Old Isabelle Reference Manual

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

With Contributions by Tobias Nipkow and Markus Wenzel

9 October 2011

Note: this document is part of the earlier Isabelle documentation and is mostly outdated. Fully obsolete parts of the original text have already been removed. The remaining material covers some aspects that did not make it into the newer manuals yet.

Acknowledgements

Tobias Nipkow, of T. U. Munich, wrote most of Chapters 4 and 7. Markus Wenzel contributed to Chapter 5. Jeremy Dawson, 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, GR/M75440) and by ESPRIT (projects 3245: Logical Frameworks, and 6453: Types), and by the DFG Schwerpunktprogramm *Deduktion*.

Contents

Chapter 1

Tactics

1.1 Other basic tactics

1.1.1 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
subgoals_tac  : 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 §**??**. It adds the resulting theorem as a new assumption to subgoal i.

`subgoal_tac` *formula* i adds the *formula* as an 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.

1.1.2 "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.

1.1.3 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 7).

1.1.4 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`.

1.2 Obscure tactics

1.2.1 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.

1.2.2 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 (§**??**). Normally flex-flex constraints can be ignored; they often disappear as unknowns get instantiated.

1.2.3 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 ψ 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.

1.3 *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.

1.3.1 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 $(\mathit{flag}, \mathit{rule})$ 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 §**??**).

`subgoals_of_brl(`*flag*`,`*rule*`)` returns the number of new subgoals that biresolution 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.

1.3.2 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 [3, 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 λ-abstractions as unknowns, since they could η-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*.

Chapter 2

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.

2.1 The basic tacticals

2.1.1 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.

2.1.2 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.

2.1.3 Repetition tacticals

```
TRY              : tactic -> tactic
REPEAT_DETERM    : tactic -> tactic
REPEAT_DETERM_N  : int -> tactic -> tactic
REPEAT           : tactic -> tactic
REPEAT1          : tactic -> tactic
DETERM_UNTIL     : (thm -> bool) -> 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_DETERM_N` *n* *tac* is like `REPEAT_DETERM` *tac* but the number of repititions is bound by *n* (unless negative).

`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.

`DETERM_UNTIL` *p tac* applies *tac* to the proof state and, recursively, to the head of the resulting sequence, until the predicate *p* (applied on the proof state) yields *true*. It fails if *tac* fails on any of the intermediate states. It is deterministic, discarding alternative outcomes.

`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.

2.1.4 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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                                  infix 1
trace_BEST_FIRST: bool ref                           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.

2.2.4 Auxiliary tacticals for searching

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

`SOLVE` *tac* applies *tac* to the proof state and then fails iff there are subgoals left.

`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.

`DETERM_UNTIL_SOLVED` *tac* forces repeated deterministic application of *tac* to the proof state until the goal is solved completely.

2.2.5 Predicates and functions useful for searching

```
has_fewer_prems : int -> thm -> bool
eq_thm          : thm * thm -> bool
eq_thm_prop     : 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 compatible signatures. Both theorems must have the same conclusions, the same hypotheses (in the same order), and the same set of sort hypotheses. Names of bound variables are ignored.

`eq_thm_prop (`thm_1`, `thm_2`)` reports whether the propositions of thm_1 and thm_2 are equal. 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`.

2.3 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`.

2.3.1 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 ϕ. 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 θ'_1, …, θ'_k of meta-level assumptions. In these theorems, the subgoal's parameters (x_1, …, 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 θ'_1, …, θ'_k, and are restored afterwards; the `METAHYPS` call cannot instantiate them. Unknowns in θ may be instantiated. New unknowns in ϕ_1, …, ϕ_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.

2.3.2 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`.

2.3.3 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` $[tacf_1(x), \ldots, tacf_n(x)]$

2.3.4 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 3

Theorems and Forward Proof

Theorems, which represent the axioms, theorems and rules of object-logics, have type `thm`. This chapter describes operations that join theorems in forward proof. Most theorem operations are intended for advanced applications, such as programming new proof procedures.

3.1 Basic operations on theorems

3.1.1 Forward proof: joining rules by resolution

```
RSN : thm * (int * thm) -> thm                          infix
RS  : thm * thm -> thm                                  infix
MRS : thm list * thm -> thm                             infix
OF  : thm * thm list -> 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` (§??).

thm_1 `RSN` (i, thm_2) resolves the conclusion of thm_1 with the ith 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 $i = n$, …, 1. This applies thm_n, …, 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.

thm `OF` $[thm_1, \ldots, thm_n]$ is the same as $[thm_1, \ldots, thm_n]$ `MRS` thm, with slightly more readable argument order, though.

$thms_1$ `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 ith 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.

3.1.2 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 it leaves the conclusion unchanged. This rule is the basis for `rewrite_goals_tac`, but it serves little purpose in forward proof.

3.1.3 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 §**??**, 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.

3.1.4 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
permute_prems    : int -> int -> thm -> thm
rearrange_prems  :   int list -> 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*, which should be 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 (to the right if $k < 0$). It simply calls `permute_prems`, below, with $j = 0$. Used with `eresolve_tac`, it gives the effect of applying the tactic to some other premise of *thm* than the first.

`permute_prems` *j k thm* rotates the premises of *thm* leaving the first j premises unchanged. It requires $0 \leq j \leq n$, where n is the number of premises. If k is positive then it rotates the remaining $n - j$ premises to the left; if k is negative then it rotates the premises to the right.

`rearrange_prems` *ps thm* permutes the premises of *thm* where the value at the i-th position (counting from 0) in the list *ps* gives the position within the original thm to be transferred to position i. Any remaining trailing positions are left unchanged.

3.1.5 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: bool * deriv, maxidx: int,
                     shyps: sort list, hyps: term list, prop: term}
crep_thm   : thm -> {sign_ref: Sign.sg_ref, der: bool * 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.

3.1.6 *Sort hypotheses

```
strip_shyps         : thm -> thm
strip_shyps_warning : thm -> thm
```

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 automatically removes extraneous sorts from the `shyps` field at the end of a proof, provided that non-emptiness can be established by looking at the theorem's signature: from the `classes` and `arities` information. This operation is performed by `strip_shyps` and `strip_shyps_warning`.

`strip_shyps` *thm* removes any extraneous sort hypotheses that can be witnessed from the type signature.

`strip_shyps_warning` is like `strip_shyps`, but issues a warning message of any pending sort hypotheses that do not have a (syntactic) witness.

3.1.7 Tracing flags for unification

```
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`;` prevents unification from searching past the depth n. Because of this bound, higher-order unification cannot return an infinite sequence, though it can return an exponentially long one. The search rarely approaches the default value of 20. If the search is cut off, unification prints a warning `Unification bound exceeded`.

3.2 *Primitive meta-level inference rules

3.2.1 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 ψ and ϕ to the conclusion $\phi \equiv \psi$; the assumptions are those of the first premise with ϕ removed, plus those of the second premise with ψ removed.

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

3.2.2 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 $a \equiv c$.

3.2.3 The λ-conversion rules

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

There is no rule for α-conversion because Isabelle regards α-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 $f(a) \equiv g(b)$.

3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 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 ψ and $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$. For each s that unifies ψ and ϕ_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 ψ 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.

3.3.4 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 ϕ is the value of *ct*. This is the initial state for a goal-directed proof of ϕ. 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.

3.4 Proof terms

Isabelle can record the full meta-level proof of each theorem. The proof term contains all logical inferences in detail. Resolution and rewriting steps are broken down to primitive rules of the meta-logic. The proof term can be inspected by a separate proof-checker, for example.

According to the well-known *Curry-Howard isomorphism*, a proof can be viewed as a λ-term. Following this idea, proofs in Isabelle are internally represented by a datatype similar to the one for terms described in §**??**.

```
infix 8 % %%;

datatype proof =
   PBound of int
 | Abst of string * typ option * proof
 | AbsP of string * term option * proof
 | op % of proof * term option
 | op %% of proof * proof
 | Hyp of term
 | PThm of (string * (string * string list) list) *
          proof * term * typ list option
 | PAxm of string * term * typ list option
 | Oracle of string * term * typ list option
 | MinProof of proof list;
```

`Abst` `(`a`,` τ`,` *prf*`)` is the abstraction over a *term variable* of type τ in the body *prf*. Logically, this corresponds to $\bigwedge$ introduction. The name a is used only for parsing and printing.

`AbsP` `(`a`,` φ`,` *prf*`)` is the abstraction over a *proof variable* standing for a proof of proposition φ in the body *prf*. This corresponds to $\Longrightarrow$ introduction.

prf `%` t is the application of proof *prf* to term t which corresponds to $\bigwedge$ elimination.

prf_1 `%%` prf_2 is the application of proof prf_1 to proof prf_2 which corresponds to $\Longrightarrow$ elimination.

`PBound` i is a *proof variable* with de Bruijn [4] index i.

`Hyp` φ corresponds to the use of a meta level hypothesis φ.

`PThm` `((`*name*`,` *tags*`),` *prf*`,` φ`,` $\overline{\tau}$`)` stands for a pre-proved theorem, where *name* is the name of the theorem, *prf* is its actual proof, φ is the proven proposition, and $\overline{\tau}$ is a type assignment for the type variables occurring in the proposition.

`PAxm` `(`*name*`,` φ`,` $\overline{\tau}$`)` corresponds to the use of an axiom with name *name* and proposition φ, where $\overline{\tau}$ is a type assignment for the type variables occurring in the proposition.

`Oracle` `(`*name*`,` φ`,` $\overline{\tau}$`)` denotes the invocation of an oracle with name *name* which produced a proposition φ, where $\overline{\tau}$ is a type assignment for the type variables occurring in the proposition.

`MinProof` *prfs* represents a *minimal proof* where *prfs* is a list of theorems, axioms or oracles.

Note that there are no separate constructors for abstraction and application on the level of *types*, since instantiation of type variables is accomplished via the type assignments attached to `Thm`, `Axm` and `Oracle`.

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

```
#2 (#der (rep_thm conjI));
  PThm (("HOL.conjI", []),
    AbsP ("H", None, AbsP ("H", None, ...)), ..., None) %
      None % None : Proofterm.proof
```

This proof term identifies a labelled theorem, `conjI` of theory `HOL`, whose underlying proof is `AbsP ("H", None, AbsP ("H", None, ...))`. The theorem is applied to two (implicit) term arguments, which correspond to the two variables occurring in its proposition.

Isabelle's inference kernel can produce proof objects with different levels of detail. This is controlled via the global reference variable `proofs`:

`proofs := 0;` only record uses of oracles

`proofs := 1;` record uses of oracles as well as dependencies on other theorems and axioms

`proofs := 2;` record inferences in full detail

Reconstruction and checking of proofs as described in §3.4.1 will not work for proofs constructed with `proofs` set to `0` or `1`. Theorems involving oracles will be printed with a suffixed `[!]` to point out the different quality of confidence achieved.

The dependencies of theorems can be viewed using the function `thm_deps`:

`thm_deps [`thm_1`, ..., `thm_n`];`

generates the dependency graph of the theorems thm_1, ..., thm_n and displays it using Isabelle's graph browser. For this to work properly, the theorems in question have to be proved with `proofs` set to a value greater than `0`. You can use

```
ThmDeps.enable : unit -> unit
ThmDeps.disable : unit -> unit
```

to set `proofs` appropriately.

3.4.1 Reconstructing and checking proof terms

When looking at the above datatype of proofs more closely, one notices that some arguments of constructors are *optional*. The reason for this is that keeping a full proof term for each theorem would result in enormous memory requirements. Fortunately, typical proof terms usually contain quite a lot of redundant information that can be reconstructed from the context. Therefore, Isabelle's inference kernel creates only *partial* (or *implicit*) proof terms, in which all typing information in terms, all term and type labels of abstractions `AbsP` and `Abst`, and (if possible) some argument terms of `%` are omitted. The following functions are available for reconstructing and checking proof terms:

```
Reconstruct.reconstruct_proof :
  Sign.sg -> term -> Proofterm.proof -> Proofterm.proof
Reconstruct.expand_proof :
  Sign.sg -> string list -> Proofterm.proof -> Proofterm.proof
ProofChecker.thm_of_proof : theory -> Proofterm.proof -> thm
```

`Reconstruct.reconstruct_proof` *sg t prf* turns the partial proof *prf* into a full proof of the proposition denoted by t, with respect to signature *sg*. Reconstruction will fail with an error message if *prf* is not a proof of t, is ill-formed, or does not contain sufficient information for reconstruction by *higher order pattern unification* [7, 1]. The latter may only happen for proofs built up "by hand" but not for those produced automatically by Isabelle's inference kernel.

`Reconstruct.expand_proof` *sg* `[`$name_1$`, ..., `$name_n$`]` *prf* expands and reconstructs the proofs of all theorems with names $name_1$, ..., $name_n$ in the (full) proof *prf*.

`ProofChecker.thm_of_proof` *thy prf* turns the (full) proof *prf* into a theorem with respect to theory *thy* by replaying it using only primitive rules from Isabelle's inference kernel.

3.4.2 Parsing and printing proof terms

Isabelle offers several functions for parsing and printing proof terms. The concrete syntax for proof terms is described in Fig. 3.1. Implicit term arguments in partial proofs are indicated by "`_`". Type arguments for theorems and axioms may be specified using `%` or "·" with an argument of the form `TYPE(`*type*`)` (see §**??**). They must appear before any other term argument of a theorem or axiom. In contrast to term arguments, type arguments may be completely omitted.

$$
\begin{array}{rcl}
\mathit{proof} & = & \texttt{Lam}\ \mathit{params}\,.\ \mathit{proof} \quad | \quad \Lambda\mathit{params}\,.\ \mathit{proof} \\
 & | & \mathit{proof}\ \%\ \mathit{any} \quad | \quad \mathit{proof} \cdot \mathit{any} \\
 & | & \mathit{proof}\ \%\%\ \mathit{proof} \quad | \quad \mathit{proof} \bullet \mathit{proof} \\
 & | & \mathit{id} \quad | \quad \mathit{longid} \\
 & & \\
\mathit{param} & = & \mathit{idt} \quad | \quad \mathit{idt}\ \texttt{:}\ \mathit{prop} \quad | \quad \texttt{(}\ \mathit{param}\ \texttt{)} \\
 & & \\
\mathit{params} & = & \mathit{param} \quad | \quad \mathit{param}\ \mathit{params}
\end{array}
$$

Figure 3.1: Proof term syntax

```
ProofSyntax.read_proof : theory -> bool -> string -> Proofterm.proof
ProofSyntax.pretty_proof : Sign.sg -> Proofterm.proof -> Pretty.T
ProofSyntax.pretty_proof_of : bool -> thm -> Pretty.T
ProofSyntax.print_proof_of : bool -> thm -> unit
```

The function `read_proof` reads in a proof term with respect to a given theory. The boolean flag indicates whether the proof term to be parsed contains explicit typing information to be taken into account. Usually, typing information is left implicit and is inferred during proof reconstruction. The pretty printing functions operating on theorems take a boolean flag as an argument which indicates whether the proof term should be reconstructed before printing.

The following example (based on Isabelle/HOL) illustrates how to parse and check proof terms. We start by parsing a partial proof term

```
val prf = ProofSyntax.read_proof Main.thy false
  "impI % _ % _ %% (Lam H : _. conjE % _ % _ % _ %% H %%
     (Lam (H1 : _) H2 : _. conjI % _ % _ %% H2 %% H1))";
  val prf = PThm (("HOL.impI", []), ..., ..., None) % None % None %%
    AbsP ("H", None, PThm (("HOL.conjE", []), ..., ..., None) %
      None % None % None %% PBound 0 %%
      AbsP ("H1", None, AbsP ("H2", None, ...))) : Proofterm.proof
```

The statement to be established by this proof is

```
val t = term_of
  (read_cterm (sign_of Main.thy) ("A & B --> B & A", propT));
  val t = Const ("Trueprop", "bool => prop") $
    (Const ("op -->", "[bool, bool] => bool") $
      ... $ ... : Term.term
```

Using `t` we can reconstruct the full proof

```
val prf' = Reconstruct.reconstruct_proof (sign_of Main.thy) t prf;
  val prf' = PThm (("HOL.impI", []), ..., ..., Some []) %
    Some (Const ("op &", ...) $ Free ("A", ...) $ Free ("B", ...)) %
    Some (Const ("op &", ...) $ Free ("B", ...) $ Free ("A", ...)) %%
    AbsP ("H", Some (Const ("Trueprop", ...) $ ...), ...)
      : Proofterm.proof
```

This proof can finally be turned into a theorem

```
val thm = ProofChecker.thm_of_proof Main.thy prf';
  val thm = "A & B --> B & A" : Thm.thm
```

Chapter 4

Defining Logics

4.1 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 λ-calculus syntax with readable notation. They can express any context-free priority grammar. Isabelle syntax definitions are inspired by OBJ [5]; 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.

4.1.1 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:

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

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 $\texttt{("}c\texttt{"}\ a_1\ \ldots\ a_n\texttt{)}$, where a_i is the AST generated by parsing the i-th argument.

- The constant c, if non-empty, is declared to have type σ (`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 σ specifies the production's nonterminal symbols (or name tokens). If *template* is of the form above then σ 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 τ and τ_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 5).

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

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

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 `::` `"`σ`"`, is sensible only if c is an identifier. Otherwise you will be unable to write terms involving c.

4.1.2 Example: arithmetic expressions

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

```
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
```

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.

4.1.3 Infixes

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

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

and c `::` σ `(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 `::` σ `(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.

4.1.4 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 σ, 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 τ_1, the body P has type τ_2 and the whole term has type τ_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. **??**). 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.

4.2 *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: `""` (the empty string), `symbols`, `xsymbols`, and `latex`.

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.

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 λ-term printing mechanisms to generate completely different output, say for interfacing external tools like model checkers (see also `HOL/Modelcheck`).

4.3 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.

Chapter 5

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.

5.1 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 5.1 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 λ-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

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
string	

Figure 5.1: Parsing and printing

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; 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`).

5.2 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 §**??** for an example).

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 5.2: Parsing examples using the Pure syntax

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: $[\![t]\!] = \texttt{Variable}\; s$, where t is an `id`, `var`, `tid`, `tvar`, `num`, `xnum` or `xstr` token, and s its associated string. Note that for `xstr` this does not 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 5.2 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 λ-calculus. The Pure syntax provides predefined parse AST translations for ordinary applications, type applications, nested abstractions, meta implications and function types. Figure 5.3 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`.

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 5.3: Built-in parse AST translations

5.3 Transforming ASTs to terms

The AST, after application of macros (see §5.5), 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:

$$[\![\texttt{Appl}\ [f, x_1, \ldots, x_n]]\!] = [\![f]\!]\ \$\ [\![x_1]\!]\ \$\ \ldots\ \$\ [\![x_n]\!]$$

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.

5.4 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 (§5.6) 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:

- $[\![\texttt{Const}(x,\tau)]\!] = \texttt{Constant}\ x$.
- $[\![\texttt{Free}(x,\tau)]\!] = \mathit{constrain}(\texttt{Variable}\ x, \tau)$.
- $[\![\texttt{Var}((x,i),\tau)]\!] = \mathit{constrain}(\texttt{Variable}\ \texttt{"?}xi\texttt{"}, \tau)$, where $\texttt{?}xi$ 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 $\texttt{Free}(x', \texttt{dummyT})$:

  $$[\![\texttt{Abs}(x,\tau,t)]\!] = \texttt{Appl}\ [\texttt{Constant}\ \texttt{"\_abs"}, \mathit{constrain}(\texttt{Variable}\ x', \tau), [\![t']\!]]$$

- $[\![\texttt{Bound}\ i]\!] = \texttt{Variable}\ \texttt{"B.}i\texttt{"}$. 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,

  $$[\![f \ \texttt{\$}\ x_1 \ \texttt{\$}\ \ldots \ \texttt{\$}\ x_n]\!] = \texttt{Appl}\ [[\![f]\!], [\![x_1]\!], \ldots, [\![x_n]\!]]$$

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`.
- $\mathit{constrain}(x,\tau) = \texttt{Appl}\ [\texttt{Constant}\ \texttt{"\_constrain"}, x, [\![\tau]\!]]$ otherwise.

  Here, $[\![\tau]\!]$ is the AST encoding of τ: 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 §5.5), is transformed into the final output string. The built-in **print AST translations** reverse the parse AST translations of Fig. 5.3.

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.

5.5 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 §5.6) 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 5.4 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

[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 5.4: Macro example: set theory

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. 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.

5.5.1 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

$$(\mathit{root})\ \mathit{string} \quad \left\{ \begin{array}{c} \texttt{=>} \\ \texttt{<=} \\ \texttt{==} \end{array} \right\} \quad (\mathit{root})\ \mathit{string}$$

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.
- 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 §5.1), 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. 5.4 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 η-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.

5.5.2 Applying rules

As a term is being parsed or printed, an AST is generated as an intermediate form (recall Fig. 5.1). 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 §5.1 describes, former parse tree heads appear in ASTs as `Constant`s, while the name tokens `id`, `var`, `tid`, `tvar`, `num`, `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 §5.1. 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_ast` to `true`. 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`).

5.5.3 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}`.

5.5.4 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 the function `dependent_tr'` is exported by the syntax module for public use. See §5.6 below for more of the arcane lore of translation functions.

5.6 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.

5.6.1 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.

Function `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. 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
```

5.6.2 The translation strategy

The different kinds of translation functions are called during the transformations between parse trees, ASTs and terms (recall Fig. 5.1). 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, τ) or `Const`(c, τ) `$` x_1 `$` ... `$` 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 τ 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. There may never be more than one function associated with any syntactic name.

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. Multiple functions associated with some syntactic name are tried in an unspecified order.

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

§5.2. 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 §5.3.

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.

5.6.3 Example: a print translation for dependent types

Let us continue the dependent type example (page 47) 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.

Chapter 6

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 7 discusses Isabelle's rewriting tactics.

6.1 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

[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.

```
resolve_tac [eqth RS ssubst] i,
```

where `ssubst` is the 'swapped' substitution rule

$$[\![?a = ?b; ?P(?b)]\!] \Longrightarrow ?P(?a). \qquad (ssubst)$$

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.

6.2 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

$$[\![c = a; c = b]\!] \Longrightarrow a = b.$$

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.

6.3 Setting up the package

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 rev_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 7).

`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 §7.6.

`rev_eq_reflection` is the reverse of `eq_reflection`.

`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 7

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.

7.1 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.

7.1.1 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
debug_simp        : bool ref
```

(`trace_simp` and `debug_simp`: **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.

`set debug_simp;` makes the simplifier output some extra information about internal operations. This includes any attempted invocation of simplification procedures.

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_full_simp_tac` loop because the rewrite rule $f\ ?x = g\ (f\ (g\ ?x))$ extracted from the assumption does

[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`.

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. For example, invoking `Asm_full_simp_tac` on

```
1. [| P (f x); y = x; f x = f y |] ==> Q
```

gives rise to the infinite reduction sequence

$$P\,(f\,x) \stackrel{f\,x=f\,y}{\longmapsto} P\,(f\,y) \stackrel{y=x}{\longmapsto} P\,(f\,x) \stackrel{f\,x=f\,y}{\longmapsto} \cdots$$

whereas applying the same tactic to

```
1. [| y = x; f x = f y; P (f x) |] ==> Q
```

terminates.

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).

7.1.2 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
Addsplits   : thm list -> unit
Delsplits   : 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 §7.2.4.

`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 from the current simpset.

`Addsplits` *thms*`;` adds splitting rules to the current simpset.

`Delsplits` *thms*`;` deletes splitting rules from 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.

7.2 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.

7.2.1 Inspecting simpsets

```
print_ss : simpset -> unit
rep_ss   : simpset -> {mss        : meta_simpset,
                       subgoal_tac: simpset  -> int -> tactic,
                       loop_tacs  : (string * (int -> tactic))list,
                       finish_tac : solver list,
                unsafe_finish_tac : solver list}
```

`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.

7.2.2 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].

7.2.3 Rewrite rules

```
addsimps : simpset * thm list -> simpset                    infix 4
delsimps : simpset * thm list -> simpset                    infix 4
```

[2]Actually from ss_1, but it would unwise to count on that.

Rewrite rules are theorems expressing some form of equality, for example:

$$\begin{array}{rcl} Suc(?m) + ?n & = & ?m + Suc(?n) \\ ?P \wedge ?P & \leftrightarrow & ?P \\ ?A \cup ?B & \equiv & \{x \,.\, x \in ?A \vee x \in ?B\} \end{array}$$

Conditional rewrites such as $?m < ?n \Longrightarrow ?m/?n = 0$ are also permitted; the conditions can be arbitrary formulas.

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 §7.6.2. 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* [7]. These are terms in β-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 $(\forall x.?P(x) \wedge ?Q(x)) \leftrightarrow (\forall x.?P(x)) \wedge (\forall x.?Q(x))$ 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.

7.2.4 *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:

$$[\![?P_1 \leftrightarrow ?Q_1;\ ?Q_1 \Longrightarrow ?P_2 \leftrightarrow ?Q_2]\!] \Longrightarrow (?P_1 \to ?P_2) \equiv (?Q_1 \to ?Q_2)$$

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 $x = 0 \to y + x = y$. The local assumptions are also provided as theorems to the solver; see § 7.2.6 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:

$$\begin{array}{c}[\![?A = ?B;\ \bigwedge x\,.\,x \in ?B \Longrightarrow ?P(x) = ?Q(x)]\!] \Longrightarrow \\ (\forall x \in ?A\,.\,?P(x)) = (\forall x \in ?B\,.\,?Q(x))\end{array}$$

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.

7.2.5 *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.

7.2.6 *The solver

```
mk_solver  : string -> (thm list -> int -> tactic) -> solver
setSolver  : simpset * solver -> simpset                       infix 4
addSolver  : simpset * solver -> simpset                       infix 4
setSSolver : simpset * solver -> simpset                       infix 4
addSSolver : simpset * solver -> simpset                       infix 4
```

A solver is a tactic that attempts 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. To keep things more abstract, solvers are packaged up in type `solver`. The only way to create a solver is via `mk_solver`.

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 a simpset contains two solvers, a safe and an unsafe one.

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. Note that in this way the overall tactic is not totally safe: it may instantiate unknowns that appear also in other subgoals.

`mk_solver` *s tacf* converts *tacf* into a new solver; the string *s* is only attached as a comment and has no other significance.

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 §7.2.5, 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 a premise of a congruence rule cannot be proved, then the congruence is ignored. This should only happen if the rule is *conditional* — that is, contains premises not of the form $t = ?x$; otherwise it indicates that some congruence rule, or possibly the subgoaler or solver, is faulty.

7.2.7 *The looper

```
setloop   : simpset *           (int -> tactic)  -> simpset    infix 4
addloop   : simpset * (string * (int -> tactic)) -> simpset    infix 4
delloop   : simpset *  string                    -> simpset    infix 4
addsplits : simpset * thm list -> simpset                      infix 4
delsplits : simpset * thm list -> simpset                      infix 4
```

The looper is a list of tactics that are 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 : 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 only looper tactic of *ss*.

ss `addloop` (*name*, *tacf*) adds *tacf* as an additional looper tactic with name *name*; it will be tried after the looper tactics that had already been present in *ss*.

ss `delloop` *name* deletes the looper tactic *name* from *ss*.

ss `addsplits` *thms* adds split tactics for *thms* as additional looper tactics of *ss*.

ss `addsplits` *thms* deletes the split tactics for *thms* from the looper tactics of *ss*.

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 for Cartesian products (which happens to be called *split*):

$$?P(\mathit{split}(?f, ?p)) \leftrightarrow (\forall a\ b\ .\ ?p = \langle a, b\rangle \to ?P(?f(a, b)))$$

For technical reasons, there is a distinction between case splitting in the conclusion and in the premises of a subgoal. The former is done by `split_tac` with rules like `split_if` or `option.split`, which do not split the subgoal, while the latter is done by `split_asm_tac` with rules like `split_if_asm` or `option.split_asm`, which split the subgoal. The operator `addsplits` automatically takes care of which tactic to call, analyzing the form of the rules given as argument.

! Due to `split_asm_tac`, the simplifier may split subgoals!

Case splits should be allowed only when necessary; they are expensive and hard to control. Here is an example of use, where `split_if` is the first rule above:

```
by (simp_tac (simpset()
                 addloop ("split if", split_tac [split_if])) 1);
```

Users would usually prefer the following shortcut using `addsplits`:

```
by (simp_tac (simpset() addsplits [split_if]) 1);
```

Case-splitting on conditional expressions is usually beneficial, so it is enabled by default in the object-logics `HOL` and `FOL`.

7.3 The simplification tactics

```
generic_simp_tac       : bool -> bool * bool * bool ->
                         simpset -> int -> tactic
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
```

`generic_simp_tac` is the basic tactic that is underlying any actual simplification work. The others are just instantiations of it. 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.

`generic_simp_tac` *safe* `(`*simp_asm*`,` *use_asm*`,` *mutual*`)` gives direct access to the various simplification modes:

- if *safe* is `true`, the safe solver is used as explained in §7.2.6,
- *simp_asm* determines whether the local assumptions are simplified,
- *use_asm* determines whether the assumptions are used as local rewrite rules, and
- *mutual* determines whether assumptions can simplify each other rather than being processed from left to right.

This generic interface is intended for building special tools, e.g. for combining the simplifier with the classical reasoner. It is rarely used directly.

`simp_tac, asm_simp_tac, full_simp_tac, asm_full_simp_tac` are the basic simplification tactics that work exactly like their namesakes in §7.1, except that they are explicitly supplied with a simpset.

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 §7.1) 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.

7.4 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 §7.3, but affect the whole theorem instead of just a certain subgoal. Also note that the looper / solver process as described in §7.2.7 and §7.2.6 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.

7.5 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 [2] 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 [6] discuss the theory and give many examples; other algebraic structures are amenable to ordered rewriting, such as boolean rings.

7.5.1 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 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 $\sum_{i=1}^{n} i^3 = n^2 \times (n+1)^2/4$.

7.5.2 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.

7.6 *Setting up the Simplifier

Setting up the simplifier for new logics is complicated in the general case. 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 case splitting tactic, which resides on a separate files, is not part of Pure Isabelle. It needs to be loaded explicitly by the object-logic as follows (below `~~` refers to `$ISABELLE_HOME`):

```
use "~~/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].

7.6.1 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)"];
```

7.6.2 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.

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.

For standard situations like the above, there is a generic auxiliary function `mk_atomize` that takes a list of pairs (*name*, *thms*), where *name* is an operator name and *thms* is a list of theorems to resolve with in case the pattern matches, and returns a suitable `atomize` function.

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_eq` converts a theorem to a meta-equality using the case analysis described above.

```
fun mk_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_eq` will be composed together and supplied below to `setmksimps`.

7.6.3 Making the initial simpset

It is time to assemble these items. 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];
```

We also define the function `mk_meta_cong` to convert the conclusion of congruence rules into meta-equalities.

```
fun mk_meta_cong rl = standard (mk_meta_eq (mk_meta_prems rl));
```

The basic simpset for intuitionistic FOL is `FOL_basic_ss`. It preprocess rewrites using `gen_all`, `atomize` and `mk_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_eq o atomize o gen_all)
               setmkcong mk_meta_cong;

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.

Chapter 8

The Classical Reasoner

8.1 Classical rule sets

For elimination and destruction rules there are variants of the add operations adding a rule in a way such that it is applied only if also its second premise can be unified with an assumption of the current proof state:

```
addSE2      : claset * (string * thm) -> claset                infix 4
addSD2      : claset * (string * thm) -> claset                infix 4
addE2       : claset * (string * thm) -> claset                infix 4
addD2       : claset * (string * thm) -> claset                infix 4
```

! A rule to be added in this special way must be given a name, which is used to delete it again – when desired – using `delSWrappers` or `delWrappers`, respectively. This is because these add operations are implemented as wrappers (see 8.1.1 below).

8.1.1 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 §8.3 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 the unsafe parts of `step_tac`, `slow_step_tac`, and the tactics that call them. A wrapper transforms each step of the search, for example by attempting other tactics before or after 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
addSafter    : 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
addafter     : 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 a safe wrapper, such that it is tried *before* each safe step of the search.

cs `addSafter` (*name*, *tac*) adds the given tactic as a safe wrapper, such that it is tried 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 an unsafe wrapper, such that it its result is concatenated *before* the result of each unsafe step.

cs `addafter` (*name*, *tac*) adds the given tactic as an unsafe wrapper, such that it its result is concatenated *after* the result of each unsafe step.

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.

Strictly speaking, the operators `addss` and `addSss` are not part of the classical reasoner. , which are used as primitives for the automatic tactics described in §??, are implemented as wrapper tacticals. they

! 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.

8.2 The classical tactics

8.2.1 Other classical tactics

```
slow_best_tac : claset -> int -> tactic
```

`slow_best_tac` *cs* *i* applies `slow_step_tac` with best-first search to prove subgoal i.

8.2.2 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.

8.3 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 6). 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] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52. Springer-Verlag, 2000.

[2] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[3] E. Charniak, C. K. Riesbeck, and D. V. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, 1980.

[4] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34:381–392, 1972.

[5] 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.

[6] 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.

[7] 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.

[8] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

Index