$, where m' refers to a different more part. Fixed records are special instances of record schemes, where "…" is properly terminated by the $()$ :: *unit* element. In fact, $(\!| x = a,\ y = b |\!)$ is just an abbreviation for $(\!| x = a,\ y = b,\ \ldots = () |\!)$.

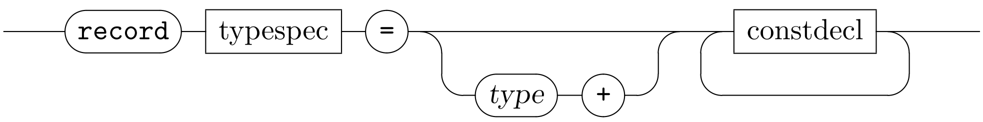
Two key observations make extensible records in a simply typed language like HOL work out:

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 [13] for further instructions on using records in practice.

8.3.2 Record specifications

record : *theory* → *theory*



record $(\alpha_1, \ldots, \alpha_m)\ t = \tau + c_1 :: \sigma_1 \ \ldots \ c_n :: \sigma_n$ defines extensible record type $(\alpha_1, \ldots, \alpha_m)\ t$, derived from the optional parent record τ by adding new field components $c_i :: \sigma_i$ etc.

The type variables of τ and σ_i need to be covered by the (distinct) parameters $\alpha_1, \ldots, \alpha_m$. Type constructor t has to be new, while τ needs to specify an instance of an existing record type. At least one new field c_i 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, $(\alpha_1, \ldots, \alpha_m)\ t$ is made a type abbreviation for the fixed record type $(\!| c_1 :: \sigma_1, \ldots, c_n :: \sigma_n |\!)$, likewise is $(\alpha_1, \ldots, \alpha_m, \zeta)$ *t_scheme* made an abbreviation for $(\!| c_1 :: \sigma_1, \ldots, c_n :: \sigma_n, \ldots :: \zeta |\!)$.

8.3.3 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 $(\alpha_1, \ldots, \alpha_m)\ t$ is a root record with fields $c_1 :: \sigma_1, \ldots, c_n :: \sigma_n$.

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

$$\begin{array}{lll} c_i & :: & (\!| \overline{c} :: \overline{\sigma}, \ldots :: \zeta |\!) \Rightarrow \sigma_i \\ c_i\mathit{\_update} & :: & \sigma_i \Rightarrow (\!| \overline{c} :: \overline{\sigma}, \ldots :: \zeta |\!) \Rightarrow (\!| \overline{c} :: \overline{\sigma}, \ldots :: \zeta |\!) \end{array}$$

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:

$$t.\mathit{make}\ ::\ \sigma_1 \Rightarrow \ldots\ \sigma_n \Rightarrow (\!| \overline{c} :: \overline{\sigma} |\!)$$

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 $(\alpha_1, \ldots, \alpha_m)\ t$ has ancestor fields $b_1 :: \varrho_1, \ldots, b_k :: \varrho_k$, the above record operations will get the following types:

$$\begin{array}{lll} c_i & :: & (\!| \overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}, \ldots :: \zeta |\!) \Rightarrow \sigma_i \\ c_i\mathit{\_update} & :: & \sigma_i \Rightarrow (\!| \overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}, \ldots :: \zeta |\!) \Rightarrow (\!| \overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}, \ldots :: \zeta |\!) \\ t.\mathit{make} & :: & \varrho_1 \Rightarrow \ldots\ \varrho_k \Rightarrow \sigma_1 \Rightarrow \ldots\ \sigma_n \Rightarrow (\!| \overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma} |\!) \end{array}$$

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

$$
\begin{array}{lll}
t.\mathit{fields} & :: & \sigma_1 \Rightarrow \ldots \sigma_n \Rightarrow (\!|\overline{c} :: \overline{\sigma}|\!) \\
t.\mathit{extend} & :: & (\!|\overline{b} :: \overline{\varrho},\ \overline{c} :: \overline{\sigma}|\!) \Rightarrow \zeta \Rightarrow (\!|\overline{b} :: \overline{\varrho},\ \overline{c} :: \overline{\sigma},\ \ldots :: \zeta|\!) \\
t.\mathit{truncate} & :: & (\!|\overline{b} :: \overline{\varrho},\ \overline{c} :: \overline{\sigma},\ \ldots :: \zeta|\!) \Rightarrow (\!|\overline{b} :: \overline{\varrho},\ \overline{c} :: \overline{\sigma}|\!)
\end{array}
$$

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

8.3.4 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 $x\ r = x\ r' \Longrightarrow y\ r = y\ r' \ldots \Longrightarrow r = r'$ is declared to the Simplifier, and as the basic rule context as "*intro?*". The rule is called *t.equality*.

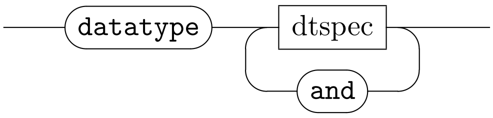
5. Representations of arbitrary record expressions as canonical constructor terms are provided both in *cases* and *induct* format (cf. the generic proof methods of the same name, §4.12). Several variations are available, for fixed records, record schemes, more parts etc.

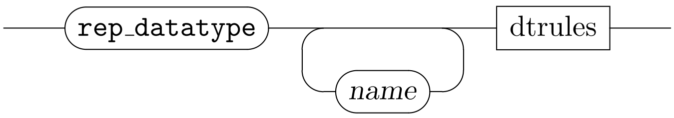
   The generic proof methods are sufficiently smart to pick the most sensible rule according to the type of the indicated record expression: users just need to apply something like "(*cases* r)" to a certain proof problem.

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*.

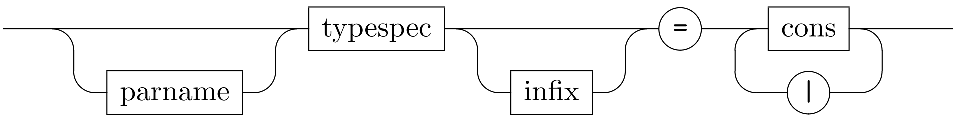
8.4 Datatypes

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

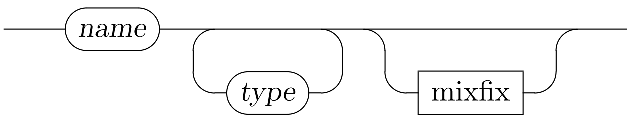




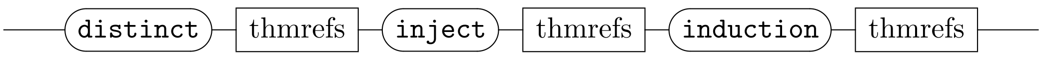
dtspec



cons



dtrules



datatype defines inductive datatypes in HOL.

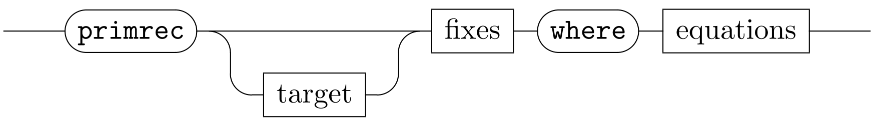
rep_datatype represents existing types as inductive ones, generating the standard infrastructure of derived concepts (primitive recursion etc.).

The induction and exhaustion theorems generated provide case names according to the constructors involved, while parameters are named after the types (see also §4.12).

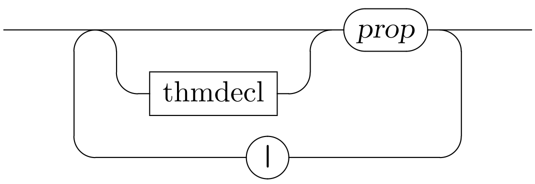
See [12] for more details on datatypes, but beware of the old-style theory syntax being used there! Apart from proper proof methods for case-analysis and induction, there are also emulations of ML tactics *case_tac* and *induct_tac* available, see §8.8; these admit to refer directly to the internal structure of subgoals (including internally bound parameters).

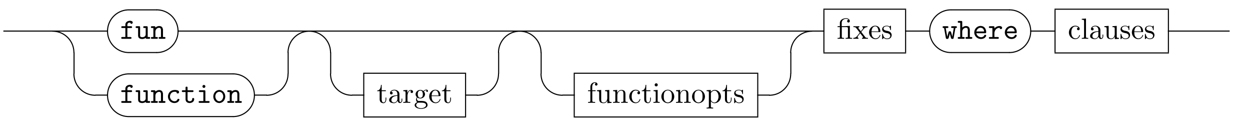
8.5 Recursive functions

primrec	:	*local-theory* → *local-theory*
fun	:	*local-theory* → *local-theory*
function	:	*local-theory* → *proof*(*prove*)
termination	:	*local-theory* → *proof*(*prove*)

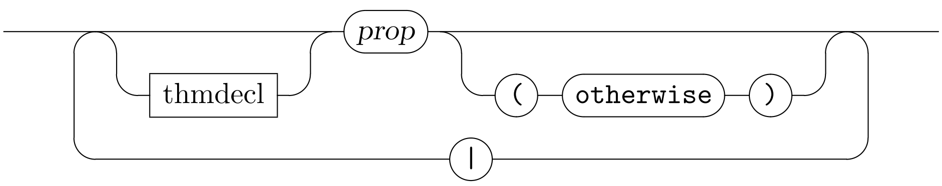


equations

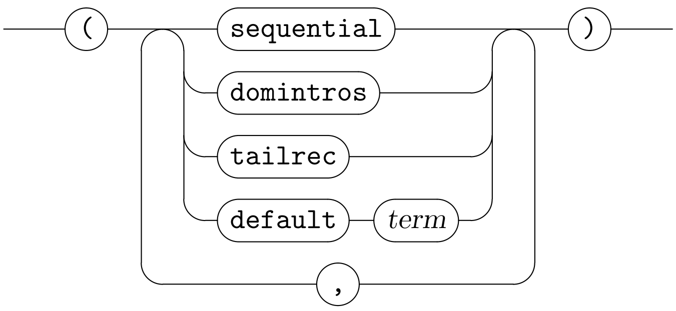


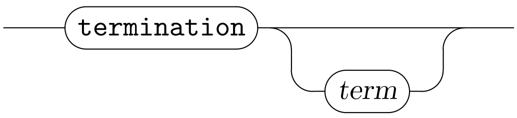


clauses



functionopts





primrec defines primitive recursive functions over datatypes, see also [12].

function defines functions by general wellfounded recursion. A detailed description with examples can be found in [7]. The function is specified by a set of (possibly conditional) recursive equations with arbitrary pattern matching. The command generates proof obligations for the completeness and the compatibility of patterns.

The defined function is considered partial, and the resulting simplification rules (named *f.psimps*) and induction rule (named *f.pinduct*) are guarded by a generated domain predicate *f_dom*. The **termination** command can then be used to establish that the function is total.

fun is a shorthand notation for "**function** (*sequential*), followed by automated proof attempts regarding pattern matching and termination. See [7] for further details.

termination *f* commences a termination proof for the previously defined function *f*. If this is omitted, the command refers to the most recent function definition. After the proof is closed, the recursive equations and the induction principle is established.

Recursive definitions introduced by both the **primrec** and the **function** command accommodate reasoning by induction (cf. §4.12): 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 **function** 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.

The **function** command accepts the following options.

sequential enables a preprocessor which disambiguates overlapping patterns by making them mutually disjoint. Earlier equations take precedence over later ones. This allows to give the specification in a format very similar to functional programming. Note that the resulting simplification and induction rules correspond to the transformed specification, not the one given originally. This usually means that each equation given by the user may result in several theroems. Also note that this automatic transformation only works for ML-style datatype patterns.

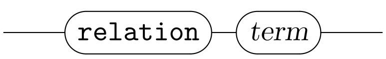
domintros enables the automated generation of introduction rules for the domain predicate. While mostly not needed, they can be helpful in some proofs about partial functions.

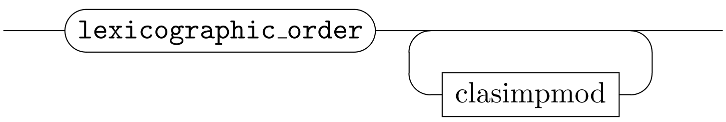
tailrec generates the unconstrained recursive equations even without a termination proof, provided that the function is tail-recursive. This currently only works

default d allows to specify a default value for a (partial) function, which will ensure that $f\ x = d\ x$ whenever $x \notin f\_dom$.

8.5.1 Proof methods related to recursive definitions

$$\begin{array}{rcl} pat\_completeness & : & method \\ relation & : & method \\ lexicographic\_order & : & method \end{array}$$





pat_completeness is a specialized method to solve goals regarding the completeness of pattern matching, as required by the **function** package (cf. [7]).

relation R introduces a termination proof using the relation R. The resulting proof state will contain goals expressing that R is wellfounded, and that the arguments of recursive calls decrease with respect to R. Usually, this method is used as the initial proof step of manual termination proofs.

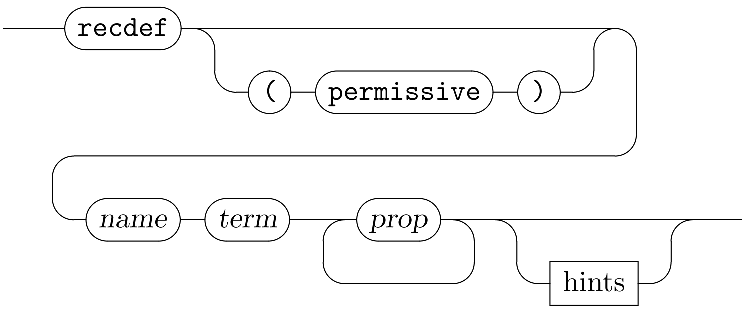
lexicographic_order attempts a fully automated termination proof by searching for a lexicographic combination of size measures on the arguments of the function. The method accepts the same arguments as the *auto* method, which it uses internally to prove local descents. The same context modifiers as for *auto* are accepted, see §7.4.3.

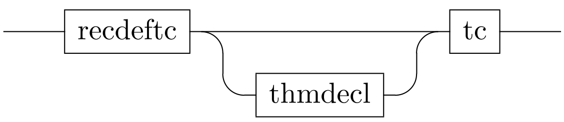
In case of failure, extensive information is printed, which can help to analyse the situation (cf. [7]).

8.5.2 Old-style recursive function definitions (TFL)

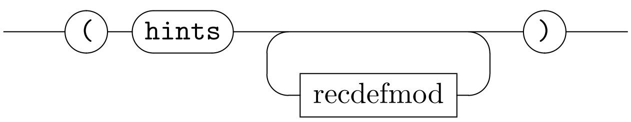
The old TFL commands **recdef** and **recdef_tc** for defining recursive are mostly obsolete; **function** or **fun** should be used instead.

$$\begin{array}{rcl} \textbf{recdef} & : & \mathit{theory} \to \mathit{theory} \\ \textbf{recdef\_tc}^{*} & : & \mathit{theory} \to \mathit{proof(prove)} \end{array}$$

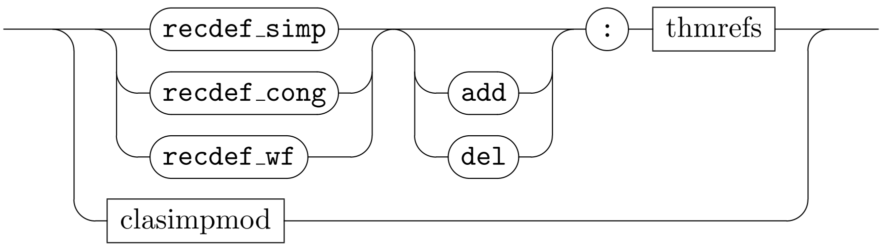




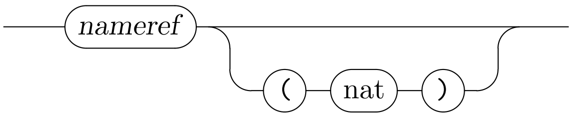
hints



recdefmod



tc



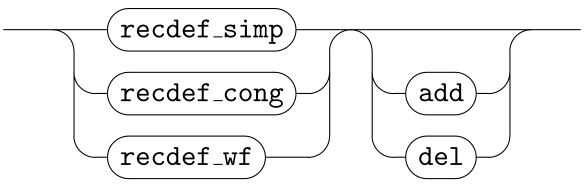
recdef defines general well-founded recursive functions (using the TFL package), see also [12]. The "(*permissive*)" option tells TFL to recover from failed proof attempts, returning unfinished results. The *recdef_simp*, *recdef_cong*, and *recdef_wf* hints refer to auxiliary rules to be used in the internal automated proof process of TFL. Additional *clasimpmod* declarations (cf. §7.4.3) may be given to tune the context of the Simplifier (cf. §7.3) and Classical reasoner (cf. §7.4).

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.

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

recdef_simp	:	*attribute*
recdef_cong	:	*attribute*
recdef_wf	:	*attribute*



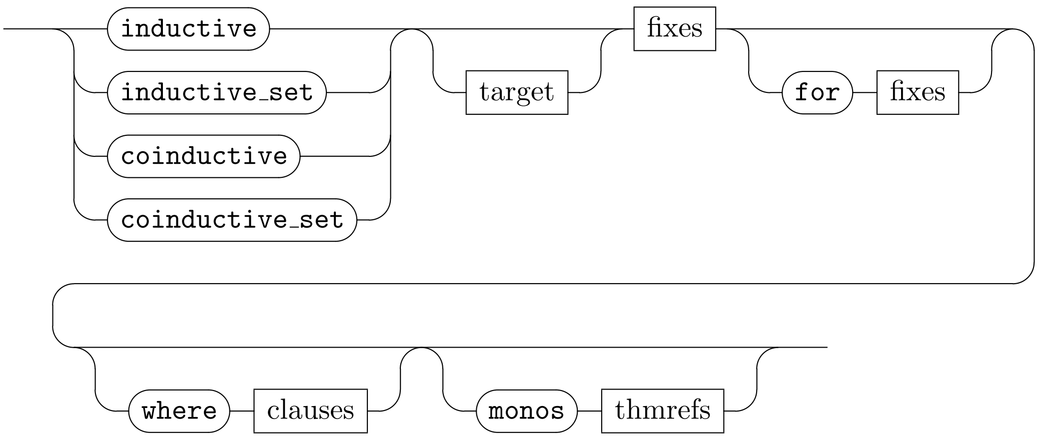
8.6 Inductive and coinductive definitions

An **inductive definition** specifies the least predicate (or set) R closed under given rules: applying a rule to elements of R yields a result within R. For example, a structural operational semantics is an inductive definition of an evaluation relation.

Dually, a **coinductive definition** specifies the greatest predicate / set R that is consistent with given rules: every element of R can be seen as arising by applying a rule to elements of R. An important example is using bisimulation relations to formalise equivalence of processes and infinite data structures.

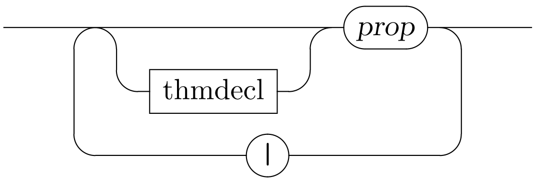
The HOL package is related to the ZF one, which is described in a separate paper,[1] which you should refer to in case of difficulties. The package is simpler than that of ZF thanks to implicit type-checking in HOL. The types of the (co)inductive predicates (or sets) determine the domain of the fixedpoint definition, and the package does not have to use inference rules for type-checking.

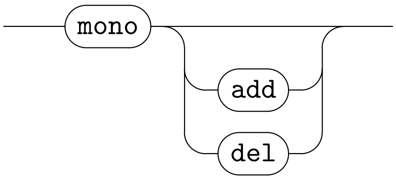
inductive	:	*local-theory* → *local-theory*
inductive_set	:	*local-theory* → *local-theory*
coinductive	:	*local-theory* → *local-theory*
coinductive_set	:	*local-theory* → *local-theory*
mono	:	*attribute*



[1]It appeared in CADE [17]; a longer version is distributed with Isabelle.

clauses





inductive and **coinductive** define (co)inductive predicates from the introduction rules given in the **where** part. The optional **for** part contains a list of parameters of the (co)inductive predicates that remain fixed throughout the definition. The optional **monos** section contains *monotonicity theorems*, which are required for each operator applied to a recursive set in the introduction rules. There *must* be a theorem of the form $A \leq B \Longrightarrow M\ A \leq M\ B$, for each premise $M\ R_i\ t$ in an introduction rule!

inductive_set and **coinductive_set** are wrappers for to the previous commands, allowing the definition of (co)inductive sets.

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

8.6.1 Derived rules

Each (co)inductive definition R adds definitions to the theory and also proves some theorems:

R.intros is the list of introduction rules as proven theorems, for the recursive predicates (or sets). The rules are also available individually, using the names given them in the theory file;

R.cases is the case analysis (or elimination) rule;

R.induct **or** *R.coinduct* is the (co)induction rule.

When several predicates $R_1, \ldots, R_n$ are defined simultaneously, the list of introduction rules is called $R_1\_\ldots\_R_n.intros$, the case analysis rules are called $R_1.cases, \ldots, R_n.cases$, and the list of mutual induction rules is called $R_1\_\ldots\_R_n.inducts$.

8.6.2 Monotonicity theorems

Each theory contains a default set of theorems that are used in monotonicity proofs. New rules can be added to this set via the *mono* attribute. The HOL theory *Inductive* shows how this is done. In general, the following monotonicity theorems may be added:

- Theorems of the form $A \leq B \Longrightarrow M\ A \leq M\ B$, for proving monotonicity of inductive definitions whose introduction rules have premises involving terms such as $M\ R_i\ t$.
- Monotonicity theorems for logical operators, which are of the general form $(\ldots \longrightarrow \ldots) \Longrightarrow \ldots (\ldots \longrightarrow \ldots) \Longrightarrow \ldots \longrightarrow \ldots$. For example, in the case of the operator $\vee$, the corresponding theorem is

$$\frac{P_1 \longrightarrow Q_1 \quad P_2 \longrightarrow Q_2}{P_1 \vee P_2 \longrightarrow Q_1 \vee Q_2}$$

- De Morgan style equations for reasoning about the "polarity" of expressions, e.g.

$$\neg\,\neg\,P \longleftrightarrow P \qquad\qquad \neg\,(P \wedge Q) \longleftrightarrow \neg\,P \vee \neg\,Q$$

- Equations for reducing complex operators to more primitive ones whose monotonicity can easily be proved, e.g.

$$(P \longrightarrow Q) \longleftrightarrow \neg\,P \vee Q \qquad\qquad Ball\ A\ P \equiv \forall x.\ x \in A \longrightarrow P\ x$$

8.7 Arithmetic proof support

$$\begin{array}{rcl} arith & : & method \\ arith\_split & : & attribute \end{array}$$

The *arith* method decides linear arithmetic problems (on types *nat*, *int*, *real*). Any current facts are inserted into the goal before running the procedure.

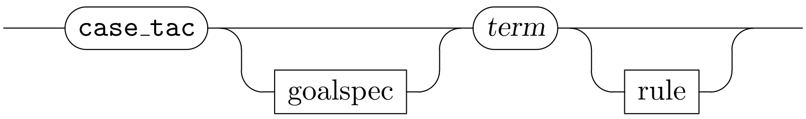
The *arith_split* attribute declares case split rules to be expanded before the arithmetic procedure is invoked.

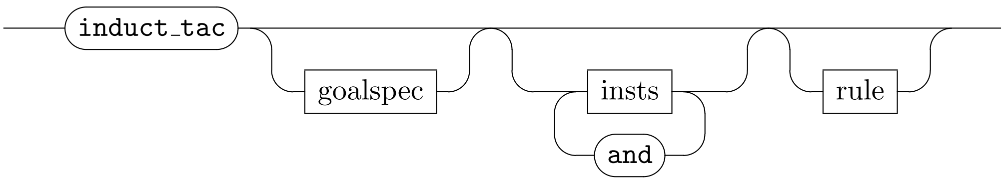
Note that a simpler (but faster) version of arithmetic reasoning is already performed by the Simplifier.

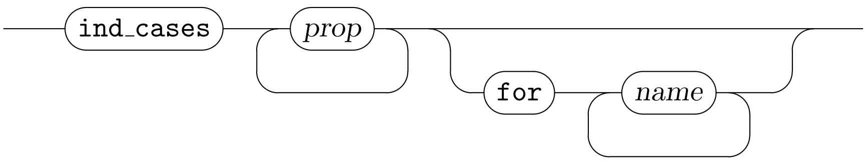
8.8 Cases and induction: emulating tactic scripts

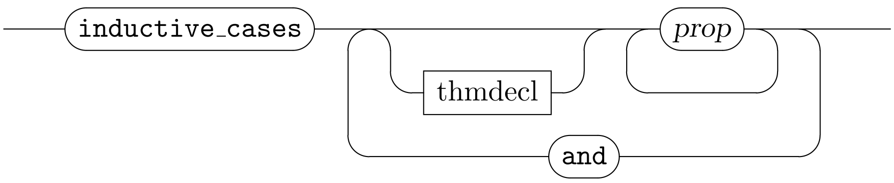
The following important tactical tools of Isabelle/HOL have been ported to Isar. These should be never used in proper proof texts!

*case_tac**	:	*method*
*induct_tac**	:	*method*
*ind_cases**	:	*method*
inductive_cases	:	*theory* → *theory*

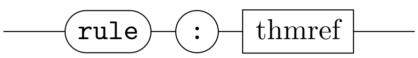








rule



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 §4.12).

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

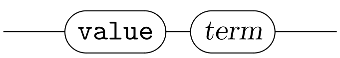
While *ind_cases* is a proof method to apply the result immediately as elimination rules, **inductive_cases** provides case split theorems at the theory level for later use. The **for** argument of the *ind_cases* method allows to specify a list of variables that should be generalized before applying the resulting rule.

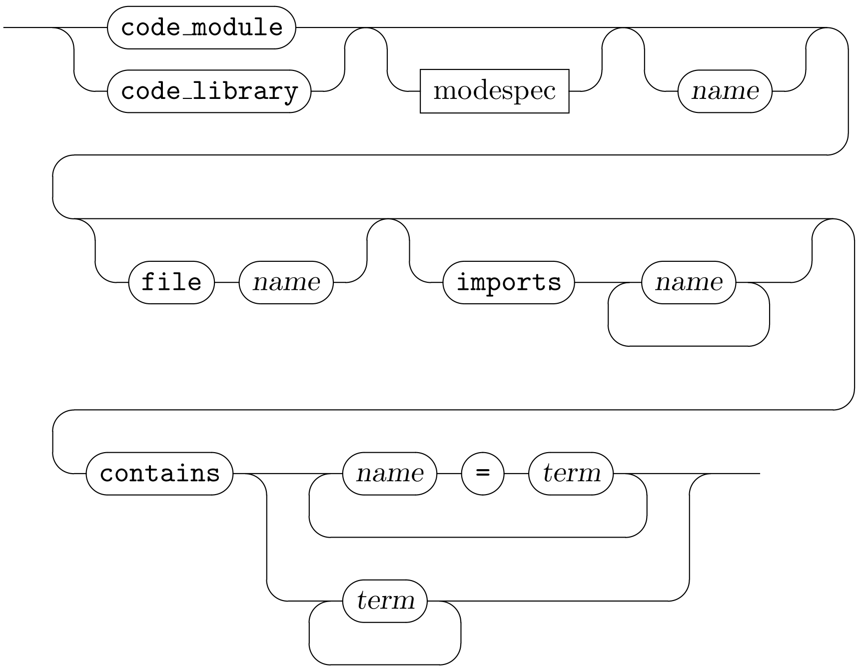
8.9 Executable code

Isabelle/Pure provides two generic frameworks to support code generation from executable specifications. Isabelle/HOL instantiates these mechanisms in a way that is amenable to end-user applications.

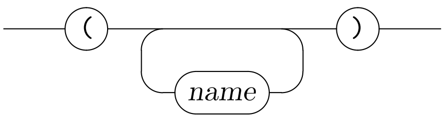
One framework generates code from both functional and relational programs to SML. See [12] for further information (this actually covers the new-style theory format as well).

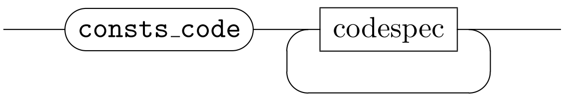
value*	:	*theory* | *proof* → *theory* | *proof*
code_module	:	*theory* → *theory*
code_library	:	*theory* → *theory*
consts_code	:	*theory* → *theory*
types_code	:	*theory* → *theory*
code	:	*attribute*



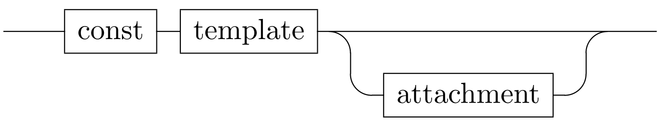


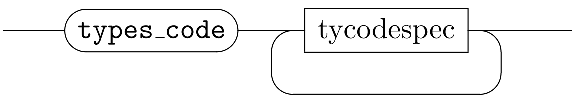
modespec



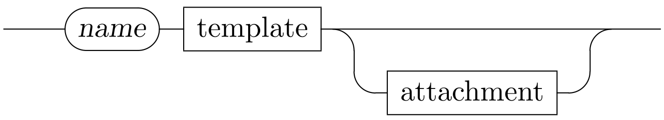


codespec

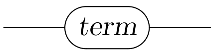




tycodespec

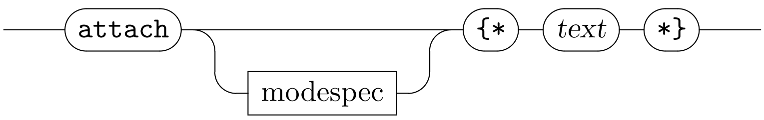


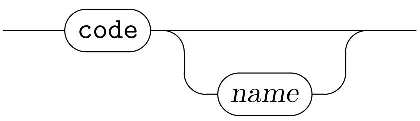
const



template

attachment

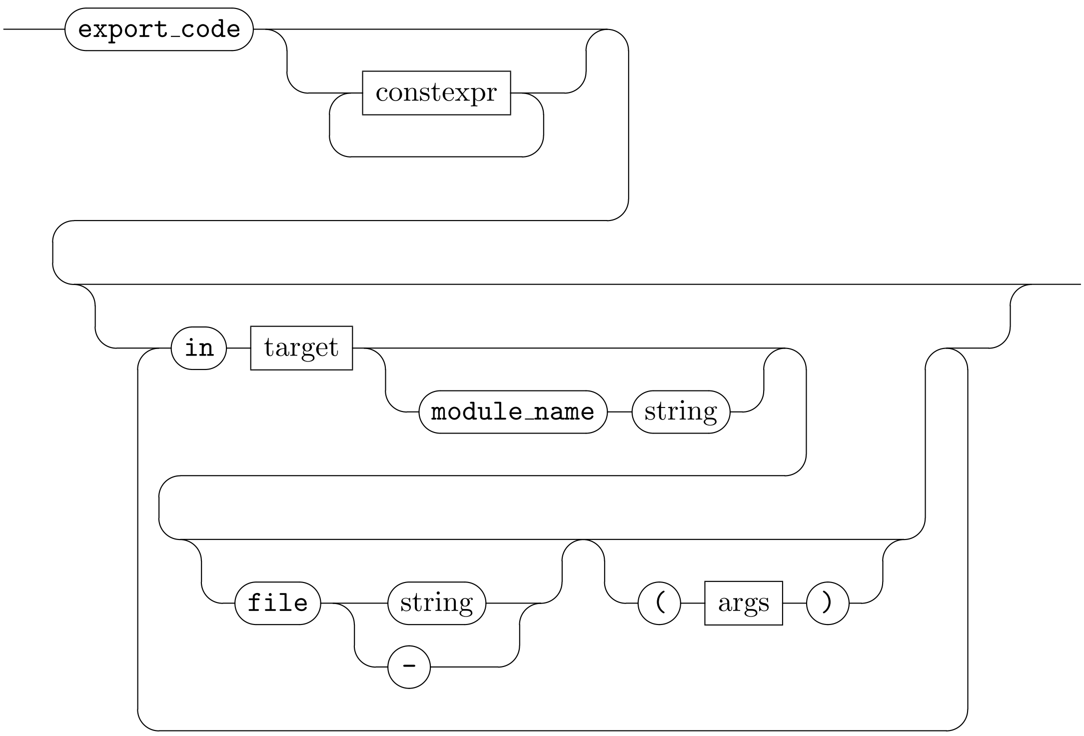


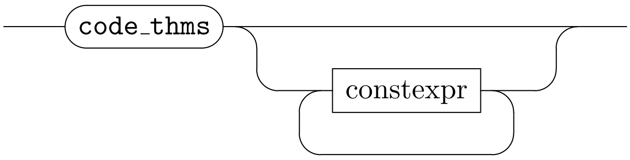


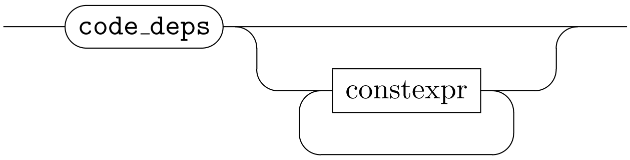
value t evaluates and prints a term using the code generator.

The other framework generates code from functional programs (including overloading using type classes) to SML [9], OCaml [8] and Haskell [18]. Conceptually, code generation is split up in three steps: *selection* of code theorems, *translation* into an abstract executable view and *serialization* to a specific *target language*. See [5] for an introduction on how to use it.

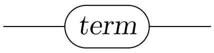
export_code*	:	*theory* | *proof* → *theory* | *proof*
code_thms*	:	*theory* | *proof* → *theory* | *proof*
code_deps*	:	*theory* | *proof* → *theory* | *proof*
code_datatype	:	*theory* → *theory*
code_const	:	*theory* → *theory*
code_type	:	*theory* → *theory*
code_class	:	*theory* → *theory*
code_instance	:	*theory* → *theory*
code_monad	:	*theory* → *theory*
code_reserved	:	*theory* → *theory*
code_include	:	*theory* → *theory*
code_modulename	:	*theory* → *theory*
code_exception	:	*theory* → *theory*
print_codesetup*	:	*theory* | *proof* → *theory* | *proof*
code	:	*attribute*



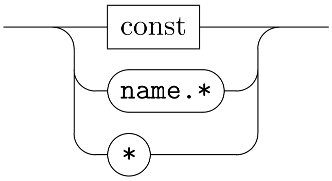




const



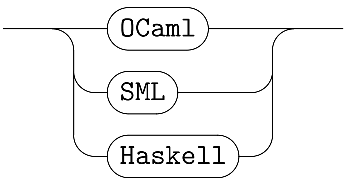
constexpr

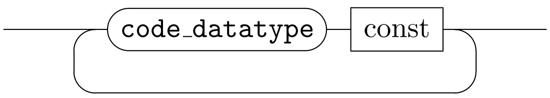


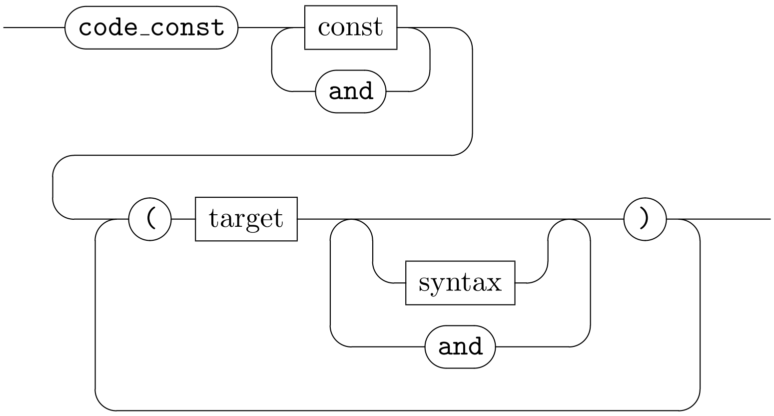
typeconstructor

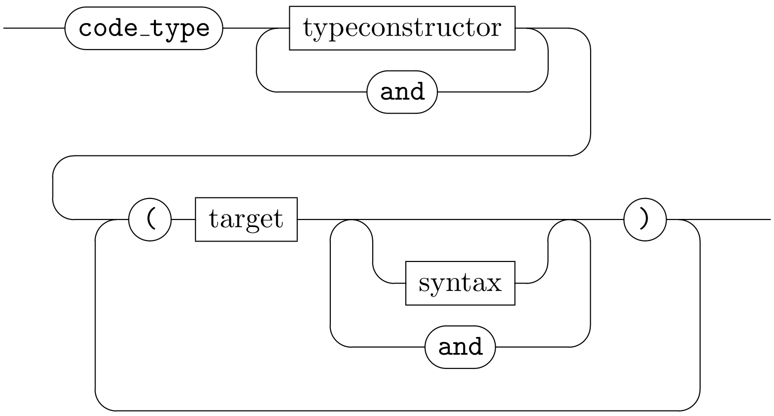
class

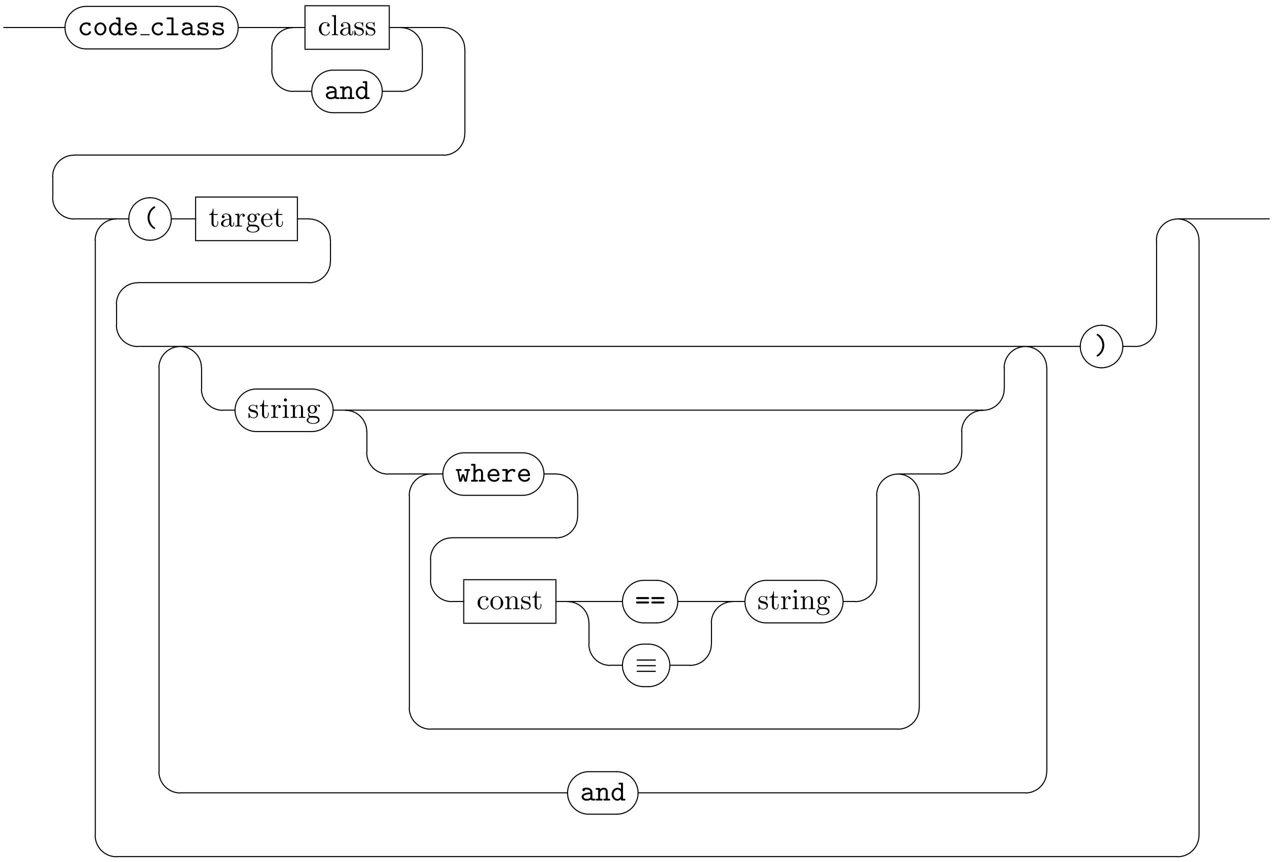
target

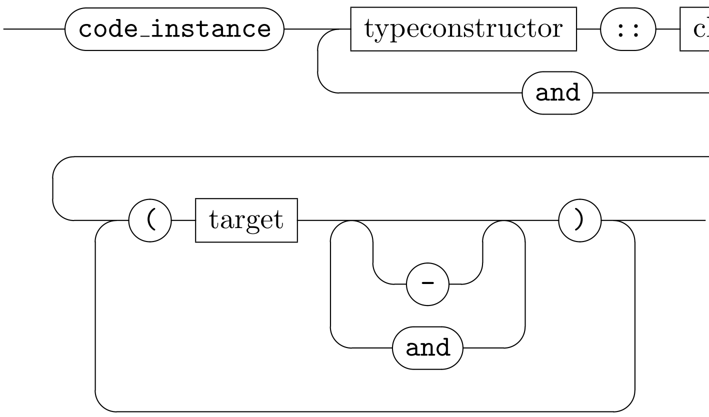


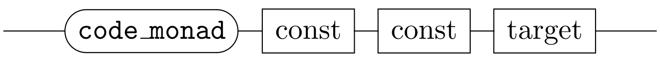


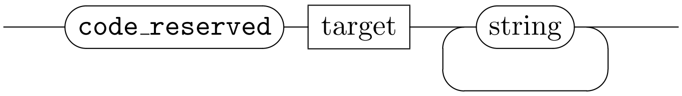

code_const
const
and
(
target
syntax
and
)

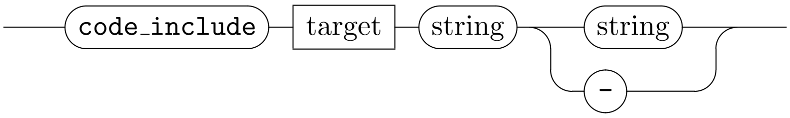

code_type
typeconstructor
and
(
target
syntax
and
)

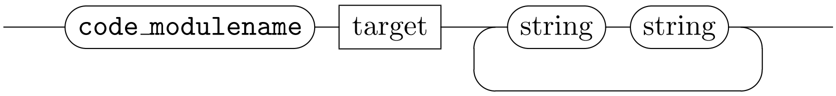

code_class
class
and
(
target
)
string
where
const
==
≡
string
and

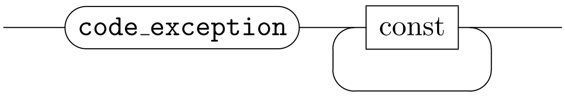

code_instance
typeconstructor
::
class
and
(
target
)
-
and


code_monad
const
const
target

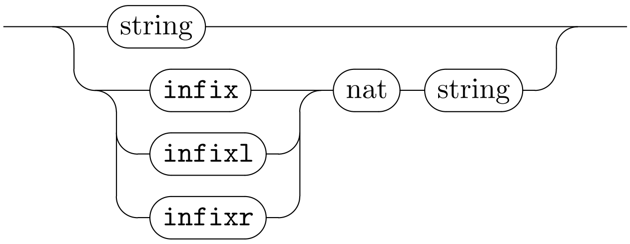


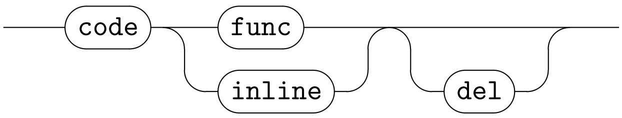






syntax





export_code is the canonical interface for generating and serializing code: for a given list of constants, code is generated for the specified target languages. Abstract code is cached incrementally. If no constant is given, the currently cached code is serialized. If no serialization instruction is given, only abstract code is cached.

Constants may be specified by giving them literally, referring to all executable contants within a certain theory by giving *name*.$*$, or referring to *all* executable constants currently available by giving $*$.

By default, for each involved theory one corresponding name space module is generated. Alternativly, a module name may be specified after the **module_name** keyword; then *all* code is placed in this module.

For *SML* and *OCaml*, the file specification refers to a single file; for *Haskell*, it refers to a whole directory, where code is generated in multiple files reflecting the module hierarchy. The file specification "–" denotes standard output. For *SML*, omitting the file specification compiles code internally in the context of the current ML session.

Serializers take an optional list of arguments in parentheses. For *Haskell* a module name prefix may be given using the "*root:*" argument; "*string_classes*" adds a "`deriving (Read, Show)`" clause to each appropriate datatype declaration.

code_thms prints a list of theorems representing the corresponding program containing all given constants; if no constants are given, the currently cached code theorems are printed.

code_deps visualizes dependencies of theorems representing the corresponding program containing all given constants; if no constants are given, the currently cached code theorems are visualized.

code_datatype specifies a constructor set for a logical type.

code_const associates a list of constants with target-specific serializations; omitting a serialization deletes an existing serialization.

code_type associates a list of type constructors with target-specific serializations; omitting a serialization deletes an existing serialization.

code_class associates a list of classes with target-specific class names; in addition, constants associated with this class may be given target-specific names used for instance declarations; omitting a serialization deletes an existing serialization. This applies only to *Haskell*.

code_instance declares a list of type constructor / class instance relations as "already present" for a given target. Omitting a "–" deletes an existing "already present" declaration. This applies only to *Haskell*.

code_monad provides an auxiliary mechanism to generate monadic code.

code_reserved declares a list of names as reserved for a given target, preventing it to be shadowed by any generated code.

code_include adds arbitrary named content ("include") to generated code. A as last argument "–" will remove an already added "include".

code_modulename declares aliasings from one module name onto another.

code_exception declares constants which are not required to have a definition by a defining equations; these are mapped on exceptions instead.

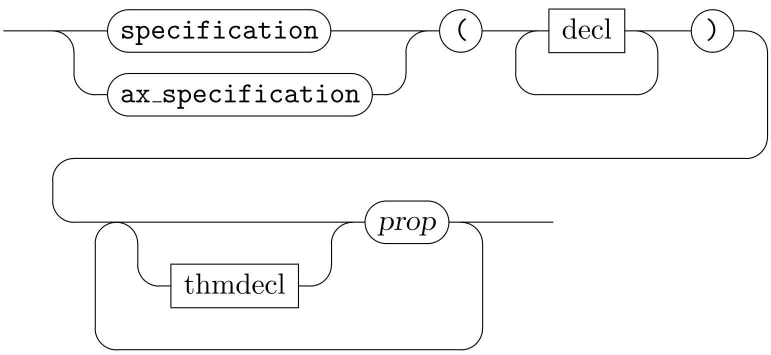
code func explicitly selects (or with option "*del*:" deselects) a defining equation for code generation. Usually packages introducing defining equations provide a resonable default setup for selection.

codeinline declares (or with option "*del*:" removes) inlining theorems which are applied as rewrite rules to any defining equation during preprocessing.

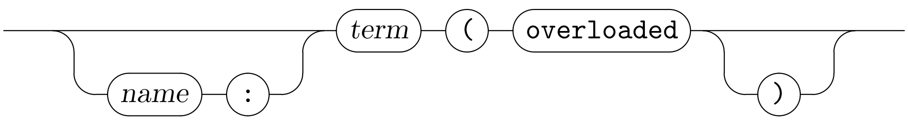
print_codesetup gives an overview on selected defining equations, code generator datatypes and preprocessor setup.

8.10 Definition by specification

specification	:	*theory* → *proof*(*prove*)
ax_specification	:	*theory* → *proof*(*prove*)



decl



specification *decls* φ sets up a goal stating the existence of terms with the properties specified to hold for the constants given in *decls*. After finishing the proof, the theory will be augmented with definitions for the given constants, as well as with theorems stating the properties for these constants.

ax_specification *decls* φ sets up a goal stating the existence of terms with the properties specified to hold for the constants given in *decls*. After finishing the proof, the theory will be augmented with axioms expressing the properties given in the first place.

decl declares a constant to be defined by the specification given. The definition for the constant c is bound to the name *c_def* unless a theorem name is given in the declaration. Overloaded constants should be declared as such.

Whether to use **specification** or **ax_specification** is to some extent a matter of style. **specification** introduces no new axioms, and so by construction cannot introduce inconsistencies, whereas **ax_specification** does introduce axioms, but only after the user has explicitly proven it to be safe. A practical issue must be considered, though: After introducing two constants with the same properties using **specification**, one can prove that the two constants are, in fact, equal. If this might be a problem, one should use **ax_specification**.

Chapter 9

Isabelle/HOLCF

9.1 Mixfix syntax for continuous operations

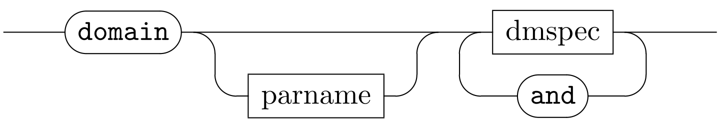
consts : *theory* → *theory*

HOLCF provides a separate type for continuous functions $\alpha \to \beta$, 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 $f\,x$.

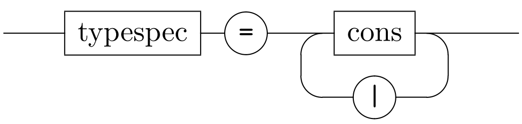
The HOLCF variant of **consts** modifies that of Pure Isabelle (cf. §3.9.3) such that declarations involving continuous function types are treated specifically. Any given syntax template is transformed internally, generating translation rules for the abstract and concrete representation of continuous application. Note that mixing of HOLCF and Pure application is *not* supported!

9.2 Recursive domains

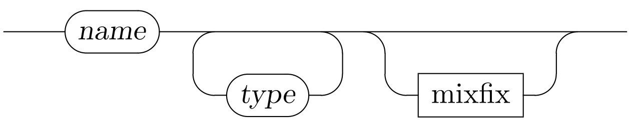
domain : *theory* → *theory*



dmspec



cons



dtrules

`distinct` thmrefs `inject` thmrefs `induction` thmrefs

Recursive domains in HOLCF are analogous to datatypes in classical HOL (cf. §8.4). 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 [10] for a general discussion of HOLCF domains.

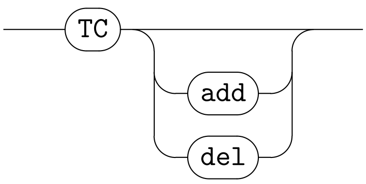
Chapter 10

Isabelle/ZF

10.1 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* | *proof* → *theory* | *proof*
typecheck	:	*method*
TC	:	*attribute*



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

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

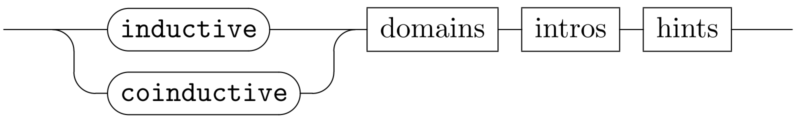
TC adds or deletes type-checking rules from the context.

10.2 (Co)Inductive sets and datatypes

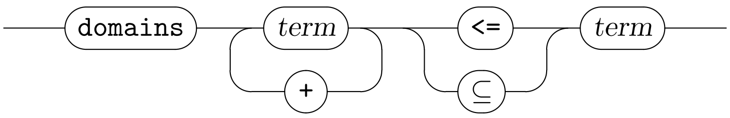
10.2.1 Set definitions

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.

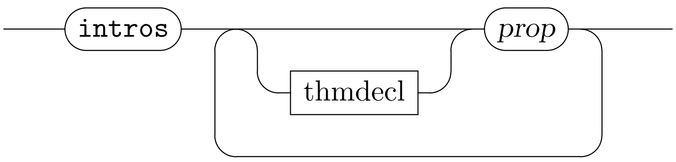
inductive	:	*theory* → *theory*
coinductive	:	*theory* → *theory*
datatype	:	*theory* → *theory*
codatatype	:	*theory* → *theory*



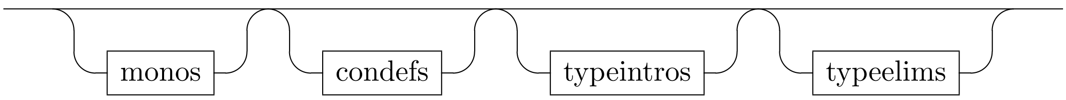
domains



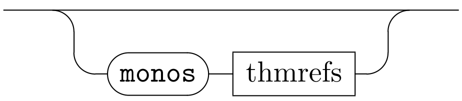
intros



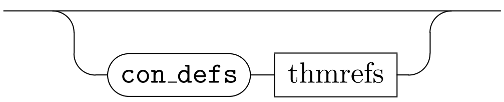
hints



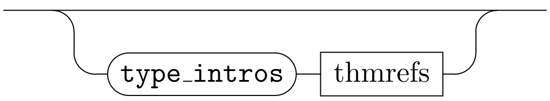
monos



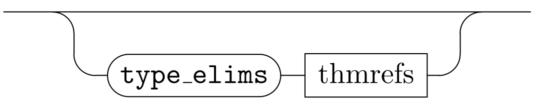
condefs



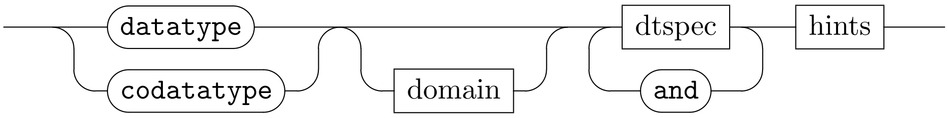
typeintros



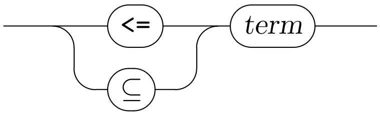
typeelims



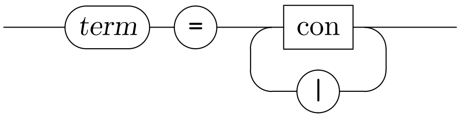
In the following syntax specification *monos*, *typeintros*, and *typeelims* are the same as above.



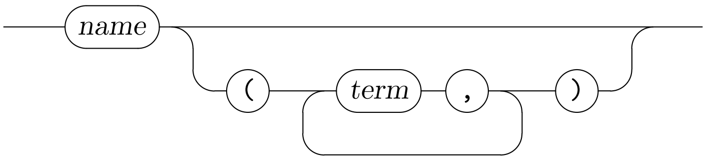
domain



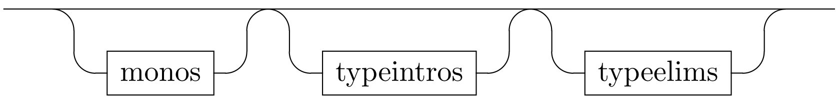
dtspec



con



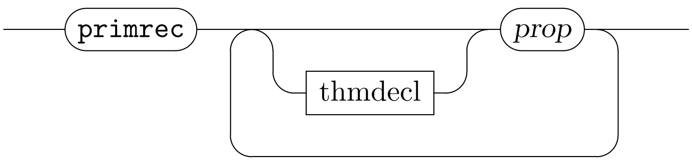
hints



See [16] for further information on inductive definitions in ZF, but note that this covers the old-style theory format.

10.2.2 Primitive recursive functions

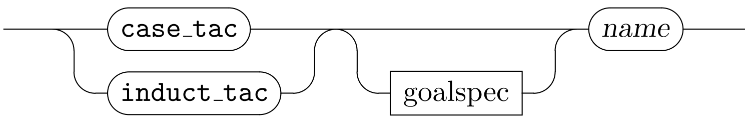
primrec : *theory* $\rightarrow$ *theory*

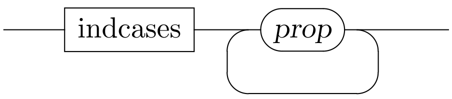


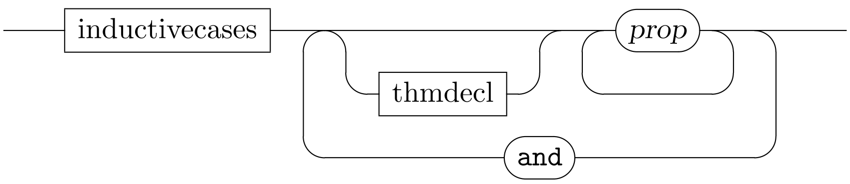
10.2.3 Cases and induction: emulating tactic scripts

The following important tactical tools of Isabelle/ZF have been ported to Isar. These should not be used in proper proof texts.

*case_tac**	:	*method*
*induct_tac**	:	*method*
*ind_cases**	:	*method*
inductive_cases	:	*theory* → *theory*







Appendix A

Isabelle/Isar quick reference

A.1 Proof commands

A.1.1 Primitives and basic syntax

fix x	augment context by $\bigwedge x.\ \Box$
assume a: φ	augment context by $\varphi \Longrightarrow \Box$
then	indicate forward chaining of facts
have a: φ	prove local result
show a: φ	prove local result, refining some goal
using a	indicate use of additional facts
unfolding a	unfold definitional equations
proof m_1 ... **qed** m_2	indicate proof structure and refinements
{ ... **}**	indicate explicit blocks
next	switch blocks
note $a = b$	reconsider facts
let $p = t$	abbreviate terms by higher-order matching

theory-stmt	=	**theorem** *name*: *props proof* | **definition** ... | ...
proof	=	*prfx** **proof** *method stmt** **qed** *method*
	|	*prfx** **done**
prfx	=	**apply** *method*
	|	**using** *facts*
	|	**unfolding** *facts*
stmt	=	**{** *stmt** **}**
	|	**next**
	|	**note** *name* = *facts*
	|	**let** *term* = *term*
	|	**fix** *var*$^+$
	|	**assume** *name*: *props*
	|	**then**$^?$ *goal*
goal	=	**have** *name*: *props proof*
	|	**show** *name*: *props proof*

A.1.2 Abbreviations and synonyms

by m_1 m_2	$\equiv$	**proof** m_1 **qed** m_2
..	$\equiv$	**by** *rule*
.	$\equiv$	**by** *this*
hence	$\equiv$	**then have**
thus	$\equiv$	**then show**
from a	$\equiv$	**note** a **then**
with a	$\equiv$	**from** a **and** *this*
from *this*	$\equiv$	**then**
from *this* **have**	$\equiv$	**hence**
from *this* **show**	$\equiv$	**thus**

A.1.3 Derived elements

also$_0$	$\approx$	**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: φ	$\approx$	**assume** a: φ
def a: $x \equiv t$	$\approx$	**fix** x **assume** a: $x \equiv t$
obtain x **where** a: φ	$\approx$	... **fix** x **assume** a: φ
case c	$\approx$	**fix** x **assume** c: φ
sorry	$\approx$	**by** *cheating*

A.1.4 Diagnostic commands

pr	print current state
thm a	print fact
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)

assumption	apply some assumption
this	apply current facts
rule a	apply some rule
rule	apply standard rule (default for **proof**)
contradiction	apply ¬ elimination rule (any order)
cases t	case analysis (provides cases)
induct x	proof by induction (provides cases)

Repeated steps (inserting facts)

−	no rules
intro a	introduction rules
intro_classes	class introduction rules
elim a	elimination rules
unfold a	definitional rewrite rules

Automated proof tools (inserting facts)

iprover	intuitionistic proof search
blast, *fast*	Classical Reasoner
simp, *simp_all*	Simplifier (+ Splitter)
auto, *force*	Simplifier + Classical Reasoner
arith	Arithmetic procedures

A.3 Attributes

Operations

OF a	rule resolved with facts (skipping "_")
of t	rule instantiated with terms (skipping "_")
where $x = t$	rule instantiated with terms, by variable name
symmetric	resolution with symmetry rule
THEN b	resolution with another rule
rule_format	result put into standard rule format
elim_format	destruct rule turned into elimination rule format

Declarations

simp	Simplifier rule
intro, *elim*, *dest*	Pure or Classical Reasoner rule
iff	Simplifier + Classical Reasoner rule
split	case split rule
trans	transitivity rule
sym	symmetry rule

A.4 Rule declarations and methods

	rule	*iprover*	*blast* *fast*	*simp* *simp_all*	*auto* *force*
Pure.elim! *Pure.intro*!	×	×			
Pure.elim *Pure.intro*	×	×			
elim! *intro*!	×		×		×
elim *intro*	×		×		×
iff	×		×	×	×
iff?	×				
elim? *intro?*	×				
simp				×	×
cong				×	×
split				×	×

A.5 Emulating tactic scripts

A.5.1 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

A.5.2 Methods

rule_tac insts	resolution (with instantiation)
erule_tac insts	elim-resolution (with instantiation)
drule_tac insts	destruct-resolution (with instantiation)
frule_tac insts	forward-resolution (with instantiation)
cut_tac insts	insert facts (with instantiation)
thin_tac φ	delete assumptions
subgoal_tac φ	new claims
rename_tac x	rename innermost goal parameters
rotate_tac n	rotate assumptions of goal
tactic text	arbitrary ML tactic
case_tac t	exhaustion (datatypes)
induct_tac x	induction (datatypes)
ind_cases t	exhaustion + simplification (inductive predicates)

Appendix B

ML tactic expressions

Isar Proof methods closely resemble traditional tactics, when used in unstructured sequences of **apply** commands. 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.

B.1 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 §7.2.3) 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 §4.5), or any of its "improper" variants *erule*, *drule*, *frule* (see §7.2.1). Note that explicit goal addressing is only supported by the actual *rule_tac* version.

With this in mind, plain resolution tactics correspond to Isar methods as follows.

`rtac` a 1	*rule* a
`resolve_tac` $[a_1, \ldots]$ 1	*rule* $a_1 \ldots$
`res_inst_tac` $[(x_1,\ t_1), \ldots]$ a 1	*rule_tac* $x_1 = t_1$ **and** $\ldots$ **in** a
`rtac` a i	*rule_tac* $[i]$ a
`resolve_tac` $[a_1, \ldots]$ i	*rule_tac* $[i]$ $a_1 \ldots$
`res_inst_tac` $[(x_1,\ t_1), \ldots]$ a i	*rule_tac* $[i]$ $x_1 = t_1$ **and** $\ldots$ **in** a

Note that explicit goal addressing may be usually avoided by changing the order of subgoals with **defer** or **prefer** (see §4.8).

B.2 Simplifier tactics

The main Simplifier tactics `simp_tac` and variants (cf. [15]) are all covered by the *simp* and *simp_all* methods (see §7.3). Note that there is no individual goal addressing available, simplification acts either on the first goal (*simp*) or all goals (*simp_all*).

`asm_full_simp_tac` @{*simpset*} 1	*simp*
`ALLGOALS` (`asm_full_simp_tac` @{*simpset*})	*simp_all*
`simp_tac` @{*simpset*} 1	*simp* (*no_asm*)
`asm_simp_tac` @{*simpset*} 1	*simp* (*no_asm_simp*)
`full_simp_tac` @{*simpset*} 1	*simp* (*no_asm_use*)
`asm_lr_simp_tac` @{*simpset*} 1	*simp* (*asm_lr*)

B.3 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 [15]). The corresponding Isar methods usually share the same base name, such as *blast*, *fast*, *clarify* etc. (see §7.4).

B.4 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.

`stac` *a* 1		*subst a*
`hyp_subst_tac` 1		*hypsubst*
`strip_tac` 1	$\approx$	*intro strip*
`split_all_tac` 1		*simp* (*no_asm_simp*) *only*: *split_tupled_all*
	$\approx$	*simp only*: *split_tupled_all*
	$\ll$	*clarify*

B.5 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 §7.2.3).

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

tac_1 `THEN` tac_2	$meth_1$, $meth_2$
tac_1 `ORELSE` tac_2	$meth_1$ | $meth_2$
`TRY` *tac*	*meth?*
`REPEAT1` *tac*	*meth*+
`REPEAT` *tac*	(*meth*+)*?*
`EVERY` [tac_1, ...]	$meth_1$, ...
`FIRST` [tac_1, ...]	$meth_1$ | ...

`CHANGED` (see [15]) 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 [15]) are not available in Isar, since there is no direct goal addressing. Nevertheless, some basic methods address all goals internally, notably *simp_all* (see §7.3). Also note that `ALLGOALS` can 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 \<dots>))`, is usually better expressed using the *intro* and *elim* methods of Isar (see §7.4).

Bibliography

[1] David Aspinall. Proof General. http://proofgeneral.inf.ed.ac.uk/.

[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] Florian Haftmann. *Code generation from Isabelle theories.* http://isabelle.in.tum.de/doc/codegen.pdf.

[6] Florian Haftmann. *Haskell-style type classes with Isabelle/Isar.* http://isabelle.in.tum.de/doc/classes.pdf.

[7] Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL.* http://isabelle.in.tum.de/doc/functions.pdf.

[8] Xavier Leroy et al. *The Objective Caml system – Documentation and user's manual.* http://caml.inria.fr/pub/docs/manual-ocaml/.

[9] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[10] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[11] 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.

[12] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle's Logics: HOL*. http://isabelle.in.tum.de/doc/logics-HOL.pdf.

[13] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS 2283.

[14] Lawrence C. Paulson. *Introduction to Isabelle*. http://isabelle.in.tum.de/doc/intro.pdf.

[15] Lawrence C. Paulson. *The Isabelle Reference Manual*. http://isabelle.in.tum.de/doc/ref.pdf.

[16] Lawrence C. Paulson. *Isabelle's Logics: FOL and ZF*. http://isabelle.in.tum.de/doc/logics-ZF.pdf.

[17] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Automated Deduction — CADE-12 International Conference*, LNAI 814, pages 148–161. Springer, 1994.

[18] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[19] Christoph Wedler. Emacs package "X-Symbol". http://x-symbol.sourceforge.net.

[20] Makarius Wenzel. Isabelle/Isar — a generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*. University of Białystok, 2007. http://www.in.tum.de/~wenzelm/papers/isar-framework.pdf.

[21] 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.

[22] 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.

[23] 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.

[24] Markus Wenzel and Stefan Berghofer. *The Isabelle System Manual*. http://isabelle.in.tum.de/doc/system.pdf.

[25] Freek Wiedijk. The mathematical vernacular. Unpublished paper, 2000. http://www.cs.kun.nl/~freek/notes/mv.ps.gz.

Index