# The Isabelle Reference Manual

*Lawrence C. Paulson*
Computer Laboratory
University of Cambridge
lcp@cl.cam.ac.uk

With Contributions by Tobias Nipkow and Markus Wenzel[1]

28 August 1998

[1]Tobias Nipkow, of T. U. Munich, wrote most of Chapters 7 and 10, and part of Chapter 6. Carsten Clasohm also contributed to Chapter 6. Markus Wenzel contributed to Chapter 8. Sara Kalvala, Martin Simons and others suggested changes and corrections. The research has been funded by the EPSRC (grants GR/G53279, GR/H40570, GR/K57381, GR/K77051) and by ESPRIT project 6453: Types.

# Contents

**Chapter 1**

# Basic Use of Isabelle

The Reference Manual is a comprehensive description of Isabelle proper, including all ML commands, functions and packages. It really is intended for reference, perhaps for browsing, but not for reading through. It is not a tutorial, but assumes familiarity with the basic logical concepts of Isabelle.

When you are looking for a way of performing some task, scan the Table of Contents for a relevant heading. Functions are organized by their purpose, by their operands (subgoals, tactics, theorems), and by their usefulness. In each section, basic functions appear first, then advanced functions, and finally esoteric functions. Use the Index when you are looking for the definition of a particular Isabelle function.

A few examples are presented. Many examples files are distributed with Isabelle, however; please experiment interactively.

## 1.1 Basic interaction with Isabelle

We assume that your local Isabelle administrator (this might be you!) has already installed the `Pure` system and several object-logics properly — otherwise see the `INSTALL` file in the top-level directory of the distribution on how to build it.

Let ⟨*isabellehome*⟩ denote the location where the distribution has been installed. To run Isabelle from a the shell prompt within an ordinary text terminal session, simply type

⟨*isabellehome*⟩`/bin/isabelle`

This should start an interactive ML session with the default object-logic already preloaded.

Subsequently we assume that ⟨*isabellehome*⟩`/bin` has been added to your shell's search path, in order to avoid typing full path specifications of the executable files.

The object-logic image to load may be also specified explicitly as an argument to the `isabelle` command, e.g.

```
isabelle FOL
```

This should put you into the world of polymorphic first-order logic (assuming that `FOL` has been pre-built).

Isabelle provides no means of storing theorems or internal proof objects on files. Theorems are simply part of the ML state. To save your work between sessions, you must dump the ML system state to a file. This is done automatically when ending the session normally (e.g. by typing control-D), provided that the image has been opened *writable* in the first place. The standard object-logic images are usually read-only, so you probably have to create a private working copy first. For example, the following shell command puts you into a writable Isabelle session of name `Foo` that initially contains just `FOL`:

```
isabelle FOL Foo
```

Ending the `Foo` session with control-D will cause the complete ML world to be saved somewhere in your home directory[1]. Make sure there is enough space available! Then one may later continue at exactly the same point by running

```
isabelle Foo
```

More details about the `isabelle` command may be found in the *System Manual.*

Saving the state is not enough. Record, on a file, the top-level commands that generate your theories and proofs. Such a record allows you to replay the proofs whenever required, for instance after making minor changes to the axioms. Ideally, your record will be somewhat intelligible to others as a formal description of your work.

There are more comfortable user interfaces than the bare-bones ML top-level run from a text terminal. The `Isabelle` executable (note the capital I) runs one such interface, depending on your local configuration. Furthermore there are a number of external utilities available. These are started uniformly via the `isatool` wrapper. See the *System Manual* for more information user interfaces and utilities.

## 1.2 Ending a session

```
quit    : unit -> unit
exit    : int -> unit
commit  : unit -> unit
```

`quit();` ends the Isabelle session, without saving the state.

`exit` $i$`;` similar to `quit`, passing return code $i$ to the operating system.

`commit();` saves the current state without ending the session, provided that the logic image is opened read-write.

[1] The default location is in `~/isabelle/heaps`, but this depends on your local configuration.

Typing control-D also finishes the session in essentially the same way as the sequence `commit(); quit();` would.

## 1.3 Reading ML files

```
cd            : string -> unit
pwd           : unit -> string
use           : string -> unit
time_use      : string -> unit
```

`cd "`*dir*`";` changes the current directory to *dir*. This is the default directory for reading files.

`pwd();` returns the full path of the current directory.

`use "`*file*`";` reads the given *file* as input to the ML session. Reading a file of Isabelle commands is the usual way of replaying a proof.

`time_use "`*file*`";` performs `use "`*file*`"` and prints the total execution time.

The *dir* and *file* specifications of the `cd` and `use` commands may contain path variables that are expanded appropriately, e.g. `$ISABELLE_HOME` or `~` (which abbreviates `$HOME`). Section 6.2 describes commands for loading theory files.

## 1.4 Setting flags

```
set     : bool ref -> bool
reset   : bool ref -> bool
toggle  : bool ref -> bool
```

These are some shorthands for manipulating boolean references. The new value is returned.

## 1.5 Printing of terms and theorems

Isabelle's pretty printer is controlled by a number of parameters.

### 1.5.1 Printing limits

```
Pretty.setdepth  : int -> unit
Pretty.setmargin : int -> unit
print_depth      : int -> unit
```

These set limits for terminal output. See also `goals_limit`, which limits the number of subgoals printed (§2.1.7).

`Pretty.setdepth` $d$; tells Isabelle's pretty printer to limit the printing depth to $d$. This affects Isabelle's display of theorems and terms. The default value is 0, which permits printing to an arbitrary depth. Useful values for $d$ are 10 and 20.

`Pretty.setmargin` $m$; tells Isabelle's pretty printer to assume a right margin (page width) of $m$. The initial margin is 76.

`print_depth` $n$; limits the printing depth of complex ML values, such as theorems and terms. This command affects the ML top level and its effect is compiler-dependent. Typically $n$ should be less than 10.

### 1.5.2 Printing of hypotheses, brackets, types etc.

| | |
|---|---|
| `show_hyps : bool ref` | **initially true** |
| `show_brackets : bool ref` | **initially false** |
| `show_types : bool ref` | **initially false** |
| `show_sorts : bool ref` | **initially false** |
| `show_consts : bool ref` | **initially false** |
| `long_names : bool ref` | **initially false** |

These flags allow you to control how much information is displayed for types, terms and theorems. The hypotheses of theorems *are* normally shown. Superfluous parentheses of types and terms are not. Types and sorts of variables are normally hidden.

Note that displaying types and sorts may explain why a polymorphic inference rule fails to resolve with some goal, or why a rewrite rule does not apply as expected.

`reset show_hyps;` makes Isabelle show each meta-level hypothesis as a dot.

`set show_brackets;` makes Isabelle show full bracketing. In particular, this reveals the grouping of infix operators.

`set show_types;` makes Isabelle show types when printing a term or theorem.

`set show_sorts;` makes Isabelle show both types and the sorts of type variables, independently of the value of `show_types`.

`set show_consts;` makes Isabelle show types of constants, provided that showing of types is enabled at all. This is supported for printing of proof states only. Note that the output can be enormous as polymorphic constants often occur at several different type instances.

`set long_names;` forces names of all objects (types, constants, theorems, etc.) to be printed in their fully qualified internal form.

### 1.5.3 $\eta$-contraction before printing

`eta_contract: bool ref` **initially false**

The $\eta$-**contraction law** asserts $(\lambda x \,.\, f(x)) \equiv f$, provided $x$ is not free in $f$. It asserts **extensionality** of functions: $f \equiv g$ if $f(x) \equiv g(x)$ for all $x$. Higher-order unification frequently puts terms into a fully $\eta$-expanded form. For example, if $F$ has type $(\tau \Rightarrow \tau) \Rightarrow \tau$ then its expanded form is $\lambda h \,.\, F(\lambda x \,.\, h(x))$. By default, the user sees this expanded form.

`set eta_contract;` makes Isabelle perform $\eta$-contractions before printing, so that $\lambda h \,.\, F(\lambda x \,.\, h(x))$ appears simply as $F$. The distinction between a term and its $\eta$-expanded form occasionally matters.

## 1.6 Diagnostic messages

Isabelle conceptually provides three output channels for different kinds of messages: ordinary text, warnings, errors. Depending on the user interface involved, these messages may appear in different text styles or colours, even within separate windows.

The default setup of an `isabelle` terminal session is as follows: plain output of ordinary text, warnings prefixed by `###`'s, errors prefixed by `***`'s. For example, a typical warning would look like this:

```
### Beware the Jabberwock, my son!
### The jaws that bite, the claws that catch!
### Beware the Jubjub Bird, and shun
### The frumious Bandersnatch!
```

`ML` programs may output diagnostic messages using the following functions:

```
writeln : string -> unit
warning : string -> unit
error   : string -> 'a
```

Note that `error` fails by raising exception `ERROR` after having output the text, while `writeln` and `warning` resume normal program execution.

## 1.7 Displaying exceptions as error messages

`print_exn: exn -> 'a`

Certain Isabelle primitives, such as the forward proof functions `RS` and `RSN`, are called both interactively and from programs. They indicate errors not by printing messages, but by raising exceptions. For interactive use, ML's reporting of an uncaught exception may be uninformative. The Poly/ML function `exception_trace` can generate a backtrace.

`print_exn` *e* displays the exception *e* in a readable manner, and then re-raises *e*. Typical usage is *EXP* `handle e => print_exn e;`, where *EXP* is an expression that may raise an exception.

`print_exn` can display the following common exceptions, which concern types, terms, theorems and theories, respectively. Each carries a message and related information.

```
exception TYPE   of string * typ list * term list
exception TERM   of string * term list
exception THM    of string * int * thm list
exception THEORY of string * theory list
```

**!** `print_exn` prints terms by calling `prin`, which obtains pretty printing information from the proof state last stored in the subgoal module. The appearance of the output thus depends upon the theory used in the last interactive proof.

Chapter 2

# Proof Management: The Subgoal Module

The subgoal module stores the current proof state and many previous states; commands can produce new states or return to previous ones. The *state list* at level $n$ is a list of pairs

$$[(\psi_n, \Psi_n),\ (\psi_{n-1}, \Psi_{n-1}),\ \ldots,\ (\psi_0, [])]$$

where $\psi_n$ is the current proof state, $\psi_{n-1}$ is the previous one, …, and $\psi_0$ is the initial proof state. The $\Psi_i$ are sequences (lazy lists) of proof states, storing branch points where a tactic returned a list longer than one. The state lists permit various forms of backtracking.

Chopping elements from the state list reverts to previous proof states. Besides this, the `undo` command keeps a list of state lists. The module actually maintains a stack of state lists, to support several proofs at the same time.

The subgoal module always contains some proof state. At the start of the Isabelle session, this state consists of a dummy formula.

## 2.1 Basic commands

Most proofs begin with `Goal` or `Goalw` and require no other commands than `by`, `chop` and `undo`. They typically end with a call to `qed`.

### 2.1.1 Starting a backward proof

```
Goal        :                         string -> thm list
Goalw       :             thm list -> string -> thm list
goal        : theory ->               string -> thm list
goalw       : theory -> thm list -> string -> thm list
goalw_cterm : thm list -> cterm -> thm list
premises    : unit -> thm list
```

The goal commands start a new proof by setting the goal. They replace the current state list by a new one consisting of the initial proof state. They also empty the `undo` list; this command cannot be undone!

They all return a list of meta-hypotheses taken from the main goal. If this list is non-empty, bind its value to an ML identifier by typing something like

`val prems = goal` *theory formula*`;`

These assumptions typically serve as the premises when you are deriving a rule. They are also stored internally and can be retrieved later by the function `premises`. When the proof is finished, `qed` compares the stored assumptions with the actual assumptions in the proof state.

The capital versions of `Goal` are the basic user level commands and should be preferred over the more primitive lowercase `goal` commands. Apart from taking the current theory context as implicit argument, the former ones try to be smart in handling meta-hypotheses when deriving rules. Thus `prems` have to be seldom bound explicitly, the effect is as if `cut_facts_tac` had been called automatically.

Some of the commands unfold definitions using meta-rewrite rules. This expansion affects both the initial subgoal and the premises, which would otherwise require use of `rewrite_goals_tac` and `rewrite_rule`.

`Goal` *formula*; begins a new proof, where *formula* is written as an ML string.

`Goalw` *defs formula*; is like `Goal` but also applies the list of *defs* as meta-rewrite rules to the first subgoal and the premises.

`goal` *theory formula*; begins a new proof, where *theory* is usually an ML identifier and the *formula* is written as an ML string.

`goalw` *theory defs formula*; is like `goal` but also applies the list of *defs* as meta-rewrite rules to the first subgoal and the premises.

`goalw_cterm` *theory defs ct*; is a version of `goalw` for programming applications. The main goal is supplied as a cterm, not as a string. Typically, the cterm is created from a term $t$ by `cterm_of (sign_of thy)` $t$.

`premises()` returns the list of meta-hypotheses associated with the current proof (in case you forgot to bind them to an ML identifier).

### 2.1.2 Applying a tactic

```
by   : tactic -> unit
byev : tactic list -> unit
```

These commands extend the state list. They apply a tactic to the current proof state. If the tactic succeeds, it returns a non-empty sequence of next states. The head of the sequence becomes the next state, while the tail is retained for backtracking (see `back`).

`by` *tactic*; applies the *tactic* to the proof state.

`byev` *tactics*; applies the list of *tactics*, one at a time. It is useful for testing calls to `prove_goal`, and abbreviates `by (EVERY` *tactics*`)`.

*Error indications:*

- `"by: tactic failed"` means that the tactic returned an empty sequence when applied to the current proof state.
- `"Warning: same as previous level"` means that the new proof state is identical to the previous state.
- `"Warning: signature of proof state has changed"` means that some rule was applied whose theory is outside the theory of the initial proof state. This could signify a mistake such as expressing the goal in intuitionistic logic and proving it using classical logic.

### 2.1.3 Extracting and storing the proved theorem

```
qed       : string -> unit
result    : unit -> thm
uresult   : unit -> thm
bind_thm  : string * thm -> unit
store_thm : string * thm -> thm
```

`qed` *name*; is the usual way of ending a proof. It combines `result` and `bind_thm`: it gets the theorem using `result()` and stores it the theorem database associated with its theory. See below for details.

`result()` returns the final theorem, after converting the free variables to schematics. It discharges the assumptions supplied to the matching `goal` command.

It raises an exception unless the proof state passes certain checks. There must be no assumptions other than those supplied to `goal`. There must be no subgoals. The theorem proved must be a (first-order) instance of the original goal, as stated in the `goal` command. This allows **answer extraction** — instantiation of variables — but no other changes to the main goal. The theorem proved must have the same signature as the initial proof state.

These checks are needed because an Isabelle tactic can return any proof state at all.

`uresult()` is like `result()` but omits the checks. It is needed when the initial goal contains function unknowns, when definitions are unfolded in the main goal (by calling `rewrite_tac`), or when `assume_ax` has been used.

`bind_thm` (*name*, *thm*); stores `standard` *thm* (see §5.1.5) in the theorem database associated with its theory and in the ML variable *name*. The theorem can be retrieved from the database using `get_thm` (see §6.4).

`store_thm` (*name*, *thm*) stores *thm* in the theorem database associated with its theory and returns that theorem.

### 2.1.4 Extracting axioms and stored theorems

```
thm       : xstring -> thm
thms      : xstring -> thm list
get_axiom : theory -> xstring -> thm
get_thm   : theory -> xstring -> thm
get_thms  : theory -> xstring -> thm list
axioms_of : theory -> (string * thm) list
thms_of   : theory -> (string * thm) list
assume_ax : theory -> string -> thm
```

`thm` *name* is the basic user function for extracting stored theorems from the current theory context.

`thms` *name* is like `thm`, but returns a whole list associated with *name* rather than a single theorem. Typically this will be collections stored by packages, e.g. `list.simps`.

`get_axiom` *thy name* returns an axiom with the given *name* from *thy* or any of its ancestors, raising exception `THEORY` if none exists. Merging theories can cause several axioms to have the same name; `get_axiom` returns an arbitrary one. Usually, axioms are also stored as theorems and may be retrieved via `get_thm` as well.

`get_thm` *thy name* is analogous to `get_axiom`, but looks for a theorem stored in the theory's database. Like `get_axiom` it searches all parents of a theory if the theorem is not found directly in *thy*.

`get_thms` *thy name* is like `get_thm` for retrieving theorem lists stored within the theory. It returns a singleton list if *name* actually refers to a theorem rather than a theorem list.

`axioms_of` *thy* returns the axioms of this theory node, not including the ones of its ancestors.

`thms_of` *thy* returns all theorems stored within the database of this theory node, not including the ones of its ancestors.

`assume_ax` *thy* *formula* reads the *formula* using the syntax of *thy*, following the same conventions as axioms in a theory definition. You can thus pretend that *formula* is an axiom and use the resulting theorem like an axiom. Actually `assume_ax` returns an assumption; `qed` and `result` complain about additional assumptions, but `uresult` does not.

For example, if *formula* is `a=b ==> b=a` then the resulting theorem has the form `?a=?b ==> ?b=?a  [!!a b. a=b ==> b=a]`

### 2.1.5 Retrieving theorems

The following functions retrieve theorems (together with their names) from the theorem database that is associated with the current proof state's theory. They can only find theorems that have explicitly been stored in the database using `qed`, `bind_thm` or related functions.

```
findI           :          int -> (string * thm) list
findE           :   int -> int -> (string * thm) list
findEs          :          int -> (string * thm) list
thms_containing : xstring list -> (string * thm) list
```

`findI` *i* returns all "introduction rules" applicable to subgoal *i* — all theorems whose conclusion matches (rather than unifies with) subgoal *i*. Useful in connection with `resolve_tac`.

`findE` *n* *i* returns all "elimination rules" applicable to premise *n* of subgoal *i* — all those theorems whose first premise matches premise *n* of subgoal *i*. Useful in connection with `eresolve_tac` and `dresolve_tac`.

`findEs` *i* returns all "elimination rules" applicable to subgoal *i* — all those theorems whose first premise matches some premise of subgoal *i*. Useful in connection with `eresolve_tac` and `dresolve_tac`.

`thms_containing` *consts* returns all theorems that contain all of a given set of constants. Note that a few basic constants like `==>` are ignored.

### 2.1.6 Undoing and backtracking

```
chop    : unit -> unit
choplev : int -> unit
back    : unit -> unit
undo    : unit -> unit
```

`chop();` deletes the top level of the state list, cancelling the last `by` command. It provides a limited undo facility, and the `undo` command can cancel it.

`choplev` $n$; truncates the state list to level $n$, if $n \geq 0$. A negative value of $n$ refers to the $n$th previous level: `choplev ~1` has the same effect as `chop`.

`back();` searches the state list for a non-empty branch point, starting from the top level. The first one found becomes the current proof state — the most recent alternative branch is taken. This is a form of interactive backtracking.

`undo();` cancels the most recent change to the proof state by the commands `by`, `chop`, `choplev`, and `back`. It **cannot** cancel `goal` or `undo` itself. It can be repeated to cancel a series of commands.

*Error indications for* ***`back`***:

- `"Warning: same as previous choice at this level"` means `back` found a non-empty branch point, but that it contained the same proof state as the current one.
- `"Warning: signature of proof state has changed"` means the signature of the alternative proof state differs from that of the current state.
- `"back: no alternatives"` means `back` could find no alternative proof state.

### 2.1.7 Printing the proof state

```
pr    : unit -> unit
prlev : int -> unit
prlim : int -> unit
goals_limit: int ref
```
**initially 10**

See also the printing control options described in §1.5.

`pr();` prints the current proof state.

`prlev` $n$; prints the proof state at level $n$, if $n \geq 0$. A negative value of $n$ refers to the $n$th previous level. This command allows you to review earlier stages of the proof.

`prlim` $k$; prints the current proof state, limiting the number of subgoals to $k$. It updates `goals_limit` (see below) and is helpful when there are many subgoals.

`goals_limit :=` $k$; specifies $k$ as the maximum number of subgoals to print.

### 2.1.8 Timing

```
proof_timing: bool ref
```
**initially false**

`set proof_timing;` makes the `by` and `prove_goal` commands display how much processor time was spent. This information is compiler-dependent.

## 2.2 Shortcuts for applying tactics

These commands call `by` with common tactics. Their chief purpose is to minimise typing, although the scanning shortcuts are useful in their own right. Chapter 3 explains the tactics themselves.

### 2.2.1 Refining a given subgoal

```
ba  :             int -> unit
br  : thm      -> int -> unit
be  : thm      -> int -> unit
bd  : thm      -> int -> unit
brs : thm list -> int -> unit
bes : thm list -> int -> unit
bds : thm list -> int -> unit
```

`ba` *i*; performs `by (assume_tac` *i*`);`

`br` *thm i*; performs `by (resolve_tac [`*thm*`]` *i*`);`

`be` *thm i*; performs `by (eresolve_tac [`*thm*`]` *i*`);`

`bd` *thm i*; performs `by (dresolve_tac [`*thm*`]` *i*`);`

`brs` *thms i*; performs `by (resolve_tac` *thms i*`);`

`bes` *thms i*; performs `by (eresolve_tac` *thms i*`);`

`bds` *thms i*; performs `by (dresolve_tac` *thms i*`);`

### 2.2.2 Scanning shortcuts

These shortcuts scan for a suitable subgoal (starting from subgoal 1). They refine the first subgoal for which the tactic succeeds. Thus, they require less typing than `br`, etc. They display the selected subgoal's number; please watch this, for it may not be what you expect!

```
fa  : unit     -> unit
fr  : thm      -> unit
fe  : thm      -> unit
fd  : thm      -> unit
frs : thm list -> unit
fes : thm list -> unit
fds : thm list -> unit
```

`fa();` solves some subgoal by assumption.

`fr` *thm*; refines some subgoal using `resolve_tac` [*thm*]

`fe` *thm*; refines some subgoal using `eresolve_tac` [*thm*]

`fd` *thm*; refines some subgoal using `dresolve_tac` [*thm*]

`frs` *thms*; refines some subgoal using `resolve_tac` *thms*

`fes` *thms*; refines some subgoal using `eresolve_tac` *thms*

`fds` *thms*; refines some subgoal using `dresolve_tac` *thms*

### 2.2.3 Other shortcuts

```
bw  : thm -> unit
bws : thm list -> unit
ren : string -> int -> unit
```

`bw` *def*; performs `by (rewrite_goals_tac [`*def*`]);` It unfolds definitions in the subgoals (but not the main goal), by meta-rewriting with the given definition (see also §3.2.4).

`bws` is like `bw` but takes a list of definitions.

`ren` *names* $i$; performs `by (rename_tac` *names* $i$`);` it renames parameters in subgoal $i$. (Ignore the message `Warning: same as previous level`.)

## 2.3 Executing batch proofs

To save space below, let type `tacfun` abbreviate `thm list -> tactic list`, which is the type of a tactical proof.

```
prove_goal :           theory ->            string -> tacfun -> thm
qed_goal   : string -> theory ->            string -> tacfun -> unit
prove_goalw:           theory -> thm list -> string -> tacfun -> thm
qed_goalw  : string -> theory -> thm list -> string -> tacfun -> unit
prove_goalw_cterm:               thm list -> cterm  -> tacfun -> thm
```

These batch functions create an initial proof state, then apply a tactic to it, yielding a sequence of final proof states. The head of the sequence is returned, provided it is an instance of the theorem originally proposed. The forms `prove_goal`, `prove_goalw` and `prove_goalw_cterm` are analogous to `goal`, `goalw` and `goalw_cterm`.

The tactic is specified by a function from theorem lists to tactic lists. The function is applied to the list of meta-assumptions taken from the main goal. The resulting tactics are applied in sequence (using `EVERY`). For example, a proof consisting of the commands

```
val prems = goal theory formula;
by tac1;  ...  by tacn;
qed "my_thm";
```

can be transformed to an expression as follows:

```
qed_goal "my_thm" theory formula
 (fn prems=> [ tac1, ..., tacn ]);
```

The methods perform identical processing of the initial *formula* and the final proof state. But `prove_goal` executes the tactic as a atomic operation, bypassing the subgoal module; the current interactive proof is unaffected.

`prove_goal` *theory formula tacsf*; executes a proof of the *formula* in the given *theory*, using the given tactic function.

`qed_goal` *name theory formula tacsf*; acts like `prove_goal` but also stores the resulting theorem in the theorem database associated with its theory and in the ML variable *name* (see §2.1.3).

`prove_goalw` *theory defs formula tacsf*; is like `prove_goal` but also applies the list of *defs* as meta-rewrite rules to the first subgoal and the premises.

`qed_goalw` *name theory defs formula tacsf*; is analogous to `qed_goal`.

`prove_goalw_cterm` *theory defs ct tacsf*; is a version of `prove_goalw` for programming applications. The main goal is supplied as a cterm, not as a string. Typically, the cterm is created from a term $t$ as follows:

```
cterm_of (sign_of thy) t
```

## 2.4 Managing multiple proofs

You may save the current state of the subgoal module and resume work on it later. This serves two purposes.

1. At some point, you may be uncertain of the next step, and wish to experiment.
2. During a proof, you may see that a lemma should be proved first.

Each saved proof state consists of a list of levels; `chop` behaves independently for each of the saved proofs. In addition, each saved state carries a separate `undo` list.

### 2.4.1 The stack of proof states

```
push_proof   : unit -> unit
pop_proof    : unit -> thm list
rotate_proof : unit -> thm list
```

The subgoal module maintains a stack of proof states. Most subgoal commands affect only the top of the stack. The `Goal` command *replaces* the top of the stack; the only command that pushes a proof on the stack is `push_proof`.

To save some point of the proof, call `push_proof`. You may now state a lemma using `goal`, or simply continue to apply tactics. Later, you can return to the saved point by calling `pop_proof` or `rotate_proof`.

To view the entire stack, call `rotate_proof` repeatedly; as it rotates the stack, it prints the new top element.

`push_proof();` duplicates the top element of the stack, pushing a copy of the current proof state on to the stack.

`pop_proof();` discards the top element of the stack. It returns the list of assumptions associated with the new proof; you should bind these to an ML identifier. They can also be obtained by calling `premises`.

`rotate_proof();` rotates the stack, moving the top element to the bottom. It returns the list of assumptions associated with the new proof.

### 2.4.2 Saving and restoring proof states

```
save_proof    : unit -> proof
restore_proof : proof -> thm list
```

States of the subgoal module may be saved as ML values of type `proof`, and later restored.

`save_proof();` returns the current state, which is on top of the stack.

`restore_proof` *prf*; replaces the top of the stack by *prf*. It returns the list of assumptions associated with the new proof.

## 2.5 *Debugging and inspecting

These functions can be useful when you are debugging a tactic. They refer to the current proof state stored in the subgoal module. A tactic should never call them; it should operate on the proof state supplied as its argument.

### 2.5.1 Reading and printing terms

```
read    : string -> term
prin    : term -> unit
printyp : typ -> unit
```

These read and print terms (or types) using the syntax associated with the proof state.

`read` *string* reads the *string* as a term, without type checking.

`prin` $t$; prints the term $t$ at the terminal.

`printyp` $T$; prints the type $T$ at the terminal.

### 2.5.2 Inspecting the proof state

```
topthm  : unit -> thm
getgoal : int -> term
gethyps : int -> thm list
```

`topthm()` returns the proof state as an Isabelle theorem. This is what `by` would supply to a tactic at this point. It omits the post-processing of `result` and `uresult`.

`getgoal` $i$ returns subgoal $i$ of the proof state, as a term. You may print this using `prin`, though you may have to examine the internal data structure in order to locate the problem!

`gethyps` *i* returns the hypotheses of subgoal *i* as meta-level assumptions. In these theorems, the subgoal's parameters become free variables. This command is supplied for debugging uses of `METAHYPS`.

### 2.5.3 Filtering lists of rules

```
filter_goal: (term*term->bool) -> thm list -> int -> thm list
```

`filter_goal` *could ths i* applies `filter_thms` *could* to subgoal *i* of the proof state and returns the list of theorems that survive the filtering.

Chapter 3

# Tactics

Tactics have type `tactic`. This is just an abbreviation for functions from theorems to theorem sequences, where the theorems represent states of a backward proof. Tactics seldom need to be coded from scratch, as functions; instead they are expressed using basic tactics and tacticals.

This chapter only presents the primitive tactics. Substantial proofs require the power of automatic tools like simplification (Chapter 10) and classical tableau reasoning (Chapter 11).

## 3.1 Resolution and assumption tactics

**Resolution** is Isabelle's basic mechanism for refining a subgoal using a rule. **Elim-resolution** is particularly suited for elimination rules, while **destruct-resolution** is particularly suited for destruction rules. The `r`, `e`, `d` naming convention is maintained for several different kinds of resolution tactics, as well as the shortcuts in the subgoal module.

All the tactics in this section act on a subgoal designated by a positive integer $i$. They fail (by returning the empty sequence) if $i$ is out of range.

### 3.1.1 Resolution tactics

```
resolve_tac  : thm list -> int -> tactic
eresolve_tac : thm list -> int -> tactic
dresolve_tac : thm list -> int -> tactic
forward_tac  : thm list -> int -> tactic
```

These perform resolution on a list of theorems, *thms*, representing a list of object-rules. When generating next states, they take each of the rules in the order given. Each rule may yield several next states, or none: higher-order resolution may yield multiple resolvents.

`resolve_tac` *thms* $i$ refines the proof state using the rules, which should normally be introduction rules. It resolves a rule's conclusion with subgoal $i$ of the proof state.

`eresolve_tac` *thms* $i$ performs elim-resolution with the rules, which should normally be elimination rules. It resolves with a rule, proves its first premise by assumption, and finally *deletes* that assumption from any new subgoals. (To rotate a rule's premises, see `rotate_prems` in §5.1.5.)

`dresolve_tac` *thms* $i$ performs destruct-resolution with the rules, which normally should be destruction rules. This replaces an assumption by the result of applying one of the rules.

`forward_tac` is like `dresolve_tac` except that the selected assumption is not deleted. It applies a rule to an assumption, adding the result as a new assumption.

### 3.1.2 Assumption tactics

```
assume_tac    : int -> tactic
eq_assume_tac : int -> tactic
```

`assume_tac` $i$ attempts to solve subgoal $i$ by assumption.

`eq_assume_tac` is like `assume_tac` but does not use unification. It succeeds (with a *unique* next state) if one of the assumptions is identical to the subgoal's conclusion. Since it does not instantiate variables, it cannot make other subgoals unprovable. It is intended to be called from proof strategies, not interactively.

### 3.1.3 Matching tactics

```
match_tac  : thm list -> int -> tactic
ematch_tac : thm list -> int -> tactic
dmatch_tac : thm list -> int -> tactic
```

These are just like the resolution tactics except that they never instantiate unknowns in the proof state. Flexible subgoals are not updated willy-nilly, but are left alone. Matching — strictly speaking — means treating the unknowns in the proof state as constants; these tactics merely discard unifiers that would update the proof state.

`match_tac` *thms* $i$ refines the proof state using the rules, matching a rule's conclusion with subgoal $i$ of the proof state.

`ematch_tac` is like `match_tac`, but performs elim-resolution.

`dmatch_tac` is like `match_tac`, but performs destruct-resolution.

### 3.1.4 Resolution with instantiation

```
res_inst_tac  : (string*string)list -> thm -> int -> tactic
eres_inst_tac : (string*string)list -> thm -> int -> tactic
dres_inst_tac : (string*string)list -> thm -> int -> tactic
forw_inst_tac : (string*string)list -> thm -> int -> tactic
```

These tactics are designed for applying rules such as substitution and induction, which cause difficulties for higher-order unification. The tactics accept explicit instantiations for unknowns in the rule — typically, in the rule's conclusion. Each instantiation is a pair `(`$v$`,`$e$`)`, where $v$ is an unknown *without* its leading question mark!

- If $v$ is the type unknown `'a`, then the rule must contain a type unknown `?'a` of some sort $s$, and $e$ should be a type of sort $s$.
- If $v$ is the unknown `P`, then the rule must contain an unknown `?P` of some type $\tau$, and $e$ should be a term of some type $\sigma$ such that $\tau$ and $\sigma$ are unifiable. If the unification of $\tau$ and $\sigma$ instantiates any type unknowns in $\tau$, these instantiations are recorded for application to the rule.

Types are instantiated before terms. Because type instantiations are inferred from term instantiations, explicit type instantiations are seldom necessary — if `?t` has type `?'a`, then the instantiation list `[("'a","bool"),("t","True")]` may be simplified to `[("t","True")]`. Type unknowns in the proof state may cause failure because the tactics cannot instantiate them.

The instantiation tactics act on a given subgoal. Terms in the instantiations are type-checked in the context of that subgoal — in particular, they may refer to that subgoal's parameters. Any unknowns in the terms receive subscripts and are lifted over the parameters; thus, you may not refer to unknowns in the subgoal.

`res_inst_tac` *insts thm i* instantiates the rule *thm* with the instantiations *insts*, as described above, and then performs resolution on subgoal *i*. Resolution typically causes further instantiations; you need not give explicit instantiations for every unknown in the rule.

`eres_inst_tac` is like `res_inst_tac`, but performs elim-resolution.

`dres_inst_tac` is like `res_inst_tac`, but performs destruct-resolution.

`forw_inst_tac` is like `dres_inst_tac` except that the selected assumption is not deleted. It applies the instantiated rule to an assumption, adding the result as a new assumption.

## 3.2 Other basic tactics

### 3.2.1 Tactic shortcuts

```
rtac     :      thm -> int -> tactic
etac     :      thm -> int -> tactic
dtac     :      thm -> int -> tactic
atac     :             int -> tactic
ares_tac : thm list -> int -> tactic
rewtac   :      thm ->        tactic
```

These abbreviate common uses of tactics.

`rtac` *thm i* abbreviates `resolve_tac [`*thm*`]` *i*, doing resolution.

`etac` *thm i* abbreviates `eresolve_tac [`*thm*`]` *i*, doing elim-resolution.

`dtac` *thm i* abbreviates `dresolve_tac [`*thm*`]` *i*, doing destruct-resolution.

`atac` *i* abbreviates `assume_tac` *i*, doing proof by assumption.

`ares_tac` *thms i* tries proof by assumption and resolution; it abbreviates

`assume_tac` *i* `ORELSE resolve_tac` *thms i*

`rewtac` *def* abbreviates `rewrite_goals_tac [`*def*`]`, unfolding a definition.

### 3.2.2 Inserting premises and facts

```
cut_facts_tac : thm list -> int -> tactic
cut_inst_tac  : (string*string)list -> thm -> int -> tactic
subgoal_tac   : string -> int -> tactic
subgoal_tacs  : string list -> int -> tactic
```

These tactics add assumptions to a subgoal.

`cut_facts_tac` *thms i* adds the *thms* as new assumptions to subgoal $i$. Once they have been inserted as assumptions, they become subject to tactics such as `eresolve_tac` and `rewrite_goals_tac`. Only rules with no premises are inserted: Isabelle cannot use assumptions that contain $\Longrightarrow$ or $\bigwedge$. Sometimes the theorems are premises of a rule being derived, returned by `goal`; instead of calling this tactic, you could state the goal with an outermost meta-quantifier.

`cut_inst_tac` *insts thm i* instantiates the *thm* with the instantiations *insts*, as described in §3.1.4. It adds the resulting theorem as a new assumption to subgoal $i$.

`subgoal_tac` *formula i* adds the *formula* as a assumption to subgoal $i$, and inserts the same *formula* as a new subgoal, $i + 1$.

`subgoals_tac` *formulae* $i$ uses `subgoal_tac` to add the members of the list of *formulae* as assumptions to subgoal $i$.

### 3.2.3 "Putting off" a subgoal

```
defer_tac : int -> tactic
```

`defer_tac` $i$ moves subgoal $i$ to the last position in the proof state. It can be useful when correcting a proof script: if the tactic given for subgoal $i$ fails, calling `defer_tac` instead will let you continue with the rest of the script.

The tactic fails if subgoal $i$ does not exist or if the proof state contains type unknowns.

### 3.2.4 Definitions and meta-level rewriting

Definitions in Isabelle have the form $t \equiv u$, where $t$ is typically a constant or a constant applied to a list of variables, for example $sqr(n) \equiv n \times n$. Conditional definitions, $\phi \Longrightarrow t \equiv u$, are also supported. **Unfolding** the definition $t \equiv u$ means using it as a rewrite rule, replacing $t$ by $u$ throughout a theorem. **Folding** $t \equiv u$ means replacing $u$ by $t$. Rewriting continues until no rewrites are applicable to any subterm.

There are rules for unfolding and folding definitions; Isabelle does not do this automatically. The corresponding tactics rewrite the proof state, yielding a single next state. See also the `goalw` command, which is the easiest way of handling definitions.

```
rewrite_goals_tac : thm list -> tactic
rewrite_tac       : thm list -> tactic
fold_goals_tac    : thm list -> tactic
fold_tac          : thm list -> tactic
```

`rewrite_goals_tac` *defs* unfolds the *defs* throughout the subgoals of the proof state, while leaving the main goal unchanged. Use `SELECT_GOAL` to restrict it to a particular subgoal.

`rewrite_tac` *defs* unfolds the *defs* throughout the proof state, including the main goal — not normally desirable!

`fold_goals_tac` *defs* folds the *defs* throughout the subgoals of the proof state, while leaving the main goal unchanged.

`fold_tac` *defs* folds the *defs* throughout the proof state.

**!** These tactics only cope with definitions expressed as meta-level equalities ($\equiv$). More general equivalences are handled by the simplifier, provided that it is set up appropriately for your logic (see Chapter 10).

### 3.2.5 Theorems useful with tactics

```
asm_rl: thm
cut_rl: thm
```

`asm_rl` is $\psi \Longrightarrow \psi$. Under elim-resolution it does proof by assumption, and `eresolve_tac (asm_rl::`*thms*`)` $i$ is equivalent to

`assume_tac` $i$ `ORELSE` `eresolve_tac` *thms* $i$

`cut_rl` is $[\![\psi \Longrightarrow \theta, \psi]\!] \Longrightarrow \theta$. It is useful for inserting assumptions; it underlies `forward_tac`, `cut_facts_tac` and `subgoal_tac`.

## 3.3 Obscure tactics

### 3.3.1 Renaming parameters in a goal

```
rename_tac        : string -> int -> tactic
rename_last_tac   : string -> string list -> int -> tactic
Logic.set_rename_prefix : string -> unit
Logic.auto_rename       : bool ref
```
**initially false**

When creating a parameter, Isabelle chooses its name by matching variable names via the object-rule. Given the rule $(\forall I)$ formalized as $(\bigwedge x\,.\,P(x)) \Longrightarrow \forall x\,.\,P(x)$, Isabelle will note that the $\bigwedge$-bound variable in the premise has the same name as the $\forall$-bound variable in the conclusion.

Sometimes there is insufficient information and Isabelle chooses an arbitrary name. The renaming tactics let you override Isabelle's choice. Because renaming parameters has no logical effect on the proof state, the `by` command prints the message `Warning: same as previous level`.

Alternatively, you can suppress the naming mechanism described above and have Isabelle generate uniform names for parameters. These names have the form $p$`a`, $p$`b`, $p$`c`, ..., where $p$ is any desired prefix. They are ugly but predictable.

`rename_tac` *str* $i$ interprets the string *str* as a series of blank-separated variable names, and uses them to rename the parameters of subgoal $i$. The names must be distinct. If there are fewer names than parameters, then the tactic renames the innermost parameters and may modify the remaining ones to ensure that all the parameters are distinct.

`rename_last_tac` *prefix suffixes* $i$ generates a list of names by attaching each of the *suffixes* to the *prefix*. It is intended for coding structural induction tactics, where several of the new parameters should have related names.

`Logic.set_rename_prefix` *prefix*; sets the prefix for uniform renaming to *prefix*. The default prefix is `"k"`.

`set Logic.auto_rename;` makes Isabelle generate uniform names for parameters.

### 3.3.2 Manipulating assumptions

```
thin_tac   : string -> int -> tactic
rotate_tac : int -> int -> tactic
```

`thin_tac` *formula* $i$ deletes the specified assumption from subgoal $i$. Often the assumption can be abbreviated, replacing subformulæ by unknowns; the first matching assumption will be deleted. Removing useless assumptions from a subgoal increases its readability and can make search tactics run faster.

`rotate_tac` $n$ $i$ rotates the assumptions of subgoal $i$ by $n$ positions: from right to left if $n$ is positive, and from left to right if $n$ is negative. This is sometimes necessary in connection with `asm_full_simp_tac`, which processes assumptions from left to right.

### 3.3.3 Tidying the proof state

```
distinct_subgoals_tac : tactic
prune_params_tac      : tactic
flexflex_tac          : tactic
```

`distinct_subgoals_tac` removes duplicate subgoals from a proof state. (These arise especially in `ZF`, where the subgoals are essentially type constraints.)

`prune_params_tac` removes unused parameters from all subgoals of the proof state. It works by rewriting with the theorem $(\bigwedge x \,.\, V) \equiv V$. This tactic can make the proof state more readable. It is used with `rule_by_tactic` to simplify the resulting theorem.

`flexflex_tac` removes all flex-flex pairs from the proof state by applying the trivial unifier. This drastic step loses information, and should only be done as the last step of a proof.

Flex-flex constraints arise from difficult cases of higher-order unification. To prevent this, use `res_inst_tac` to instantiate some variables in a rule (§3.1.4). Normally flex-flex constraints can be ignored; they often disappear as unknowns get instantiated.

### 3.3.4 Composition: resolution without lifting

```
compose_tac: (bool * thm * int) -> int -> tactic
```

**Composing** two rules means resolving them without prior lifting or renaming of unknowns. This low-level operation, which underlies the resolution tactics, may occasionally be useful for special effects. A typical application is `res_inst_tac`, which lifts and instantiates a rule, then passes the result to `compose_tac`.

`compose_tac` (*flag*, *rule*, *m*) *i* refines subgoal *i* using *rule*, without lifting. The *rule* is taken to have the form $[\![\psi_1; \ldots ; \psi_m]\!] \Longrightarrow \psi$, where $\psi$ need not be atomic; thus $m$ determines the number of new subgoals. If *flag* is `true` then it performs elim-resolution — it solves the first premise of *rule* by assumption and deletes that assumption.

## 3.4 *Managing lots of rules

These operations are not intended for interactive use. They are concerned with the processing of large numbers of rules in automatic proof strategies. Higher-order resolution involving a long list of rules is slow. Filtering techniques can shorten the list of rules given to resolution, and can also detect whether a subgoal is too flexible, with too many rules applicable.

### 3.4.1 Combined resolution and elim-resolution

```
biresolve_tac   : (bool*thm)list -> int -> tactic
bimatch_tac     : (bool*thm)list -> int -> tactic
subgoals_of_brl : bool*thm -> int
lessb           : (bool*thm) * (bool*thm) -> bool
```

**Bi-resolution** takes a list of (*flag*, *rule*) pairs. For each pair, it applies resolution if the flag is `false` and elim-resolution if the flag is `true`. A single tactic call handles a mixture of introduction and elimination rules.

`biresolve_tac` *brls* *i* refines the proof state by resolution or elim-resolution on each rule, as indicated by its flag. It affects subgoal *i* of the proof state.

`bimatch_tac` is like `biresolve_tac`, but performs matching: unknowns in the proof state are never updated (see §3.1.3).

`subgoals_of_brl`(*flag*,*rule*) returns the number of new subgoals that bi-resolution would yield for the pair (if applied to a suitable subgoal). This is $n$ if the flag is `false` and $n - 1$ if the flag is `true`, where $n$ is the number of premises of the rule. Elim-resolution yields one fewer subgoal than ordinary resolution because it solves the major premise by assumption.

`lessb` (*brl1*,*brl2*) returns the result of

> `subgoals_of_brl` *brl1* `<` `subgoals_of_brl` *brl2*

Note that `sort lessb` *brls* sorts a list of (*flag*, *rule*) pairs by the number of new subgoals they will yield. Thus, those that yield the fewest subgoals should be tried first.

### 3.4.2 Discrimination nets for fast resolution

```
net_resolve_tac  : thm list -> int -> tactic
net_match_tac    : thm list -> int -> tactic
net_biresolve_tac: (bool*thm) list -> int -> tactic
net_bimatch_tac  : (bool*thm) list -> int -> tactic
filt_resolve_tac : thm list -> int -> int -> tactic
could_unify      : term*term->bool
filter_thms      : (term*term->bool) -> int*term*thm list -> thm list
```

The module `Net` implements a discrimination net data structure for fast selection of rules [2, Chapter 14]. A term is classified by the symbol list obtained by flattening it in preorder. The flattening takes account of function applications, constants, and free and bound variables; it identifies all unknowns and also regards $\lambda$-abstractions as unknowns, since they could $\eta$-contract to anything.

A discrimination net serves as a polymorphic dictionary indexed by terms. The module provides various functions for inserting and removing items from nets. It provides functions for returning all items whose term could match or unify with a target term. The matching and unification tests are overly lax (due to the identifications mentioned above) but they serve as useful filters.

A net can store introduction rules indexed by their conclusion, and elimination rules indexed by their major premise. Isabelle provides several functions for 'compiling' long lists of rules into fast resolution tactics. When supplied with a list of theorems, these functions build a discrimination net; the net is used when the tactic is applied to a goal. To avoid repeatedly constructing the nets, use currying: bind the resulting tactics to ML identifiers.

`net_resolve_tac` *thms* builds a discrimination net to obtain the effect of a similar call to `resolve_tac`.

`net_match_tac` *thms* builds a discrimination net to obtain the effect of a similar call to `match_tac`.

`net_biresolve_tac` *brls* builds a discrimination net to obtain the effect of a similar call to `biresolve_tac`.

`net_bimatch_tac` *brls* builds a discrimination net to obtain the effect of a similar call to `bimatch_tac`.

`filt_resolve_tac` *thms maxr i* uses discrimination nets to extract the *thms* that are applicable to subgoal $i$. If more than *maxr* theorems are applicable then the tactic fails. Otherwise it calls `resolve_tac`.

This tactic helps avoid runaway instantiation of unknowns, for example in type inference.

`could_unify` $(t,u)$ returns `false` if $t$ and $u$ are 'obviously' non-unifiable, and otherwise returns `true`. It assumes all variables are distinct, reporting that `?a=?a` may unify with `0=1`.

`filter_thms` *could* (*limit*, *prem*, *thms*) returns the list of potentially resolvable rules (in *thms*) for the subgoal *prem*, using the predicate *could* to compare the conclusion of the subgoal with the conclusion of each rule. The resulting list is no longer than *limit*.

## 3.5 Programming tools for proof strategies

Do not consider using the primitives discussed in this section unless you really need to code tactics from scratch.

### 3.5.1 Operations on type `tactic`

A tactic maps theorems to sequences of theorems. The type constructor for sequences (lazy lists) is called `Seq.seq`. To simplify the types of tactics and tacticals, Isabelle defines a type abbreviation:

```
type tactic = thm -> thm Seq.seq
```

The following operations provide means for coding tactics in a clean style.

```
PRIMITIVE :                  (thm -> thm) -> tactic
SUBGOAL   : ((term*int) -> tactic) -> int -> tactic
```

`PRIMITIVE` $f$ packages the meta-rule $f$ as a tactic that applies $f$ to the proof state and returns the result as a one-element sequence. If $f$ raises an exception, then the tactic's result is the empty sequence.

`SUBGOAL` $f$ $i$ extracts subgoal $i$ from the proof state as a term $t$, and computes a tactic by calling $f(t,i)$. It applies the resulting tactic to the same state. The tactic body is expressed using tactics and tacticals, but may peek at a particular subgoal:

`SUBGOAL (fn (t,i) =>` *tactic-valued expression*`)`

### 3.5.2 Tracing

```
pause_tac: tactic
print_tac: tactic
```

These tactics print tracing information when they are applied to a proof state. Their output may be difficult to interpret. Note that certain of the searching tacticals, such as `REPEAT`, have built-in tracing options.

`pause_tac` prints `** Press RETURN to continue:` and then reads a line from the terminal. If this line is blank then it returns the proof state unchanged; otherwise it fails (which may terminate a repetition).

`print_tac` returns the proof state unchanged, with the side effect of printing it at the terminal.

## 3.6 *Sequences

The module `Seq` declares a type of lazy lists. It uses Isabelle's type `option` to represent the possible presence (`Some`) or absence (`None`) of a value:

```
datatype 'a option = None  |  Some of 'a;
```

The `Seq` structure is supposed to be accessed via fully qualified names and should not be `open`.

### 3.6.1 Basic operations on sequences

```
Seq.empty   : 'a seq
Seq.make    : (unit -> ('a * 'a seq) option) -> 'a seq
Seq.single  : 'a -> 'a seq
Seq.pull    : 'a seq -> ('a * 'a seq) option
```

`Seq.empty` is the empty sequence.

`Seq.make (fn () => Some (`$x$`, `$xq$`))` constructs the sequence with head $x$ and tail $xq$, neither of which is evaluated.

`Seq.single` $x$ constructs the sequence containing the single element $x$.

`Seq.pull` $xq$ returns `None` if the sequence is empty and `Some (`$x$`, `$xq'$`)` if the sequence has head $x$ and tail $xq'$. Warning: calling Seq.pull $xq$ again will *recompute* the value of $x$; it is not stored!

### 3.6.2 Converting between sequences and lists

```
Seq.chop    : int * 'a seq -> 'a list * 'a seq
Seq.list_of : 'a seq -> 'a list
Seq.of_list : 'a list -> 'a seq
```

`Seq.chop` ($n$, $xq$) returns the first $n$ elements of $xq$ as a list, paired with the remaining elements of $xq$. If $xq$ has fewer than $n$ elements, then so will the list.

`Seq.list_of` $xq$ returns the elements of $xq$, which must be finite, as a list.

`Seq.of_list` $xs$ creates a sequence containing the elements of $xs$.

### 3.6.3 Combining sequences

```
Seq.append     : 'a seq * 'a seq -> 'a seq
Seq.interleave : 'a seq * 'a seq -> 'a seq
Seq.flat       : 'a seq seq -> 'a seq
Seq.map        : ('a -> 'b) -> 'a seq -> 'b seq
Seq.filter     : ('a -> bool) -> 'a seq -> 'a seq
```

`Seq.append` ($xq$, $yq$) concatenates $xq$ to $yq$.

`Seq.interleave` ($xq$, $yq$) joins $xq$ with $yq$ by interleaving their elements. The result contains all the elements of the sequences, even if both are infinite.

`Seq.flat` $xqq$ concatenates a sequence of sequences.

`Seq.map` $f$ $xq$ applies $f$ to every element of $xq = x_1, x_2, \ldots$, yielding the sequence $f(x_1), f(x_2), \ldots$.

`Seq.filter` $p$ $xq$ returns the sequence consisting of all elements $x$ of $xq$ such that $p(x)$ is `true`.

Chapter 4

# Tacticals

Tacticals are operations on tactics. Their implementation makes use of functional programming techniques, especially for sequences. Most of the time, you may forget about this and regard tacticals as high-level control structures.

## 4.1 The basic tacticals

### 4.1.1 Joining two tactics

The tacticals `THEN` and `ORELSE`, which provide sequencing and alternation, underlie most of the other control structures in Isabelle. `APPEND` and `INTLEAVE` provide more sophisticated forms of alternation.

```
THEN     : tactic * tactic -> tactic                    infix 1
ORELSE   : tactic * tactic -> tactic                    infix
APPEND   : tactic * tactic -> tactic                    infix
INTLEAVE : tactic * tactic -> tactic                    infix
```

$tac_1$ `THEN` $tac_2$ is the sequential composition of the two tactics. Applied to a proof state, it returns all states reachable in two steps by applying $tac_1$ followed by $tac_2$. First, it applies $tac_1$ to the proof state, getting a sequence of next states; then, it applies $tac_2$ to each of these and concatenates the results.

$tac_1$ `ORELSE` $tac_2$ makes a choice between the two tactics. Applied to a state, it tries $tac_1$ and returns the result if non-empty; if $tac_1$ fails then it uses $tac_2$. This is a deterministic choice: if $tac_1$ succeeds then $tac_2$ is excluded.

$tac_1$ `APPEND` $tac_2$ concatenates the results of $tac_1$ and $tac_2$. By not making a commitment to either tactic, `APPEND` helps avoid incompleteness during search.

$tac_1$ `INTLEAVE` $tac_2$ interleaves the results of $tac_1$ and $tac_2$. Thus, it includes all possible next states, even if one of the tactics returns an infinite sequence.

### 4.1.2 Joining a list of tactics

```
EVERY : tactic list -> tactic
FIRST : tactic list -> tactic
```

`EVERY` and `FIRST` are block structured versions of `THEN` and `ORELSE`.

`EVERY` $[tac_1, \ldots, tac_n]$ abbreviates $tac_1$ `THEN ... THEN` $tac_n$. It is useful for writing a series of tactics to be executed in sequence.

`FIRST` $[tac_1, \ldots, tac_n]$ abbreviates $tac_1$ `ORELSE ... ORELSE` $tac_n$. It is useful for writing a series of tactics to be attempted one after another.

### 4.1.3 Repetition tacticals

```
TRY           : tactic -> tactic
REPEAT_DETERM : tactic -> tactic
REPEAT        : tactic -> tactic
REPEAT1       : tactic -> tactic
trace_REPEAT  : bool ref                                    initially false
```

`TRY` *tac* applies *tac* to the proof state and returns the resulting sequence, if non-empty; otherwise it returns the original state. Thus, it applies *tac* at most once.

`REPEAT_DETERM` *tac* applies *tac* to the proof state and, recursively, to the head of the resulting sequence. It returns the first state to make *tac* fail. It is deterministic, discarding alternative outcomes.

`REPEAT` *tac* applies *tac* to the proof state and, recursively, to each element of the resulting sequence. The resulting sequence consists of those states that make *tac* fail. Thus, it applies *tac* as many times as possible (including zero times), and allows backtracking over each invocation of *tac*. It is more general than `REPEAT_DETERM`, but requires more space.

`REPEAT1` *tac* is like `REPEAT` *tac* but it always applies *tac* at least once, failing if this is impossible.

`set trace_REPEAT;` enables an interactive tracing mode for the tacticals `REPEAT_DETERM` and `REPEAT`. To view the tracing options, type `h` at the prompt.

### 4.1.4 Identities for tacticals

```
all_tac : tactic
no_tac  : tactic
```

`all_tac` maps any proof state to the one-element sequence containing that state. Thus, it succeeds for all states. It is the identity element of the tactical `THEN`.

`no_tac` maps any proof state to the empty sequence. Thus it succeeds for no state. It is the identity element of `ORELSE`, `APPEND`, and `INTLEAVE`. Also, it is a zero element for `THEN`, which means that *tac* `THEN no_tac` is equivalent to `no_tac`.

These primitive tactics are useful when writing tacticals. For example, `TRY` and `REPEAT` (ignoring tracing) can be coded as follows:

```
fun TRY tac = tac ORELSE all_tac;

fun REPEAT tac =
     (fn state => ((tac THEN REPEAT tac) ORELSE all_tac) state);
```

If *tac* can return multiple outcomes then so can `REPEAT` *tac*. Since `REPEAT` uses `ORELSE` and not `APPEND` or `INTLEAVE`, it applies *tac* as many times as possible in each outcome.

**!** Note `REPEAT`'s explicit abstraction over the proof state. Recursive tacticals must be coded in this awkward fashion to avoid infinite recursion. With the following definition, `REPEAT` *tac* would loop due to ML's eager evaluation strategy:

```
fun REPEAT tac = (tac THEN REPEAT tac) ORELSE all_tac;
```

The built-in `REPEAT` avoids `THEN`, handling sequences explicitly and using tail recursion. This sacrifices clarity, but saves much space by discarding intermediate proof states.

## 4.2 Control and search tacticals

A predicate on theorems, namely a function of type `thm->bool`, can test whether a proof state enjoys some desirable property — such as having no subgoals. Tactics that search for satisfactory states are easy to express. The main search procedures, depth-first, breadth-first and best-first, are provided as tacticals. They generate the search tree by repeatedly applying a given tactic.

### 4.2.1 Filtering a tactic's results

```
FILTER  : (thm -> bool) -> tactic -> tactic
CHANGED : tactic -> tactic
```

`FILTER` *p tac* applies *tac* to the proof state and returns a sequence consisting of those result states that satisfy *p*.

`CHANGED` *tac* applies *tac* to the proof state and returns precisely those states that differ from the original state. Thus, `CHANGED` *tac* always has some effect on the state.

### 4.2.2 Depth-first search

```
DEPTH_FIRST   : (thm->bool) -> tactic -> tactic
DEPTH_SOLVE   :                tactic -> tactic
DEPTH_SOLVE_1 :                tactic -> tactic
trace_DEPTH_FIRST: bool ref
```
**initially false**

`DEPTH_FIRST` *satp tac* returns the proof state if *satp* returns true. Otherwise it applies *tac*, then recursively searches from each element of the resulting sequence. The code uses a stack for efficiency, in effect applying *tac* `THEN DEPTH_FIRST` *satp tac* to the state.

`DEPTH_SOLVE` *tac* uses `DEPTH_FIRST` to search for states having no subgoals.

`DEPTH_SOLVE_1` *tac* uses `DEPTH_FIRST` to search for states having fewer subgoals than the given state. Thus, it insists upon solving at least one subgoal.

`set trace_DEPTH_FIRST;` enables interactive tracing for `DEPTH_FIRST`. To view the tracing options, type `h` at the prompt.

### 4.2.3 Other search strategies

```
BREADTH_FIRST   :             (thm->bool) -> tactic -> tactic
BEST_FIRST      : (thm->bool)*(thm->int) -> tactic -> tactic
THEN_BEST_FIRST : tactic * ((thm->bool) * (thm->int) * tactic)
                  -> tactic
trace_BEST_FIRST: bool ref
```
**infix 1**

**initially false**

These search strategies will find a solution if one exists. However, they do not enumerate all solutions; they terminate after the first satisfactory result from *tac*.

`BREADTH_FIRST` *satp tac* uses breadth-first search to find states for which *satp* is true. For most applications, it is too slow.

`BEST_FIRST` $(satp, distf)$ *tac* does a heuristic search, using *distf* to estimate the distance from a satisfactory state. It maintains a list of states ordered by distance. It applies *tac* to the head of this list; if the result contains any satisfactory states, then it returns them. Otherwise, `BEST_FIRST` adds the new states to the list, and continues.

The distance function is typically `size_of_thm`, which computes the size of the state. The smaller the state, the fewer and simpler subgoals it has.

$tac_0$ `THEN_BEST_FIRST` $(satp, distf, tac)$ is like `BEST_FIRST`, except that the priority queue initially contains the result of applying $tac_0$ to the proof state. This tactical permits separate tactics for starting the search and continuing the search.

`set trace_BEST_FIRST;` enables an interactive tracing mode for the tactical `BEST_FIRST`. To view the tracing options, type `h` at the prompt.

### 4.2.4 Auxiliary tacticals for searching

```
COND        : (thm->bool) -> tactic -> tactic -> tactic
IF_UNSOLVED : tactic -> tactic
DETERM      : tactic -> tactic
```

`COND` $p$ $tac_1$ $tac_2$ applies $tac_1$ to the proof state if it satisfies $p$, and applies $tac_2$ otherwise. It is a conditional tactical in that only one of $tac_1$ and $tac_2$ is applied to a proof state. However, both $tac_1$ and $tac_2$ are evaluated because ML uses eager evaluation.

`IF_UNSOLVED` *tac* applies *tac* to the proof state if it has any subgoals, and simply returns the proof state otherwise. Many common tactics, such as `resolve_tac`, fail if applied to a proof state that has no subgoals.

`DETERM` *tac* applies *tac* to the proof state and returns the head of the resulting sequence. `DETERM` limits the search space by making its argument deterministic.

### 4.2.5 Predicates and functions useful for searching

```
has_fewer_prems : int -> thm -> bool
eq_thm          : thm * thm -> bool
size_of_thm     : thm -> int
```

`has_fewer_prems` $n$ *thm* reports whether *thm* has fewer than $n$ premises. By currying, `has_fewer_prems` $n$ is a predicate on theorems; it may be given to the searching tacticals.

`eq_thm (`*thm*_1`,` *thm*_2`)` reports whether *thm*_1 and *thm*_2 are equal. Both theorems must have identical signatures. Both theorems must have the same conclusions, and the same hypotheses, in the same order. Names of bound variables are ignored.

`size_of_thm` *thm* computes the size of *thm*, namely the number of variables, constants and abstractions in its conclusion. It may serve as a distance function for `BEST_FIRST`.

## 4.3 Tacticals for subgoal numbering

When conducting a backward proof, we normally consider one goal at a time. A tactic can affect the entire proof state, but many tactics — such as `resolve_tac` and `assume_tac` — work on a single subgoal. Subgoals are designated by a positive integer, so Isabelle provides tacticals for combining values of type `int->tactic`.

### 4.3.1 Restricting a tactic to one subgoal

```
SELECT_GOAL : tactic -> int -> tactic
METAHYPS    : (thm list -> tactic) -> int -> tactic
```

`SELECT_GOAL` *tac* $i$ restricts the effect of *tac* to subgoal $i$ of the proof state. It fails if there is no subgoal $i$, or if *tac* changes the main goal (do not use `rewrite_tac`). It applies *tac* to a dummy proof state and uses the result to refine the original proof state at subgoal $i$. If *tac* returns multiple results then so does `SELECT_GOAL` *tac* $i$.

`SELECT_GOAL` works by creating a state of the form $\phi \Longrightarrow \phi$, with the one subgoal $\phi$. If subgoal $i$ has the form $\psi \Longrightarrow \theta$ then $(\psi \Longrightarrow \theta) \Longrightarrow (\psi \Longrightarrow \theta)$ is in fact $[\![\psi \Longrightarrow \theta;\ \psi]\!] \Longrightarrow \theta$, a proof state with two subgoals. Such a proof state might cause tactics to go astray. Therefore `SELECT_GOAL` inserts a quantifier to create the state

$$(\bigwedge x\,.\,\psi \Longrightarrow \theta) \Longrightarrow (\bigwedge x\,.\,\psi \Longrightarrow \theta).$$

`METAHYPS` *tacf* $i$ takes subgoal $i$, of the form

$$\bigwedge x_1 \ldots x_l\,.\,[\![\theta_1; \ldots ; \theta_k]\!] \Longrightarrow \theta,$$

and creates the list $\theta'_1, \ldots, \theta'_k$ of meta-level assumptions. In these theorems, the subgoal's parameters ($x_1, \ldots, x_l$) become free variables. It supplies the assumptions to *tacf* and applies the resulting tactic to the proof state $\theta \Longrightarrow \theta$.

If the resulting proof state is $[\![\phi_1; \ldots ; \phi_n]\!] \Longrightarrow \phi$, possibly containing $\theta'_1, \ldots, \theta'_k$ as assumptions, then it is lifted back into the original context, yielding $n$ subgoals.

Meta-level assumptions may not contain unknowns. Unknowns in the hypotheses $\theta_1, \ldots, \theta_k$ become free variables in $\theta'_1, \ldots, \theta'_k$, and are restored afterwards; the `METAHYPS` call cannot instantiate them. Unknowns in $\theta$ may be instantiated. New unknowns in $\phi_1, \ldots, \phi_n$ are lifted over the parameters.

Here is a typical application. Calling `hyp_res_tac` $i$ resolves subgoal $i$ with one of its own assumptions, which may itself have the form of an inference rule (these are called **higher-level assumptions**).

```
val hyp_res_tac = METAHYPS (fn prems => resolve_tac prems 1);
```

The function `gethyps` is useful for debugging applications of `METAHYPS`.

**!** `METAHYPS` fails if the context or new subgoals contain type unknowns. In principle, the tactical could treat these like ordinary unknowns.

### 4.3.2 Scanning for a subgoal by number

```
ALLGOALS         : (int -> tactic) -> tactic
TRYALL           : (int -> tactic) -> tactic
SOMEGOAL         : (int -> tactic) -> tactic
FIRSTGOAL        : (int -> tactic) -> tactic
REPEAT_SOME      : (int -> tactic) -> tactic
REPEAT_FIRST     : (int -> tactic) -> tactic
trace_goalno_tac : (int -> tactic) -> int -> tactic
```

These apply a tactic function of type `int -> tactic` to all the subgoal numbers of a proof state, and join the resulting tactics using `THEN` or `ORELSE`. Thus, they apply the tactic to all the subgoals, or to one subgoal.

Suppose that the original proof state has $n$ subgoals.

`ALLGOALS` *tacf* is equivalent to *tacf*$(n)$ `THEN ... THEN` *tacf*$(1)$.

It applies *tacf* to all the subgoals, counting downwards (to avoid problems when subgoals are added or deleted).

`TRYALL` *tacf* is equivalent to `TRY`(*tacf*$(n)$) `THEN ... THEN` `TRY`(*tacf*$(1)$).

It attempts to apply *tacf* to all the subgoals. For instance, the tactic `TRYALL assume_tac` attempts to solve all the subgoals by assumption.

`SOMEGOAL` *tacf* is equivalent to *tacf*$(n)$ `ORELSE ... ORELSE` *tacf*$(1)$.

It applies *tacf* to one subgoal, counting downwards. For instance, the tactic `SOMEGOAL assume_tac` solves one subgoal by assumption, failing if this is impossible.

`FIRSTGOAL` *tacf* is equivalent to *tacf*$(1)$ `ORELSE ... ORELSE` *tacf*$(n)$.

It applies *tacf* to one subgoal, counting upwards.

`REPEAT_SOME` *tacf* applies *tacf* once or more to a subgoal, counting downwards.

`REPEAT_FIRST` *tacf* applies *tacf* once or more to a subgoal, counting upwards.

`trace_goalno_tac` *tac* $i$ applies *tac* $i$ to the proof state. If the resulting sequence is non-empty, then it is returned, with the side-effect of printing `Subgoal` $i$ `selected`. Otherwise, `trace_goalno_tac` returns the empty sequence and prints nothing.

It indicates that 'the tactic worked for subgoal $i$' and is mainly used with `SOMEGOAL` and `FIRSTGOAL`.

### 4.3.3 Joining tactic functions

```
THEN'     : ('a -> tactic) * ('a -> tactic) -> 'a -> tactic       infix 1
ORELSE'   : ('a -> tactic) * ('a -> tactic) -> 'a -> tactic         infix
APPEND'   : ('a -> tactic) * ('a -> tactic) -> 'a -> tactic         infix
INTLEAVE' : ('a -> tactic) * ('a -> tactic) -> 'a -> tactic         infix
EVERY'    : ('a -> tactic) list -> 'a -> tactic
FIRST'    : ('a -> tactic) list -> 'a -> tactic
```

These help to express tactics that specify subgoal numbers. The tactic

```
SOMEGOAL (fn i => resolve_tac rls i  ORELSE  eresolve_tac erls i)
```

can be simplified to

```
SOMEGOAL (resolve_tac rls  ORELSE'  eresolve_tac erls)
```

Note that `TRY'`, `REPEAT'`, `DEPTH_FIRST'`, etc. are not provided, because function composition accomplishes the same purpose. The tactic

```
ALLGOALS (fn i => REPEAT (etac exE i  ORELSE  atac i))
```

can be simplified to

```
ALLGOALS (REPEAT o (etac exE  ORELSE'  atac))
```

These tacticals are polymorphic; $x$ need not be an integer.

| | | |
|---|---|---|
| $(tacf_1$ `THEN'` $tacf_2)(x)$ | yields | $tacf_1(x)$ `THEN` $tacf_2(x)$ |
| $(tacf_1$ `ORELSE'` $tacf_2)(x)$ | yields | $tacf_1(x)$ `ORELSE` $tacf_2(x)$ |
| $(tacf_1$ `APPEND'` $tacf_2)(x)$ | yields | $tacf_1(x)$ `APPEND` $tacf_2(x)$ |
| $(tacf_1$ `INTLEAVE'` $tacf_2)(x)$ | yields | $tacf_1(x)$ `INTLEAVE` $tacf_2(x)$ |
| `EVERY'` $[tacf_1, \ldots, tacf_n]\ (x)$ | yields | `EVERY` $[tacf_1(x), \ldots, tacf_n(x)]$ |
| `FIRST'` $[tacf_1, \ldots, tacf_n]\ (x)$ | yields | `FIRST` $[tacf_1(x), \ldots, tacf_n(x)]$ |

### 4.3.4 Applying a list of tactics to 1

```
EVERY1: (int -> tactic) list -> tactic
FIRST1: (int -> tactic) list -> tactic
```

A common proof style is to treat the subgoals as a stack, always restricting attention to the first subgoal. Such proofs contain long lists of tactics, each applied to 1. These can be simplified using `EVERY1` and `FIRST1`:

| | | |
|---|---|---|
| `EVERY1` $[tacf_1, \ldots, tacf_n]$ | abbreviates | `EVERY` $[tacf_1(1), \ldots, tacf_n(1)]$ |
| `FIRST1` $[tacf_1, \ldots, tacf_n]$ | abbreviates | `FIRST` $[tacf_1(1), \ldots, tacf_n(1)]$ |

Chapter 5

# Theorems and Forward Proof

Theorems, which represent the axioms, theorems and rules of object-logics, have type `thm`. This chapter begins by describing operations that print theorems and that join them in forward proof. Most theorem operations are intended for advanced applications, such as programming new proof procedures. Many of these operations refer to signatures, certified terms and certified types, which have the ML types `Sign.sg`, `cterm` and `ctyp` and are discussed in Chapter 6. Beginning users should ignore such complexities — and skip all but the first section of this chapter.

The theorem operations do not print error messages. Instead, they raise exception `THM`. Use `print_exn` to display exceptions nicely:

```
allI RS mp  handle e => print_exn e;
  Exception THM raised:
  RSN: no unifiers -- premise 1
  (!!x. ?P(x)) ==> ALL x. ?P(x)
  [| ?P --> ?Q; ?P |] ==> ?Q

  uncaught exception THM
```

## 5.1 Basic operations on theorems

### 5.1.1 Pretty-printing a theorem

```
prth          : thm -> thm
prths         : thm list -> thm list
prthq         : thm Seq.seq -> thm Seq.seq
print_thm     : thm -> unit
print_goals   : int -> thm -> unit
string_of_thm : thm -> string
```

The first three commands are for interactive use. They are identity functions that display, then return, their argument. The ML identifier `it` will refer to the value just displayed.

The others are for use in programs. Functions with result type `unit` are convenient for imperative programming.

`prth` *thm* prints *thm* at the terminal.

`prths` *thms* prints *thms*, a list of theorems.

`prthq` *thmq* prints *thmq*, a sequence of theorems. It is useful for inspecting the output of a tactic.

`print_thm` *thm* prints *thm* at the terminal.

`print_goals` *limit thm* prints *thm* in goal style, with the premises as subgoals. It prints at most *limit* subgoals. The subgoal module calls `print_goals` to display proof states.

`string_of_thm` *thm* converts *thm* to a string.

### 5.1.2 Forward proof: joining rules by resolution

```
RSN : thm * (int * thm) -> thm                          infix
RS  : thm * thm -> thm                                  infix
MRS : thm list * thm -> thm                             infix
RLN : thm list * (int * thm list) -> thm list           infix
RL  : thm list * thm list -> thm list                   infix
MRL : thm list list * thm list -> thm list              infix
```

Joining rules together is a simple way of deriving new rules. These functions are especially useful with destruction rules. To store the result in the theorem database, use `bind_thm` (§2.1.3).

$thm_1$ `RSN` $(i, thm_2)$ resolves the conclusion of $thm_1$ with the $i$th premise of $thm_2$. Unless there is precisely one resolvent it raises exception `THM`; in that case, use `RLN`.

$thm_1$ `RS` $thm_2$ abbreviates $thm_1$ `RSN` $(1, thm_2)$. Thus, it resolves the conclusion of $thm_1$ with the first premise of $thm_2$.

$[thm_1, \ldots, thm_n]$ `MRS` *thm* uses `RSN` to resolve $thm_i$ against premise $i$ of *thm*, for $i = n, \ldots, 1$. This applies $thm_n, \ldots, thm_1$ to the first $n$ premises of *thm*. Because the theorems are used from right to left, it does not matter if the $thm_i$ create new premises. `MRS` is useful for expressing proof trees.

$thms_1$ `RLN` $(i, thms_2)$ joins lists of theorems. For every $thm_1$ in $thms_1$ and $thm_2$ in $thms_2$, it resolves the conclusion of $thm_1$ with the $i$th premise of $thm_2$, accumulating the results.

$thms_1$ `RL` $thms_2$ abbreviates $thms_1$ `RLN` $(1, thms_2)$.

$[thms_1, \ldots, thms_n]$ `MRL` *thms* is analogous to `MRS`, but combines theorem lists rather than theorems. It too is useful for expressing proof trees.

### 5.1.3 Expanding definitions in theorems

```
rewrite_rule       : thm list -> thm -> thm
rewrite_goals_rule : thm list -> thm -> thm
```

`rewrite_rule` *defs thm* unfolds the *defs* throughout the theorem *thm*.

`rewrite_goals_rule` *defs thm* unfolds the *defs* in the premises of *thm*, but leaves the conclusion unchanged. This rule underlies `rewrite_goals_tac`, but serves little purpose in forward proof.

### 5.1.4 Instantiating unknowns in a theorem

```
read_instantiate    :                   (string*string) list -> thm -> thm
read_instantiate_sg :     Sign.sg -> (string*string) list -> thm -> thm
cterm_instantiate   :                     (cterm*cterm) list -> thm -> thm
instantiate'       : ctyp option list -> cterm option list -> thm -> thm
```

These meta-rules instantiate type and term unknowns in a theorem. They are occasionally useful. They can prevent difficulties with higher-order unification, and define specialized versions of rules.

`read_instantiate` *insts thm* processes the instantiations *insts* and instantiates the rule *thm*. The processing of instantiations is described in §3.1.4, under `res_inst_tac`.

Use `res_inst_tac`, not `read_instantiate`, to instantiate a rule and refine a particular subgoal. The tactic allows instantiation by the subgoal's parameters, and reads the instantiations using the signature associated with the proof state.

Use `read_instantiate_sg` below if *insts* appears to be treated incorrectly.

`read_instantiate_sg` *sg insts thm* is like `read_instantiate` *insts thm*, but it reads the instantiations under signature *sg*. This is necessary to instantiate a rule from a general theory, such as first-order logic, using the notation of some specialized theory. Use the function `sign_of` to get a theory's signature.

`cterm_instantiate` *ctpairs thm* is similar to `read_instantiate`, but the instantiations are provided as pairs of certified terms, not as strings to be read.

`instantiate'` *ctyps cterms thm* instantiates *thm* according to the positional arguments *ctyps* and *cterms*. Counting from left to right, schematic variables $?x$ are either replaced by $t$ for any argument `Some` $t$, or left unchanged in case of `None` or if the end of the argument list is encountered. Types are instantiated before terms.

### 5.1.5 Miscellaneous forward rules

```
standard         :            thm -> thm
zero_var_indexes :            thm -> thm
make_elim        :            thm -> thm
rule_by_tactic   : tactic -> thm -> thm
rotate_prems     :    int -> thm -> thm
```

`standard` *thm* puts *thm* into the standard form of object-rules. It discharges all meta-assumptions, replaces free variables by schematic variables, renames schematic variables to have subscript zero, also strips outer (meta) quantifiers and removes dangling sort hypotheses.

`zero_var_indexes` *thm* makes all schematic variables have subscript zero, renaming them to avoid clashes.

`make_elim` *thm* converts *thm*, a destruction rule of the form $[\![P_1; \ldots; P_m]\!] \Longrightarrow Q$, to the elimination rule $[\![P_1; \ldots; P_m; Q \Longrightarrow R]\!] \Longrightarrow R$. This is the basis for destruct-resolution: `dresolve_tac`, etc.

`rule_by_tactic` *tac thm* applies *tac* to the *thm*, freezing its variables first, then yields the proof state returned by the tactic. In typical usage, the *thm* represents an instance of a rule with several premises, some with contradictory assumptions (because of the instantiation). The tactic proves those subgoals and does whatever else it can, and returns whatever is left.

`rotate_prems` *k thm* rotates the premises of *thm* to the left by *k* positions. It requires $0 \leq k \leq n$, where $n$ is the number of premises; the rotation has no effect if $k$ is at either extreme. Used with `eresolve_tac`, it gives the effect of applying the tactic to some other premise of *thm* than the first.

### 5.1.6 Taking a theorem apart

```
cprop_of      : thm -> cterm
concl_of      : thm -> term
prems_of      : thm -> term list
cprems_of     : thm -> cterm list
nprems_of     : thm -> int
tpairs_of     : thm -> (term*term) list
sign_of_thm   : thm -> Sign.sg
theory_of_thm : thm -> theory
dest_state    : thm * int -> (term*term) list * term list * term * term
rep_thm       : thm -> {sign_ref: Sign.sg_ref, der: deriv, maxidx: int,
                        shyps: sort list, hyps: term list, prop: term}
crep_thm      : thm -> {sign_ref: Sign.sg_ref, der: deriv, maxidx: int,
                        shyps: sort list, hyps: cterm list, prop: cterm}
```

`cprop_of` *thm* returns the statement of *thm* as a certified term.

`concl_of` *thm* returns the conclusion of *thm* as a term.

`prems_of` *thm* returns the premises of *thm* as a list of terms.

`cprems_of` *thm* returns the premises of *thm* as a list of certified terms.

`nprems_of` *thm* returns the number of premises in *thm*, and is equivalent to `length (prems_of` *thm*`)`.

`tpairs_of` *thm* returns the flex-flex constraints of *thm*.

`sign_of_thm` *thm* returns the signature associated with *thm*.

`theory_of_thm` *thm* returns the theory associated with *thm*. Note that this does a lookup in Isabelle's global database of loaded theories.

`dest_state` $(thm, i)$ decomposes *thm* as a tuple containing a list of flex-flex constraints, a list of the subgoals 1 to $i-1$, subgoal $i$, and the rest of the theorem (this will be an implication if there are more than $i$ subgoals).

`rep_thm` *thm* decomposes *thm* as a record containing the statement of *thm* (`prop`), its list of meta-assumptions (`hyps`), its derivation (`der`), a bound on the maximum subscript of its unknowns (`maxidx`), and a reference to its signature (`sign_ref`). The `shyps` field is discussed below.

`crep_thm` *thm* like `rep_thm`, but returns the hypotheses and statement as certified terms.

### 5.1.7 *Sort hypotheses

```
force_strip_shyps : bool ref                    initially true
```

`force_strip_shyps` causes sort hypotheses to be deleted, printing a warning.

Isabelle's type variables are decorated with sorts, constraining them to certain ranges of types. This has little impact when sorts only serve for syntactic classification of types — for example, FOL distinguishes between terms and other types. But when type classes are introduced through axioms, this may result in some sorts becoming *empty*: where one cannot exhibit a type belonging to it because certain sets of axioms are unsatisfiable.

If a theorem contains a type variable that is constrained by an empty sort, then that theorem has no instances. It is basically an instance of *ex falso quodlibet*. But what if it is used to prove another theorem that no longer involves that sort? The latter theorem holds only if under an additional non-emptiness assumption.

Therefore, Isabelle's theorems carry around sort hypotheses. The `shyps` field is a list of sorts occurring in type variables in the current `prop` and `hyps` fields. It may also includes sorts used in the theorem's proof that no longer appear in the `prop` or `hyps` fields — so-called *dangling* sort constraints. These are the critical ones, asserting non-emptiness of the corresponding sorts.

Isabelle tries to remove extraneous sorts from the `shyps` field whenever non-emptiness can be established by looking at the theorem's signature: from the `arities` information, etc. Because its current implementation is highly incomplete, the flag shown above is available. Setting it to true (the default) allows existing proofs to run.

### 5.1.8 Tracing flags for unification

```
Unify.trace_simp   : bool ref
Unify.trace_types  : bool ref
Unify.trace_bound  : int ref
Unify.search_bound : int ref
```

| | |
|---|---|
| `Unify.trace_simp : bool ref` | **initially false** |
| `Unify.trace_types : bool ref` | **initially false** |
| `Unify.trace_bound : int ref` | **initially 10** |
| `Unify.search_bound : int ref` | **initially 20** |

Tracing the search may be useful when higher-order unification behaves unexpectedly. Letting `res_inst_tac` circumvent the problem is easier, though.

`set Unify.trace_simp;` causes tracing of the simplification phase.

`set Unify.trace_types;` generates warnings of incompleteness, when unification is not considering all possible instantiations of type unknowns.

`Unify.trace_bound :=` $n$`;` causes unification to print tracing information once it reaches depth $n$. Use $n = 0$ for full tracing. At the default value of 10, tracing information is almost never printed.

`Unify.search_bound :=` $n$`;` causes unification to limit its search to depth $n$. Because of this bound, higher-order unification cannot return an infinite sequence, though it can return a very long one. The search rarely approaches the default value of 20. If the search is cut off, unification prints a warning `Unification bound exceeded`.

## 5.2 *Primitive meta-level inference rules

These implement the meta-logic in the style of the LCF system, as functions from theorems to theorems. They are, rarely, useful for deriving results in the pure theory. Mainly, they are included for completeness, and most users should not bother with them. The meta-rules raise exception `THM` to signal malformed premises, incompatible signatures and similar errors.

The meta-logic uses natural deduction. Each theorem may depend on meta-level assumptions. Certain rules, such as $(\Longrightarrow I)$, discharge assumptions; in most other rules, the conclusion depends on all of the assumptions of the premises. Formally, the system works with assertions of the form

$$\phi \quad [\phi_1, \ldots, \phi_n],$$

where $\phi_1$, ..., $\phi_n$ are the assumptions. This can be also read as a single conclusion sequent $\phi_1, \ldots, \phi_n \vdash \phi$. Do not confuse meta-level assumptions with the object-level assumptions in a subgoal, which are represented in the meta-logic using $\Longrightarrow$.

Each theorem has a signature. Certified terms have a signature. When a rule takes several premises and certified terms, it merges the signatures to make a signature for the conclusion. This fails if the signatures are incompatible.

The **implication** rules are ($\Longrightarrow I$) and ($\Longrightarrow E$):

$$\frac{\begin{array}{c}[\phi]\\ \vdots\\ \psi\end{array}}{\phi \Longrightarrow \psi}\,(\Longrightarrow I) \qquad \frac{\phi \Longrightarrow \psi \quad \phi}{\psi}\,(\Longrightarrow E)$$

Equality of truth values means logical equivalence:

$$\frac{\phi \Longrightarrow \psi \quad \psi \Longrightarrow \phi}{\phi \equiv \psi}\,(\equiv I) \qquad \frac{\phi \equiv \psi \quad \phi}{\psi}\,(\equiv E)$$

The **equality** rules are reflexivity, symmetry, and transitivity:

$$a \equiv a\,(\mathit{refl}) \qquad \frac{a \equiv b}{b \equiv a}\,(\mathit{sym}) \qquad \frac{a \equiv b \quad b \equiv c}{a \equiv c}\,(\mathit{trans})$$

The $\lambda$-conversions are $\alpha$-conversion, $\beta$-conversion, and extensionality:[1]

$$(\lambda x\,.\,a) \equiv (\lambda y\,.\,a[y/x]) \qquad ((\lambda x\,.\,a)(b)) \equiv a[b/x] \qquad \frac{f(x) \equiv g(x)}{f \equiv g}\,(\mathit{ext})$$

The **abstraction** and **combination** rules let conversions be applied to subterms:[2]

$$\frac{a \equiv b}{(\lambda x\,.\,a) \equiv (\lambda x\,.\,b)}\,(\mathit{abs}) \qquad \frac{f \equiv g \quad a \equiv b}{f(a) \equiv g(b)}\,(\mathit{comb})$$

The **universal quantification** rules are ($\bigwedge I$) and ($\bigwedge E$):[3]

$$\frac{\phi}{\bigwedge x\,.\,\phi}\,(\bigwedge I) \qquad \frac{\bigwedge x\,.\,\phi}{\phi[b/x]}\,(\bigwedge E)$$

[1] $\alpha$-conversion holds if $y$ is not free in $a$; ($\mathit{ext}$) holds if $x$ is not free in the assumptions, $f$, or $g$.

[2] Abstraction holds if $x$ is not free in the assumptions.

[3] ($\bigwedge I$) holds if $x$ is not free in the assumptions.

### 5.2.1 Assumption rule

```
assume: cterm -> thm
```

`assume` *ct* makes the theorem $\phi\ [\phi]$, where $\phi$ is the value of *ct*. The rule checks that *ct* has type *prop* and contains no unknowns, which are not allowed in assumptions.

### 5.2.2 Implication rules

```
implies_intr      : cterm -> thm -> thm
implies_intr_list : cterm list -> thm -> thm
implies_intr_hyps : thm -> thm
implies_elim      : thm -> thm -> thm
implies_elim_list : thm -> thm list -> thm
```

`implies_intr` *ct thm* is $(\Longrightarrow I)$, where *ct* is the assumption to discharge, say $\phi$. It maps the premise $\psi$ to the conclusion $\phi \Longrightarrow \psi$, removing all occurrences of $\phi$ from the assumptions. The rule checks that *ct* has type *prop*.

`implies_intr_list` *cts thm* applies $(\Longrightarrow I)$ repeatedly, on every element of the list *cts*.

`implies_intr_hyps` *thm* applies $(\Longrightarrow I)$ to discharge all the hypotheses (assumptions) of *thm*. It maps the premise $\phi\ [\phi_1, \ldots, \phi_n]$ to the conclusion $[\![\phi_1, \ldots, \phi_n]\!] \Longrightarrow \phi$.

`implies_elim` $thm_1$ $thm_2$ applies $(\Longrightarrow E)$ to $thm_1$ and $thm_2$. It maps the premises $\phi \Longrightarrow \psi$ and $\phi$ to the conclusion $\psi$.

`implies_elim_list` *thm thms* applies $(\Longrightarrow E)$ repeatedly to *thm*, using each element of *thms* in turn. It maps the premises $[\![\phi_1, \ldots, \phi_n]\!] \Longrightarrow \psi$ and $\phi_1,\ldots,\phi_n$ to the conclusion $\psi$.

### 5.2.3 Logical equivalence rules

```
equal_intr : thm -> thm -> thm
equal_elim : thm -> thm -> thm
```

`equal_intr` $thm_1$ $thm_2$ applies $(\equiv I)$ to $thm_1$ and $thm_2$. It maps the premises $\psi$ and $\phi$ to the conclusion $\phi \equiv \psi$; the assumptions are those of the first premise with $\phi$ removed, plus those of the second premise with $\psi$ removed.

`equal_elim` $thm_1$ $thm_2$ applies $(\equiv E)$ to $thm_1$ and $thm_2$. It maps the premises $\phi \equiv \psi$ and $\phi$ to the conclusion $\psi$.

### 5.2.4 Equality rules

```
reflexive  : cterm -> thm
symmetric  : thm -> thm
transitive : thm -> thm -> thm
```

`reflexive` *ct* makes the theorem $ct \equiv ct$.

`symmetric` *thm* maps the premise $a \equiv b$ to the conclusion $b \equiv a$.

`transitive` $thm_1$ $thm_2$ maps the premises $a \equiv b$ and $b \equiv c$ to the conclusion $a \equiv c$.

### 5.2.5 The $\lambda$-conversion rules

```
beta_conversion : cterm -> thm
extensional     : thm -> thm
abstract_rule   : string -> cterm -> thm -> thm
combination     : thm -> thm -> thm
```

There is no rule for $\alpha$-conversion because Isabelle regards $\alpha$-convertible theorems as equal.

`beta_conversion` *ct* makes the theorem $((\lambda x\,.\,a)(b)) \equiv a[b/x]$, where *ct* is the term $(\lambda x\,.\,a)(b)$.

`extensional` *thm* maps the premise $f(x) \equiv g(x)$ to the conclusion $f \equiv g$. Parameter $x$ is taken from the premise. It may be an unknown or a free variable (provided it does not occur in the assumptions); it must not occur in $f$ or $g$.

`abstract_rule` $v$ $x$ *thm* maps the premise $a \equiv b$ to the conclusion $(\lambda x\,.\,a) \equiv (\lambda x\,.\,b)$, abstracting over all occurrences (if any!) of $x$. Parameter $x$ is supplied as a cterm. It may be an unknown or a free variable (provided it does not occur in the assumptions). In the conclusion, the bound variable is named $v$.

`combination` $thm_1$ $thm_2$ maps the premises $f \equiv g$ and $a \equiv b$ to the conclusion $f(a) \equiv g(b)$.

### 5.2.6 Forall introduction rules

```
forall_intr       : cterm      -> thm -> thm
forall_intr_list  : cterm list -> thm -> thm
forall_intr_frees :               thm -> thm
```

`forall_intr` $x$ *thm* applies $(\bigwedge I)$, abstracting over all occurrences (if any!) of $x$. The rule maps the premise $\phi$ to the conclusion $\bigwedge x\,.\,\phi$. Parameter $x$ is supplied as a cterm. It may be an unknown or a free variable (provided it does not occur in the assumptions).

`forall_intr_list` *xs thm* applies $(\bigwedge I)$ repeatedly, on every element of the list *xs*.

`forall_intr_frees` *thm* applies $(\bigwedge I)$ repeatedly, generalizing over all the free variables of the premise.

### 5.2.7 Forall elimination rules

```
forall_elim       : cterm      -> thm -> thm
forall_elim_list  : cterm list -> thm -> thm
forall_elim_var   :        int -> thm -> thm
forall_elim_vars  :        int -> thm -> thm
```

`forall_elim` *ct thm* applies $(\bigwedge E)$, mapping the premise $\bigwedge x.\phi$ to the conclusion $\phi[ct/x]$. The rule checks that *ct* and *x* have the same type.

`forall_elim_list` *cts thm* applies $(\bigwedge E)$ repeatedly, on every element of the list *cts*.

`forall_elim_var` *k thm* applies $(\bigwedge E)$, mapping the premise $\bigwedge x . \phi$ to the conclusion $\phi[?x_k/x]$. Thus, it replaces the outermost $\bigwedge$-bound variable by an unknown having subscript $k$.

`forall_elim_vars` *ks thm* applies `forall_elim_var` repeatedly, for every element of the list *ks*.

### 5.2.8 Instantiation of unknowns

```
instantiate: (indexname * ctyp) list * (cterm * cterm) list -> thm -> thm
```

`instantiate (`*tyinsts*`,` *insts*`)` *thm* simultaneously substitutes types for type unknowns (the *tyinsts*) and terms for term unknowns (the *insts*). Instantiations are given as $(v, t)$ pairs, where $v$ is an unknown and $t$ is a term (of the same type as $v$) or a type (of the same sort as $v$). All the unknowns must be distinct. The rule normalizes its conclusion.

Note that `instantiate'` (see §5.1.4) provides a more convenient interface to this rule.

### 5.2.9 Freezing/thawing type unknowns

```
freezeT: thm -> thm
varifyT: thm -> thm
```

`freezeT` *thm* converts all the type unknowns in *thm* to free type variables.

`varifyT` *thm* converts all the free type variables in *thm* to type unknowns.

## 5.3 Derived rules for goal-directed proof

Most of these rules have the sole purpose of implementing particular tactics. There are few occasions for applying them directly to a theorem.

### 5.3.1 Proof by assumption

```
assumption    : int -> thm -> thm Seq.seq
eq_assumption : int -> thm -> thm
```

`assumption` *i thm* attempts to solve premise *i* of *thm* by assumption.

`eq_assumption` is like `assumption` but does not use unification.

### 5.3.2 Resolution

```
biresolution : bool -> (bool*thm)list -> int -> thm
               -> thm Seq.seq
```

`biresolution` *match rules i state* performs bi-resolution on subgoal *i* of *state*, using the list of (*flag*, *rule*) pairs. For each pair, it applies resolution if the flag is `false` and elim-resolution if the flag is `true`. If *match* is `true`, the *state* is not instantiated.

### 5.3.3 Composition: resolution without lifting

```
compose   : thm * int * thm -> thm list
COMP      : thm * thm -> thm
bicompose : bool -> bool * thm * int -> int -> thm
            -> thm Seq.seq
```

In forward proof, a typical use of composition is to regard an assertion of the form $\phi \Longrightarrow \psi$ as atomic. Schematic variables are not renamed, so beware of clashes!

`compose` `(`$thm_1$`,` $i$`,` $thm_2$`)` uses $thm_1$, regarded as an atomic formula, to solve premise $i$ of $thm_2$. Let $thm_1$ and $thm_2$ be $\psi$ and $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$. For each $s$ that unifies $\psi$ and $\phi_i$, the result list contains the theorem

$$([\![\phi_1; \ldots; \phi_{i-1}; \phi_{i+1}; \ldots; \phi_n]\!] \Longrightarrow \phi)s.$$

$thm_1$ `COMP` $thm_2$ calls `compose` `(`$thm_1$`, 1,` $thm_2$`)` and returns the result, if unique; otherwise, it raises exception `THM`. It is analogous to `RS`.

For example, suppose that $thm_1$ is $a = b \Longrightarrow b = a$, a symmetry rule, and that $thm_2$ is $[\![P \Longrightarrow Q; \neg Q]\!] \Longrightarrow \neg P$, which is the principle of contrapositives. Then the result would be the derived rule $\neg(b = a) \Longrightarrow \neg(a = b)$.

`bicompose` *match* `(`*flag*`,` *rule*`,` *m*`)` *i* *state* refines subgoal $i$ of *state* using *rule*, without lifting. The *rule* is taken to have the form $[\![\psi_1; \ldots ; \psi_m]\!] \Longrightarrow \psi$, where $\psi$ need not be atomic; thus $m$ determines the number of new subgoals. If *flag* is `true` then it performs elim-resolution — it solves the first premise of *rule* by assumption and deletes that assumption. If *match* is `true`, the *state* is not instantiated.

### 5.3.4 Other meta-rules

```
trivial           : cterm -> thm
lift_rule         : (thm * int) -> thm -> thm
rename_params_rule : string list * int -> thm -> thm
flexflex_rule     : thm -> thm Seq.seq
```

`trivial` *ct* makes the theorem $\phi \Longrightarrow \phi$, where $\phi$ is the value of *ct*. This is the initial state for a goal-directed proof of $\phi$. The rule checks that *ct* has type *prop*.

`lift_rule (`*state*`,` *i*`)` *rule* prepares *rule* for resolution by lifting it over the parameters and assumptions of subgoal $i$ of *state*.

`rename_params_rule (`*names*`,` *i*`)` *thm* uses the *names* to rename the parameters of premise $i$ of *thm*. The names must be distinct. If there are fewer names than parameters, then the rule renames the innermost parameters and may modify the remaining ones to ensure that all the parameters are distinct.

`flexflex_rule` *thm* removes all flex-flex pairs from *thm* using the trivial unifier.

## 5.4 Proof objects

Isabelle can record the full meta-level proof of each theorem. The proof object contains all logical inferences in detail, while omitting bookkeeping steps that have no logical meaning to an outside observer. Rewriting steps are recorded in similar detail as the output of simplifier tracing. The proof object can be inspected by a separate proof-checker, for example.

Full proof objects are large. They multiply storage requirements by about seven; attempts to build large logics (such as ZF and HOL) may fail. Isabelle normally builds minimal proof objects, which include only uses of oracles. You can also request an intermediate level of detail, containing uses of oracles, axioms and theorems. These smaller proof objects indicate a theorem's dependencies.

Isabelle provides proof objects for the sake of transparency. Their aim is to increase your confidence in Isabelle. They let you inspect proofs constructed by the classical reasoner or simplifier, and inform you of all uses of oracles. Seldom will proof objects be given whole to an automatic proof-checker: none has been written. It is up to you to examine and interpret them sensibly. For example, when scrutinizing a theorem's derivation for dependence upon some oracle or axiom, remember to scrutinize all of its lemmas. Their proofs are included in the main derivation, through the `Theorem` constructor.

Proof objects are expressed using a polymorphic type of variable-branching trees. Proof objects (formally known as *derivations*) are trees labelled by rules, where `rule` is a complicated datatype declared in the file `Pure/thm.ML`.

```
datatype 'a mtree = Join of 'a * 'a mtree list;
datatype rule     = ...;
type deriv        = rule mtree;
```

Each theorem's derivation is stored as the `der` field of its internal record:

```
#der (rep_thm conjI);
  Join (Theorem "HOL.conjI", [Join (MinProof,[])]) : deriv
```

This proof object identifies a labelled theorem, `conjI` of theory `HOL`, whose underlying proof has not been recorded; all we have is `MinProof`.

Nontrivial proof objects are unreadably large and complex. Isabelle provides several functions to help you inspect them informally. These functions omit the more obscure inferences and attempt to restructure the others into natural formats, linear or tree-structured.

```
keep_derivs  : deriv_kind ref
Deriv.size   : deriv -> int
Deriv.drop   : 'a mtree * int -> 'a mtree
Deriv.linear : deriv -> deriv list
Deriv.tree   : deriv -> Deriv.orule mtree
```

`keep_derivs := MinDeriv | ThmDeriv | FullDeriv;` specifies one of the options for keeping derivations. They can be minimal (oracles only), include theorems and axioms, or be full.

`Deriv.size` *der* yields the size of a derivation, excluding lemmas.

`Deriv.drop` (*tree*,*n*) returns the subtree *n* levels down, always following the first child. It is good for stripping off outer level inferences that are used to put a theorem into standard form.

`Deriv.linear` *der* converts a derivation into a linear format, replacing the deep nesting by a list of rules. Intuitively, this reveals the single-step Isabelle proof that is constructed internally by tactics.

`Deriv.tree` *der* converts a derivation into an object-level proof tree. A resolution by an object-rule is converted to a tree node labelled by that rule.

> Complications arise if the object-rule is itself derived in some way. Nested resolutions are unravelled, but other operations on rules (such as rewriting) are left as-is.

Functions `Deriv.linear` and `Deriv.tree` omit the proof of any named theorems (constructor `Theorem`) they encounter in a derivation. Applying them directly to the derivation of a named theorem is therefore pointless. Use `Deriv.drop` with argument 1 to skip over the initial `Theorem` constructor.

**Chapter 6**

# Theories, Terms and Types

Theories organize the syntax, declarations and axioms of a mathematical development. They are built, starting from the `Pure` or `CPure` theory, by extending and merging existing theories. They have the ML type `theory`. Theory operations signal errors by raising exception `THEORY`, returning a message and a list of theories.

Signatures, which contain information about sorts, types, constants and syntax, have the ML type `Sign.sg`. For identification, each signature carries a unique list of **stamps**, which are ML references to strings. The strings serve as human-readable names; the references serve as unique identifiers. Each primitive signature has a single stamp. When two signatures are merged, their lists of stamps are also merged. Every theory carries a unique signature.

Terms and types are the underlying representation of logical syntax. Their ML definitions are irrelevant to naive Isabelle users. Programmers who wish to extend Isabelle may need to know such details, say to code a tactic that looks for subgoals of a particular form. Terms and types may be 'certified' to be well-formed with respect to a given signature.

## 6.1 Defining theories

Theories are usually defined using theory definition files (which have a name suffix `.thy`). There is also a low level interface provided by certain ML functions (see §6.4.2). Appendix A presents the concrete syntax for theory definitions; here is an explanation of the constituent parts:

*theoryDef* is the full definition. The new theory is called *id*. It is the union of the named **parent theories**, possibly extended with new components. `Pure` and `CPure` are the basic theories, which contain only the meta-logic. They differ just in their concrete syntax for function applications.

Normally each *name* is an identifier, the name of the parent theory. Quoted strings can be used to document additional file dependencies; see §6.2 for details.

*classes* is a series of class declarations. Declaring *id* `<` $id_1$ ... $id_n$ makes *id* a subclass of the existing classes $id_1 \ldots id_n$. This rules out cyclic class structures. Isabelle automatically computes the transitive closure of subclass hierarchies; it is not necessary to declare `c < e` in addition to `c < d` and `d < e`.

*default* introduces *sort* as the new default sort for type variables. This applies to unconstrained type variables in an input string but not to type variables created internally. If omitted, the default sort is the listwise union of the default sorts of the parent theories (i.e. their logical intersection).

*sort* is a finite set of classes. A single class *id* abbreviates the sort *–id"*.

*types* is a series of type declarations. Each declares a new type constructor or type synonym. An $n$-place type constructor is specified by $(\alpha_1, \ldots, \alpha_n)name$, where the type variables serve only to indicate the number $n$.

A **type synonym** is an abbreviation $(\alpha_1, \ldots, \alpha_n)name = \tau$, where *name* and $\tau$ can be strings.

*infix* declares a type or constant to be an infix operator of priority *nat* associating to the left (`infixl`) or right (`infixr`). Only 2-place type constructors can have infix status; an example is `('a,'b) "*" (infixr 20)`, which may express binary product types.

*arities* is a series of type arity declarations. Each assigns arities to type constructors. The *name* must be an existing type constructor, which is given the additional arity *arity*.

*nonterminals* declares purely syntactic types to be used as nonterminal symbols of the context free grammar.

*consts* is a series of constant declarations. Each new constant *name* is given the specified type. The optional *mixfix* annotations may attach concrete syntax to the constant.

*syntax* is a variant of *consts* which adds just syntax without actually declaring logical constants. This gives full control over a theory's context free grammar. The optional *mode* specifies the print mode where the mixfix productions should be added. If there is no `output` option given, all productions are also added to the input syntax (regardless of the print mode).

*mixfix* annotations can take three forms:

- A mixfix template given as a *string* of the form `"..._..._..."` where the $i$-th underscore indicates the position where the $i$-th argument should go. The list of numbers gives the priority of each argument. The final number gives the priority of the whole construct.

- A constant $f$ of type $\tau_1 \Rightarrow (\tau_2 \Rightarrow \tau)$ can be given **infix** status.
- A constant $f$ of type $(\tau_1 \Rightarrow \tau_2) \Rightarrow \tau$ can be given **binder** status. The declaration `binder` $\mathcal{Q}$ $p$ causes $\mathcal{Q}\, x\, .\, F(x)$ to be treated like $f(F)$, where $p$ is the priority.

*trans* specifies syntactic translation rules (macros). There are three forms: parse rules (`=>`), print rules (`<=`), and parse/print rules (`==`).

*rules* is a series of rule declarations. Each has a name *id* and the formula is given by the *string*. Rule names must be distinct within any single theory.

*defs* is a series of definitions. They are just like *rules*, except that every *string* must be a definition (see below for details).

*constdefs* combines the declaration of constants and their definition. The first *string* is the type, the second the definition.

*axclass* 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, which is automatically employed by *instance* to prove instantiations of this class.

*instance* proves class inclusions or type arities at the logical level and then transfers these to the type signature. The instantiation is proven and checked properly. The user has to supply sufficient witness information: theorems (*longident*), axioms (*string*), or even arbitrary ML tactic code *verbatim*.

*oracle* links the theory to a trusted external reasoner. It is allowed to create theorems, but each theorem carries a proof object describing the oracle invocation. See §6.10 for details.

*local*, *global* change the current name declaration mode. Initially, theories start in *local* mode, causing all names of types, constants, axioms etc. to be automatically qualified by the theory name. Changing this to *global* causes all names to be declared as short base names only.

The *local* and *global* declarations act like switches, affecting all following theory sections until changed again explicitly. Also note that the final state at the end of the theory will persist. In particular, this determines how the names of theorems stored later on are handled.

*setup* applies a list of ML functions to the theory. The argument should denote a value of type `(theory -> theory) list`. Typically, ML packages are initialized in this way.

*ml* consists of ML code, typically for parse and print translation functions.

Chapters 7 and 8 explain mixfix declarations, translation rules and the `ML` section in more detail.

### 6.1.1 Definitions

**Definitions** are intended to express abbreviations. The simplest form of a definition is $f \equiv t$, where $f$ is a constant. Isabelle also allows a derived forms where the arguments of $f$ appear on the left, abbreviating a string of $\lambda$-abstractions.

Isabelle makes the following checks on definitions:

- 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 definitions such as `(zero::nat) == length ([]::'a list)`.
- The definition must not be recursive. Most object-logics provide definitional principles that can be used to express recursion safely.

These checks are intended to catch the sort of errors that might be made accidentally. Misspellings, for instance, might result in additional variables appearing on the right-hand side. More elaborate checks could be made, but the cost might be overly strict rules on declaration order, etc.

### 6.1.2 *Classes and arities

In order to guarantee principal types [7], arity declarations must obey two conditions:

- There must not be any two declarations $ty :: (\vec{r})c$ and $ty :: (\vec{s})c$ with $\vec{r} \neq \vec{s}$. For example, this excludes the following:

  ```
  arities
    foo :: ({logic}) logic
    foo :: ({})logic
  ```

- If there are two declarations $ty :: (s_1, \ldots, s_n)c$ and $ty :: (s'_1, \ldots, s'_n)c'$ such that $c' < c$ then $s'_i \preceq s_i$ must hold for $i = 1, \ldots, n$. The relationship $\preceq$, defined as
$$s' \preceq s \iff \forall c \in s \, . \, \exists c' \in s' \, . \; c' \leq c,$$

expresses that the set of types represented by $s'$ is a subset of the set of types represented by $s$. Assuming *term* $\preceq$ *logic*, the following is forbidden:

```
arities
  foo :: ({logic})logic
  foo :: ({})term
```

## 6.2 Loading a new theory

```
use_thy         : string -> unit
time_use_thy    : string -> unit
loadpath        : string list ref                    initially ["."]
delete_tmpfiles : bool ref                           initially true
```

`use_thy` *thyname* reads the theory *thyname* and creates an ML structure as described below.

`time_use_thy` *thyname* calls `use_thy` *thyname* and reports the time taken.

`loadpath` contains a list of directories to search when locating the files that define a theory. This list is only used if the theory name in `use_thy` does not specify the path explicitly.

`reset delete_tmpfiles;` suppresses the deletion of temporary files.

Each theory definition must reside in a separate file. Let the file $T$`.thy` contain the definition of a theory called $T$, whose parent theories are $TB_1 \ldots TB_n$. Calling `use_thy "`$T$`"` reads the file $T$`.thy`, writes a temporary ML file `.`$T$`.thy.ML`, and reads the latter file. Recursive `use_thy` calls load those parent theories that have not been loaded previously; the recursive calls may continue to any depth. One `use_thy` call can read an entire logic provided all theories are linked appropriately.

The result is an ML structure $T$ containing at least a component `thy` for the new theory and components for each of the rules. The structure also contains the definitions of the `ML` section, if present. The file `.`$T$`.thy.ML` is then deleted if `delete_tmpfiles` is set and no errors occurred.

Finally the file $T$`.ML` is read, if it exists. The structure $T$ is automatically open in this context. Proof scripts typically refer to its components by unqualified names.

Some applications construct theories directly by calling ML functions. In this situation there is no `.thy` file, only an `.ML` file. The `.ML` file must declare an ML structure having the theory's name and a component `thy` containing the new theory object. Section 6.3.1 below describes a way of linking such theories to their parents.

## 6.3 Reloading modified theories

```
update     : unit -> unit
unlink_thy : string -> unit
```

Changing a theory on disk often makes it necessary to reload all theories descended from it. However, `use_thy` reads only one theory, even if some of the parent theories are out of date. In this case you should call `update()`.

Isabelle keeps track of all loaded theories and their files. If `use_thy` finds that the theory to be loaded has been read before, it determines whether to reload the theory as follows. First it looks for the theory's files in their previous location. If it finds them, it compares their modification times to the internal data and stops if they are equal. If the files have been moved, `use_thy` searches for them as it would for a new theory. After `use_thy` reloads a theory, it marks the children as out-of-date.

`update()` reloads all modified theories and their descendants in the correct order.

`unlink_thy` *thyname* informs Isabelle that theory *thyname* no longer exists. If you delete the theory files for *thyname* then you must execute `unlink_thy`; otherwise `update` will complain about a missing file.

### 6.3.1 *Pseudo theories

Any automatic reloading facility requires complete knowledge of all dependencies. Sometimes theories depend on objects created in ML files with no associated theory definition file. These objects may be theories but they could also be theorems, proof procedures, etc.

Unless such dependencies are documented, `update` fails to reload these ML files and the system is left in a state where some objects, such as theorems, still refer to old versions of theories. This may lead to the error

```
Attempt to merge different versions of theories: ...
```

Therefore there is a way to link theories and **orphaned** ML files — those not associated with a theory definition.

Let us assume we have an orphaned ML file named `orphan.ML` and a theory $B$ that depends on `orphan.ML` — for example, `B.ML` uses theorems proved in `orphan.ML`. Then `B.thy` should mention this dependency as follows:

```
B = ... + "orphan" + ...
```

Quoted strings stand for theories which have to be loaded before the current theory is read but which are not used in building the base of theory $B$. Whenever `orphan` changes and is reloaded, Isabelle knows that $B$ has to be updated, too.

Note that it's necessary for `orphan` to declare a special ML object of type `theory` which is present in all theories. This is normally achieved by adding the file `orphan.thy` to make `orphan` a **pseudo theory**. A minimum version of `orphan.thy` would be

```
orphan = Pure
```

which uses `Pure` to make a dummy theory. Normally though the orphaned file has its own dependencies. If `orphan.ML` depends on theories or files $A_1$, ..., $A_n$, record this by creating the pseudo theory in the following way:

```
orphan = A1 + ... + An
```

The resulting theory ensures that `update` reloads `orphan` whenever it reloads one of the $A_i$.

For an extensive example of how this technique can be used to link lots of theory files and load them by just a few `use_thy` calls see the sources of one of the major object-logics (e.g. `ZF`).

# 6.4 Basic operations on theories

## 6.4.1 *Theory inclusion

```
subthy      : theory * theory -> bool
eq_thy      : theory * theory -> bool
transfer    : theory -> thm -> thm
transfer_sg : Sign.sg -> thm -> thm
```

Inclusion and equality of theories is determined by unique identification stamps that are created when declaring new components. Theorems contain a reference to the theory (actually to its signature) they have been derived in. Transferring theorems to super theories has no logical significance, but may affect some operations in subtle ways (e.g. implicit merges of signatures when applying rules, or pretty printing of theorems).

`subthy` `(`$thy_1$`,` $thy_2$`)` determines if $thy_1$ is included in $thy_2$ wrt. identification stamps.

`eq_thy` `(`$thy_1$`,` $thy_2$`)` determines if $thy_1$ is exactly the same as $thy_2$.

`transfer` *thy thm* transfers theorem *thm* to theory *thy*, provided the latter includes the theory of *thm*.

`transfer_sg` *sign thm* is similar to `transfer`, but identifies the super theory via its signature.

### 6.4.2 *Building a theory

```
ProtoPure.thy  : theory
Pure.thy       : theory
CPure.thy      : theory
merge_theories : string -> theory * theory -> theory
```

`ProtoPure.thy`, `Pure.thy`, `CPure.thy` contain the syntax and signature of the meta-logic. There are basically no axioms: meta-level inferences are carried out by ML functions. `Pure` and `CPure` just differ in their concrete syntax of prefix function application: $t(u_1, \ldots, u_n)$ in `Pure` vs. $t\, u_1, \ldots\, u_n$ in `CPure`. `ProtoPure` is their common parent, containing no syntax for printing prefix applications at all!

`merge_theories` *name* ($thy_1$, $thy_2$) merges the two theories $thy_1$ and $thy_2$, creating a new named theory node. The resulting theory contains all of the syntax, signature and axioms of the constituent theories. Merging theories that contain different identification stamps of the same name fails with the following message

```
Attempt to merge different versions of theories: "T1", ..., "Tn"
```

This error may especially occur when a theory is redeclared — say to change an inappropriate definition — and bindings to old versions persist. Isabelle ensures that old and new theories of the same name are not involved in a proof.

### 6.4.3 Inspecting a theory

```
print_syntax        : theory -> unit
print_theory        : theory -> unit
parents_of          : theory -> theory list
ancestors_of        : theory -> theory list
sign_of             : theory -> Sign.sg
Sign.stamp_names_of : Sign.sg -> string list
```

These provide means of viewing a theory's components.

`print_syntax` *thy* prints the syntax of *thy* (grammar, macros, translation functions etc., see page 73 for more details).

`print_theory` *thy* prints the logical parts of *thy*, excluding the syntax.

`parents_of` *thy* returns the direct ancestors of *thy*.

`ancestors_of` *thy* returns all ancestors of *thy* (not including *thy* itself).

`sign_of` *thy* returns the signature associated with *thy*. It is useful with functions like `read_instantiate_sg`, which take a signature as an argument.

`Sign.stamp_names_of` *sg* returns the names of the identification stamps of ax signature. These coincide with the names of its full ancestry including that of *sg* itself.

## 6.5 Terms

Terms belong to the ML type `term`, which is a concrete datatype with six constructors:

```
type indexname = string * int;
infix 9 $;
datatype term = Const of string * typ
              | Free  of string * typ
              | Var   of indexname * typ
              | Bound of int
              | Abs   of string * typ * term
              | op $  of term * term;
```

`Const` $(a,\ T)$ is the **constant** with name $a$ and type $T$. Constants include connectives like $\wedge$ and $\forall$ as well as constants like 0 and $Suc$. Other constants may be required to define a logic's concrete syntax.

`Free` $(a,\ T)$ is the **free variable** with name $a$ and type $T$.

`Var` $(v,\ T)$ is the **scheme variable** with indexname $v$ and type $T$. An `indexname` is a string paired with a non-negative index, or subscript; a term's scheme variables can be systematically renamed by incrementing their subscripts. Scheme variables are essentially free variables, but may be instantiated during unification.

`Bound` $i$ is the **bound variable** with de Bruijn index $i$, which counts the number of lambdas, starting from zero, between a variable's occurrence and its binding. The representation prevents capture of variables. For more information see de Bruijn [3] or Paulson [9, page 336].

`Abs` $(a,\ T,\ u)$ is the **$\lambda$-abstraction** with body $u$, and whose bound variable has name $a$ and type $T$. The name is used only for parsing and printing; it has no logical significance.

$t$ `$` $u$ is the **application** of $t$ to $u$.

Application is written as an infix operator to aid readability. Here is an ML pattern to recognize FOL formulae of the form $A \to B$, binding the subformulae to $A$ and $B$:

```
Const("Trueprop",_) $ (Const("op -->",_) $ A $ B)
```

## 6.6 *Variable binding

```
loose_bnos     : term -> int list
incr_boundvars : int -> term -> term
abstract_over  : term*term -> term
variant_abs    : string * typ * term -> string * term
aconv          : term * term -> bool                      infix
```

These functions are all concerned with the de Bruijn representation of bound variables.

`loose_bnos` $t$ returns the list of all dangling bound variable references. In particular, `Bound 0` is loose unless it is enclosed in an abstraction. Similarly `Bound 1` is loose unless it is enclosed in at least two abstractions; if enclosed in just one, the list will contain the number 0. A well-formed term does not contain any loose variables.

`incr_boundvars` $j$ increases a term's dangling bound variables by the offset $j$. This is required when moving a subterm into a context where it is enclosed by a different number of abstractions. Bound variables with a matching abstraction are unaffected.

`abstract_over` $(v, t)$ forms the abstraction of $t$ over $v$, which may be any well-formed term. It replaces every occurrence of $v$ by a `Bound` variable with the correct index.

`variant_abs` $(a, T, u)$ substitutes into $u$, which should be the body of an abstraction. It replaces each occurrence of the outermost bound variable by a free variable. The free variable has type $T$ and its name is a variant of $a$ chosen to be distinct from all constants and from all variables free in $u$.

$t$ `aconv` $u$ tests whether terms $t$ and $u$ are $\alpha$-convertible: identical up to renaming of bound variables.

- Two constants, `Free`s, or `Var`s are $\alpha$-convertible if their names and types are equal. (Variables having the same name but different types are thus distinct. This confusing situation should be avoided!)
- Two bound variables are $\alpha$-convertible if they have the same number.
- Two abstractions are $\alpha$-convertible if their bodies are, and their bound variables have the same type.

- Two applications are $\alpha$-convertible if the corresponding subterms are.

## 6.7 Certified terms

A term $t$ can be **certified** under a signature to ensure that every type in $t$ is well-formed and every constant in $t$ is a type instance of a constant declared in the signature. The term must be well-typed and its use of bound variables must be well-formed. Meta-rules such as `forall_elim` take certified terms as arguments.

Certified terms belong to the abstract type `cterm`. Elements of the type can only be created through the certification process. In case of error, Isabelle raises exception `TERM`.

### 6.7.1 Printing terms

```
     string_of_cterm :           cterm -> string
Sign.string_of_term  : Sign.sg -> term -> string
```

`string_of_cterm` *ct* displays *ct* as a string.

`Sign.string_of_term` *sign t* displays $t$ as a string, using the syntax of *sign*.

### 6.7.2 Making and inspecting certified terms

```
cterm_of          : Sign.sg -> term -> cterm
read_cterm        : Sign.sg -> string * typ -> cterm
cert_axm          : Sign.sg -> string * term -> string * term
read_axm          : Sign.sg -> string * string -> string * term
rep_cterm         : cterm -> {T:typ, t:term, sign:Sign.sg, maxidx:int}
Sign.certify_term : Sign.sg -> term -> term * typ * int
```

`cterm_of` *sign t* certifies $t$ with respect to signature *sign*.

`read_cterm` *sign* `(`*s*`,` *T*`)` reads the string $s$ using the syntax of *sign*, creating a certified term. The term is checked to have type $T$; this type also tells the parser what kind of phrase to parse.

`cert_axm` *sign* `(`*name*`,` *t*`)` certifies $t$ with respect to *sign* as a meta-proposition and converts all exceptions to an error, including the final message

```
The error(s) above occurred in axiom "name"
```

`read_axm` *sign* `(`*name*`,` *s*`)` similar to `cert_axm`, but first reads the string $s$ using the syntax of *sign*.

`rep_cterm` *ct* decomposes *ct* as a record containing its type, the term itself, its signature, and the maximum subscript of its unknowns. The type and maximum subscript are computed during certification.

`Sign.certify_term` is a more primitive version of `cterm_of`, returning the internal representation instead of an abstract `cterm`.

## 6.8 Types

Types belong to the ML type `typ`, which is a concrete datatype with three constructor functions. These correspond to type constructors, free type variables and schematic type variables. Types are classified by sorts, which are lists of classes (representing an intersection). A class is represented by a string.

```
type class = string;
type sort  = class list;

datatype typ = Type  of string * typ list
             | TFree of string * sort
             | TVar  of indexname * sort;

infixr 5 -->;
fun S --> T = Type ("fun", [S, T]);
```

`Type (`*a*`, `*Ts*`)` applies the **type constructor** named *a* to the type operands *Ts*. Type constructors include *fun*, the binary function space constructor, as well as nullary type constructors such as *prop*. Other type constructors may be introduced. In expressions, but not in patterns, *S*`-->`*T* is a convenient shorthand for function types.

`TFree (`*a*`, `*s*`)` is the **type variable** with name *a* and sort *s*.

`TVar (`*v*`, `*s*`)` is the **type unknown** with indexname *v* and sort *s*. Type unknowns are essentially free type variables, but may be instantiated during unification.

## 6.9 Certified types

Certified types, which are analogous to certified terms, have type `ctyp`.

### 6.9.1 Printing types

```
     string_of_ctyp :           ctyp -> string
Sign.string_of_typ  : Sign.sg -> typ -> string
```

`string_of_ctyp` *cT* displays *cT* as a string.

`Sign.string_of_typ` *sign T* displays *T* as a string, using the syntax of *sign*.

### 6.9.2 Making and inspecting certified types

```
ctyp_of           : Sign.sg -> typ -> ctyp
rep_ctyp          : ctyp -> {T: typ, sign: Sign.sg}
Sign.certify_typ : Sign.sg -> typ -> typ
```

`ctyp_of` *sign T* certifies *T* with respect to signature *sign*.

`rep_ctyp` *cT* decomposes *cT* as a record containing the type itself and its signature.

`Sign.certify_typ` is a more primitive version of `ctyp_of`, returning the internal representation instead of an abstract `ctyp`.

## 6.10 Oracles: calling trusted external reasoners

Oracles allow Isabelle to take advantage of external reasoners such as arithmetic decision procedures, model checkers, fast tautology checkers or computer algebra systems. Invoked as an oracle, an external reasoner can create arbitrary Isabelle theorems. It is your responsibility to ensure that the external reasoner is as trustworthy as your application requires. Isabelle's proof objects (§5.4) record how each theorem depends upon oracle calls.

```
invoke_oracle     : theory -> xstring -> Sign.sg * object -> thm
Theory.add_oracle : bstring * (Sign.sg * object -> term) -> theory
                    -> theory
```

`invoke_oracle` *thy name* `(`*sign*`,` *data*`)` invokes the oracle *name* of theory *thy* passing the information contained in the exception value *data* and creating a theorem having signature *sign*. Note that type `object` is just an abbreviation for `exn`. Errors arise if *thy* does not have an oracle called *name*, if the oracle rejects its arguments or if its result is ill-typed.

`Theory.add_oracle` *name fun thy* extends *thy* by oracle *fun* called *name*. It is seldom called explicitly, as there is concrete syntax for oracles in theory files.

A curious feature of ML exceptions is that they are ordinary constructors. The ML type `exn` is a datatype that can be extended at any time. (See my *ML for the Working Programmer* [10], especially page 136.) The oracle mechanism takes advantage of this to allow an oracle to take any information whatever.

There must be some way of invoking the external reasoner from ML, either because it is coded in ML or via an operating system interface. Isabelle expects the ML function to take two arguments: a signature and an exception object.

- The signature will typically be that of a desendant of the theory declaring the oracle. The oracle will use it to distinguish constants from variables, etc., and it will be attached to the generated theorems.
- The exception is used to pass arbitrary information to the oracle. This information must contain a full description of the problem to be solved by the external reasoner, including any additional information that might be required. The oracle may raise the exception to indicate that it cannot solve the specified problem.

A trivial example is provided in theory `FOL/ex/IffOracle`. This oracle generates tautologies of the form $P \leftrightarrow \cdots \leftrightarrow P$, with an even number of $P$s.

The `ML` section of `IffOracle.thy` begins by declaring a few auxiliary functions (suppressed below) for creating the tautologies. Then it declares a new exception constructor for the information required by the oracle: here, just an integer. It finally defines the oracle function itself.

```
exception IffOracleExn of int;

fun mk_iff_oracle (sign, IffOracleExn n) =
  if n > 0 andalso n mod 2 = 0
  then Trueprop $ mk_iff n
  else raise IffOracleExn n;
```

Observe the function's two arguments, the signature `sign` and the exception given as a pattern. The function checks its argument for validity. If $n$ is positive and even then it creates a tautology containing $n$ occurrences of $P$. Otherwise it signals error by raising its own exception (just by happy coincidence). Errors may be signalled by other means, such as returning the theorem `True`. Please ensure that the oracle's result is correctly typed; Isabelle will reject ill-typed theorems by raising a cryptic exception at top level.

The `oracle` section of `IffOracle.thy` installs above `ML` function as follows:

```
IffOracle = FOL +

oracle
  iff = mk_iff_oracle

end
```

Now in `IffOracle.ML` we first define a wrapper for invoking the oracle:

```
fun iff_oracle n = invoke_oracle IffOracle.thy "iff"
                      (sign_of IffOracle.thy, IffOracleExn n);
```

Here are some example applications of the `iff` oracle. An argument of 10 is allowed, but one of 5 is forbidden:

```
iff_oracle 10;
   "P <-> P <-> P <-> P <-> P <-> P <-> P <-> P <-> P <-> P" : thm
iff_oracle 5;
  Exception- IffOracleExn 5 raised
```

Chapter 7

# Defining Logics

This chapter explains how to define new formal systems — in particular, their concrete syntax. While Isabelle can be regarded as a theorem prover for set theory, higher-order logic or the sequent calculus, its distinguishing feature is support for the definition of new logics.

Isabelle logics are hierarchies of theories, which are described and illustrated in *Introduction to Isabelle*. That material, together with the theory files provided in the examples directories, should suffice for all simple applications. The easiest way to define a new theory is by modifying a copy of an existing theory.

This chapter documents the meta-logic syntax, mixfix declarations and pretty printing. The extended examples in §7.6 demonstrate the logical aspects of the definition of theories.

## 7.1 Priority grammars

A context-free grammar contains a set of **nonterminal symbols**, a set of **terminal symbols** and a set of **productions**. Productions have the form $A = \gamma$, where $A$ is a nonterminal and $\gamma$ is a string of terminals and nonterminals. One designated nonterminal is called the **start symbol**. The language defined by the grammar consists of all strings of terminals that can be derived from the start symbol by applying productions as rewrite rules.

The syntax of an Isabelle logic is specified by a **priority grammar**. Each nonterminal is decorated by an integer priority, as in $A^{(p)}$. A nonterminal $A^{(p)}$ in a derivation may be rewritten using a production $A^{(q)} = \gamma$ only if $p \leq q$. Any priority grammar can be translated into a normal context free grammar by introducing new nonterminals and productions.

Formally, a set of context free productions $G$ induces a derivation relation $\longrightarrow_G$. Let $\alpha$ and $\beta$ denote strings of terminal or nonterminal symbols. Then

$$\alpha\, A^{(p)}\, \beta \;\longrightarrow_G\; \alpha\, \gamma\, \beta$$

if and only if $G$ contains some production $A^{(q)} = \gamma$ for $p \leq q$.

The following simple grammar for arithmetic expressions demonstrates how binding power and associativity of operators can be enforced by priorities.

$$\begin{array}{rcl}
A^{(9)} & = & \mathtt{0} \\
A^{(9)} & = & \mathtt{(}\ A^{(0)}\ \mathtt{)} \\
A^{(0)} & = & A^{(0)}\ \mathtt{+}\ A^{(1)} \\
A^{(2)} & = & A^{(3)}\ \mathtt{*}\ A^{(2)} \\
A^{(3)} & = & \mathtt{-}\ A^{(3)}
\end{array}$$

The choice of priorities determines that `-` binds tighter than `*`, which binds tighter than `+`. Furthermore `+` associates to the left and `*` to the right.

For clarity, grammars obey these conventions:

- All priorities must lie between 0 and `max_pri`, which is a some fixed integer. Sometimes `max_pri` is written as $\infty$.
- Priority 0 on the right-hand side and priority `max_pri` on the left-hand side may be omitted.
- The production $A^{(p)} = \alpha$ is written as $A = \alpha\ (p)$; the priority of the left-hand side actually appears in a column on the far right.
- Alternatives are separated by $|$.
- Repetition is indicated by dots $(\dots)$ in an informal but obvious way.

Using these conventions and assuming $\infty = 9$, the grammar takes the form

$$\begin{array}{rcll}
A & = & \mathtt{0} & \\
& | & \mathtt{(}\ A\ \mathtt{)} & \\
& | & A\ \mathtt{+}\ A^{(1)} & (0) \\
& | & A^{(3)}\ \mathtt{*}\ A^{(2)} & (2) \\
& | & \mathtt{-}\ A^{(3)} & (3)
\end{array}$$

## 7.2 The Pure syntax

At the root of all object-logics lies the theory `Pure`. It contains, among many other things, the Pure syntax. An informal account of this basic syntax (types, terms and formulae) appears in *Introduction to Isabelle*. A more precise description using a priority grammar appears in Fig. 7.1. It defines the following nonterminals:

`any` denotes any term.

`prop` denotes terms of type `prop`. These are formulae of the meta-logic. Note that user constants of result type `prop` (i.e. $c :: \ldots \Rightarrow prop$) should always provide concrete syntax. Otherwise atomic propositions with head $c$ may be printed incorrectly.

$$
\begin{array}{rcll}
\mathit{any} & = & \mathit{prop} \quad \mid \quad \mathit{logic} & \\[1ex]
\mathit{prop} & = & \texttt{(}\ \mathit{prop}\ \texttt{)} & \\
 & \mid & \mathit{prop}^{(4)}\ \texttt{::}\ \mathit{type} & (3) \\
 & \mid & \texttt{PROP}\ \mathit{aprop} & \\
 & \mid & \mathit{any}^{(3)}\ \texttt{==}\ \mathit{any}^{(2)} & (2) \\
 & \mid & \mathit{any}^{(3)}\ \texttt{=?=}\ \mathit{any}^{(2)} & (2) \\
 & \mid & \mathit{prop}^{(2)}\ \texttt{==>}\ \mathit{prop}^{(1)} & (1) \\
 & \mid & \texttt{[|}\ \mathit{prop}\ \texttt{;}\ \ldots\texttt{;}\ \mathit{prop}\ \texttt{|]}\ \texttt{==>}\ \mathit{prop}^{(1)} & (1) \\
 & \mid & \texttt{!!}\ \mathit{idts}\ \texttt{.}\ \mathit{prop} & (0) \\
 & \mid & \texttt{OFCLASS}\ \texttt{(}\ \mathit{type}\ \texttt{,}\ \mathit{logic}\ \texttt{)} & \\[1ex]
\mathit{aprop} & = & \mathit{id} \quad \mid \quad \mathit{longid} \quad \mid \quad \mathit{var} \quad \mid \quad \mathit{logic}^{(\infty)}\ \texttt{(}\ \mathit{any}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{any}\ \texttt{)} & \\[1ex]
\mathit{logic} & = & \texttt{(}\ \mathit{logic}\ \texttt{)} & \\
 & \mid & \mathit{logic}^{(4)}\ \texttt{::}\ \mathit{type} & (3) \\
 & \mid & \mathit{id} \quad \mid \quad \mathit{longid} \quad \mid \quad \mathit{var} \quad \mid \quad \mathit{logic}^{(\infty)}\ \texttt{(}\ \mathit{any}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{any}\ \texttt{)} & \\
 & \mid & \texttt{\%}\ \mathit{pttrns}\ \texttt{.}\ \mathit{any}^{(3)} & (3) \\[1ex]
\mathit{idts} & = & \mathit{idt} \quad \mid \quad \mathit{idt}^{(1)}\ \mathit{idts} & \\[1ex]
\mathit{idt} & = & \mathit{id} \quad \mid \quad \texttt{(}\ \mathit{idt}\ \texttt{)} & \\
 & \mid & \mathit{id}\ \texttt{::}\ \mathit{type} & (0) \\[1ex]
\mathit{pttrns} & = & \mathit{pttrn} \quad \mid \quad \mathit{pttrn}^{(1)}\ \mathit{pttrns} & \\[1ex]
\mathit{pttrn} & = & \mathit{idt} & \\[1ex]
\mathit{type} & = & \texttt{(}\ \mathit{type}\ \texttt{)} & \\
 & \mid & \mathit{tid} \quad \mid \quad \mathit{tvar} \quad \mid \quad \mathit{tid}\ \texttt{::}\ \mathit{sort} \quad \mid \quad \mathit{tvar}\ \texttt{::}\ \mathit{sort} & \\
 & \mid & \mathit{id} \quad \mid \quad \mathit{type}^{(\infty)}\ \mathit{id} \quad \mid \quad \texttt{(}\ \mathit{type}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{type}\ \texttt{)}\ \mathit{id} & \\
 & \mid & \mathit{longid} \quad \mid \quad \mathit{type}^{(\infty)}\ \mathit{longid} \quad \mid \quad \texttt{(}\ \mathit{type}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{type}\ \texttt{)}\ \mathit{longid} & \\
 & \mid & \mathit{type}^{(1)}\ \texttt{=>}\ \mathit{type} & (0) \\
 & \mid & \texttt{[}\ \mathit{type}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{type}\ \texttt{]}\ \texttt{=>}\ \mathit{type} & (0) \\[1ex]
\mathit{sort} & = & \mathit{id} \quad \mid \quad \mathit{longid} \quad \mid \quad \texttt{\{\}} \quad \mid \quad \texttt{\{}\ \mathit{id} \mid \mathit{longid}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{id} \mid \mathit{longid}\ \texttt{\}} &
\end{array}
$$

Figure 7.1: Meta-logic syntax

`aprop` denotes atomic propositions.

`logic` denotes terms whose type belongs to class `logic`, excluding type *prop*.

`idts` denotes a list of identifiers, possibly constrained by types.

`pttrn, pttrns` denote patterns for abstraction, cases etc. Initially the same as *idt* and *idts*, these are indetended to be augmented by user extensions.

`type` denotes types of the meta-logic.

`sort` denotes meta-level sorts.

**!** In `idts`, note that `x::nat y` is parsed as `x::(nat y)`, treating `y` like a type constructor applied to `nat`. The likely result is an error message. To avoid this interpretation, use parentheses and write `(x::nat) y`.

Similarly, `x::nat y::nat` is parsed as `x::(nat y::nat)` and yields an error. The correct form is `(x::nat) (y::nat)`.

**!** Type constraints bind very weakly. For example, `x<y::nat` is normally parsed as `(x<y)::nat`, unless `<` has priority of 3 or less, in which case the string is likely to be ambiguous. The correct form is `x<(y::nat)`.

### 7.2.1 Logical types and default syntax

Isabelle's representation of mathematical languages is based on the simply typed $\lambda$-calculus. All logical types, namely those of class `logic`, are automatically equipped with a basic syntax of types, identifiers, variables, parentheses, $\lambda$-abstraction and application.

**!** Isabelle combines the syntaxes for all types of class `logic` by mapping all those types to the single nonterminal *logic*. Thus all productions of *logic*, in particular *id*, *var* etc, become available.

### 7.2.2 Lexical matters

The parser does not process input strings directly. It operates on token lists provided by Isabelle's **lexer**. There are two kinds of tokens: **delimiters** and **name tokens**.

Delimiters can be regarded as reserved words of the syntax. You can add new ones when extending theories. In Fig. 7.1 they appear in typewriter font, for example `==`, `=?=` and `PROP`.

Name tokens have a predefined syntax. The lexer distinguishes six disjoint classes of names: identifiers, unknowns, type identifiers, type unknowns, numerals, strings. They are denoted by `id`, `var`, `tid`, `tvar`, `xnum`, `xstr`, respectively.

Typical examples are `x`, `?x7`, `'a`, `?'a3`, `#42`, `''foo bar''`. Here is the precise syntax:

$$
\begin{array}{rcl}
\mathit{id} & = & \mathit{letter}\ \mathit{quasiletter}^* \\
\mathit{longid} & = & \mathit{id}\,\texttt{.}\,\mathit{id}\ \ldots\ \mathit{id} \\
\mathit{var} & = & \texttt{?}\mathit{id} \ \mid\ \texttt{?}\mathit{id}\,\texttt{.}\,\mathit{nat} \\
\mathit{tid} & = & \texttt{'}\mathit{id} \\
\mathit{tvar} & = & \texttt{?}\mathit{tid} \ \mid\ \texttt{?}\mathit{tid}\,\texttt{.}\,\mathit{nat} \\
\mathit{xnum} & = & \texttt{\#}\mathit{nat} \ \mid\ \texttt{\#\~{}}\mathit{nat} \\
\mathit{xstr} & = & \texttt{''}\mathrm{text}\texttt{''} \\
\\
\mathit{letter} & = & \text{one of } \texttt{a}\ldots\texttt{z}\ \texttt{A}\ldots\texttt{Z} \\
\mathit{digit} & = & \text{one of } \texttt{0}\ldots\texttt{9} \\
\mathit{quasiletter} & = & \mathit{letter} \ \mid\ \mathit{digit} \ \mid\ \texttt{\_} \ \mid\ \texttt{'} \\
\mathit{nat} & = & \mathit{digit}^+
\end{array}
$$

The lexer repeatedly takes the longest prefix of the input string that forms a valid token. A maximal prefix that is both a delimiter and a name is treated as a delimiter. Spaces, tabs, newlines and formfeeds are separators; they never occur within tokens, except those of class *xstr*.

Delimiters need not be separated by white space. For example, if `-` is a delimiter but `--` is not, then the string `--` is treated as two consecutive occurrences of the token `-`. In contrast, ML treats `--` as a single symbolic name. The consequence of Isabelle's more liberal scheme is that the same string may be parsed in different ways after extending the syntax: after adding `--` as a delimiter, the input `--` is treated as a single token.

A `var` or `tvar` describes an unknown, which is internally a pair of base name and index (ML type `indexname`). These components are either separated by a dot as in `?x.1` or `?x7.3` or run together as in `?x1`. The latter form is possible if the base name does not end with digits. If the index is 0, it may be dropped altogether: `?x` abbreviates both `?x0` and `?x.0`.

Tokens of class *xnum* or *xstr* are not used by the meta-logic. Object-logics may provide numerals and string constants by adding appropriate productions and translation functions.

Although name tokens are returned from the lexer rather than the parser, it is more logical to regard them as nonterminals. Delimiters, however, are terminals; they are just syntactic sugar and contribute nothing to the abstract syntax tree.

### 7.2.3 *Inspecting the syntax

```
syn_of              : theory -> Syntax.syntax
print_syntax        : theory -> unit
Syntax.print_syntax : Syntax.syntax -> unit
Syntax.print_gram   : Syntax.syntax -> unit
Syntax.print_trans  : Syntax.syntax -> unit
```

The abstract type `Syntax.syntax` allows manipulation of syntaxes in ML. You can display values of this type by calling the following functions:

`syn_of` *thy* returns the syntax of the Isabelle theory *thy* as an ML value.

`print_syntax` *thy* displays the syntax part of *thy* using `Syntax.print_syntax`.

`Syntax.print_syntax` *syn* shows virtually all information contained in the syntax *syn*. The displayed output can be large. The following two functions are more selective.

`Syntax.print_gram` *syn* shows the grammar part of *syn*, namely the lexicon, logical types and productions. These are discussed below.

`Syntax.print_trans` *syn* shows the translation part of *syn*, namely the constants, parse/print macros and parse/print translations.

Let us demonstrate these functions by inspecting Pure's syntax. Even that is too verbose to display in full.

```
Syntax.print_syntax (syn_of Pure.thy);
  lexicon: "!!" "%" "(" ")" "," "." "::" ";" "==" "==>" ...
  logtypes: fun itself
  prods:
    type = tid  (1000)
    type = tvar  (1000)
    type = id  (1000)
    type = tid "::" sort[0]  => "_ofsort" (1000)
    type = tvar "::" sort[0]  => "_ofsort" (1000)
    ⋮
  print modes: "symbols" "xterm"
  consts: "_K" "_appl" "_aprop" "_args" "_asms" "_bigimpl" ...
  parse_ast_translation: "_appl" "_bigimpl" "_bracket"
    "_idtyp" "_lambda" "_tapp" "_tappl"
  parse_rules:
  parse_translation: "!!" "_K" "_abs" "_aprop"
  print_translation: "all"
  print_rules:
  print_ast_translation: "==>" "_abs" "_idts" "fun"
```

As you can see, the output is divided into labelled sections. The grammar is represented by `lexicon`, `logtypes` and `prods`. The rest refers to syntactic translations and macro expansion. Here is an explanation of the various sections.

`lexicon` lists the delimiters used for lexical analysis.

`logtypes` lists the types that are regarded the same as `logic` syntactically. Thus types of object-logics (e.g. `nat`, say) will be automatically equipped with the standard syntax of $\lambda$-calculus.

`prods` lists the productions of the priority grammar. The nonterminal $A^{(n)}$ is rendered in ASCII as `A[`$n$`]`. Each delimiter is quoted. Some productions are shown with `=>` and an attached string. These strings later become the heads of parse trees; they also play a vital role when terms are printed (see §8.1).

Productions with no strings attached are called **copy productions**. Their right-hand side must have exactly one nonterminal symbol (or name token). The parser does not create a new parse tree node for copy productions, but simply returns the parse tree of the right-hand symbol.

If the right-hand side consists of a single nonterminal with no delimiters, then the copy production is called a **chain production**. Chain productions act as abbreviations: conceptually, they are removed from the grammar by adding new productions. Priority information attached to chain productions is ignored; only the dummy value $-1$ is displayed.

`print_modes` lists the alternative print modes provided by this syntax (see §7.4).

`consts`, `parse_rules`, `print_rules` relate to macros (see §8.5).

`parse_ast_translation`, `print_ast_translation` list sets of constants that invoke translation functions for abstract syntax trees. Section §8.1 below discusses this obscure matter.

`parse_translation`, `print_translation` list the sets of constants that invoke translation functions for terms (see §8.6).

## 7.3 Mixfix declarations

When defining a theory, you declare new constants by giving their names, their type, and an optional **mixfix annotation**. Mixfix annotations allow you to extend Isabelle's basic $\lambda$-calculus syntax with readable notation. They can express any context-free priority grammar. Isabelle syntax definitions are inspired by OBJ [4]; they are more general than the priority declarations of ML and Prolog.

A mixfix annotation defines a production of the priority grammar. It describes the concrete syntax, the translation to abstract syntax, and the pretty printing. Special case annotations provide a simple means of specifying infix operators and binders.

### 7.3.1 The general mixfix form

Here is a detailed account of mixfix declarations. Suppose the following line occurs within a `consts` or `syntax` section of a `.thy` file:

$$c \; \texttt{::} \; \texttt{"}\sigma\texttt{"} \; \texttt{("}\textit{template}\texttt{"} \; ps \; p\texttt{)}$$

This constant declaration and mixfix annotation are interpreted as follows:

- The string $c$ is the name of the constant associated with the production; unless it is a valid identifier, it must be enclosed in quotes. If $c$ is empty (given as `""`) then this is a copy production. Otherwise, parsing an instance of the phrase *template* generates the AST `("`$c$`"` $a_1$ ... $a_n$`)`, where $a_i$ is the AST generated by parsing the $i$-th argument.

- The constant $c$, if non-empty, is declared to have type $\sigma$ (`consts` section only).

- The string *template* specifies the right-hand side of the production. It has the form

$$w_0 \; \texttt{\_} \; w_1 \; \texttt{\_} \; \ldots \; \texttt{\_} \; w_n,$$

  where each occurrence of `_` denotes an argument position and the $w_i$ do not contain `_`. (If you want a literal `_` in the concrete syntax, you must escape it as described below.) The $w_i$ may consist of delimiters, spaces or pretty printing annotations (see below).

- The type $\sigma$ specifies the production's nonterminal symbols (or name tokens). If *template* is of the form above then $\sigma$ must be a function type with at least $n$ argument positions, say $\sigma = [\tau_1, \ldots, \tau_n] \Rightarrow \tau$. Nonterminal symbols are derived from the types $\tau_1, \ldots, \tau_n, \tau$ as described below. Any of these may be function types.

- The optional list $ps$ may contain at most $n$ integers, say `[`$p_1$`,` ...`,` $p_m$`]`, where $p_i$ is the minimal priority required of any phrase that may appear as the $i$-th argument. Missing priorities default to 0.

- The integer $p$ is the priority of this production. If omitted, it defaults to the maximal priority. Priorities range between 0 and `max_pri` (= 1000).

The resulting production is

$$A^{(p)} = w_0 \, A_1^{(p_1)} \, w_1 \, A_2^{(p_2)} \, \ldots \, A_n^{(p_n)} \, w_n$$

where $A$ and the $A_i$ are the nonterminals corresponding to the types $\tau$ and $\tau_i$ respectively. The nonterminal symbol associated with a type $(\ldots)ty$ is `logic`, if this is a logical type (namely one of class `logic` excluding `prop`). Otherwise it is *ty* (note that only the outermost type constructor is taken into account). Finally, the nonterminal of a type variable is `any`.

**!** Theories must sometimes declare types for purely syntactic purposes — merely playing the role of nonterminals. One example is *type*, the built-in type of types. This is a 'type of all types' in the syntactic sense only. Do not declare such types under `arities` as belonging to class `logic`, for that would make them useless as separate nonterminal symbols.

Associating nonterminals with types allows a constant's type to specify syntax as well. We can declare the function $f$ to have type $[\tau_1, \ldots, \tau_n] \Rightarrow \tau$ and, through a mixfix annotation, specify the layout of the function's $n$ arguments. The constant's name, in this case $f$, will also serve as the label in the abstract syntax tree.

You may also declare mixfix syntax without adding constants to the theory's signature, by using a `syntax` section instead of `consts`. Thus a production need not map directly to a logical function (this typically requires additional syntactic translations, see also Chapter 8).

As a special case of the general mixfix declaration, the form

$$c \; \texttt{::} \; \texttt{"}\sigma\texttt{"} \; \texttt{("}\textit{template}\texttt{")}$$

specifies no priorities. The resulting production puts no priority constraints on any of its arguments and has maximal priority itself. Omitting priorities in this manner is prone to syntactic ambiguities unless the production's right-hand side is fully bracketed, as in `"if _ then _ else _ fi"`.

Omitting the mixfix annotation completely, as in $c$ `::` `"`$\sigma$`"`, is sensible only if $c$ is an identifier. Otherwise you will be unable to write terms involving $c$.

### 7.3.2 Example: arithmetic expressions

This theory specification contains a `syntax` section with mixfix declarations encoding the priority grammar from §7.1:

```
ExpSyntax = Pure +
types
  exp
syntax
  "0" :: exp                  ("0"      9)
  "+" :: [exp, exp] => exp    ("_ + _"  [0, 1] 0)
  "*" :: [exp, exp] => exp    ("_ * _"  [3, 2] 2)
  "-" :: exp => exp           ("- _"    [3] 3)
end
```

If you put this into a file `ExpSyntax.thy` and load it via `use_thy"ExpSyntax"`, you can run some tests:

```
val read_exp = Syntax.test_read (syn_of ExpSyntax.thy) "exp";
  val it = fn : string -> unit
read_exp "0 * 0 * 0 * 0 + 0 + 0 + 0";
  tokens: "0" "*" "0" "*" "0" "*" "0" "+" "0" "+" "0" "+" "0"
  raw: ("+" ("+" ("+" ("*" "0" ("*" "0" ("*" "0" "0"))) "0") "0") "0")
  ⋮
read_exp "0 + - 0 + 0";
  tokens: "0" "+" "-" "0" "+" "0"
  raw: ("+" ("+" "0" ("-" "0")) "0")
  ⋮
```

The output of `Syntax.test_read` includes the token list (`tokens`) and the raw AST directly derived from the parse tree, ignoring parse AST translations. The rest is tracing information provided by the macro expander (see §8.5).

Executing `Syntax.print_gram` reveals the productions derived from the above mixfix declarations (lots of additional information deleted):

```
Syntax.print_gram (syn_of ExpSyntax.thy);
  exp = "0"  => "0" (9)
  exp = exp[0] "+" exp[1]  => "+" (0)
  exp = exp[3] "*" exp[2]  => "*" (2)
  exp = "-" exp[3]  => "-" (3)
```

Note that because `exp` is not of class `logic`, it has been retained as a separate nonterminal. This also entails that the syntax does not provide for identifiers or paranthesized expressions. Normally you would also want to add the declaration `arities exp::logic` after `types` and use `consts` instead of `syntax`. Try this as an exercise and study the changes in the grammar.

### 7.3.3 The mixfix template

Let us now take a closer look at the string *template* appearing in mixfix annotations. This string specifies a list of parsing and printing directives: delimiters, arguments, spaces, blocks of indentation and line breaks. These are encoded by the following character sequences:

*d* is a delimiter, namely a non-empty sequence of characters other than the special characters `_`, `(`, `)` and `/`. Even these characters may appear if escaped; this means preceding it with a `'` (single quote). Thus you have to write `''` if you really want a single quote. Furthermore, a `'` followed by a space separates delimiters without extra white space being added for printing.

_ is an argument position, which stands for a nonterminal symbol or name token.

$s$ is a non-empty sequence of spaces for printing. This and the following specifications do not affect parsing at all.

`(`$n$ opens a pretty printing block. The optional number $n$ specifies how much indentation to add when a line break occurs within the block. If `(` is not followed by digits, the indentation defaults to 0.

`)` closes a pretty printing block.

`//` forces a line break.

`/`$s$ allows a line break. Here $s$ stands for the string of spaces (zero or more) right after the `/` character. These spaces are printed if the break is not taken.

For example, the template `"(_ +/ _)"` specifies an infix operator. There are two argument positions; the delimiter `+` is preceded by a space and followed by a space or line break; the entire phrase is a pretty printing block. Other examples appear in Fig. 8.4 below. Isabelle's pretty printer resembles the one described in Paulson [9].

### 7.3.4 Infixes

Infix operators associating to the left or right can be declared using `infixl` or `infixr`. Basically, the form $c$ `::` $\sigma$ `(infixl` $p$`)` abbreviates the mixfix declarations

```
"op c" :: σ   ("(_ c/ _)" [p, p+1] p)
"op c" :: σ   ("op c")
```

and $c$ `::` $\sigma$ `(infixr` $p$`)` abbreviates the mixfix declarations

```
"op c" :: σ   ("(_ c/ _)" [p+1, p] p)
"op c" :: σ   ("op c")
```

The infix operator is declared as a constant with the prefix `op`. Thus, prefixing infixes with `op` makes them behave like ordinary function symbols, as in ML. Special characters occurring in $c$ must be escaped, as in delimiters, using a single quote.

A slightly more general form of infix declarations allows constant names to be independent from their concrete syntax, namely $c$ `::` $\sigma$ `(infixl "`$sy$`"` $p$`)`, the same for `infixr`. As an example consider:

```
and :: [bool, bool] => bool  (infixr "&" 35)
```

The internal constant name will then be just `and`, without any `op` prefixed.

### 7.3.5 Binders

A **binder** is a variable-binding construct such as a quantifier. The constant declaration

```
c :: σ   (binder "Q" [pb] p)
```

introduces a constant $c$ of type $\sigma$, which must have the form $(\tau_1 \Rightarrow \tau_2) \Rightarrow \tau_3$. Its concrete syntax is $\mathcal{Q}\ x\ .\ P$, where $x$ is a bound variable of type $\tau_1$, the body $P$ has type $\tau_2$ and the whole term has type $\tau_3$. The optional integer $pb$ specifies the body's priority, by default $p$. Special characters in $\mathcal{Q}$ must be escaped using a single quote.

The declaration is expanded internally to something like

```
c     :: (τ1 => τ2) => τ3
"Q"   :: [idts, τ2] => τ3   ("(3Q_./ _)" [0, pb] p)
```

Here `idts` is the nonterminal symbol for a list of identifiers with optional type constraints (see Fig. 7.1). The declaration also installs a parse translation for $\mathcal{Q}$ and a print translation for $c$ to translate between the internal and external forms.

A binder of type $(\sigma \Rightarrow \tau) \Rightarrow \tau$ can be nested by giving a list of variables. The external form $\mathcal{Q}\ x_1\ x_2 \ldots x_n\ .\ P$ corresponds to the internal form

$$c(\lambda x_1\ .\ c(\lambda x_2\ .\ \ldots\ c(\lambda x_n\ .\ P) \ldots)).$$

For example, let us declare the quantifier $\forall$:

```
All :: ('a => o) => o   (binder "ALL " 10)
```

This lets us write $\forall x.P$ as either `All(%`$x\,.\,P$`)` or `ALL` $x\,.\,P$. When printing, Isabelle prefers the latter form, but must fall back on `All`$(P)$ if $P$ is not an abstraction. Both $P$ and `ALL` $x\,.\,P$ have type $o$, the type of formulae, while the bound variable can be polymorphic.

## 7.4 *Alternative print modes

Isabelle's pretty printer supports alternative output syntaxes. These may be used independently or in cooperation. The currently active print modes (with precedence from left to right) are determined by a reference variable.

```
print_mode: string list ref
```

Initially this may already contain some print mode identifiers, depending on how Isabelle has been invoked (e.g. by some user interface). So changes should be incremental — adding or deleting modes relative to the current value.

Any ML string is a legal print mode identifier, without any predeclaration required. The following names should be considered reserved, though: `""` (yes, the empty string), `symbols`, `latex`, `xterm`.

There is a separate table of mixfix productions for pretty printing associated with each print mode. The currently active ones are conceptually just concatenated from left to right, with the standard syntax output table always coming last as default. Thus mixfix productions of preceding modes in the list may override those of later ones. Also note that token translations are always relative to some print mode (see §8.7).

The canonical application of print modes is optional printing of mathematical symbols from a special screen font instead of ASCII. Another example is to re-use Isabelle's advanced $\lambda$-term printing mechanisms to generate completely different output, say for interfacing external tools like model checkers (see also `HOL/Modelcheck`).

## 7.5 Ambiguity of parsed expressions

To keep the grammar small and allow common productions to be shared all logical types (except `prop`) are internally represented by one nonterminal, namely `logic`. This and omitted or too freely chosen priorities may lead to ways of parsing an expression that were not intended by the theory's maker. In most cases Isabelle is able to select one of multiple parse trees that an expression has lead to by checking which of them can be typed correctly. But this may not work in every case and always slows down parsing. The warning and error messages that can be produced during this process are as follows:

If an ambiguity can be resolved by type inference the following warning is shown to remind the user that parsing is (unnecessarily) slowed down. In cases where it's not easily possible to eliminate the ambiguity the frequency of the warning can be controlled by changing the value of `Syntax.ambiguity_level` which has type `int ref`. Its default value is 1 and by increasing it one can control how many parse trees are necessary to generate the warning.

```
Ambiguous input "..."
produces the following parse trees:
...
Fortunately, only one parse tree is type correct.
You may still want to disambiguate your grammar or your input.
```

The following message is normally caused by using the same syntax in two different productions:

```
Ambiguous input "..."
produces the following parse trees:
...
More than one term is type correct:
...
```

Ambiguities occuring in syntax translation rules cannot be resolved by type inference because it is not necessary for these rules to be type correct. Therefore Isabelle always generates an error message and the ambiguity should be eliminated by changing the grammar or the rule.

## 7.6 Example: some minimal logics

This section presents some examples that have a simple syntax. They demonstrate how to define new object-logics from scratch.

First we must define how an object-logic syntax is embedded into the meta-logic. Since all theorems must conform to the syntax for `prop` (see Fig. 7.1), that syntax has to be extended with the object-level syntax. Assume that the syntax of your object-logic defines a meta-type $o$ of formulae which refers to the nonterminal `logic`. These formulae can now appear in axioms and theorems wherever `prop` does if you add the production

$$prop \;=\; logic.$$

This is not supposed to be a copy production but an implicit coercion from formulae to propositions:

```
Base = Pure +
types
  o
arities
  o :: logic
consts
  Trueprop :: o => prop   ("_" 5)
end
```

The constant `Trueprop` (the name is arbitrary) acts as an invisible coercion function. Assuming this definition resides in a file `Base.thy`, you have to load it with the command `use_thy "Base"`.

One of the simplest nontrivial logics is **minimal logic** of implication. Its definition in Isabelle needs no advanced features but illustrates the overall mechanism nicely:

```
Hilbert = Base +
consts
  "-->" :: [o, o] => o   (infixr 10)
rules
  K     "P --> Q --> P"
  S     "(P --> Q --> R) --> (P --> Q) --> P --> R"
  MP    "[| P --> Q; P |] ==> Q"
end
```

After loading this definition from the file `Hilbert.thy`, you can start to prove theorems in the logic:

```
Goal "P --> P";
  Level 0
  P --> P
   1.  P --> P
by (resolve_tac [Hilbert.MP] 1);
  Level 1
  P --> P
   1.  ?P --> P --> P
   2.  ?P
by (resolve_tac [Hilbert.MP] 1);
  Level 2
  P --> P
   1.  ?P1 --> ?P --> P --> P
   2.  ?P1
   3.  ?P
by (resolve_tac [Hilbert.S] 1);
  Level 3
  P --> P
   1.  P --> ?Q2 --> P
   2.  P --> ?Q2
by (resolve_tac [Hilbert.K] 1);
  Level 4
  P --> P
   1.  P --> ?Q2
by (resolve_tac [Hilbert.K] 1);
  Level 5
  P --> P
  No subgoals!
```

As we can see, this Hilbert-style formulation of minimal logic is easy to define but difficult to use. The following natural deduction formulation is better:

```
MinI = Base +
consts
  "-->" :: [o, o] => o   (infixr 10)
rules
  impI  "(P ==> Q) ==> P --> Q"
  impE  "[| P --> Q; P |] ==> Q"
end
```

Note, however, that although the two systems are equivalent, this fact cannot be proved within Isabelle. Axioms `S` and `K` can be derived in `MinI` (exercise!), but `impI` cannot be derived in `Hilbert`. The reason is that `impI` is only an **admissible** rule in `Hilbert`, something that can only be shown by induction over all possible proofs in `Hilbert`.

We may easily extend minimal logic with falsity:

```
MinIF = MinI +
consts
  False :: o
rules
  FalseE "False ==> P"
end
```

On the other hand, we may wish to introduce conjunction only:

```
MinC = Base +
consts
  "&" :: [o, o] => o   (infixr 30)
rules
  conjI  "[| P; Q |] ==> P & Q"
  conjE1 "P & Q ==> P"
  conjE2 "P & Q ==> Q"
end
```

And if we want to have all three connectives together, we create and load a theory file consisting of a single line:

```
MinIFC = MinIF + MinC
```

Now we can prove mixed theorems like

```
Goal "P & False --> Q";
by (resolve_tac [MinI.impI] 1);
by (dresolve_tac [MinC.conjE2] 1);
by (eresolve_tac [MinIF.FalseE] 1);
```

Try this as an exercise!

**Chapter 8**

# Syntax Transformations

This chapter is intended for experienced Isabelle users who need to define macros or code their own translation functions. It describes the transformations between parse trees, abstract syntax trees and terms.

## 8.1 Abstract syntax trees

The parser, given a token list from the lexer, applies productions to yield a parse tree. By applying some internal transformations the parse tree becomes an abstract syntax tree, or AST. Macro expansion, further translations and finally type inference yields a well-typed term. The printing process is the reverse, except for some subtleties to be discussed later.

Figure 8.1 outlines the parsing and printing process. Much of the complexity is due to the macro mechanism. Using macros, you can specify most forms of concrete syntax without writing any ML code.

Abstract syntax trees are an intermediate form between the raw parse trees and the typed $\lambda$-terms. An AST is either an atom (constant or variable) or a list of *at least two* subtrees. Internally, they have type `Syntax.ast`:

```
datatype ast = Constant of string
             | Variable of string
             | Appl of ast list
```

Isabelle uses an S-expression syntax for abstract syntax trees. Constant atoms are shown as quoted strings, variable atoms as non-quoted strings and applications as a parenthesised list of subtrees. For example, the AST

```
Appl [Constant "_constrain",
      Appl [Constant "_abs", Variable "x", Variable "t"],
      Appl [Constant "fun", Variable "'a", Variable "'b"]]
```

is shown as `("_constrain" ("_abs" x t) ("fun" 'a 'b))`. Both `()` and `(f)` are illegal because they have too few subtrees.

The resemblance to Lisp's S-expressions is intentional, but there are two kinds of atomic symbols: `Constant` $x$ and `Variable` $x$. Do not take the names `Constant` and `Variable` too literally; in the later translation to terms, `Variable` $x$ may become a constant, free or bound variable, even a type constructor or class name;

| | |
|---|---|
| string | |
| $\downarrow$ | lexer, parser |
| parse tree | |
| $\downarrow$ | parse AST translation |
| AST | |
| $\downarrow$ | AST rewriting (macros) |
| AST | |
| $\downarrow$ | parse translation, type inference |
| — well-typed term — | |
| $\downarrow$ | print translation |
| AST | |
| $\downarrow$ | AST rewriting (macros) |
| AST | |
| $\downarrow$ | print AST translation, token translation |
| string | |

Figure 8.1: Parsing and printing

the actual outcome depends on the context.

Similarly, you can think of ($f$ $x_1$ ... $x_n$) as the application of $f$ to the arguments $x_1, \ldots, x_n$. But the kind of application is determined later by context; it could be a type constructor applied to types.

Forms like `(("_abs" x` $t$`)` $u$`)` are legal, but ASTs are first-order: the `"_abs"` does not bind the `x` in any way. Later at the term level, `("_abs" x` $t$`)` will become an `Abs` node and occurrences of `x` in $t$ will be replaced by bound variables (the term constructor `Bound`).

## 8.2 Transforming parse trees to ASTs

The parse tree is the raw output of the parser. Translation functions, called **parse AST translations**, transform the parse tree into an abstract syntax tree.

The parse tree is constructed by nesting the right-hand sides of the productions used to recognize the input. Such parse trees are simply lists of tokens and constituent parse trees, the latter representing the nonterminals of the productions. Let us refer to the actual productions in the form displayed by `print_syntax` (see §6.4.3 for an example).

Ignoring parse AST translations, parse trees are transformed to ASTs by stripping out delimiters and copy productions. More precisely, the mapping $[\![-]\!]$ is derived from the productions as follows:

- Name tokens: $[\![t]\!]$ = `Variable` $s$, where $t$ is an `id`, `var`, `tid`, `tvar`, `xnum` or `xstr` token, and $s$ its associated string. Note that for `xstr` this does not

| input string | AST |
|---|---|
| `"f"` | `f` |
| `"'a"` | `'a` |
| `"t == u"` | `("==" t u)` |
| `"f(x)"` | `("_appl" f x)` |
| `"f(x, y)"` | `("_appl" f ("_args" x y))` |
| `"f(x, y, z)"` | `("_appl" f ("_args" x ("_args" y z)))` |
| `"%x y. t"` | `("_lambda" ("_idts" x y) t)` |

Figure 8.2: Parsing examples using the Pure syntax

| input string | AST |
|---|---|
| `"f(x, y, z)"` | `(f x y z)` |
| `"'a ty"` | `(ty 'a)` |
| `"('a, 'b) ty"` | `(ty 'a 'b)` |
| `"%x y z. t"` | `("_abs" x ("_abs" y ("_abs" z t)))` |
| `"%x :: 'a. t"` | `("_abs" ("_constrain" x 'a) t)` |
| `"[\| P; Q; R \|] => S"` | `("==>" P ("==>" Q ("==>" R S)))` |
| `"['a, 'b, 'c] => 'd"` | `("fun" 'a ("fun" 'b ("fun" 'c 'd)))` |

Figure 8.3: Built-in parse AST translations

include the quotes.

- Copy productions: $[\![\ldots P \ldots]\!] = [\![P]\!]$. Here $\ldots$ stands for strings of delimiters, which are discarded. $P$ stands for the single constituent that is not a delimiter; it is either a nonterminal symbol or a name token.

- 0-ary productions: $[\![\ldots \texttt{=>} c]\!] = \texttt{Constant}\ c$. Here there are no constituents other than delimiters, which are discarded.

- $n$-ary productions, where $n \geq 1$: delimiters are discarded and the remaining constituents $P_1, \ldots, P_n$ are built into an application whose head constant is $c$:
$$[\![\ldots P_1 \ldots P_n \ldots \texttt{=>} c]\!] = \texttt{Appl}\ [\texttt{Constant}\ c, [\![P_1]\!], \ldots, [\![P_n]\!]]$$

Figure 8.2 presents some simple examples, where `==`, `_appl`, `_args`, and so forth name productions of the Pure syntax. These examples illustrate the need for further translations to make ASTs closer to the typed $\lambda$-calculus. The Pure syntax provides predefined parse AST translations for ordinary applications, type applications, nested abstractions, meta implications and function types. Figure 8.3 shows their effect on some representative input strings.

The names of constant heads in the AST control the translation process. The list of constants invoking parse AST translations appears in the output of `print_syntax` under `parse_ast_translation`.

## 8.3 Transforming ASTs to terms

The AST, after application of macros (see §8.5), is transformed into a term. This term is probably ill-typed since type inference has not occurred yet. The term may contain type constraints consisting of applications with head `"_constrain"`; the second argument is a type encoded as a term. Type inference later introduces correct types or rejects the input.

Another set of translation functions, namely parse translations, may affect this process. If we ignore parse translations for the time being, then ASTs are transformed to terms by mapping AST constants to constants, AST variables to schematic or free variables and AST applications to applications.

More precisely, the mapping $[\![-]\!]$ is defined by

- Constants: $[\![\texttt{Constant}\ x]\!] = \texttt{Const}(x, \texttt{dummyT})$.
- Schematic variables: $[\![\texttt{Variable}\ \texttt{"?}xi\texttt{"}]\!] = \texttt{Var}((x, i), \texttt{dummyT})$, where $x$ is the base name and $i$ the index extracted from $xi$.
- Free variables: $[\![\texttt{Variable}\ x]\!] = \texttt{Free}(x, \texttt{dummyT})$.
- Function applications with $n$ arguments:

$$[\![\texttt{Appl}\ [f, x_1, \ldots, x_n]]\!] = [\![f]\!]\ \texttt{\$}\ [\![x_1]\!]\ \texttt{\$}\ \ldots\ \texttt{\$}\ [\![x_n]\!]$$

Here `Const`, `Var`, `Free` and `$` are constructors of the datatype `term`, while `dummyT` stands for some dummy type that is ignored during type inference.

So far the outcome is still a first-order term. Abstractions and bound variables (constructors `Abs` and `Bound`) are introduced by parse translations. Such translations are attached to `"_abs"`, `"!!"` and user-defined binders.

## 8.4 Printing of terms

The output phase is essentially the inverse of the input phase. Terms are translated via abstract syntax trees into strings. Finally the strings are pretty printed.

Print translations (§8.6) may affect the transformation of terms into ASTs. Ignoring those, the transformation maps term constants, variables and applications to the corresponding constructs on ASTs. Abstractions are mapped to applications of the special constant `_abs`.

More precisely, the mapping $[\![-]\!]$ is defined as follows:

- $[\![\texttt{Const}(x, \tau)]\!] = \texttt{Constant}\ x$.
- $[\![\texttt{Free}(x, \tau)]\!] = \mathit{constrain}(\texttt{Variable}\ x, \tau)$.
- $[\![\texttt{Var}((x, i), \tau)]\!] = \mathit{constrain}(\texttt{Variable "?}xi\texttt{"}, \tau)$, where ?$xi$ is the string representation of the `indexname` $(x, i)$.
- For the abstraction $\lambda x :: \tau . t$, let $x'$ be a variant of $x$ renamed to differ from all names occurring in $t$, and let $t'$ be obtained from $t$ by replacing all bound occurrences of $x$ by the free variable $x'$. This replaces corresponding occurrences of the constructor `Bound` by the term $\texttt{Free}(x', \texttt{dummyT})$:

  $$[\![\texttt{Abs}(x, \tau, t)]\!] = \texttt{Appl}\ [\texttt{Constant "\_abs"}, \mathit{constrain}(\texttt{Variable}\ x', \tau), [\![t']\!]].$$

- $[\![\texttt{Bound}\ i]\!] = \texttt{Variable "B.}i\texttt{"}$. The occurrence of constructor `Bound` should never happen when printing well-typed terms; it indicates a de Bruijn index with no matching abstraction.
- Where $f$ is not an application,

  $$[\![f\ \$\ x_1\ \$\ \ldots\ \$\ x_n]\!] = \texttt{Appl}\ [[\![f]\!], [\![x_1]\!], \ldots, [\![x_n]\!]]$$

Type constraints are inserted to allow the printing of types. This is governed by the boolean variable `show_types`:

- $\mathit{constrain}(x, \tau) = x$ if $\tau = \texttt{dummyT}$ or `show_types` is set to `false`.
- $\mathit{constrain}(x, \tau) = \texttt{Appl}\ [\texttt{Constant "\_constrain"}, x, [\![\tau]\!]]$ otherwise.

  Here, $[\![\tau]\!]$ is the AST encoding of $\tau$: type constructors go to `Constant`s; type identifiers go to `Variable`s; type applications go to `Appl`s with the type constructor as the first element. If `show_sorts` is set to `true`, some type variables are decorated with an AST encoding of their sort.

The AST, after application of macros (see §8.5), is transformed into the final output string. The built-in **print AST translations** reverse the parse AST translations of Fig. 8.3.

For the actual printing process, the names attached to productions of the form $\ldots A_1^{(p_1)} \ldots A_n^{(p_n)} \ldots$ `=>`$c$ play a vital role. Each AST with constant head $c$, namely "$c$" or ("$c$" $x_1 \ldots x_n$), is printed according to the production for $c$. Each argument $x_i$ is converted to a string, and put in parentheses if its priority $(p_i)$ requires this. The resulting strings and their syntactic sugar (denoted by … above) are joined to make a single string.

If an application ("$c$" $x_1 \ldots x_m$) has more arguments than the corresponding production, it is first split into (("$c$" $x_1 \ldots x_n$) $x_{n+1} \ldots x_m$). Applications with too few arguments or with non-constant head or without a corresponding production are printed as $f(x_1, \ldots, x_l)$ or $(\alpha_1, \ldots, \alpha_l)ty$. Multiple productions associated with some name $c$ are tried in order of appearance. An occurrence of `Variable` $x$ is simply printed as $x$.

Blanks are *not* inserted automatically. If blanks are required to separate tokens, specify them in the mixfix declaration, possibly preceded by a slash (`/`) to allow a line break.

## 8.5 Macros: syntactic rewriting

Mixfix declarations alone can handle situations where there is a direct connection between the concrete syntax and the underlying term. Sometimes we require a more elaborate concrete syntax, such as quantifiers and list notation. Isabelle's **macros** and **translation functions** can perform translations such as

```
ALL x:A.P  ⇌  Ball(A, %x.P)
[x, y, z]  ⇌  Cons(x, Cons(y, Cons(z, Nil)))
```

Translation functions (see §8.6) must be coded in ML; they are the most powerful translation mechanism but are difficult to read or write. Macros are specified by first-order rewriting systems that operate on abstract syntax trees. They are usually easy to read and write, and can express all but the most obscure translations.

Figure 8.4 defines a fragment of first-order logic and set theory.[1] Theory `SetSyntax` declares constants for set comprehension (`Collect`), replacement (`Replace`) and bounded universal quantification (`Ball`). Each of these binds some variables. Without additional syntax we should have to write $\forall x \in A . P$ as `Ball(A,%x.P)`, and similarly for the others.

The theory specifies a variable-binding syntax through additional productions that have mixfix declarations. Each non-copy production must specify some constant, which is used for building ASTs. The additional constants are decorated with `@` to stress their purely syntactic purpose; they may not occur within the final well-typed terms, being declared as `syntax` rather than `consts`.

The translations cause the replacement of external forms by internal forms after parsing, and vice versa before printing of terms. As a specification of the set theory notation, they should be largely self-explanatory. The syntactic constants, `@Collect`, `@Replace` and `@Ball`, appear implicitly in the macro rules via their mixfix forms.

Macros can define variable-binding syntax because they operate on ASTs, which have no inbuilt notion of bound variable. The macro variables `x` and `y` have type `idt` and therefore range over identifiers, in this case bound variables.

[1]This and the following theories are complete working examples, though they specify only syntax, no axioms. The file `ZF/ZF.thy` presents a full set theory definition, including many macro rules.

```
SetSyntax = Pure +
types
  i o
arities
  i, o :: logic
consts
  Trueprop      :: o => prop                ("_" 5)
  Collect       :: [i, i => o] => i
  Replace       :: [i, [i, i] => o] => i
  Ball          :: [i, i => o] => o
syntax
  "@Collect"    :: [idt, i, o] => i         ("(1{_:_./ _})")
  "@Replace"    :: [idt, idt, i, o] => i    ("(1{_./ _:_, _})")
  "@Ball"       :: [idt, i, o] => o         ("(3ALL _:_./ _)" 10)
translations
  "{x:A. P}"    == "Collect(A, %x. P)"
  "{y. x:A, Q}" == "Replace(A, %x y. Q)"
  "ALL x:A. P"  == "Ball(A, %x. P)"
end
```

Figure 8.4: Macro example: set theory

The macro variables `P` and `Q` range over formulae containing bound variable occurrences.

Other applications of the macro system can be less straightforward, and there are peculiarities. The rest of this section will describe in detail how Isabelle macros are preprocessed and applied.

### 8.5.1 Specifying macros

Macros are basically rewrite rules on ASTs. But unlike other macro systems found in programming languages, Isabelle's macros work in both directions. Therefore a syntax contains two lists of rewrites: one for parsing and one for printing.

The `translations` section specifies macros. The syntax for a macro is

$$(root)\ string \quad \left\{ \begin{array}{c} \texttt{=>} \\ \texttt{<=} \\ \texttt{==} \end{array} \right\} \quad (root)\ string$$

This specifies a parse rule (`=>`), a print rule (`<=`), or both (`==`). The two strings specify the left and right-hand sides of the macro rule. The $(root)$ specification is optional; it specifies the nonterminal for parsing the *string* and if omitted defaults to `logic`. AST rewrite rules $(l, r)$ must obey certain conditions:

- Rules must be left linear: $l$ must not contain repeated variables.
- Rules must have constant heads, namely $l = \texttt{"}c\texttt{"}$ or $l = (\texttt{"}c\texttt{"}\ x_1 \ldots x_n)$.

- Every variable in $r$ must also occur in $l$.

Macro rules may refer to any syntax from the parent theories. They may also refer to anything defined before the current `translations` section — including any mixfix declarations.

Upon declaration, both sides of the macro rule undergo parsing and parse AST translations (see §8.1), but do not themselves undergo macro expansion. The lexer runs in a different mode that additionally accepts identifiers of the form _ *letter quasiletter** (like `_idt`, `_K`). Thus, a constant whose name starts with an underscore can appear in macro rules but not in ordinary terms.

Some atoms of the macro rule's AST are designated as constants for matching. These are all names that have been declared as classes, types or constants (logical and syntactic).

The result of this preprocessing is two lists of macro rules, each stored as a pair of ASTs. They can be viewed using `print_syntax` (sections `parse_rules` and `print_rules`). For theory `SetSyntax` of Fig. 8.4 these are

```
parse_rules:
  ("@Collect" x A P)  ->  ("Collect" A ("_abs" x P))
  ("@Replace" y x A Q)  ->  ("Replace" A ("_abs" x ("_abs" y Q)))
  ("@Ball" x A P)  ->  ("Ball" A ("_abs" x P))
print_rules:
  ("Collect" A ("_abs" x P))  ->  ("@Collect" x A P)
  ("Replace" A ("_abs" x ("_abs" y Q)))  ->  ("@Replace" y x A Q)
  ("Ball" A ("_abs" x P))  ->  ("@Ball" x A P)
```

**!** Avoid choosing variable names that have previously been used as constants, types or type classes; the `consts` section in the output of `print_syntax` lists all such names. If a macro rule works incorrectly, inspect its internal form as shown above, recalling that constants appear as quoted strings and variables without quotes.

**!** If `eta_contract` is set to `true`, terms will be $\eta$-contracted *before* the AST rewriter sees them. Thus some abstraction nodes needed for print rules to match may vanish. For example, `Ball(A, %x. P(x))` contracts to `Ball(A, P)`; the print rule does not apply and the output will be `Ball(A, P)`. This problem would not occur if ML translation functions were used instead of macros (as is done for binder declarations).

**!** Another trap concerns type constraints. If `show_types` is set to `true`, bound variables will be decorated by their meta types at the binding place (but not at occurrences in the body). Matching with `Collect(A, %x. P)` binds `x` to something like `("_constrain" y "i")` rather than only `y`. AST rewriting will cause the constraint to appear in the external form, say `{y::i:A::i. P::o}`.

To allow such constraints to be re-read, your syntax should specify bound variables using the nonterminal `idt`. This is the case in our example. Choosing `id` instead of `idt` is a common error.

### 8.5.2 Applying rules

As a term is being parsed or printed, an AST is generated as an intermediate form (recall Fig. 8.1). The AST is normalised by applying macro rules in the manner of a traditional term rewriting system. We first examine how a single rule is applied.

Let $t$ be the abstract syntax tree to be normalised and $(l, r)$ some translation rule. A subtree $u$ of $t$ is a **redex** if it is an instance of $l$; in this case $l$ is said to **match** $u$. A redex matched by $l$ may be replaced by the corresponding instance of $r$, thus **rewriting** the AST $t$. Matching requires some notion of **place-holders** that may occur in rule patterns but not in ordinary ASTs; `Variable` atoms serve this purpose.

The matching of the object $u$ by the pattern $l$ is performed as follows:

- Every constant matches itself.
- `Variable` $x$ in the object matches `Constant` $x$ in the pattern. This point is discussed further below.
- Every AST in the object matches `Variable` $x$ in the pattern, binding $x$ to $u$.
- One application matches another if they have the same number of subtrees and corresponding subtrees match.
- In every other case, matching fails. In particular, `Constant` $x$ can only match itself.

A successful match yields a substitution that is applied to $r$, generating the instance that replaces $u$.

The second case above may look odd. This is where `Variable`s of non-rule ASTs behave like `Constant`s. Recall that ASTs are not far removed from parse trees; at this level it is not yet known which identifiers will become constants, bounds, frees, types or classes. As §8.1 describes, former parse tree heads appear in ASTs as `Constant`s, while the name tokens `id`, `var`, `tid`, `tvar`, `xnum` and `xstr` become `Variable`s. On the other hand, when ASTs generated from terms for printing, all constants and type constructors become `Constant`s; see §8.1. Thus ASTs may contain a messy mixture of `Variable`s and `Constant`s. This is insignificant at macro level because matching treats them alike.

Because of this behaviour, different kinds of atoms with the same name are indistinguishable, which may make some rules prone to misbehaviour. Example:

```
types
  Nil
consts
  Nil     :: 'a list
syntax
  "[]"    :: 'a list    ("[]")
translations
  "[]"    == "Nil"
```

The term `Nil` will be printed as `[]`, just as expected. The term `%Nil.t` will be printed as `%[].t`, which might not be expected! Guess how type `Nil` is printed?

Normalizing an AST involves repeatedly applying macro rules until none are applicable. Macro rules are chosen in order of appearance in the theory definitions. You can watch the normalization of ASTs during parsing and printing by setting `Syntax.trace_norm_ast` to `true`. Alternatively, use `Syntax.test_read`. The information displayed when tracing includes the AST before normalization (`pre`), redexes with results (`rewrote`), the normal form finally reached (`post`) and some statistics (`normalize`). If tracing is off, `Syntax.stat_norm_ast` can be set to `true` in order to enable printing of the normal form and statistics only.

### 8.5.3 Example: the syntax of finite sets

This example demonstrates the use of recursive macros to implement a convenient notation for finite sets.

```
FinSyntax = SetSyntax +
types
  is
syntax
  ""            :: i => is                  ("_")
  "@Enum"       :: [i, is] => is            ("_,/ _")
consts
  empty         :: i                        ("{}")
  insert        :: [i, i] => i
syntax
  "@Finset"     :: is => i                  ("{(_)}")
translations
  "{x, xs}"     == "insert(x, {xs})"
  "{x}"         == "insert(x, {})"
end
```

Finite sets are internally built up by `empty` and `insert`. The declarations above specify `{x, y, z}` as the external representation of

```
insert(x, insert(y, insert(z, empty)))
```

The nonterminal symbol `is` stands for one or more objects of type `i` separated by commas. The mixfix declaration `"_,/ _"` allows a line break after the comma for pretty printing; if no line break is required then a space is printed instead.

The nonterminal is declared as the type `is`, but with no `arities` declaration. Hence `is` is not a logical type and may be used safely as a new nonterminal for custom syntax. The nonterminal `is` can later be re-used for other enumerations of type `i` like lists or tuples. If we had needed polymorphic enumerations, we could have used the predefined nonterminal symbol `args` and skipped this part altogether.

Next follows `empty`, which is already equipped with its syntax `{}`, and `insert` without concrete syntax. The syntactic constant `@Finset` provides concrete syntax for enumerations of `i` enclosed in curly braces. Remember that a pair of parentheses, as in `"{(_)}"`, specifies a block of indentation for pretty printing.

The translations may look strange at first. Macro rules are best understood in their internal forms:

```
parse_rules:
  ("@Finset" ("@Enum" x xs))  ->  ("insert" x ("@Finset" xs))
  ("@Finset" x)  ->  ("insert" x "empty")
print_rules:
  ("insert" x ("@Finset" xs))  ->  ("@Finset" ("@Enum" x xs))
  ("insert" x "empty")  ->  ("@Finset" x)
```

This shows that `{x,xs}` indeed matches any set enumeration of at least two elements, binding the first to `x` and the rest to `xs`. Likewise, `{xs}` and `{x}` represent any set enumeration. The parse rules only work in the order given.

**!** The AST rewriter cannot distinguish constants from variables and looks only for names of atoms. Thus the names of `Constants` occurring in the (internal) left-hand side of translation rules should be regarded as reserved words. Choose non-identifiers like `@Finset` or sufficiently long and strange names. If a bound variable's name gets rewritten, the result will be incorrect; for example, the term

```
%empty insert. insert(x, empty)
```

is incorrectly printed as `%empty insert. {x}`.

### 8.5.4 Example: a parse macro for dependent types

As stated earlier, a macro rule may not introduce new `Variables` on the right-hand side. Something like `"K(B)" => "%x.B"` is illegal; if allowed, it could cause variable capture. In such cases you usually must fall back on translation functions. But a trick can make things readable in some cases: *calling* translation functions by parse macros:

```
ProdSyntax = SetSyntax +
consts
  Pi            :: [i, i => i] => i
syntax
  "@PROD"       :: [idt, i, i] => i       ("(3PROD _:_./ _)" 10)
  "@->"         :: [i, i] => i            ("(_ ->/ _)" [51, 50] 50)
translations
  "PROD x:A. B" => "Pi(A, %x. B)"
  "A -> B"      => "Pi(A, _K(B))"
end
ML
  val print_translation = [("Pi", dependent_tr' ("@PROD", "@->"))];
```

Here `Pi` is a logical constant for constructing general products. Two external forms exist: the general case `PROD x:A.B` and the function space `A -> B`, which abbreviates `Pi(A, %x.B)` when `B` does not depend on `x`.

The second parse macro introduces `_K(B)`, which later becomes `%x.B` due to a parse translation associated with `_K`. Unfortunately there is no such trick for printing, so we have to add a `ML` section for the print translation `dependent_tr'`.

Recall that identifiers with a leading `_` are allowed in translation rules, but not in ordinary terms. Thus we can create ASTs containing names that are not directly expressible.

The parse translation for `_K` is already installed in Pure, and `dependent_tr'` is exported by the syntax module for public use. See §8.6 below for more of the arcane lore of translation functions.

## 8.6 Translation functions

This section describes the translation function mechanism. By writing ML functions, you can do almost everything to terms or ASTs during parsing and printing. The logic `LK` is a good example of sophisticated transformations between internal and external representations of sequents; here, macros would be useless.

A full understanding of translations requires some familiarity with Isabelle's internals, especially the datatypes `term`, `typ`, `Syntax.ast` and the encodings of types and terms as such at the various stages of the parsing or printing process. Most users should never need to use translation functions.

### 8.6.1 Declaring translation functions

There are four kinds of translation functions, with one of these coming in two variants. Each such function is associated with a name, which triggers calls to it. Such names can be constants (logical or syntactic) or type constructors.

`print_syntax` displays the sets of names associated with the translation functions of a theory under `parse_ast_translation`, etc. You can add new ones via the `ML` section of a theory definition file. There may never be more than one function of the same kind per name. Even though the `ML` section is the very last part of the file, newly installed translation functions are already effective when processing all of the preceding sections.

The `ML` section's contents are simply copied verbatim near the beginning of the ML file generated from a theory definition file. Definitions made here are accessible as components of an ML structure; to make some parts private, use an ML `local` declaration. The ML code may install translation functions by declaring any of the following identifiers:

```
val parse_ast_translation   : (string * (ast list -> ast)) list
val print_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
```

### 8.6.2 The translation strategy

The different kinds of translation functions are called during the transformations between parse trees, ASTs and terms (recall Fig. 8.1). Whenever a combination of the form ("$c$" $x_1 \ldots x_n$) is encountered, and a translation function $f$ of appropriate kind exists for $c$, the result is computed by the ML function call $f$ [$x_1, \ldots, x_n$].

For AST translations, the arguments $x_1, \ldots, x_n$ are ASTs. A combination has the form `Constant` $c$ or `Appl` [`Constant` $c, x_1, \ldots, x_n$]. For term translations, the arguments are terms and a combination has the form `Const`$(c, \tau)$ or `Const`$(c, \tau)$ `$` $x_1$ `$` ... `$` $x_n$. Terms allow more sophisticated transformations than ASTs do, typically involving abstractions and bound variables. *Typed* print translations may even peek at the type $\tau$ of the constant they are invoked on; they are also passed the current value of the `show_sorts` flag.

Regardless of whether they act on terms or ASTs, translation functions called during the parsing process differ from those for printing more fundamentally in their overall behaviour:

**Parse translations** are applied bottom-up. The arguments are already in translated form. The translations must not fail; exceptions trigger an error message.

**Print translations** are applied top-down. They are supplied with arguments that are partly still in internal form. The result again undergoes translation; therefore a print translation should not introduce as head the very constant that invoked it. The function may raise exception `Match` to indicate failure; in this event it has no effect.

Only constant atoms — constructor `Constant` for ASTs and `Const` for terms — can invoke translation functions. This causes another difference between parsing and printing.

Parsing starts with a string and the constants are not yet identified. Only parse tree heads create `Constant`s in the resulting AST, as described in §8.2. Macros and parse AST translations may introduce further `Constant`s. When the final AST is converted to a term, all `Constant`s become `Const`s, as described in §8.3.

Printing starts with a well-typed term and all the constants are known. So all logical constants and type constructors may invoke print translations. These, and macros, may introduce further constants.

### 8.6.3 Example: a print translation for dependent types

Let us continue the dependent type example (page 94) by examining the parse translation for `_K` and the print translation `dependent_tr'`, which are both built-in. By convention, parse translations have names ending with `_tr` and print translations have names ending with `_tr'`. Search for such names in the Isabelle sources to locate more examples.

Here is the parse translation for `_K`:

```
fun k_tr [t] = Abs ("x", dummyT, incr_boundvars 1 t)
  | k_tr ts = raise TERM ("k_tr", ts);
```

If `k_tr` is called with exactly one argument $t$, it creates a new `Abs` node with a body derived from $t$. Since terms given to parse translations are not yet typed, the type of the bound variable in the new `Abs` is simply `dummyT`. The function increments all `Bound` nodes referring to outer abstractions by calling `incr_boundvars`, a basic term manipulation function defined in `Pure/term.ML`.

Here is the print translation for dependent types:

```
fun dependent_tr' (q, r) (A :: Abs (x, T, B) :: ts) =
      if 0 mem (loose_bnos B) then
        let val (x', B') = Syntax.variant_abs' (x, dummyT, B) in
          list_comb
            (Const (q,dummyT) $ Syntax.mark_boundT (x',T) $ A $ B',ts)
        end
      else list_comb (Const (r, dummyT) $ A $ B, ts)
  | dependent_tr' _ _ = raise Match;
```

The argument `(q,r)` is supplied to the curried function `dependent_tr'` by a partial application during its installation. For example, we could set up print translations for both `Pi` and `Sigma` by including

```
val print_translation =
  [("Pi",    dependent_tr' ("@PROD", "@->")),
   ("Sigma", dependent_tr' ("@SUM", "@*"))];
```

within the `ML` section. The first of these transforms `Pi`$(A,$ `Abs`$(x, T, B))$ into `@PROD`$(x', A, B')$ or `@->`$(A, B)$, choosing the latter form if $B$ does not depend on $x$. It checks this using `loose_bnos`, yet another function from `Pure/term.ML`. Note that $x'$ is a version of $x$ renamed away from all names in $B$, and $B'$ is the body $B$ with `Bound` nodes referring to the `Abs` node replaced by `Free`$(x',$ `dummyT`$)$ (but marked as representing a bound variable).

We must be careful with types here. While types of `Const`s are ignored, type constraints may be printed for some `Free`s and `Var`s if `show_types` is set to `true`. Variables of type `dummyT` are never printed with constraint, though. The line

```
let val (x', B') = Syntax.variant_abs' (x, dummyT, B);
```

replaces bound variable occurrences in $B$ by the free variable $x'$ with type `dummyT`. Only the binding occurrence of $x'$ is given the correct type `T`, so this is the only place where a type constraint might appear.

Also note that we are responsible to mark free identifiers that actually represent bound variables. This is achieved by `Syntax.variant_abs'` and `Syntax.mark_boundT` above. Failing to do so may cause these names to be printed in the wrong style.

## 8.7 Token translations

Isabelle's meta-logic features quite a lot of different kinds of identifiers, namely *class*, *tfree*, *tvar*, *free*, *bound*, *var*. One might want to have these printed in different styles, e.g. in bold or italic, or even transcribed into something more readable like $\alpha, \alpha', \beta$ instead of `'a`, `'aa`, `'b` for type variables. Token translations provide a means to such ends, enabling the user to install certain ML functions associated with any logical token class and depending on some print mode.

The logical class of identifiers can not necessarily be determined by its syntactic category, though. For example, consider free vs. bound variables. So Isabelle's pretty printing mechanism, starting from fully typed terms, has to be careful to preserve this additional information[2]. In particular, user-supplied print translation functions operating on terms have to be well-behaved in this respect. Free identifiers introduced to represent bound variables have to be marked appropriately (cf. the example at the end of §8.6).

[2] This is done by marking atoms in abstract syntax trees appropriately. The marks are actually visible by print translation functions – they are just special constants applied to atomic asts, for example `("_bound" x)`.

Token translations may be installed by declaring the `token_translation` value within the `ML` section of a theory definition file:

```
val token_translation: (string * string * (string -> string * int)) list
```

The elements of this list are of the form $(m, c, f)$, where $m$ is a print mode identifier, $c$ a token class, and $f\colon \mathit{string} \to \mathit{string} \times \mathit{int}$ the actual translation function. Assuming that $x$ is of identifier class $c$, and print mode $m$ is the first one of all currently active modes that provide some translation for $c$, then $x$ is output according to $f(x) = (x', \mathit{len})$. Thereby $x'$ is the modified identifier name and $\mathit{len}$ its visual length approximated in terms of whole characters. Thus $x'$ may include non-printing parts like control sequences or markup information for typesetting systems.

**Chapter 9**

# Substitution Tactics

Replacing equals by equals is a basic form of reasoning. Isabelle supports several kinds of equality reasoning. **Substitution** means replacing free occurrences of $t$ by $u$ in a subgoal. This is easily done, given an equality $t = u$, provided the logic possesses the appropriate rule. The tactic `hyp_subst_tac` performs substitution even in the assumptions. But it works via object-level implication, and therefore must be specially set up for each suitable object-logic.

Substitution should not be confused with object-level **rewriting**. Given equalities of the form $t = u$, rewriting replaces instances of $t$ by corresponding instances of $u$, and continues until it reaches a normal form. Substitution handles 'one-off' replacements by particular equalities while rewriting handles general equations. Chapter 10 discusses Isabelle's rewriting tactics.

## 9.1 Substitution rules

Many logics include a substitution rule of the form

$$[\![ ?a = ?b; ?P(?a) ]\!] \Longrightarrow ?P(?b) \qquad (subst)$$

In backward proof, this may seem difficult to use: the conclusion $?P(?b)$ admits far too many unifiers. But, if the theorem `eqth` asserts $t = u$, then `eqth RS subst` is the derived rule

$$?P(t) \Longrightarrow ?P(u).$$

Provided $u$ is not an unknown, resolution with this rule is well-behaved.[1] To replace $u$ by $t$ in subgoal $i$, use

`resolve_tac [eqth RS subst]` $i$.

To replace $t$ by $u$ in subgoal $i$, use

`resolve_tac [eqth RS ssubst]` $i$,

where `ssubst` is the 'swapped' substitution rule

$$[\![ ?a = ?b; ?P(?b) ]\!] \Longrightarrow ?P(?a). \qquad (ssubst)$$

[1] Unifying $?P(u)$ with a formula $Q$ expresses $Q$ in terms of its dependence upon $u$. There are still $2^k$ unifiers, if $Q$ has $k$ occurrences of $u$, but Isabelle ensures that the first unifier includes all the occurrences.

If `sym` denotes the symmetry rule $?a = ?b \Longrightarrow ?b = ?a$, then `ssubst` is just `sym RS subst`. Many logics with equality include the rules `subst` and `ssubst`, as well as `refl`, `sym` and `trans` (for the usual equality laws). Examples include `FOL` and `HOL`, but not `CTT` (Constructive Type Theory).

Elim-resolution is well-behaved with assumptions of the form $t = u$. To replace $u$ by $t$ or $t$ by $u$ in subgoal $i$, use

`eresolve_tac [subst]` $i$ or `eresolve_tac [ssubst]` $i$.

Logics `HOL`, `FOL` and `ZF` define the tactic `stac` by

```
fun stac eqth = CHANGED o rtac (eqth RS ssubst);
```

Now `stac eqth` is like `resolve_tac [eqth RS ssubst]` but with the valuable property of failing if the substitution has no effect.

## 9.2 Substitution in the hypotheses

Substitution rules, like other rules of natural deduction, do not affect the assumptions. This can be inconvenient. Consider proving the subgoal

$$\llbracket c = a; c = b \rrbracket \Longrightarrow a = b.$$

Calling `eresolve_tac [ssubst]` $i$ simply discards the assumption $c = a$, since $c$ does not occur in $a = b$. Of course, we can work out a solution. First apply `eresolve_tac [subst]` $i$, replacing $a$ by $c$:

$$c = b \Longrightarrow c = b$$

Equality reasoning can be difficult, but this trivial proof requires nothing more sophisticated than substitution in the assumptions. Object-logics that include the rule (*subst*) provide tactics for this purpose:

```
hyp_subst_tac       : int -> tactic
bound_hyp_subst_tac : int -> tactic
```

`hyp_subst_tac` $i$ selects an equality assumption of the form $t = u$ or $u = t$, where $t$ is a free variable or parameter. Deleting this assumption, it replaces $t$ by $u$ throughout subgoal $i$, including the other assumptions.

`bound_hyp_subst_tac` $i$ is similar but only substitutes for parameters (bound variables). Uses for this are discussed below.

The term being replaced must be a free variable or parameter. Substitution for constants is usually unhelpful, since they may appear in other theorems. For instance, the best way to use the assumption $0 = 1$ is to contradict a theorem that states $0 \neq 1$, rather than to replace 0 by 1 in the subgoal!

Substitution for unknowns, such as $?x = 0$, is a bad idea: we might prove the subgoal more easily by instantiating $?x$ to 1. Substitution for free variables is unhelpful if they appear in the premises of a rule being derived: the substitution affects object-level assumptions, not meta-level assumptions. For instance, replacing $a$ by $b$ could make the premise $P(a)$ worthless. To avoid this problem, use `bound_hyp_subst_tac`; alternatively, call `cut_facts_tac` to insert the atomic premises as object-level assumptions.

## 9.3 Setting up `hyp_subst_tac`

Many Isabelle object-logics, such as `FOL`, `HOL` and their descendants, come with `hyp_subst_tac` already defined. A few others, such as `CTT`, do not support this tactic because they lack the rule (*subst*). When defining a new logic that includes a substitution rule and implication, you must set up `hyp_subst_tac` yourself. It is packaged as the ML functor `HypsubstFun`, which takes the argument signature `HYPSUBST_DATA`:

```
signature HYPSUBST_DATA =
  sig
  structure Simplifier : SIMPLIFIER
  val dest_Trueprop    : term -> term
  val dest_eq          : term -> term*term*typ
  val dest_imp         : term -> term*term
  val eq_reflection    : thm            (* a=b ==> a==b *)
  val imp_intr         : thm            (* (P ==> Q) ==> P-->Q *)
  val rev_mp           : thm            (* [| P;  P-->Q |] ==> Q *)
  val subst            : thm            (* [| a=b;  P(a) |] ==> P(b) *)
  val sym              : thm            (* a=b ==> b=a *)
  val thin_refl        : thm            (* [|x=x; P|] ==> P *)
  end;
```

Thus, the functor requires the following items:

`Simplifier` should be an instance of the simplifier (see Chapter 10).

`dest_Trueprop` should coerce a meta-level formula to the corresponding object-level one. Typically, it should return $P$ when applied to the term `Trueprop` $P$ (see example below).

`dest_eq` should return the triple $(t, u, T)$, where $T$ is the type of $t$ and $u$, when applied to the ML term that represents $t = u$. For other terms, it should raise an exception.

`dest_imp` should return the pair $(P, Q)$ when applied to the ML term that represents the implication $P \to Q$. For other terms, it should raise an exception.

`eq_reflection` is the theorem discussed in §10.8.

`imp_intr` should be the implies introduction rule $(?P \Longrightarrow ?Q) \Longrightarrow ?P \to ?Q$.

`rev_mp` should be the 'reversed' implies elimination rule $[\![?P;\ ?P \to ?Q]\!] \Longrightarrow ?Q$.

`subst` should be the substitution rule $[\![?a = ?b;\ ?P(?a)]\!] \Longrightarrow ?P(?b)$.

`sym` should be the symmetry rule $?a = ?b \Longrightarrow ?b = ?a$.

`thin_refl` should be the rule $[\![?a = ?a;\ ?P]\!] \Longrightarrow ?P$, which is used to erase trivial equalities.

The functor resides in file `Provers/hypsubst.ML` in the Isabelle distribution directory. It is not sensitive to the precise formalization of the object-logic. It is not concerned with the names of the equality and implication symbols, or the types of formula and terms.

Coding the functions `dest_Trueprop`, `dest_eq` and `dest_imp` requires knowledge of Isabelle's representation of terms. For `FOL`, they are declared by

```
fun dest_Trueprop (Const ("Trueprop", _) $ P) = P
  | dest_Trueprop t = raise TERM ("dest_Trueprop", [t]);

fun dest_eq (Const("op =",T)  $ t $ u) = (t, u, domain_type T)

fun dest_imp (Const("op -->",_) $ A $ B) = (A, B)
  | dest_imp  t = raise TERM ("dest_imp", [t]);
```

Recall that `Trueprop` is the coercion from type $o$ to type *prop*, while `op =` is the internal name of the infix operator `=`. Function `domain_type`, given the function type $S \Rightarrow T$, returns the type $S$. Pattern-matching expresses the function concisely, using wildcards (`_`) for the types.

The tactic `hyp_subst_tac` works as follows. First, it identifies a suitable equality assumption, possibly re-orienting it using `sym`. Then it moves other assumptions into the conclusion of the goal, by repeatedly calling `etac rev_mp`. Then, it uses `asm_full_simp_tac` or `ssubst` to substitute throughout the subgoal. (If the equality involves unknowns then it must use `ssubst`.) Then, it deletes the equality. Finally, it moves the assumptions back to their original positions by calling `resolve_tac [imp_intr]`.

Chapter 10

# Simplification

This chapter describes Isabelle's generic simplification package. It performs conditional and unconditional rewriting and uses contextual information ('local assumptions'). It provides several general hooks, which can provide automatic case splits during rewriting, for example. The simplifier is already set up for many of Isabelle's logics: `FOL`, `ZF`, `HOL`, `HOLCF`.

The first section is a quick introduction to the simplifier that should be sufficient to get started. The later sections explain more advanced features.

## 10.1 Simplification for dummies

Basic use of the simplifier is particularly easy because each theory is equipped with sensible default information controlling the rewrite process — namely the implicit *current simpset*. A suite of simple commands is provided that refer to the implicit simpset of the current theory context.

**!** Make sure that you are working within the correct theory context. Executing proofs interactively, or loading them from ML files without associated theories may require setting the current theory manually via the `context` command.

### 10.1.1 Simplification tactics

```
Simp_tac          : int -> tactic
Asm_simp_tac      : int -> tactic
Full_simp_tac     : int -> tactic
Asm_full_simp_tac : int -> tactic
trace_simp        : bool ref
```
**initially false**

`Simp_tac` $i$ simplifies subgoal $i$ using the current simpset. It may solve the subgoal completely if it has become trivial, using the simpset's solver tactic.

`Asm_simp_tac` is like `Simp_tac`, but extracts additional rewrite rules from the local assumptions.

`Full_simp_tac` is like `Simp_tac`, but also simplifies the assumptions (without using the assumptions to simplify each other or the actual goal).

`Asm_full_simp_tac` is like `Asm_simp_tac`, but also simplifies the assumptions. In particular, assumptions can simplify each other. [1]

`set trace_simp;` makes the simplifier output internal operations. This includes rewrite steps, but also bookkeeping like modifications of the simpset.

As an example, consider the theory of arithmetic in `HOL`. The (rather trivial) goal $0 + (x + 0) = x + 0 + 0$ can be solved by a single call of `Simp_tac` as follows:

```
context Arith.thy;
Goal "0 + (x + 0) = x + 0 + 0";
  1. 0 + (x + 0) = x + 0 + 0
by (Simp_tac 1);
 Level 1
 0 + (x + 0) = x + 0 + 0
 No subgoals!
```

The simplifier uses the current simpset of `Arith.thy`, which contains suitable theorems like $?n + 0 = ?n$ and $0 + ?n = ?n$.

In many cases, assumptions of a subgoal are also needed in the simplification process. For example, `x = 0 ==> x + x = 0` is solved by `Asm_simp_tac` as follows:

```
  1. x = 0 ==> x + x = 0
by (Asm_simp_tac 1);
```

`Asm_full_simp_tac` is the most powerful of this quartet of tactics but may also loop where some of the others terminate. For example,

```
  1. ALL x. f x = g (f (g x)) ==> f 0 = f 0 + 0
```

is solved by `Simp_tac`, but `Asm_simp_tac` and `Asm_simp_tac` loop because the rewrite rule $f\,?x = g\,(f\,(g\,?x))$ extracted from the assumption does not terminate. Isabelle notices certain simple forms of nontermination, but not this one. Because assumptions may simplify each other, there can be very subtle cases of nontermination.

**!** `Asm_full_simp_tac` may miss opportunities to simplify an assumption $A_i$ using an assumption $A_j$ in case $A_j$ is to the right of $A_i$. For example, given the subgoal

```
 [| ... f a ...; P a; ALL x. P x --> f x = g x |] ==> ...
```

`Asm_full_simp_tac` will not simplify the `f a` on the left. This problem can be overcome by reordering assumptions (see §10.5.4). Future versions of `Asm_full_simp_tac` will not suffer from this deficiency.

[1] `Asm_full_simp_tac` used to process the assumptions from left to right. For backwards compatibilty reasons only there is now `Asm_lr_simp_tac` that behaves like the old `Asm_full_simp_tac`.

Using the simplifier effectively may take a bit of experimentation. Set the `trace_simp` flag to get a better idea of what is going on. The resulting output can be enormous, especially since invocations of the simplifier are often nested (e.g. when solving conditions of rewrite rules).

### 10.1.2 Modifying the current simpset

```
Addsimps    : thm list -> unit
Delsimps    : thm list -> unit
Addsimprocs : simproc list -> unit
Delsimprocs : simproc list -> unit
Addcongs    : thm list -> unit
Delcongs    : thm list -> unit
```

Depending on the theory context, the `Add` and `Del` functions manipulate basic components of the associated current simpset. Internally, all rewrite rules have to be expressed as (conditional) meta-equalities. This form is derived automatically from object-level equations that are supplied by the user. Another source of rewrite rules are *simplification procedures*, that is ML functions that produce suitable theorems on demand, depending on the current redex. Congruences are a more advanced feature; see §10.2.6.

`Addsimps` *thms*`;` adds rewrite rules derived from *thms* to the current simpset.

`Delsimps` *thms*`;` deletes rewrite rules derived from *thms* from the current simpset.

`Addsimprocs` *procs*`;` adds simplification procedures *procs* to the current simpset.

`Delsimprocs` *procs*`;` deletes simplification procedures *procs* from the current simpset.

`Addcongs` *thms*`;` adds congruence rules to the current simpset.

`Delcongs` *thms*`;` deletes congruence rules to the current simpset.

When a new theory is built, its implicit simpset is initialized by the union of the respective simpsets of its parent theories. In addition, certain theory definition constructs (e.g. `datatype` and `primrec` in `HOL`) implicitly augment the current simpset. Ordinary definitions are not added automatically!

It is up the user to manipulate the current simpset further by explicitly adding or deleting theorems and simplification procedures.

Good simpsets are hard to design. Rules that obviously simplify, like $?n+0 = ?n$, should be added to the current simpset right after they have been proved. More specific ones (such as distributive laws, which duplicate subterms) should be added only for specific proofs and deleted afterwards. Conversely, sometimes a rule needs to be removed for a certain proof and restored afterwards. The need of frequent additions or deletions may indicate a badly designed simpset.

! The union of the parent simpsets (as described above) is not always a good starting point for the new theory. If some ancestors have deleted simplification rules because they are no longer wanted, while others have left those rules in, then the union will contain the unwanted rules. After this union is formed, changes to a parent simpset have no effect on the child simpset.

## 10.2 Simplification sets

The simplifier is controlled by information contained in **simpsets**. These consist of several components, including rewrite rules, simplification procedures, congruence rules, and the subgoaler, solver and looper tactics. The simplifier should be set up with sensible defaults so that most simplifier calls specify only rewrite rules or simplification procedures. Experienced users can exploit the other components to streamline proofs in more sophisticated manners.

### 10.2.1 Inspecting simpsets

```
print_ss : simpset -> unit
rep_ss   : simpset -> {mss        : meta_simpset,
                       subgoal_tac: simpset  -> int -> tactic,
                       loop_tac   :             int -> tactic,
                       finish_tac : thm list -> int -> tactic,
                unsafe_finish_tac : thm list -> int -> tactic}
```

`print_ss` *ss*; displays the printable contents of simpset *ss*. This includes the rewrite rules and congruences in their internal form expressed as meta-equalities. The names of the simplification procedures and the patterns they are invoked on are also shown. The other parts, functions and tactics, are non-printable.

`rep_ss` *ss*; decomposes *ss* as a record of its internal components, namely the meta˙simpset, the subgoaler, the loop, and the safe and unsafe solvers.

### 10.2.2 Building simpsets

```
empty_ss : simpset
merge_ss : simpset * simpset -> simpset
```

`empty_ss` is the empty simpset. This is not very useful under normal circumstances because it doesn't contain suitable tactics (subgoaler etc.). When setting up the simplifier for a particular object-logic, one will typically define a more appropriate "almost empty" simpset. For example, in `HOL` this is called `HOL_basic_ss`.

`merge_ss` ($ss_1$, $ss_2$) merges simpsets $ss_1$ and $ss_2$ by building the union of their respective rewrite rules, simplification procedures and congruences. The other components (tactics etc.) cannot be merged, though; they are taken from either simpset[2].

### 10.2.3 Accessing the current simpset

```
simpset        : unit -> simpset
simpset_ref    : unit -> simpset ref
simpset_of     : theory -> simpset
simpset_ref_of : theory -> simpset ref
print_simpset  : theory -> unit
```

Each theory contains a current simpset stored within a private ML reference variable. This can be retrieved and modified as follows.

`simpset();` retrieves the simpset value from the current theory context.

`simpset_ref();` retrieves the simpset reference variable from the current theory context. This can be assigned to by using `:=` in ML.

`simpset_of` *thy*; retrieves the simpset value from theory *thy*.

`simpset_ref_of` *thy*; retrieves the simpset reference variable from theory *thy*.

`print_simpset` *thy*; prints the current simpset of theory *thy* in the same way as `print_ss`.

### 10.2.4 Rewrite rules

```
addsimps : simpset * thm list -> simpset                 infix 4
delsimps : simpset * thm list -> simpset                 infix 4
```

Rewrite rules are theorems expressing some form of equality, for example:

$$\begin{aligned} Suc(?m) + ?n &\;=\; ?m + Suc(?n) \\ ?P \land ?P &\;\leftrightarrow\; ?P \\ ?A \cup ?B &\;\equiv\; \{x\,.\,x \in ?A \lor x \in ?B\} \end{aligned}$$

Conditional rewrites such as $?m < ?n \Longrightarrow ?m/?n = 0$ are also permitted; the conditions can be arbitrary formulas.

[2]Actually from $ss_1$, but it would unwise to count on that.

Internally, all rewrite rules are translated into meta-equalities, theorems with conclusion $lhs \equiv rhs$. Each simpset contains a function for extracting equalities from arbitrary theorems. For example, $\neg(?x \in \{\})$ could be turned into $?x \in \{\} \equiv False$. This function can be installed using `setmksimps` but only the definer of a logic should need to do this; see §10.8.2. The function processes theorems added by `addsimps` as well as local assumptions.

*ss* `addsimps` *thms* adds rewrite rules derived from *thms* to the simpset *ss*.

*ss* `delsimps` *thms* deletes rewrite rules derived from *thms* from the simpset *ss*.

**!** The simplifier will accept all standard rewrite rules: those where all unknowns are of base type. Hence $?i + (?j + ?k) = (?i + ?j) + ?k$ is OK.

It will also deal gracefully with all rules whose left-hand sides are so-called *higher-order patterns* [6]. These are terms in $\beta$-normal form (this will always be the case unless you have done something strange) where each occurrence of an unknown is of the form $?F(x_1, \ldots, x_n)$, where the $x_i$ are distinct bound variables. Hence $(\forall x . ?P(x) \land ?Q(x)) \leftrightarrow (\forall x . ?P(x)) \land (\forall x . ?Q(x))$ is also OK, in both directions.

In some rare cases the rewriter will even deal with quite general rules: for example $?f(?x) \in range(?f) = True$ rewrites $g(a) \in range(g)$ to $True$, but will fail to match $g(h(b)) \in range(\lambda x . g(h(x)))$. However, you can replace the offending subterms (in our case $?f(?x)$, which is not a pattern) by adding new variables and conditions: $?y = ?f(?x) \Longrightarrow ?y \in range(?f) = True$ is acceptable as a conditional rewrite rule since conditions can be arbitrary terms.

There is basically no restriction on the form of the right-hand sides. They may not contain extraneous term or type variables, though.

### 10.2.5 *Simplification procedures

```
addsimprocs : simpset * simproc list -> simpset
delsimprocs : simpset * simproc list -> simpset
```

Simplification procedures are ML objects of abstract type `simproc`. Basically they are just functions that may produce *proven* rewrite rules on demand. They are associated with certain patterns that conceptually represent left-hand sides of equations; these are shown by `print_ss`. During its operation, the simplifier may offer a simplification procedure the current redex and ask for a suitable rewrite rule. Thus rules may be specifically fashioned for particular situations, resulting in a more powerful mechanism than term rewriting by a fixed set of rules.

*ss* `addsimprocs` *procs* adds the simplification procedures *procs* to the current simpset.

*ss* `delsimprocs` *procs* deletes the simplification procedures *procs* from the current simpset.

For example, simplification procedures `nat_cancel` of `HOL/Arith` cancel common summands and constant factors out of several relations of sums over natural numbers.

Consider the following goal, which after cancelling $a$ on both sides contains a factor of 2. Simplifying with the simpset of `Arith.thy` will do the cancellation automatically:

```
  1. x + a + x < y + y + 2 + a + a + a + a + a
by (Simp_tac 1);
  1. x < Suc (a + (a + y))
```

### 10.2.6 *Congruence rules

```
addcongs   : simpset * thm list -> simpset                  infix 4
delcongs   : simpset * thm list -> simpset                  infix 4
addeqcongs : simpset * thm list -> simpset                  infix 4
deleqcongs : simpset * thm list -> simpset                  infix 4
```

Congruence rules are meta-equalities of the form

$$\ldots \Longrightarrow f(?x_1,\ldots,?x_n) \equiv f(?y_1,\ldots,?y_n).$$

This governs the simplification of the arguments of $f$. For example, some arguments can be simplified under additional assumptions:

$$[\![?P_1 \leftrightarrow ?Q_1;\ ?Q_1 \Longrightarrow ?P_2 \leftrightarrow ?Q_2]\!] \Longrightarrow (?P_1 \rightarrow ?P_2) \equiv (?Q_1 \rightarrow ?Q_2)$$

Given this rule, the simplifier assumes $Q_1$ and extracts rewrite rules from it when simplifying $P_2$. Such local assumptions are effective for rewriting formulae such as $x = 0 \rightarrow y + x = y$. The local assumptions are also provided as theorems to the solver; see § 10.2.8 below.

*ss* `addcongs` *thms* adds congruence rules to the simpset *ss*. These are derived from *thms* in an appropriate way, depending on the underlying object-logic.

*ss* `delcongs` *thms* deletes congruence rules derived from *thms*.

*ss* `addeqcongs` *thms* adds congruence rules in their internal form (conclusions using meta-equality) to simpset *ss*. This is the basic mechanism that `addcongs` is built on. It should be rarely used directly.

*ss* `deleqcongs` *thms* deletes congruence rules in internal form from simpset *ss*.

Here are some more examples. The congruence rule for bounded quantifiers also supplies contextual information, this time about the bound variable:

$$[\![?A = ?B;\ \bigwedge x\,.\, x \in ?B \Longrightarrow ?P(x) = ?Q(x)]\!] \Longrightarrow \\ (\forall x \in ?A\,.\, ?P(x)) = (\forall x \in ?B\,.\, ?Q(x))$$

The congruence rule for conditional expressions can supply contextual information for simplifying the arms:

$$[\![?p = ?q;\ ?q \Longrightarrow ?a = ?c;\ \neg ?q \Longrightarrow ?b = ?d]\!] \Longrightarrow \mathit{if}(?p, ?a, ?b) \equiv \mathit{if}(?q, ?c, ?d)$$

A congruence rule can also *prevent* simplification of some arguments. Here is an alternative congruence rule for conditional expressions:

$$?p = ?q \Longrightarrow \mathit{if}(?p, ?a, ?b) \equiv \mathit{if}(?q, ?a, ?b)$$

Only the first argument is simplified; the others remain unchanged. This can make simplification much faster, but may require an extra case split to prove the goal.

### 10.2.7 *The subgoaler

```
setsubgoaler : simpset *  (simpset -> int -> tactic) -> simpset    infix 4
prems_of_ss  : simpset -> thm list
```

The subgoaler is the tactic used to solve subgoals arising out of conditional rewrite rules or congruence rules. The default should be simplification itself. Occasionally this strategy needs to be changed. For example, if the premise of a conditional rule is an instance of its conclusion, as in $Suc(?m) < ?n \Longrightarrow ?m < ?n$, the default strategy could loop.

*ss* `setsubgoaler` *tacf* sets the subgoaler of *ss* to *tacf*. The function *tacf* will be applied to the current simplifier context expressed as a simpset.

`prems_of_ss` *ss* retrieves the current set of premises from simplifier context *ss*. This may be non-empty only if the simplifier has been told to utilize local assumptions in the first place, e.g. if invoked via `asm_simp_tac`.

As an example, consider the following subgoaler:

```
fun subgoaler ss =
    assume_tac ORELSE'
    resolve_tac (prems_of_ss ss) ORELSE'
    asm_simp_tac ss;
```

This tactic first tries to solve the subgoal by assumption or by resolving with with one of the premises, calling simplification only if that fails.

### 10.2.8 *The solver

```
setSolver  : simpset * (thm list -> int -> tactic) -> simpset      infix 4
addSolver  : simpset * (thm list -> int -> tactic) -> simpset      infix 4
setSSolver : simpset * (thm list -> int -> tactic) -> simpset      infix 4
addSSolver : simpset * (thm list -> int -> tactic) -> simpset      infix 4
```

The solver is a pair of tactics that attempt to solve a subgoal after simplification. Typically it just proves trivial subgoals such as `True` and $t = t$. It could use sophisticated means such as `blast_tac`, though that could make simplification expensive.

Rewriting does not instantiate unknowns. For example, rewriting cannot prove $a \in {?A}$ since this requires instantiating $?A$. The solver, however, is an arbitrary tactic and may instantiate unknowns as it pleases. This is the only way the simplifier can handle a conditional rewrite rule whose condition contains extra variables. When a simplification tactic is to be combined with other provers, especially with the classical reasoner, it is important whether it can be considered safe or not. For this reason the solver is split into a safe and an unsafe part.

The standard simplification strategy solely uses the unsafe solver, which is appropriate in most cases. For special applications where the simplification process is not allowed to instantiate unknowns within the goal, simplification starts with the safe solver, but may still apply the ordinary unsafe one in nested simplifications for conditional rules or congruences.

*ss* `setSSolver` *tacf* installs *tacf* as the *safe* solver of *ss*.

*ss* `addSSolver` *tacf* adds *tacf* as an additional *safe* solver; it will be tried after the solvers which had already been present in *ss*.

*ss* `setSolver` *tacf* installs *tacf* as the unsafe solver of *ss*.

*ss* `addSolver` *tacf* adds *tacf* as an additional unsafe solver; it will be tried after the solvers which had already been present in *ss*.

The solver tactic is invoked with a list of theorems, namely assumptions that hold in the local context. This may be non-empty only if the simplifier has been told to utilize local assumptions in the first place, e.g. if invoked via `asm_simp_tac`. The solver is also presented the full goal including its assumptions in any case. Thus it can use these (e.g. by calling `assume_tac`), even if the list of premises is not passed.

As explained in §10.2.7, the subgoaler is also used to solve the premises of congruence rules. These are usually of the form $s = {?x}$, where $s$ needs to be simplified and $?x$ needs to be instantiated with the result. Typically, the subgoaler will invoke the simplifier at some point, which will eventually call the solver. For this reason, solver tactics must be prepared to solve goals of the form $t = {?x}$, usually by reflexivity. In particular, reflexivity should be tried before any of the fancy tactics like `blast_tac`.

It may even happen that due to simplification the subgoal is no longer an equality. For example $False \leftrightarrow {?Q}$ could be rewritten to $\neg{?Q}$. To cover this case, the solver could try resolving with the theorem $\neg False$.

**!** If the simplifier aborts with the message `Failed congruence proof!`, then the subgoaler or solver has failed to prove a premise of a congruence rule. This should never occur under normal circumstances; it indicates that some congruence rule, or possibly the subgoaler or solver, is faulty.

### 10.2.9 *The looper

```
setloop   : simpset * (int -> tactic) -> simpset              infix 4
addloop   : simpset * (int -> tactic) -> simpset              infix 4
addsplits : simpset * thm list -> simpset                      infix 4
```

The looper is a tactic that is applied after simplification, in case the solver failed to solve the simplified goal. If the looper succeeds, the simplification process is started all over again. Each of the subgoals generated by the looper is attacked in turn, in reverse order.

A typical looper is case splitting: the expansion of a conditional. Another possibility is to apply an elimination rule on the assumptions. More adventurous loopers could start an induction.

*ss* `setloop` *tacf* installs *tacf* as the looper of *ss*.

*ss* `addloop` *tacf* adds *tacf* as an additional looper; it will be tried after the loopers which had already been present in *ss*.

*ss* `addsplits` *thms* adds (`split_tac` *thms*) as an additional looper.

## 10.3 The simplification tactics

```
simp_tac              : simpset -> int -> tactic
asm_simp_tac          : simpset -> int -> tactic
full_simp_tac         : simpset -> int -> tactic
asm_full_simp_tac     : simpset -> int -> tactic
safe_asm_full_simp_tac : simpset -> int -> tactic
SIMPSET               : (simpset -> tactic) -> tactic
SIMPSET'              : (simpset -> 'a -> tactic) -> 'a -> tactic
```

These are the basic tactics that are underlying any actual simplification work. The rewriting strategy is always strictly bottom up, except for congruence rules, which are applied while descending into a term. Conditions in conditional rewrite rules are solved recursively before the rewrite rule is applied.

`simp_tac`, `asm_simp_tac`, `full_simp_tac`, `asm_full_simp_tac` are the basic simplification tactics that work exactly like their namesakes in §10.1, except that they are explicitly supplied with a simpset.

`safe_asm_full_simp_tac` is like `asm_full_simp_tac`, but uses the safe solver as explained in §10.2.8. This tactic is mainly intended for building special tools, e.g. for combining the simplifier with the classical reasoner. It is rarely used directly.

`SIMPSET` *tacf*, `SIMPSET'` *tacf′* are tacticals that make a tactic depend on the implicit current simpset of the theory associated with the proof state they are applied on.

Local modifications of simpsets within a proof are often much cleaner by using above tactics in conjunction with explicit simpsets, rather than their capitalized counterparts. For example

```
Addsimps thms;
by (Simp_tac i);
Delsimps thms;
```

can be expressed more appropriately as

```
by (simp_tac (simpset() addsimps thms) i);
```

Also note that functions depending implicitly on the current theory context (like capital `Simp_tac` and the other commands of §10.1) should be considered harmful outside of actual proof scripts. In particular, ML programs like theory definition packages or special tactics should refer to simpsets only explicitly, via the above tactics used in conjunction with `simpset_of` or the `SIMPSET` tacticals.

**!** There is a subtle difference between `(SIMPSET'` *tacf*`)` and `(`*tacf* `(simpset()))`. For example `(SIMPSET' simp_tac)` would depend on the theory of the proof state it is applied to, while `(simp_tac (simpset()))` implicitly refers to the current theory context. Both are usually the same in proof scripts, provided that goals are only stated within the current theory. Robust programs would not count on that, of course.

## 10.4 Forward rules and conversions

```
simplify          : simpset -> thm -> thm
asm_simplify      : simpset -> thm -> thm
full_simplify     : simpset -> thm -> thm
asm_full_simplify : simpset -> thm -> thm

Simplifier.rewrite           : simpset -> cterm -> thm
Simplifier.asm_rewrite       : simpset -> cterm -> thm
Simplifier.full_rewrite      : simpset -> cterm -> thm
Simplifier.asm_full_rewrite  : simpset -> cterm -> thm
```

The first four of these functions provide *forward* rules for simplification. Their effect is analogous to the corresponding tactics described in §10.3, but affect the whole theorem instead of just a certain subgoal. Also note that the looper / solver process as described in §10.2.9 and §10.2.8 is omitted in forward simplification.

The latter four are *conversions*, establishing proven equations of the form $t \equiv u$ where the l.h.s. $t$ has been given as argument.

**!** Forward simplification rules and conversions should be used rarely in ordinary proof scripts. The main intention is to provide an internal interface to the simplifier for special utilities.

## 10.5 Examples of using the simplifier

Assume we are working within `FOL` (see the file `FOL/ex/Nat`) and that

`Nat.thy` is a theory including the constants 0, $Suc$ and $+$,

`add_0` is the rewrite rule $0 + ?n = ?n$,

`add_Suc` is the rewrite rule $Suc(?m) + ?n = Suc(?m + ?n)$,

`induct` is the induction rule $[\![?P(0);\ \bigwedge x\,.\ ?P(x) \Longrightarrow ?P(Suc(x))]\!] \Longrightarrow ?P(?n)$.

We augment the implicit simpset inherited from `Nat` with the basic rewrite rules for addition of natural numbers:

```
Addsimps [add_0, add_Suc];
```

### 10.5.1 A trivial example

Proofs by induction typically involve simplification. Here is a proof that 0 is a right identity:

```
Goal "m+0 = m";
  Level 0
  m + 0 = m
   1. m + 0 = m
```

The first step is to perform induction on the variable $m$. This returns a base case and inductive step as two subgoals:

```
by (res_inst_tac [("n","m")] induct 1);
  Level 1
  m + 0 = m
   1. 0 + 0 = 0
   2. !!x. x + 0 = x ==> Suc(x) + 0 = Suc(x)
```

Simplification solves the first subgoal trivially:

```
by (Simp_tac 1);
  Level 2
  m + 0 = m
   1. !!x. x + 0 = x ==> Suc(x) + 0 = Suc(x)
```

The remaining subgoal requires `Asm_simp_tac` in order to use the induction hypothesis as a rewrite rule:

```
by (Asm_simp_tac 1);
  Level 3
  m + 0 = m
  No subgoals!
```

### 10.5.2 An example of tracing

Let us prove a similar result involving more complex terms. We prove that addition is commutative.

```
Goal "m+Suc(n) = Suc(m+n)";
  Level 0
  m + Suc(n) = Suc(m + n)
   1. m + Suc(n) = Suc(m + n)
```

Performing induction on $m$ yields two subgoals:

```
by (res_inst_tac [("n","m")] induct 1);
  Level 1
  m + Suc(n) = Suc(m + n)
   1. 0 + Suc(n) = Suc(0 + n)
   2. !!x. x + Suc(n) = Suc(x + n) ==>
           Suc(x) + Suc(n) = Suc(Suc(x) + n)
```

Simplification solves the first subgoal, this time rewriting two occurrences of 0:

```
by (Simp_tac 1);
  Level 2
  m + Suc(n) = Suc(m + n)
   1. !!x. x + Suc(n) = Suc(x + n) ==>
           Suc(x) + Suc(n) = Suc(Suc(x) + n)
```

Switching tracing on illustrates how the simplifier solves the remaining subgoal:

```
set trace_simp;
by (Asm_simp_tac 1);
  Adding rewrite rule:
  .x + Suc n == Suc (.x + n)
```

```
Applying instance of rewrite rule:
?m + Suc ?n == Suc (?m + ?n)
Rewriting:
Suc .x + Suc n == Suc (Suc .x + n)
Applying instance of rewrite rule:
Suc ?m + ?n == Suc (?m + ?n)
Rewriting:
Suc .x + n == Suc (.x + n)
Applying instance of rewrite rule:
Suc ?m + ?n == Suc (?m + ?n)
Rewriting:
Suc .x + n == Suc (.x + n)
Applying instance of rewrite rule:
?x = ?x == True
Rewriting:
Suc (Suc (.x + n)) = Suc (Suc (.x + n)) == True
Level 3
m + Suc(n) = Suc(m + n)
No subgoals!
```

Many variations are possible. At Level 1 (in either example) we could have solved both subgoals at once using the tactical `ALLGOALS`:

```
by (ALLGOALS Asm_simp_tac);
  Level 2
  m + Suc(n) = Suc(m + n)
  No subgoals!
```

### 10.5.3 Free variables and simplification

Here is a conjecture to be proved for an arbitrary function $f$ satisfying the law $f(Suc(?n)) = Suc(f(?n))$:

```
val [prem] = Goal "(!!n. f(Suc(n)) = Suc(f(n))) ==> f(i+j) = i+f(j)";
  Level 0
  f(i + j) = i + f(j)
   1. f(i + j) = i + f(j)
  val prem = "f(Suc(?n)) = Suc(f(?n))
              [!!n. f(Suc(n)) = Suc(f(n))]" : thm
```

In the theorem `prem`, note that $f$ is a free variable while $?n$ is a schematic variable.

```
by (res_inst_tac [("n","i")] induct 1);
  Level 1
  f(i + j) = i + f(j)
   1. f(0 + j) = 0 + f(j)
   2. !!x. f(x + j) = x + f(j) ==> f(Suc(x) + j) = Suc(x) + f(j)
```

We simplify each subgoal in turn. The first one is trivial:

```
by (Simp_tac 1);
  Level 2
  f(i + j) = i + f(j)
   1. !!x. f(x + j) = x + f(j) ==> f(Suc(x) + j) = Suc(x) + f(j)
```

The remaining subgoal requires rewriting by the premise, so we add it to the current simpset:

```
by (asm_simp_tac (simpset() addsimps [prem]) 1);
  Level 3
  f(i + j) = i + f(j)
  No subgoals!
```

### 10.5.4 Reordering assumptions

As mentioned in §10.1.1, `asm_full_simp_tac` may require the assumptions to be permuted to be more effective. Given the subgoal

```
 1. [| ALL x. P x --> f x = g x; Q(f a); P a; R |] ==> S
```

we can rotate the assumptions two positions to the right

```
by (rotate_tac ~2 1);
```

to obtain

```
 1. [| P a; R; ALL x. P x --> f x = g x; Q(f a) |] ==> S
```

which enables `asm_full_simp_tac` to simplify `Q(f a)` to `Q(g a)` because now all required assumptions are to the left of `Q(f a)`.

Since rotation alone cannot produce arbitrary permutations, you can also pick out a particular assumption which needs to be rewritten and move it the the right end of the assumptions. In the above case rotation can be replaced by

```
by (dres_inst_tac [("psi","Q(f a)")] asm_rl 1);
```

which is more directed and leads to

```
 1. [| ALL x. P x --> f x = g x; P a; R; Q(f a) |] ==> S
```

**!** Reordering assumptions usually leads to brittle proofs and should be avoided. Future versions of `asm_full_simp_tac` will completely remove the need for such manipulations.

## 10.6 Permutative rewrite rules

A rewrite rule is **permutative** if the left-hand side and right-hand side are the same up to renaming of variables. The most common permutative rule is commutativity: $x + y = y + x$. Other examples include $(x - y) - z = (x - z) - y$ in arithmetic and $insert(x, insert(y, A)) = insert(y, insert(x, A))$ for sets. Such rules are common enough to merit special attention.

Because ordinary rewriting loops given such rules, the simplifier employs a special strategy, called **ordered rewriting**. There is a standard lexicographic ordering on terms. This should be perfectly OK in most cases, but can be changed for special applications.

```
settermless : simpset * (term * term -> bool) -> simpset
```
**infix 4**

*ss* `settermless` *rel* installs relation *rel* as term order in simpset *ss*.

A permutative rewrite rule is applied only if it decreases the given term with respect to this ordering. For example, commutativity rewrites $b + a$ to $a + b$, but then stops because $a + b$ is strictly less than $b + a$. The Boyer-Moore theorem prover [1] also employs ordered rewriting.

Permutative rewrite rules are added to simpsets just like other rewrite rules; the simplifier recognizes their special status automatically. They are most effective in the case of associative-commutative operators. (Associativity by itself is not permutative.) When dealing with an AC-operator $f$, keep the following points in mind:

- The associative law must always be oriented from left to right, namely $f(f(x, y), z) = f(x, f(y, z))$. The opposite orientation, if used with commutativity, leads to looping in conjunction with the standard term order.
- To complete your set of rewrite rules, you must add not just associativity (A) and commutativity (C) but also a derived rule, **left-commutativity** (LC): $f(x, f(y, z)) = f(y, f(x, z))$.

Ordered rewriting with the combination of A, C, and LC sorts a term lexicographically:

$$(b + c) + a \stackrel{A}{\longmapsto} b + (c + a) \stackrel{C}{\longmapsto} b + (a + c) \stackrel{LC}{\longmapsto} a + (b + c)$$

Martin and Nipkow [5] discuss the theory and give many examples; other algebraic structures are amenable to ordered rewriting, such as boolean rings.

### 10.6.1 Example: sums of natural numbers

This example is again set in HOL (see HOL/ex/NatSum). Theory Arith contains natural numbers arithmetic. Its associated simpset contains many arithmetic laws including distributivity of $\times$ over $+$, while add_ac is a list consisting of the A, C and LC laws for $+$ on type nat. Let us prove the theorem

$$\sum_{i=1}^{n} i = n \times (n+1)/2.$$

A functional sum represents the summation operator under the interpretation $\texttt{sum}\, f\,(n+1) = \sum_{i=0}^{n} f\ i$. We extend Arith as follows:

```
NatSum = Arith +
consts sum     :: [nat=>nat, nat] => nat
primrec "sum" nat
  "sum f 0 = 0"
  "sum f (Suc n) = f(n) + sum f n"
end
```

The primrec declaration automatically adds rewrite rules for sum to the default simpset. We now remove the nat_cancel simplification procedures (in order not to spoil the example) and insert the AC-rules for $+$:

```
Delsimprocs nat_cancel;
Addsimps add_ac;
```

Our desired theorem now reads $\texttt{sum}\,(\lambda i\,.\,i)\,(n+1) = n \times (n+1)/2$. The Isabelle goal has both sides multiplied by 2:

```
Goal "2 * sum (%i.i) (Suc n) = n * Suc n";
  Level 0
  2 * sum (%i. i) (Suc n) = n * Suc n
   1. 2 * sum (%i. i) (Suc n) = n * Suc n
```

Induction should not be applied until the goal is in the simplest form:

```
by (Simp_tac 1);
  Level 1
  2 * sum (%i. i) (Suc n) = n * Suc n
   1. n + (sum (%i. i) n + sum (%i. i) n) = n * n
```

Ordered rewriting has sorted the terms in the left-hand side. The subgoal is now ready for induction:

```
by (induct_tac "n" 1);
  Level 2
  2 * sum (%i. i) (Suc n) = n * Suc n
   1. 0 + (sum (%i. i) 0 + sum (%i. i) 0) = 0 * 0
```

```
 2. !!n. n + (sum (%i. i) n + sum (%i. i) n) = n * n
          ==> Suc n + (sum (%i. i) (Suc n) + sum (%i. i) (Suc n)) =
              Suc n * Suc n
```

Simplification proves both subgoals immediately:

```
by (ALLGOALS Asm_simp_tac);
  Level 3
  2 * sum (%i. i) (Suc n) = n * Suc n
  No subgoals!
```

Simplification cannot prove the induction step if we omit `add_ac` from the simpset. Observe that like terms have not been collected:

```
Level 3
2 * sum (%i. i) (Suc n) = n * Suc n
 1. !!n. n + sum (%i. i) n + (n + sum (%i. i) n) = n + n * n
          ==> n + (n + sum (%i. i) n) + (n + (n + sum (%i. i) n)) =
              n + (n + (n + n * n))
```

Ordered rewriting proves this by sorting the left-hand side. Proving arithmetic theorems without ordered rewriting requires explicit use of commutativity. This is tedious; try it and see!

Ordered rewriting is equally successful in proving $\sum_{i=1}^{n} i^3 = n^2 \times (n+1)^2/4$.

### 10.6.2 Re-orienting equalities

Ordered rewriting with the derived rule `symmetry` can reverse equations:

```
val symmetry = prove_goal HOL.thy "(x=y) = (y=x)"
                 (fn _ => [Blast_tac 1]);
```

This is frequently useful. Assumptions of the form $s = t$, where $t$ occurs in the conclusion but not $s$, can often be brought into the right form. For example, ordered rewriting with `symmetry` can prove the goal

$$f(a) = b \land f(a) = c \to b = c.$$

Here `symmetry` reverses both $f(a) = b$ and $f(a) = c$ because $f(a)$ is lexicographically greater than $b$ and $c$. These re-oriented equations, as rewrite rules, replace $b$ and $c$ in the conclusion by $f(a)$.

Another example is the goal $\neg(t = u) \to \neg(u = t)$. The differing orientations make this appear difficult to prove. Ordered rewriting with `symmetry` makes the equalities agree. (Without knowing more about $t$ and $u$ we cannot say whether they both go to $t = u$ or $u = t$.) Then the simplifier can prove the goal outright.

## 10.7 *Coding simplification procedures

```
mk_simproc: string -> cterm list ->
                (Sign.sg -> thm list -> term -> thm option) -> simproc
```

`mk_simproc` *name lhss proc* makes *proc* a simplification procedure for left-hand side patterns *lhss*. The name just serves as a comment. The function *proc* may be invoked by the simplifier for redex positions matched by one of *lhss* as described below.

Simplification procedures are applied in a two-stage process as follows: The simplifier tries to match the current redex position against any one of the *lhs* patterns of any simplification procedure. If this succeeds, it invokes the corresponding ML function, passing with the current signature, local assumptions and the (potential) redex. The result may be either `None` (indicating failure) or `Some` *thm*.

Any successful result is supposed to be a (possibly conditional) rewrite rule $t \equiv u$ that is applicable to the current redex. The rule will be applied just as any ordinary rewrite rule. It is expected to be already in *internal form*, though, bypassing the automatic preprocessing of object-level equivalences.

As an example of how to write your own simplification procedures, consider eta-expansion of pair abstraction (see also `HOL/Modelcheck/MCSyn` where this is used to provide external model checker syntax).

The `HOL` theory of tuples (see `HOL/Prod`) provides an operator `split` together with some concrete syntax supporting $\lambda\,(x,y)\,.\,b$ abstractions. Assume that we would like to offer a tactic that rewrites any function $\lambda\,p\,.\,f\;p$ (where $p$ is of some pair type) to $\lambda\,(x,y)\,.\,f\;(x,y)$. The corresponding rule is:

```
pair_eta_expand:  (f::'a*'b=>'c) = (%(x, y). f (x, y))
```

Unfortunately, term rewriting using this rule directly would not terminate! We now use the simplification procedure mechanism in order to stop the simplifier from applying this rule over and over again, making it rewrite only actual abstractions. The simplification procedure `pair_eta_expand_proc` is defined as follows:

```
local
  val lhss =
    [read_cterm (sign_of Prod.thy)
                ("f::'a*'b=>'c", TVar (("'z", 0), []))];
  val rew = mk_meta_eq pair_eta_expand;

  fun proc _ _ (Abs _) = Some rew
    | proc _ _ _ = None;
in
  val pair_eta_expand_proc = mk_simproc "pair_eta_expand" lhss proc;
end;
```

This is an example of using `pair_eta_expand_proc`:

```
  1. P (%p::'a * 'a. fst p + snd p + z)
by (simp_tac (simpset() addsimprocs [pair_eta_expand_proc]) 1);
  1. P (%(x::'a,y::'a). x + y + z)
```

In the above example the simplification procedure just did fine grained control over rule application, beyond higher-order pattern matching. Usually, procedures would do some more work, in particular prove particular theorems depending on the current redex.

## 10.8 *Setting up the simplifier

Setting up the simplifier for new logics is complicated. This section describes how the simplifier is installed for intuitionistic first-order logic; the code is largely taken from `FOL/simpdata.ML` of the Isabelle sources.

The simplifier and the case splitting tactic, which reside on separate files, are not part of Pure Isabelle. They must be loaded explicitly by the object-logic as follows:

```
use "$ISABELLE_HOME/src/Provers/simplifier.ML";
use "$ISABELLE_HOME/src/Provers/splitter.ML";
```

Simplification requires converting object-equalities to meta-level rewrite rules. This demands rules stating that equal terms and equivalent formulae are also equal at the meta-level. The rule declaration part of the file `FOL/IFOL.thy` contains the two lines

```
eq_reflection   "(x=y)   ==> (x==y)"
iff_reflection  "(P<->Q) ==> (P==Q)"
```

Of course, you should only assert such rules if they are true for your particular logic. In Constructive Type Theory, equality is a ternary relation of the form $a = b \in A$; the type $A$ determines the meaning of the equality essentially as a partial equivalence relation. The present simplifier cannot be used. Rewriting in

`CTT` uses another simplifier, which resides in the file `Provers/typedsimp.ML` and is not documented. Even this does not work for later variants of Constructive Type Theory that use intensional equality [8].

### 10.8.1 A collection of standard rewrite rules

We first prove lots of standard rewrite rules about the logical connectives. These include cancellation and associative laws. We define a function that echoes the desired law and then supplies it the prover for intuitionistic `FOL`:

```
fun int_prove_fun s =
 (writeln s;
  prove_goal IFOL.thy s
   (fn prems => [ (cut_facts_tac prems 1),
                  (IntPr.fast_tac 1) ]));
```

The following rewrite rules about conjunction are a selection of those proved on `FOL/simpdata.ML`. Later, these will be supplied to the standard simpset.

```
val conj_simps = map int_prove_fun
 ["P & True <-> P",      "True & P <-> P",
  "P & False <-> False", "False & P <-> False",
  "P & P <-> P",
  "P & ~P <-> False",    "~P & P <-> False",
  "(P & Q) & R <-> P & (Q & R)"];
```

The file also proves some distributive laws. As they can cause exponential blowup, they will not be included in the standard simpset. Instead they are merely bound to an ML identifier, for user reference.

```
val distrib_simps  = map int_prove_fun
 ["P & (Q | R) <-> P&Q | P&R",
  "(Q | R) & P <-> Q&P | R&P",
  "(P | Q --> R) <-> (P --> R) & (Q --> R)"];
```

### 10.8.2 Functions for preprocessing the rewrite rules

```
setmksimps : simpset * (thm -> thm list) -> simpset                infix 4
```

The next step is to define the function for preprocessing rewrite rules. This will be installed by calling `setmksimps` below. Preprocessing occurs whenever rewrite rules are added, whether by user command or automatically. Preprocessing involves extracting atomic rewrites at the object-level, then reflecting them to the meta-level.

To start, the function `gen_all` strips any meta-level quantifiers from the front of the given theorem. Usually there are none anyway.

```
fun gen_all th = forall_elim_vars (#maxidx(rep_thm th)+1) th;
```

The function `atomize` analyses a theorem in order to extract atomic rewrite rules. The head of all the patterns, matched by the wildcard `_`, is the coercion function `Trueprop`.

```
fun atomize th = case concl_of th of
    _ $ (Const("op &",_) $ _ $ _)   => atomize(th RS conjunct1) @
                                       atomize(th RS conjunct2)
  | _ $ (Const("op -->",_) $ _ $ _) => atomize(th RS mp)
  | _ $ (Const("All",_) $ _)        => atomize(th RS spec)
  | _ $ (Const("True",_))           => []
  | _ $ (Const("False",_))          => []
  | _                               => [th];
```

There are several cases, depending upon the form of the conclusion:

- Conjunction: extract rewrites from both conjuncts.
- Implication: convert $P \to Q$ to the meta-implication $P \Longrightarrow Q$ and extract rewrites from $Q$; these will be conditional rewrites with the condition $P$.
- Universal quantification: remove the quantifier, replacing the bound variable by a schematic variable, and extract rewrites from the body.
- `True` and `False` contain no useful rewrites.
- Anything else: return the theorem in a singleton list.

The resulting theorems are not literally atomic — they could be disjunctive, for example — but are broken down as much as possible. See the file `ZF/simpdata.ML` for a sophisticated translation of set-theoretic formulae into rewrite rules.

The simplified rewrites must now be converted into meta-equalities. The rule `eq_reflection` converts equality rewrites, while `iff_reflection` converts if-and-only-if rewrites. The latter possibility can arise in two other ways: the negative theorem $\neg P$ is converted to $P \equiv$ `False`, and any other theorem $P$ is converted to $P \equiv$ `True`. The rules `iff_reflection_F` and `iff_reflection_T` accomplish this conversion.

```
val P_iff_F = int_prove_fun "~P ==> (P <-> False)";
val iff_reflection_F = P_iff_F RS iff_reflection;

val P_iff_T = int_prove_fun "P ==> (P <-> True)";
val iff_reflection_T = P_iff_T RS iff_reflection;
```

The function `mk_meta_eq` converts a theorem to a meta-equality using the case analysis described above.

```
fun mk_meta_eq th = case concl_of th of
    _ $ (Const("op =",_)$_$_)   => th RS eq_reflection
  | _ $ (Const("op <->",_)$_$_) => th RS iff_reflection
  | _ $ (Const("Not",_)$_)      => th RS iff_reflection_F
  | _                           => th RS iff_reflection_T;
```

The three functions `gen_all`, `atomize` and `mk_meta_eq` will be composed together and supplied below to `setmksimps`.

### 10.8.3 Making the initial simpset

It is time to assemble these items. We define the infix operator `addcongs` to insert congruence rules; given a list of theorems, it converts their conclusions into meta-equalities and passes them to `addeqcongs`.

```
infix 4 addcongs;
fun ss addcongs congs =
    ss addeqcongs (congs RL [eq_reflection,iff_reflection]);
```

Furthermore, we define the infix operator `addsplits` providing a convenient interface for adding split tactics.

```
infix 4 addsplits;
fun ss addsplits splits = ss addloop (split_tac splits);
```

The list `IFOL_simps` contains the default rewrite rules for intuitionistic first-order logic. The first of these is the reflexive law expressed as the equivalence $(a = a) \leftrightarrow$ `True`; the rewrite rule $a = a$ is clearly useless.

```
val IFOL_simps =
   [refl RS P_iff_T] @ conj_simps @ disj_simps @ not_simps @
    imp_simps @ iff_simps @ quant_simps;
```

The list `triv_rls` contains trivial theorems for the solver. Any subgoal that is simplified to one of these will be removed.

```
val notFalseI = int_prove_fun "~False";
val triv_rls = [TrueI,refl,iff_refl,notFalseI];
```

The basic simpset for intuitionistic FOL is `FOL_basic_ss`. It preprocess rewrites using `gen_all`, `atomize` and `mk_meta_eq`. It solves simplified subgoals using `triv_rls` and assumptions, and by detecting contradictions. It uses `asm_simp_tac` to tackle subgoals of conditional rewrites.

Other simpsets built from `FOL_basic_ss` will inherit these items. In particular, `IFOL_ss`, which introduces `IFOL_simps` as rewrite rules. `FOL_ss` will later extend IFOL_ss with classical rewrite rules such as $\neg\neg P \leftrightarrow P$.

```
fun unsafe_solver prems = FIRST'[resolve_tac (triv_rls @ prems),
                                 atac, etac FalseE];

fun   safe_solver prems = FIRST'[match_tac (triv_rls @ prems),
                                 eq_assume_tac, ematch_tac [FalseE]];

val FOL_basic_ss = empty_ss setsubgoaler asm_simp_tac
                            addsimprocs [defALL_regroup, defEX_regroup]
                            setSSolver   safe_solver
                            setSolver  unsafe_solver
                            setmksimps (map mk_meta_eq o
                                        atomize o gen_all);

val IFOL_ss = FOL_basic_ss addsimps (IFOL_simps @
                                     int_ex_simps @ int_all_simps)
                           addcongs [imp_cong];
```

This simpset takes imp_cong as a congruence rule in order to use contextual information to simplify the conclusions of implications:

$$[\![ ?P \leftrightarrow ?P';\ ?P' \Longrightarrow ?Q \leftrightarrow ?Q' ]\!] \Longrightarrow (?P \to ?Q) \leftrightarrow (?P' \to ?Q')$$

By adding the congruence rule conj_cong, we could obtain a similar effect for conjunctions.

### 10.8.4 Case splitting

To set up case splitting, we must prove the theorem meta_iffD below and pass it to mk_case_split_tac. The tactic split_tac uses mk_meta_eq, defined above, to convert the splitting rules to meta-equalities.

```
val meta_iffD =
    prove_goal FOL.thy "[| P==Q; Q |] ==> P"
        (fn [prem1,prem2] => [rewtac prem1, rtac prem2 1])
fun split_tac splits =
    mk_case_split_tac meta_iffD (map mk_meta_eq splits);
```

The splitter replaces applications of a given function; the right-hand side of the replacement can be anything. For example, here is a splitting rule for conditional expressions:

$$?P(\mathit{if}(?Q, ?x, ?y)) \leftrightarrow (?Q \to ?P(?x)) \land (\neg ?Q \to ?P(?y))$$

Another example is the elimination operator (which happens to be called *split*) for Cartesian products:

$$?P(\mathit{split}(?f, ?p)) \leftrightarrow (\forall a\ b\ .\ ?p = \langle a, b\rangle \to ?P(?f(a, b)))$$

Case splits should be allowed only when necessary; they are expensive and hard to control. Here is an example of use, where `expand_if` is the first rule above:

```
by (simp_tac (simpset() addloop (split_tac [expand_if])) 1);
```

Users would usually prefer the following shortcut using `addsplits`:

```
by (simp_tac (simpset() addsplits [expand_if]) 1);
```

### 10.8.5 Theory setup

```
Simplifier.setup: (theory -> theory) list
```

Advanced theory related features of the simplifier (e.g. implicit simpset support) have to be set up explicitly. The simplifier already provides a suitable setup function definition. This has to be installed into the base theory of any new object-logic via a `setup` declaration.

For example, this is done in `FOL/IFOL.thy` as follows:

```
setup Simplifier.setup
```

Chapter 11

# The Classical Reasoner

Although Isabelle is generic, many users will be working in some extension of classical first-order logic. Isabelle's set theory `ZF` is built upon theory `FOL`, while `HOL` conceptually contains first-order logic as a fragment. Theorem-proving in predicate logic is undecidable, but many researchers have developed strategies to assist in this task.

Isabelle's classical reasoner is an ML functor that accepts certain information about a logic and delivers a suite of automatic tactics. Each tactic takes a collection of rules and executes a simple, non-clausal proof procedure. They are slow and simplistic compared with resolution theorem provers, but they can save considerable time and effort. They can prove theorems such as Pelletier's [11] problems 40 and 41 in seconds:

$$(\exists y\,.\,\forall x\,.\,J(y,x) \leftrightarrow \neg J(x,x)) \rightarrow \neg(\forall x\,.\,\exists y\,.\,\forall z\,.\,J(z,y) \leftrightarrow \neg J(z,x))$$

$$(\forall z\,.\,\exists y\,.\,\forall x\,.\,F(x,y) \leftrightarrow F(x,z) \wedge \neg F(x,x)) \rightarrow \neg(\exists z\,.\,\forall x\,.\,F(x,z))$$

The tactics are generic. They are not restricted to first-order logic, and have been heavily used in the development of Isabelle's set theory. Few interactive proof assistants provide this much automation. The tactics can be traced, and their components can be called directly; in this manner, any proof can be viewed interactively.

The simplest way to apply the classical reasoner (to subgoal $i$) is to type

```
by (Blast_tac i);
```

This command quickly proves most simple formulas of the predicate calculus or set theory. To attempt to prove *all* subgoals using a combination of rewriting and classical reasoning, try

```
by Auto_tac;
```

To do all obvious logical steps, even if they do not prove the subgoal, type one of the following:

```
by (Clarify_tac i);        applies to one subgoal
by Safe_tac;               applies to all subgoals
```

You need to know how the classical reasoner works in order to use it effectively. There are many tactics to choose from, including `Fast_tac` and `Best_tac`.

We shall first discuss the underlying principles, then present the classical reasoner. Finally, we shall see how to instantiate it for new logics. The logics `FOL`, `ZF`, `HOL` and `HOLCF` have it already installed.

## 11.1 The sequent calculus

Isabelle supports natural deduction, which is easy to use for interactive proof. But natural deduction does not easily lend itself to automation, and has a bias towards intuitionism. For certain proofs in classical logic, it can not be called natural. The **sequent calculus**, a generalization of natural deduction, is easier to automate.

A **sequent** has the form $\Gamma \vdash \Delta$, where $\Gamma$ and $\Delta$ are sets of formulae.[1] The sequent

$$P_1, \ldots, P_m \vdash Q_1, \ldots, Q_n$$

is **valid** if $P_1 \land \ldots \land P_m$ implies $Q_1 \lor \ldots \lor Q_n$. Thus $P_1, \ldots, P_m$ represent assumptions, each of which is true, while $Q_1, \ldots, Q_n$ represent alternative goals. A sequent is **basic** if its left and right sides have a common formula, as in $P, Q \vdash Q, R$; basic sequents are trivially valid.

Sequent rules are classified as **right** or **left**, indicating which side of the $\vdash$ symbol they operate on. Rules that operate on the right side are analogous to natural deduction's introduction rules, and left rules are analogous to elimination rules. Recall the natural deduction rules for first-order logic, from *Introduction to Isabelle*. The sequent calculus analogue of $(\to I)$ is the rule

$$\frac{P, \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \to Q} \qquad (\to R)$$

This breaks down some implication on the right side of a sequent; $\Gamma$ and $\Delta$ stand for the sets of formulae that are unaffected by the inference. The analogue of the pair $(\lor I1)$ and $(\lor I2)$ is the single rule

$$\frac{\Gamma \vdash \Delta, P, Q}{\Gamma \vdash \Delta, P \lor Q} \qquad (\lor R)$$

This breaks down some disjunction on the right side, replacing it by both disjuncts. Thus, the sequent calculus is a kind of multiple-conclusion logic.

To illustrate the use of multiple formulae on the right, let us prove the classical theorem $(P \to Q) \lor (Q \to P)$. Working backwards, we reduce this formula to a basic sequent:

$$\cfrac{\cfrac{\cfrac{P, Q \vdash Q, P}{P \vdash Q, (Q \to P)}\,(\to)R}{\vdash (P \to Q), (Q \to P)}\,(\to)R}{\vdash (P \to Q) \lor (Q \to P)}\,(\lor)R$$

[1]For first-order logic, sequents can equivalently be made from lists or multisets of formulae.

This example is typical of the sequent calculus: start with the desired theorem and apply rules backwards in a fairly arbitrary manner. This yields a surprisingly effective proof procedure. Quantifiers add few complications, since Isabelle handles parameters and schematic variables. See Chapter 10 of *ML for the Working Programmer* [9] for further discussion.

## 11.2 Simulating sequents by natural deduction

Isabelle can represent sequents directly, as in the object-logic `LK`. But natural deduction is easier to work with, and most object-logics employ it. Fortunately, we can simulate the sequent $P_1, \ldots, P_m \vdash Q_1, \ldots, Q_n$ by the Isabelle formula

$$[\![P_1; \ldots; P_m; \neg Q_2; \ldots; \neg Q_n]\!] \Longrightarrow Q_1,$$

where the order of the assumptions and the choice of $Q_1$ are arbitrary. Elim-resolution plays a key role in simulating sequent proofs.

We can easily handle reasoning on the left. As discussed in *Introduction to Isabelle*, elim-resolution with the rules $(\vee E)$, $(\bot E)$ and $(\exists E)$ achieves a similar effect as the corresponding sequent rules. For the other connectives, we use sequent-style elimination rules instead of destruction rules such as $(\wedge E1, 2)$ and $(\forall E)$. But note that the rule $(\neg L)$ has no effect under our representation of sequents!

$$\frac{\Gamma \vdash \Delta, P}{\neg P, \Gamma \vdash \Delta} \qquad (\neg L)$$

What about reasoning on the right? Introduction rules can only affect the formula in the conclusion, namely $Q_1$. The other right-side formulae are represented as negated assumptions, $\neg Q_2$, ..., $\neg Q_n$. In order to operate on one of these, it must first be exchanged with $Q_1$. Elim-resolution with the **swap** rule has this effect:

$$[\![\neg P;\ \neg R \Longrightarrow P]\!] \Longrightarrow R \qquad (swap)$$

To ensure that swaps occur only when necessary, each introduction rule is converted into a swapped form: it is resolved with the second premise of $(swap)$. The swapped form of $(\wedge I)$, which might be called $(\neg\wedge E)$, is

$$[\![\neg(P \wedge Q);\ \neg R \Longrightarrow P;\ \neg R \Longrightarrow Q]\!] \Longrightarrow R.$$

Similarly, the swapped form of $(\rightarrow I)$ is

$$[\![\neg(P \rightarrow Q);\ [\![\neg R; P]\!] \Longrightarrow Q]\!] \Longrightarrow R$$

Swapped introduction rules are applied using elim-resolution, which deletes the negated formula. Our representation of sequents also requires the use of ordinary introduction rules. If we had no regard for readability, we could treat the right side more uniformly by representing sequents as

$$[\![P_1; \ldots; P_m; \neg Q_1; \ldots; \neg Q_n]\!] \Longrightarrow \bot.$$

## 11.3 Extra rules for the sequent calculus

As mentioned, destruction rules such as $(\wedge E1,2)$ and $(\forall E)$ must be replaced by sequent-style elimination rules. In addition, we need rules to embody the classical equivalence between $P \to Q$ and $\neg P \vee Q$. The introduction rules $(\vee I1,2)$ are replaced by a rule that simulates $(\vee R)$:

$$(\neg Q \Longrightarrow P) \Longrightarrow P \vee Q$$

The destruction rule $(\to E)$ is replaced by

$$[\![P \to Q;\ \neg P \Longrightarrow R;\ Q \Longrightarrow R]\!] \Longrightarrow R.$$

Quantifier replication also requires special rules. In classical logic, $\exists x.P$ is equivalent to $\neg\forall x.\neg P$; the rules $(\exists R)$ and $(\forall L)$ are dual:

$$\frac{\Gamma \vdash \Delta, \exists x.P, P[t/x]}{\Gamma \vdash \Delta, \exists x.P}\,(\exists R) \qquad \frac{P[t/x], \forall x.P, \Gamma \vdash \Delta}{\forall x.P, \Gamma \vdash \Delta}\,(\forall L)$$

Thus both kinds of quantifier may be replicated. Theorems requiring multiple uses of a universal formula are easy to invent; consider

$$(\forall x\,.\,P(x) \to P(f(x))) \wedge P(a) \to P(f^n(a)),$$

for any $n > 1$. Natural examples of the multiple use of an existential formula are rare; a standard one is $\exists x\,.\,\forall y\,.\,P(x) \to P(y)$.

Forgoing quantifier replication loses completeness, but gains decidability, since the search space becomes finite. Many useful theorems can be proved without replication, and the search generally delivers its verdict in a reasonable time. To adopt this approach, represent the sequent rules $(\exists R)$, $(\exists L)$ and $(\forall R)$ by $(\exists I)$, $(\exists E)$ and $(\forall I)$, respectively, and put $(\forall E)$ into elimination form:

$$[\![\forall x.P(x); P(t) \Longrightarrow Q]\!] \Longrightarrow Q \qquad (\forall E_2)$$

Elim-resolution with this rule will delete the universal formula after a single use. To replicate universal quantifiers, replace the rule by

$$[\![\forall x.P(x);\ [\![P(t); \forall x.P(x)]\!] \Longrightarrow Q]\!] \Longrightarrow Q. \qquad (\forall E_3)$$

To replicate existential quantifiers, replace $(\exists I)$ by

$$[\![\neg(\exists x.P(x)) \Longrightarrow P(t)]\!] \Longrightarrow \exists x.P(x).$$

All introduction rules mentioned above are also useful in swapped form.

Replication makes the search space infinite; we must apply the rules with care. The classical reasoner distinguishes between safe and unsafe rules, applying the latter only when there is no alternative. Depth-first search may well go down a blind alley; best-first search is better behaved in an infinite search space. However, quantifier replication is too expensive to prove any but the simplest theorems.

## 11.4 Classical rule sets

Each automatic tactic takes a **classical set** — a collection of rules, classified as introduction or elimination and as **safe** or **unsafe**. In general, safe rules can be attempted blindly, while unsafe rules must be used with care. A safe rule must never reduce a provable goal to an unprovable set of subgoals.

The rule ($\vee I1$) is unsafe because it reduces $P \vee Q$ to $P$. Any rule is unsafe whose premises contain new unknowns. The elimination rule ($\forall E_2$) is unsafe, since it is applied via elim-resolution, which discards the assumption $\forall x.P(x)$ and replaces it by the weaker assumption $P(?t)$. The rule ($\exists I$) is unsafe for similar reasons. The rule ($\forall E_3$) is unsafe in a different sense: since it keeps the assumption $\forall x.P(x)$, it is prone to looping. In classical first-order logic, all rules are safe except those mentioned above.

The safe/unsafe distinction is vague, and may be regarded merely as a way of giving some rules priority over others. One could argue that ($\vee E$) is unsafe, because repeated application of it could generate exponentially many subgoals. Induction rules are unsafe because inductive proofs are difficult to set up automatically. Any inference is unsafe that instantiates an unknown in the proof state — thus `match_tac` must be used, rather than `resolve_tac`. Even proof by assumption is unsafe if it instantiates unknowns shared with other subgoals — thus `eq_assume_tac` must be used, rather than `assume_tac`.

### 11.4.1 Adding rules to classical sets

Classical rule sets belong to the abstract type `claset`, which supports the following operations (provided the classical reasoner is installed!):

```
empty_cs    : claset
print_cs    : claset -> unit
rep_cs      : claset -> safeEs: thm list, safeIs: thm list,
                         hazEs: thm list,  hazIs: thm list,
                         swrappers: (string * wrapper) list,
                         uwrappers: (string * wrapper) list,
                         safe0_netpair: netpair, safep_netpair: netpair,
                           haz_netpair: netpair,   dup_netpair: netpair
addSIs      : claset * thm list -> claset                          infix 4
addSEs      : claset * thm list -> claset                          infix 4
addSDs      : claset * thm list -> claset                          infix 4
addIs       : claset * thm list -> claset                          infix 4
addEs       : claset * thm list -> claset                          infix 4
addDs       : claset * thm list -> claset                          infix 4
delrules    : claset * thm list -> claset                          infix 4
```

The add operations ignore any rule already present in the claset with the same classification (such as Safe Introduction). They print a warning if the rule has already been added with some other classification, but add the rule anyway.

Calling `delrules` deletes all occurrences of a rule from the claset, but see the warning below concerning destruction rules.

`empty_cs` is the empty classical set.

`print_cs` *cs* displays the printable contents of *cs*, which is the rules. All other parts are non-printable.

`rep_cs` *cs* decomposes *cs* as a record of its internal components, namely the safe introduction and elimination rules, the unsafe introduction and elimination rules, the lists of safe and unsafe wrappers (see 11.4.2), and the internalized forms of the rules.

*cs* `addSIs` *rules* adds safe introduction *rules* to *cs*.

*cs* `addSEs` *rules* adds safe elimination *rules* to *cs*.

*cs* `addSDs` *rules* adds safe destruction *rules* to *cs*.

*cs* `addIs` *rules* adds unsafe introduction *rules* to *cs*.

*cs* `addEs` *rules* adds unsafe elimination *rules* to *cs*.

*cs* `addDs` *rules* adds unsafe destruction *rules* to *cs*.

*cs* `delrules` *rules* deletes *rules* from *cs*. It prints a warning for those rules that are not in *cs*.

! If you added *rule* using `addSDs` or `addDs`, then you must delete it as follows:

*cs* `delrules [make_elim` *rule*`]`

This is necessary because the operators `addSDs` and `addDs` convert the destruction rules to elimination rules by applying `make_elim`, and then insert them using `addSEs` and `addEs`, respectively.

Introduction rules are those that can be applied using ordinary resolution. The classical set automatically generates their swapped forms, which will be applied using elim-resolution. Elimination rules are applied using elim-resolution. In a classical set, rules are sorted by the number of new subgoals they will yield; rules that generate the fewest subgoals will be tried first (see §3.4.1).

### 11.4.2 Modifying the search step

For a given classical set, the proof strategy is simple. Perform as many safe inferences as possible; or else, apply certain safe rules, allowing instantiation of unknowns; or else, apply an unsafe rule. The tactics also eliminate assumptions of the form $x = t$ by substitution if they have been set up to do so (see `hyp_subst_tacs` in §11.6 below). They may perform a form of Modus Ponens: if there are assumptions $P \to Q$ and $P$, then replace $P \to Q$ by $Q$.

The classical reasoning tactics — except `blast_tac`! — allow you to modify this basic proof strategy by applying two lists of arbitrary **wrapper tacticals** to it. The first wrapper list, which is considered to contain safe wrappers only, affects `safe_step_tac` and all the tactics that call it. The second one, which may contain unsafe wrappers, affects `step_tac`, `slow_step_tac`, and the tactics that call them. A wrapper transforms each step of the search, for example by invoking other tactics before or alternatively to the original step tactic. All members of a wrapper list are applied in turn to the respective step tactic.

Initially the two wrapper lists are empty, which means no modification of the step tactics. Safe and unsafe wrappers are added to a claset with the functions given below, supplying them with wrapper names. These names may be used to selectively delete wrappers.

```
type wrapper = (int -> tactic) -> (int -> tactic);

addSWrapper  : claset * (string *  wrapper       ) -> claset       infix 4
addSbefore   : claset * (string * (int -> tactic)) -> claset       infix 4
addSaltern   : claset * (string * (int -> tactic)) -> claset       infix 4
delSWrapper  : claset *  string                    -> claset       infix 4

addWrapper   : claset * (string *  wrapper       ) -> claset       infix 4
addbefore    : claset * (string * (int -> tactic)) -> claset       infix 4
addaltern    : claset * (string * (int -> tactic)) -> claset       infix 4
delWrapper   : claset *  string                    -> claset       infix 4

addSss       : claset * simpset -> claset                          infix 4
addss        : claset * simpset -> claset                          infix 4
```

*cs* `addSWrapper` (*name*, *wrapper*) adds a new wrapper, which should yield a safe tactic, to modify the existing safe step tactic.

*cs* `addSbefore` (*name*, *tac*) adds the given tactic as safe wrapper, such that it is applied *before* each safe step of the search.

*cs* `addSaltern` (*name*, *tac*) adds the given tactic as safe wrapper, such that it is applied when a safe step of the search would fail.

*cs* `delSWrapper` *name* deletes the safe wrapper with the given name.

*cs* `addWrapper` (*name*, *wrapper*) adds a new wrapper to modify the existing (unsafe) step tactic.

*cs* `addbefore` (*name*, *tac*) adds the given tactic as unsafe wrapper, such that it is applied *before* each step of the search.

*cs* `addaltern` (*name*, *tac*) adds the given tactic as unsafe wrapper, such that it is applied when a step of the search would fail.

*cs* `delWrapper` *name* deletes the unsafe wrapper with the given name.

*cs* `addSss` *ss* adds the simpset *ss* to the classical set. The assumptions and goal will be simplified, in a rather safe way, after each safe step of the search.

*cs* `addss` *ss* adds the simpset *ss* to the classical set. The assumptions and goal will be simplified, before the each unsafe step of the search.

The wrapper tacticals underly the operators addss and addSss, which are used as primitives for the automatic tactics described in §11.5.3. Strictly speaking, both operators are not part of the classical reasoner. They should be defined when the simplifier is installed:

```
infix 4 addSss addss;
fun  cs addSss ss = cs addSaltern ("safe_asm_full_simp_tac",
                         CHANGED o (safe_asm_full_simp_tac ss));
fun  cs addss  ss = cs addbefore  ("asm_full_simp_tac",
                                    asm_full_simp_tac ss);
```

**!** Being defined as wrappers, these operators are inappropriate for adding more than one simpset at a time: the simpset added last overwrites any earlier ones. When a simpset combined with a claset is to be augmented, this should done *before* combining it with the claset.

## 11.5 The classical tactics

If installed, the classical module provides powerful theorem-proving tactics. Most of them have capitalized analogues that use the default claset; see §11.5.7.

### 11.5.1 Semi-automatic tactics

```
clarify_tac      : claset -> int -> tactic
clarify_step_tac : claset -> int -> tactic
```

Use these when the automatic tactics fail. They perform all the obvious logical inferences that do not split the subgoal. The result is a simpler subgoal that can be tackled by other means, such as by instantiating quantifiers yourself.

`clarify_tac` *cs* *i* performs a series of safe steps on subgoal *i* by repeatedly calling `clarify_step_tac`.

`clarify_step_tac` *cs* *i* performs a safe step on subgoal *i*. No splitting step is applied; for example, the subgoal $A \land B$ is left as a conjunction. Proof by assumption, Modus Ponens, etc., may be performed provided they do not instantiate unknowns. Assumptions of the form $x = t$ may be eliminated. The user-supplied safe wrapper tactical is applied.

### 11.5.2 The tableau prover

The tactic `blast_tac` searches for a proof using a fast tableau prover, coded directly in ML. It then reconstructs the proof using Isabelle tactics. It is faster and more powerful than the other classical reasoning tactics, but has major limitations too.

- It does not use the wrapper tacticals described above, such as `addss`.
- It ignores types, which can cause problems in `HOL`. If it applies a rule whose types are inappropriate, then proof reconstruction will fail.
- It does not perform higher-order unification, as needed by the rule `rangeI` in `HOL` and `RepFunI` in `ZF`. There are often alternatives to such rules, for example `range_eqI` and `RepFun_eqI`.
- The message `Function Var's argument not a bound variable` relates to the lack of higher-order unification. Function variables may only be applied to parameters of the subgoal.
- Its proof strategy is more general than `fast_tac`'s but can be slower. If `blast_tac` fails or seems to be running forever, try `fast_tac` and the other tactics described below.

```
blast_tac        : claset -> int -> tactic
Blast.depth_tac  : claset -> int -> int -> tactic
Blast.trace      : bool ref
```
**initially false**

The two tactics differ on how they bound the number of unsafe steps used in a proof. While `blast_tac` starts with a bound of zero and increases it successively to 20, `Blast.depth_tac` applies a user-supplied search bound.

`blast_tac` *cs* *i* tries to prove subgoal *i* using iterative deepening to increase the search bound.

`Blast.depth_tac` *cs* *lim* *i* tries to prove subgoal *i* using a search bound of *lim*. Often a slow proof using `blast_tac` can be made much faster by supplying the successful search bound to this tactic instead.

`set Blast.trace;` causes the tableau prover to print a trace of its search. At each step it displays the formula currently being examined and reports whether the branch has been closed, extended or split.

### 11.5.3 Automatic tactics

```
type clasimpset = claset * simpset;
auto_tac       : clasimpset ->        tactic
force_tac      : clasimpset -> int -> tactic
auto           : unit -> unit
force          : int  -> unit
```

The automatic tactics attempt to prove goals using a combination of simplification and classical reasoning.

`auto_tac` $(cs, ss)$ is intended for situations where there are a lot of mostly trivial subgoals; it proves all the easy ones, leaving the ones it cannot prove. (Unfortunately, attempting to prove the hard ones may take a long time.)

`force_tac` $(cs, ss)$ $i$ is intended to prove subgoal $i$ completely. It tries to apply all fancy tactics it knows about, performing a rather exhaustive search.

They must be supplied both a simpset and a claset; therefore they are most easily called as `Auto_tac` and `Force_tac`, which use the default claset and simpset (see §11.5.7 below). For interactive use, the shorthand `auto();` abbreviates `by Auto_tac;` while `force 1;` abbreviates `by (Force_tac 1);`

### 11.5.4 Other classical tactics

```
fast_tac      : claset -> int -> tactic
best_tac      : claset -> int -> tactic
slow_tac      : claset -> int -> tactic
slow_best_tac : claset -> int -> tactic
```

These tactics attempt to prove a subgoal using sequent-style reasoning. Unlike `blast_tac`, they construct proofs directly in Isabelle. Their effect is restricted (by `SELECT_GOAL`) to one subgoal; they either prove this subgoal or fail. The `slow_` versions conduct a broader search.[2]

The best-first tactics are guided by a heuristic function: typically, the total size of the proof state. This function is supplied in the functor call that sets up the classical reasoner.

`fast_tac` $cs$ $i$ applies `step_tac` using depth-first search, to prove subgoal $i$.

`best_tac` $cs$ $i$ applies `step_tac` using best-first search, to prove subgoal $i$.

`slow_tac` $cs$ $i$ applies `slow_step_tac` using depth-first search, to prove subgoal $i$.

`slow_best_tac` $cs$ $i$ applies `slow_step_tac` using best-first search, to prove subgoal $i$.

[2]They may, when backtracking from a failed proof attempt, undo even the step of proving a subgoal by assumption.

### 11.5.5 Depth-limited automatic tactics

```
depth_tac  : claset -> int -> int -> tactic
deepen_tac : claset -> int -> int -> tactic
```

These work by exhaustive search up to a specified depth. Unsafe rules are modified to preserve the formula they act on, so that it be used repeatedly. They can prove more goals than `fast_tac` can but are much slower, for example if the assumptions have many universal quantifiers.

The depth limits the number of unsafe steps. If you can estimate the minimum number of unsafe steps needed, supply this value as $m$ to save time.

`depth_tac` *cs* $m$ $i$ tries to prove subgoal $i$ by exhaustive search up to depth $m$.

`deepen_tac` *cs* $m$ $i$ tries to prove subgoal $i$ by iterative deepening. It calls `depth_tac` repeatedly with increasing depths, starting with $m$.

### 11.5.6 Single-step tactics

```
safe_step_tac : claset -> int -> tactic
safe_tac      : claset        -> tactic
inst_step_tac : claset -> int -> tactic
step_tac      : claset -> int -> tactic
slow_step_tac : claset -> int -> tactic
```

The automatic proof procedures call these tactics. By calling them yourself, you can execute these procedures one step at a time.

`safe_step_tac` *cs* $i$ performs a safe step on subgoal $i$. The safe wrapper tacticals are applied to a tactic that may include proof by assumption or Modus Ponens (taking care not to instantiate unknowns), or substitution.

`safe_tac` *cs* repeatedly performs safe steps on all subgoals. It is deterministic, with at most one outcome.

`inst_step_tac` *cs* $i$ is like `safe_step_tac`, but allows unknowns to be instantiated.

`step_tac` *cs* $i$ is the basic step of the proof procedure. The (unsafe) wrapper tacticals are applied to a tactic that tries `safe_tac`, `inst_step_tac`, or applies an unsafe rule from *cs*.

`slow_step_tac` resembles `step_tac`, but allows backtracking between using safe rules with instantiation (`inst_step_tac`) and using unsafe rules. The resulting search space is larger.

### 11.5.7 The current claset

Each theory is equipped with an implicit *current claset*. This is a default set of classical rules. The underlying idea is quite similar to that of a current simpset described in §10.1; please read that section, including its warnings. The implicit claset can be accessed as follows:

```
claset        : unit -> claset
claset_ref    : unit -> claset ref
claset_of     : theory -> claset
claset_ref_of : theory -> claset ref
print_claset  : theory -> unit
```

The tactics

```
Blast_tac        : int -> tactic
Auto_tac         :        tactic
Force_tac        : int -> tactic
Fast_tac         : int -> tactic
Best_tac         : int -> tactic
Deepen_tac       : int -> int -> tactic
Clarify_tac      : int -> tactic
Clarify_step_tac : int -> tactic
Safe_tac         :        tactic
Safe_step_tac    : int -> tactic
Step_tac         : int -> tactic
```

make use of the current claset. For example, `Blast_tac` is defined as

```
fun Blast_tac i st = blast_tac (claset()) i st;
```

and gets the current claset, only after it is applied to a proof state. The functions

```
AddSIs, AddSEs, AddSDs, AddIs, AddEs, AddDs: thm list -> unit
```

are used to add rules to the current claset. They work exactly like their lower case counterparts, such as `addSIs`. Calling

```
Delrules : thm list -> unit
```

deletes rules from the current claset.

### 11.5.8 Other useful tactics

```
contr_tac    :             int -> tactic
mp_tac       :             int -> tactic
eq_mp_tac    :             int -> tactic
swap_res_tac : thm list -> int -> tactic
```

These can be used in the body of a specialized search.

`contr_tac` $i$ solves subgoal $i$ by detecting a contradiction among two assumptions of the form $P$ and $\neg P$, or fail. It may instantiate unknowns. The tactic can produce multiple outcomes, enumerating all possible contradictions.

`mp_tac` $i$ is like `contr_tac`, but also attempts to perform Modus Ponens in subgoal $i$. If there are assumptions $P \to Q$ and $P$, then it replaces $P \to Q$ by $Q$. It may instantiate unknowns. It fails if it can do nothing.

`eq_mp_tac` $i$ is like `mp_tac` $i$, but may not instantiate unknowns — thus, it is safe.

`swap_res_tac` *thms* $i$ refines subgoal $i$ of the proof state using *thms*, which should be a list of introduction rules. First, it attempts to prove the goal using `assume_tac` or `contr_tac`. It then attempts to apply each rule in turn, attempting resolution and also elim-resolution with the swapped form.

### 11.5.9 Creating swapped rules

```
swapify   : thm list -> thm list
joinrules : thm list * thm list -> (bool * thm) list
```

`swapify` *thms* returns a list consisting of the swapped versions of *thms*, regarded as introduction rules.

`joinrules (`*intrs*`,` *elims*`)` joins introduction rules, their swapped versions, and elimination rules for use with `biresolve_tac`. Each rule is paired with `false` (indicating ordinary resolution) or `true` (indicating elim-resolution).

## 11.6 Setting up the classical reasoner

Isabelle's classical object-logics, including `FOL` and `HOL`, have the classical reasoner already set up. When defining a new classical logic, you should set up the reasoner yourself. It consists of the ML functor `ClassicalFun`, which takes the argument signature `CLASSICAL_DATA`:

```
signature CLASSICAL_DATA =
  sig
  val mp             : thm
  val not_elim       : thm
  val swap           : thm
  val sizef          : thm -> int
  val hyp_subst_tacs : (int -> tactic) list
  end;
```

Thus, the functor requires the following items:

`mp` should be the Modus Ponens rule $[\![ ?P \to ?Q;\ ?P ]\!] \Longrightarrow ?Q$.

`not_elim` should be the contradiction rule $[\![\neg ?P;\ ?P]\!] \Longrightarrow ?R$.

`swap` should be the swap rule $[\![\neg ?P;\ \neg ?R \Longrightarrow ?P]\!] \Longrightarrow ?R$.

`sizef` is the heuristic function used for best-first search. It should estimate the size of the remaining subgoals. A good heuristic function is `size_of_thm`, which measures the size of the proof state. Another size function might ignore certain subgoals (say, those concerned with type checking). A heuristic function might simply count the subgoals.

`hyp_subst_tacs` is a list of tactics for substitution in the hypotheses, typically created by `HypsubstFun` (see Chapter 9). This list can, of course, be empty. The tactics are assumed to be safe!

The functor is not at all sensitive to the formalization of the object-logic. It does not even examine the rules, but merely applies them according to its fixed strategy. The functor resides in `Provers/classical.ML` in the Isabelle sources.

# Bibliography


[1] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[2] E. Charniak, C. K. Riesbeck, and D. V. McDermott. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, 1980.

[3] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[4] K. Futatsugi, J.A. Goguen, Jean-Pierre Jouannaud, and J. Meseguer. Principles of OBJ2. In *Symposium on Principles of Programming Languages*, pages 52–66, 1985.

[5] Ursula Martin and Tobias Nipkow. Ordered rewriting and confluence. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, LNAI 449, pages 366–380. Springer, 1990.

[6] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.

[7] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[8] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford University Press, 1990.

[9] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[10] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[11] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986. Errata, JAR 4 (1988), 235–236 and JAR 18 (1997), 135.

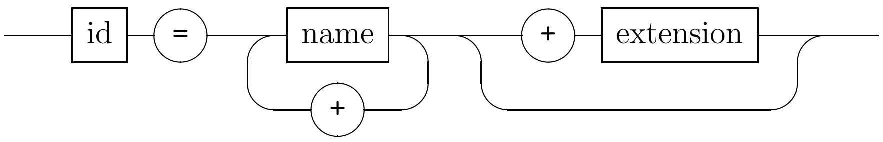
# Appendix A

# Syntax of Isabelle Theories

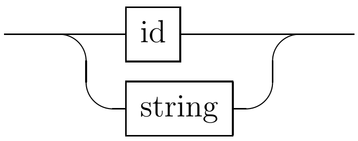
Below we present the full syntax of theory definition files as provided by `Pure` Isabelle — object-logics may add their own sections. §6.1 explains the meanings of these constructs. The syntax obeys the following conventions:

- `Typewriter font` denotes terminal symbols.
- *id*, *tid*, *nat*, *string* and *longident* are the lexical classes of identifiers, type identifiers, natural numbers, quoted strings (without the need for `\...\` between lines) and long qualified ML identifiers. The categories *id*, *tid*, *nat* are fully defined in §7.
- *text* is all text from the current position to the end of file, *verbatim* is any text enclosed in `{|...|}`
- Comments in theories take the form `(*...*)` and may be nested, just as in ML.

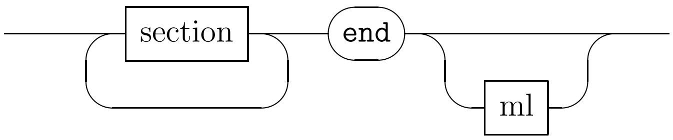
*theoryDef*



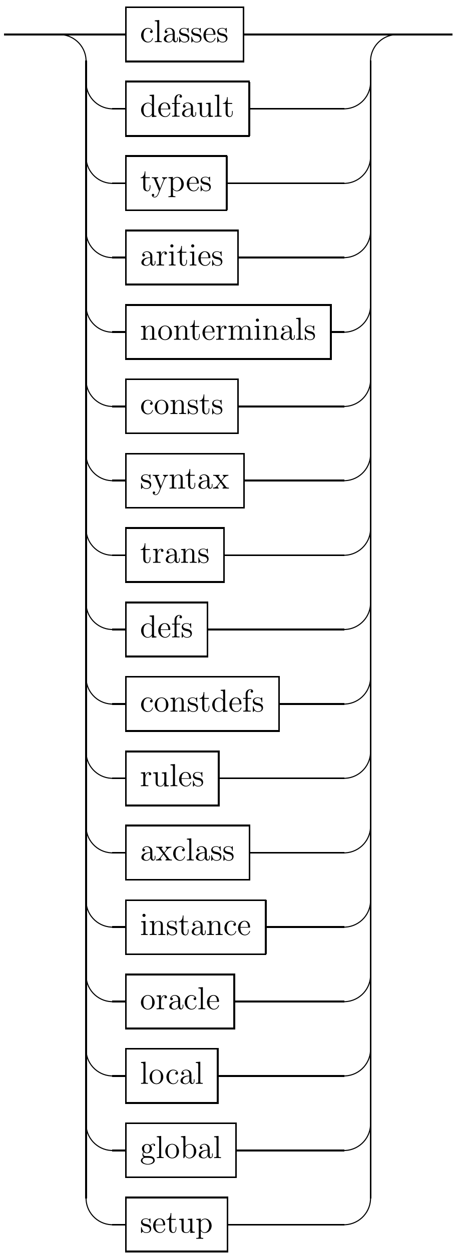

*name*



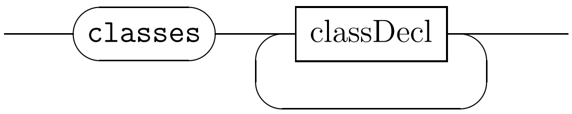

*extension*



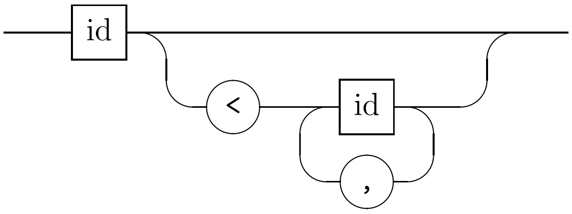
*section*



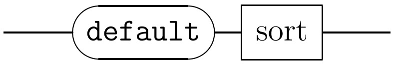

*classes*



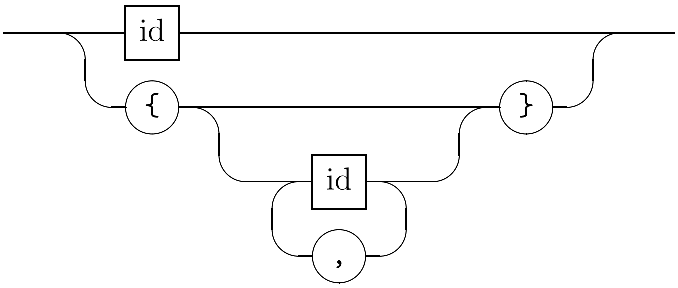
*classDecl*



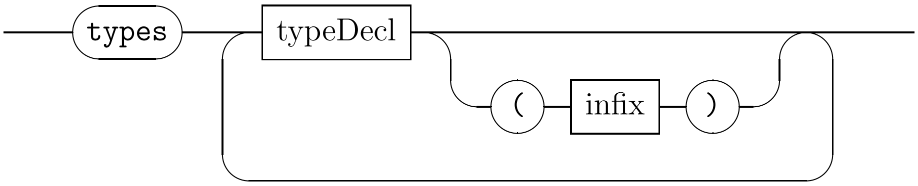

*default*



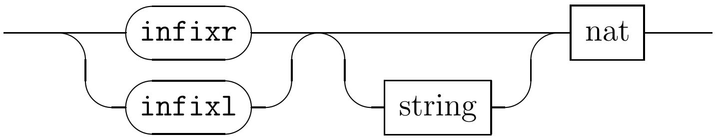

*sort*



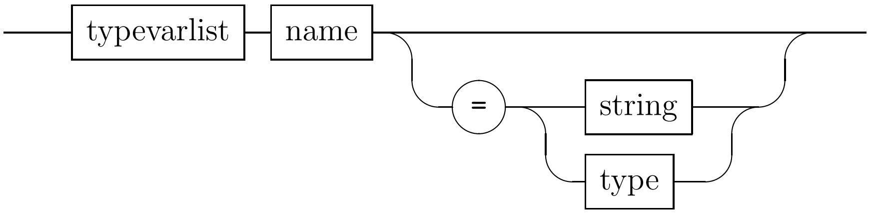

*types*



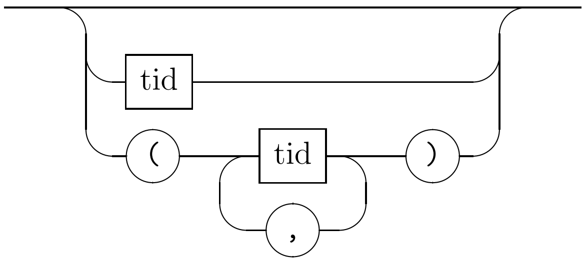

*infix*



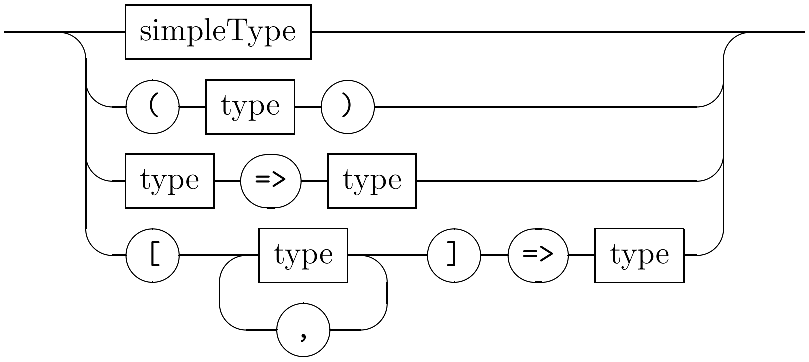

*typeDecl*



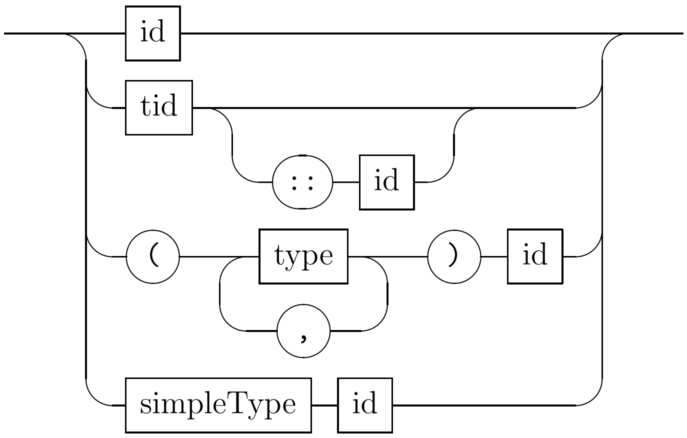
*typevarlist*



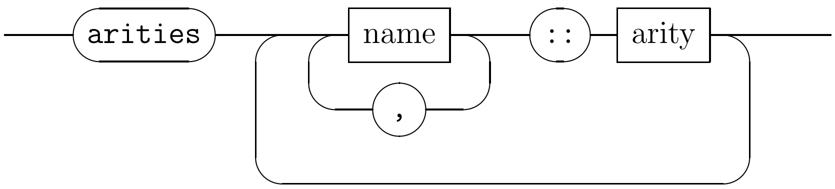

*type*



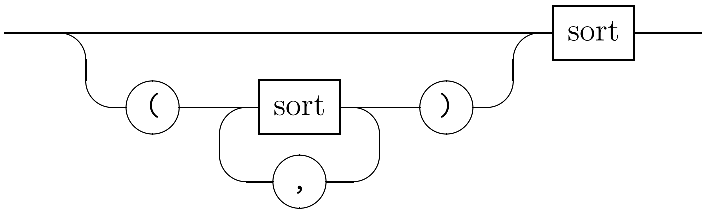

*simpleType*



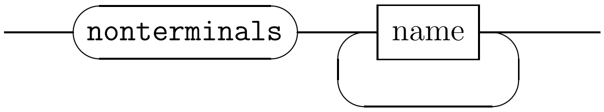

*arities*



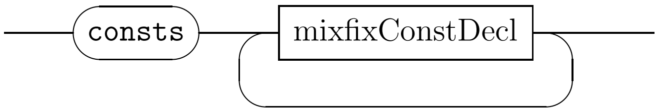
*arity*



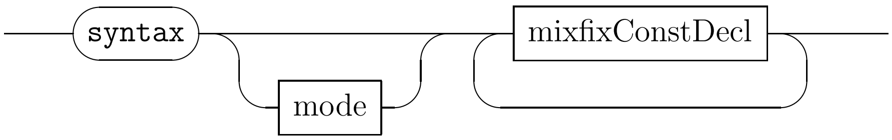

*nonterminals*



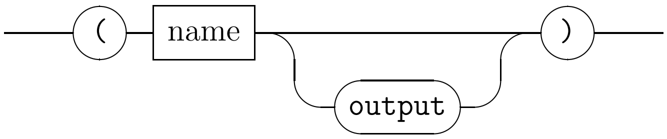

*consts*



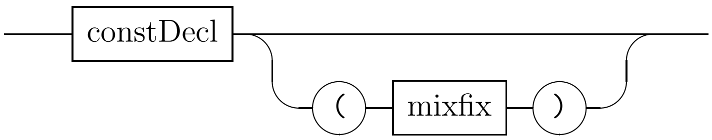

*syntax*



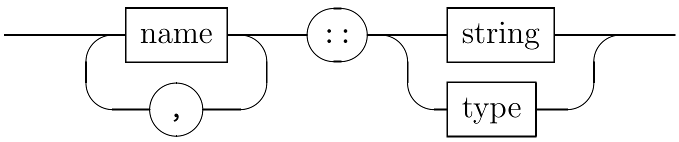

*mode*



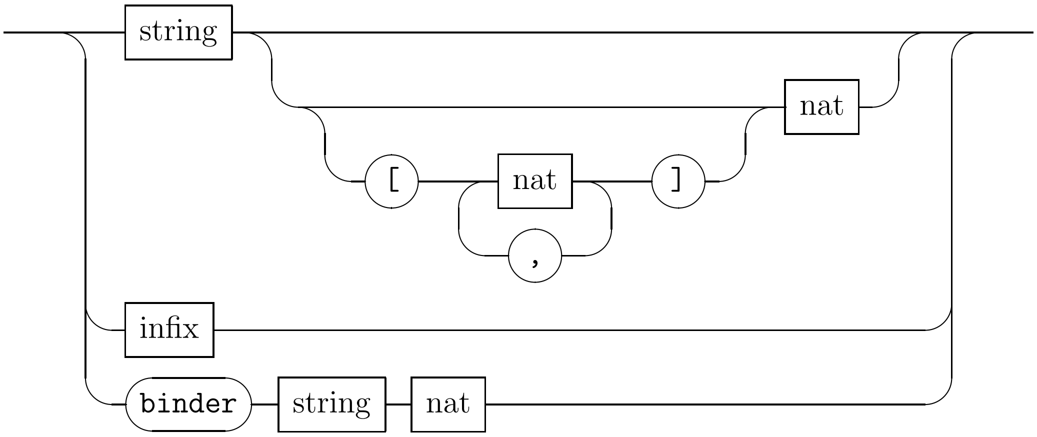

*mixfixConstDecl*



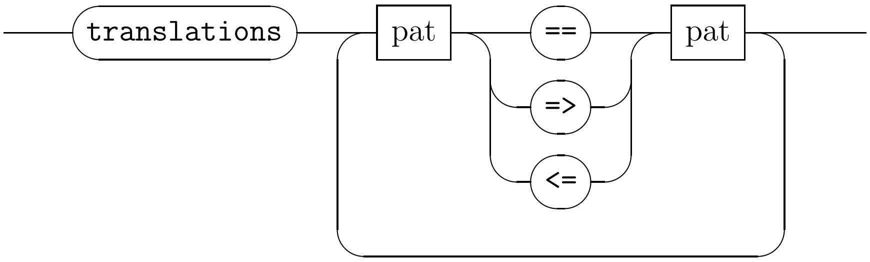

*constDecl*



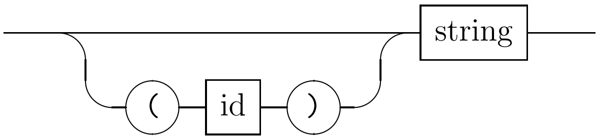
*mixfix*



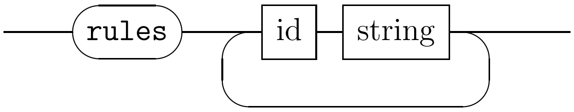

*trans*



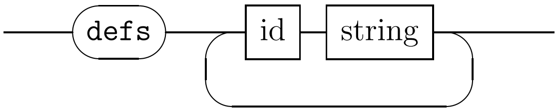

*pat*



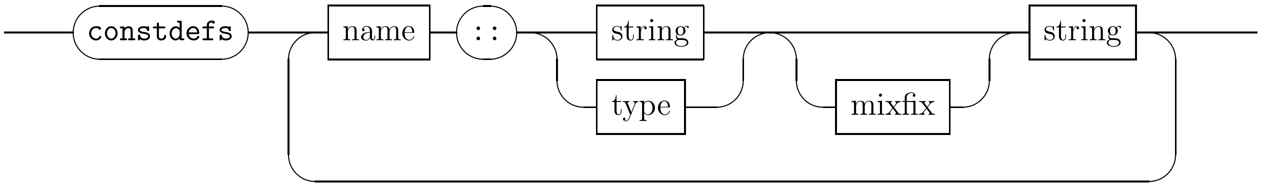

*rules*



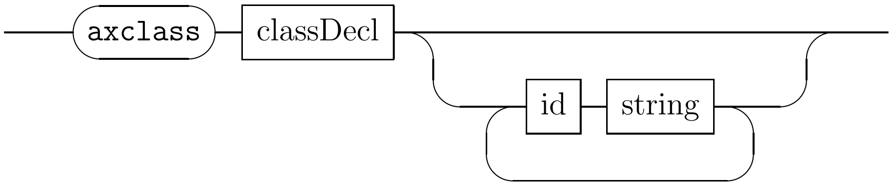

*defs*



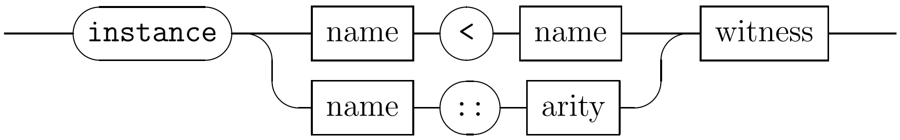
*constdefs*



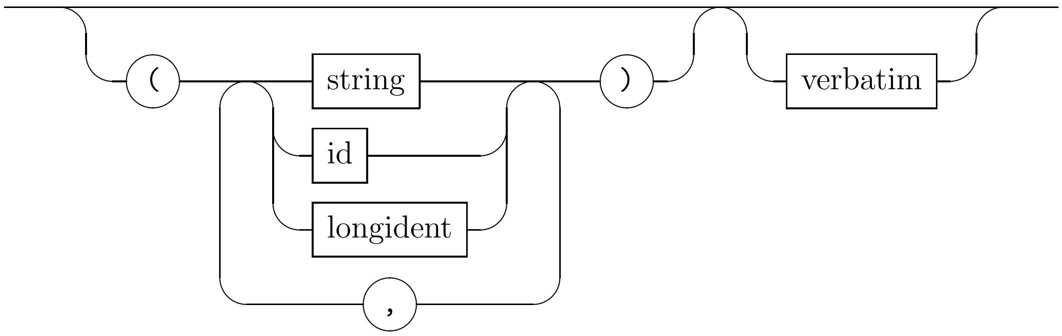

*axclass*



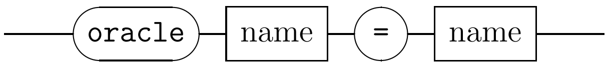

*instance*




*witness*



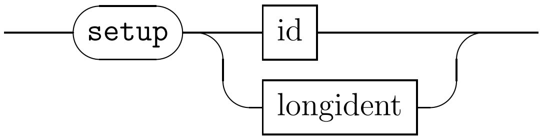

*oracle*



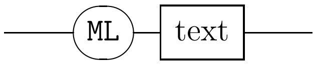

*local*


*global*

*setup*




*ml*



# Index