A Guide to UNICOM, an Inductive Theorem Prover Based on Rewriting and Completion Techniques

Bernhard Gramlich, Wolfgang Lindner*
Fachbereich Informatik, Universität Kaiserslautern
Erwin-Schrödinger-Straße, Postfach 3049
D-6750 Kaiserslautern
Germany
gramlich@informatik.uni-kl.de

December 1991

Abstract

We provide an overview of UNICOM, an inductive theorem prover for equational logic which is based on refined rewriting and completion techniques. The architecture of the system as well as its functionality are described. Moreover, an insight into the most important aspects of the internal proof process is provided. This knowledge about how the central inductive proof component of the system essentially works is crucial for human users who want to solve non-trivial proof tasks with UNICOM and thoroughly analyse potential failures. The presentation is focussed on practical aspects of understanding and using UNICOM. A brief but complete description of the command interface, an installation guide, an example session, a detailed extended example illustrating various special features and a collection of successfully handled examples are also included.

*This work was supported by the 'Deutsche Forschungsgemeinschaft, SFB 314 (D4-Projekt)'.

Contents

Appendix

1 Introduction and Overview

UNICOM is an experimental term rewriting based system for investigating hierarchically structured many-sorted algebraic specifications. Input specifications consist of definitions of total functions and of inductive conjectures (equations to be proved or disproved). The functions are specified equationally by rewrite programs and operate on domains which are generated by a set of free constructors.

The central tool of UNICOM (UNfailing Inductive COMpletion) is an inductive theorem prover based on refined unfailing completion techniques (cf. [Gra89], [Gra90b], [Gra90c]).

For the theoretical foundations of the underlying equational logic and *implicit* inductive theorem proving approach see e.g. [HO80], [HH80], [JK86], [Pla85], [Fri86], [Göb87], [Küc87], [Bac88], [Gra90a] and [Red90]. Concerning UNICOM the interested reader is referred to [AGG+87], [Sch88], [Gra90b], [Gra90c]. For other systems based on rewriting and completion techniques and incorporating inductive theorem proving components see e.g. [KZ89]. In classical *explicit* inductive theorem proving using schemata the best known and rather successful system is the Boyer-Moore (inductive) theorem prover *NQTHM* (see [BM79], [BM88]).

Before calling the *prover*, a specification has to be processed by two other of the tools of UNICOM: the *parser* and the *checker*. The parser checks the syntax of an input specification and generates an internal representation (intern-file). Then the checker gets this internal representation and examines function definitions trying to establish required properties like convergence and totality. The prover finally tries to prove or to disprove the equations (conjectures) of a specification and produces two output files. The *prove-file* primarily contains the proved conjectures and rules inferred during the proving process, together with refinements of the generated term reduction ordering. When 'proving' a specification, the prove-files of all subspecifications imported by the specification under consideration

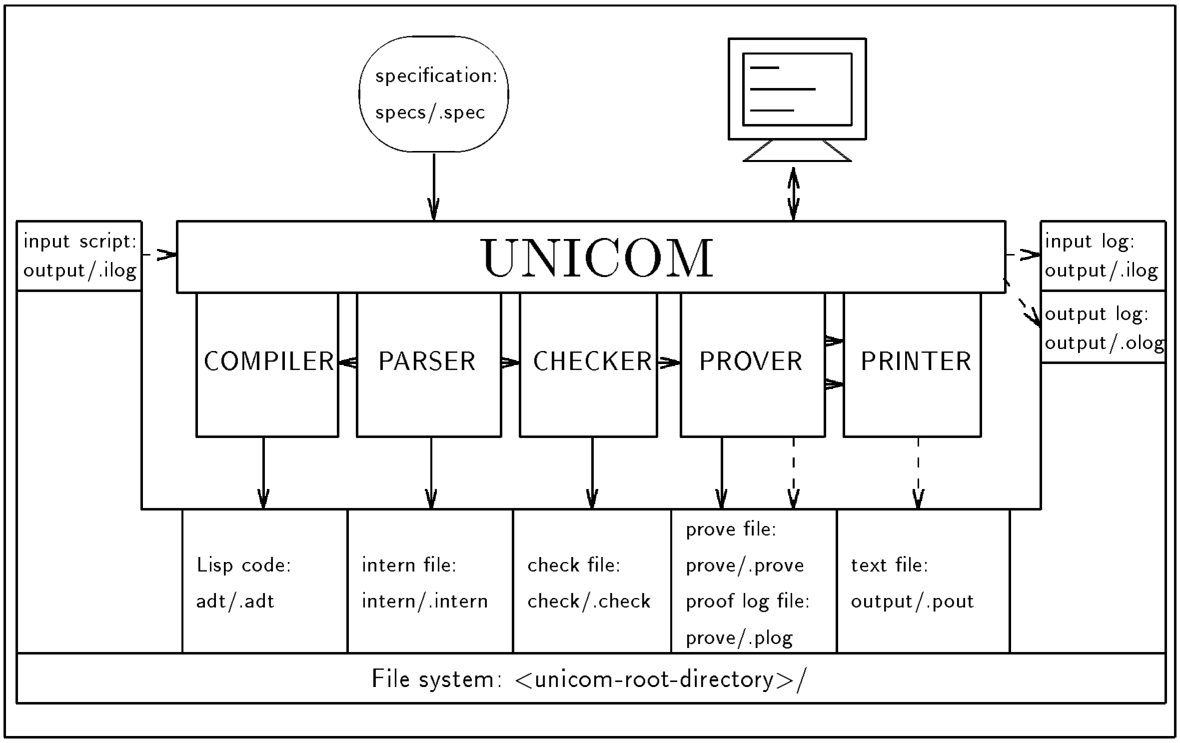

Figure 1: The structure of UNICOM.

are read by the prover in order to make available the already proved inductive knowledge about subspecifications. Secondly, the prover optionally produces *proof-log-files* in which the proof steps carried out are recorded. The *printer* can be used to display logged proofs in some desired readable form. Moreover, UNICOM includes a compiler (not described here) which is able to translate function definitions into Lisp code, so that symbolic computations are possible, i.e. normal forms of ground terms can be computed by a Lisp implementation.

Figure 1 illustrates how UNICOM manipulates the data base of hierarchically structured specifications.

Before treating the tools individually, we will describe the user interface of the system now.

2 The User Interface

Commands and their Syntax

The user interface of UNICOM mainly consists of a command and control language with some abbreviation facilities. Interaction between the system and the user is organized as follows. The input stream is cut into words consisting of non-blank characters. When a point is reached where UNICOM accepts input, the next word of the input stream is used to identify one of the commands executable at this point by matching the word with the syntax descriptions of the commands. If this correspondence is unambiguous, the piece of program code of the selected command is carried out, possibly using the entered word as parameter.

Besides words the following tokens are extracted from the stream:

’;’: The (optional) command delimiter (the character is separating).

<newline>: The newline character delimits the type-ahead area (while reading commands out of this area no prompt or menu is displayed). Secondly, it marks the end of a list of words. If ’newline’ is preceded by ’;’, the former is ignored.

’\’<rest-of-line>: The backslash and the rest of the line (esp. the closing ’newline’) are ignored (this is useful e.g. for comments and continuation lines).

<eof> or ’\\’: The end-of-file character or equivalently the double-backslash serves as abort command.

The syntax of commands is described by clauses, some of which may match with a range of words (or tokens). The syntax of a command may consist of several clauses in order to admit alternative ways to refer to the command. There are six types of clauses:

keywords: The clause ’keyword k’ (k a word) matches the word w, if the first character of k and of w coincide and the remaining characters of k occur in the rest of w in the same order.

codes: The clause ’code c’ matches the word c.

switches: There are two variants of switch clauses: ’kswitch k’ and ’cswitch c’. A word of the form ’+’|’-’w matches ’kswitch k’ (resp. ’cswitch c’), if w matches ’keyword k’ (resp. ’code c’).

numbers: All words consisting of digits.

names: All words which do not start with one of the characters ?,+,-,0,..,9.

defaults: The clause ’default’ matches with the command delimiter and ’newline’.

Help Menus

The table of commands that are locally executable can be viewed together with their syntax by entering '?'. The table entries are structured as follows:

$$< name\ of\ the\ command > : \qquad \begin{array}{c} < syntax - clause_1 > \\ \vdots \\ < syntax - clause_n > \\ [< description\ of\ the\ command >] \end{array}$$

If the parameter of a keyword-, code- or switch-clause coincides with the command name, this is indicated by '-*-'. In addition, more general information on the current halting point can be viewed by entering '??'.

The Prompt and Abortions

In order to point out that input is expected UNICOM displays a prompt which contains the following information: The first section of the prompt is the currently active command, i.e. the command by means of which UNICOM has reached the current halting point and which was entered at the next higher point. This is not always the command entered last, and the next higher halting position is

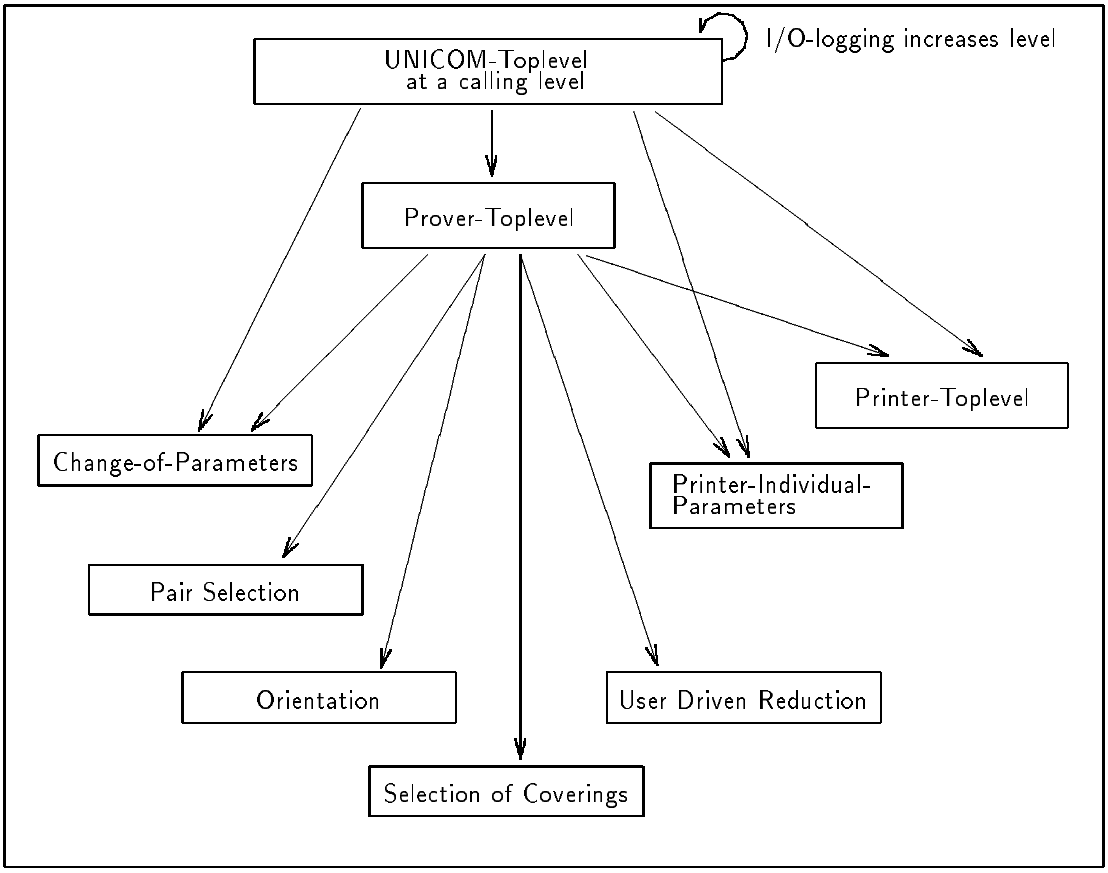

Figure 2: The hierarchy of the main interaction points.

not always the point at which the last command has been entered (see Figure 2). The second part of the prompt indicates the current point or the expected input.

By entering '\\' (resp. an eof-character) one can abort the currently active command. Then the next higher halting point is reentered. In general, all side effects caused by the aborted command are undone and the input logged during the execution of the command is discarded.

I/O-Logging

The basic idea of the I/O-logging mechanism is to record all input (or program output) produced during a session with UNICOM, i.e. from the first command entered at Unicom-Toplevel to the leaving 'quit' (see commands 'log-input', 'log-output' and 'log-proof' at Unicom-Toplevel). As the interpretation of input commands (esp. during a call of the prover) depends on the parameter settings, each input log additionally contains the parameter settings valid at the beginning of the logged command sequence. When replaying a logged sequence of commands (see command 'execute' at Unicom-Toplevel) these parameter settings are first restored. This does not include, however, parameters that control the output produced by UNICOM (namely: the prover parameters: generation-of-prover-file, generation-of-proof-log and trace-depth). These may be set as desired before executing a command script.

System Crashes, Lisp-level Commands

The only cases where UNICOM should enter the (Lisp-) debugger are: a failure in writing a file (e.g. due to non-existent directories or missing write access rights) and a user's keyboard interrupt. Besides the usual ways to continue (in Lisp), there are two Lisp functions to reenter UNICOM:

'(abort-com)': Aborts the currently active command analogous to aborting by means of '\\'.

'(leave-unicom)': The current Unicom-level is left directly without undoing side effects (this affects parameter settings). I/O-logs are closed in the current state.

'(im)': Enters a menu to call the above functions.

Note: Whether these functions work correctly depends on the used Lisp implementation which is required to allow that the debugger can be left with the (Lisp) function 'throw'.

At Lisp-level UNICOM is called with the function '(unicom)' or '(u)'.

When using an Apollo/Sun Lucid Common Lisp environment (Version 3 or 4), a keyboard interrupt causes an Unicom-specific interrupt handler to be entered. It allows: to continue or to abort the active command, to leave the current Unicom-level, to enter the debugger and to change the value of the prover parameter 'trace-depth'.

Starting UNICOM

When starting UNICOM, the program enters Unicom-Toplevel and displays the prompt 'unicom(1)>' (here '1' is the current level at which UNICOM is working; thus, '1' indicates that no i/o-logging is done). The default root directory of the specification data base is '<working-directory>/system/inout/'. It can be changed or adapted to some user's own specification data base by means of the command 'root-directory'.

3 The Parser

The parser reads in a specification (from a file in directory <root>/specs/ , with suffix '.spec') and if it is syntactically correct, produces an internal representation of its contents (saved in a file in directory <root>/intern/ , with suffix '.intern'). If some syntax error is detected, the parser stops immediately and displays the error message. The syntax of specifications is defined by the following grammar rules:[1]

<specification> =	**spec** <specification-name>
	[**use** <specification-name> { , <specification-name> }]
	[**sorts** <sort-name> { , <sort-name> }]
	[**ops** { <ops-with-arity> }]
	[**constructors** <operator-name> { , <operator-name> }]
	[**equations** { <term> = <term> }]
	[**rules** { <term> --> <term> }]
	endspec
<ops-with-arity> =	<operator-name> { , <operator-name> } **:** { <sort-name> } --> <sort-name>
<term> =	<variable-name> | <operator-name> [**(** <term> { , <term> } **)**]

<specification-name>, <sort-name>, <operator-name>, <variable-name> = <name>

Restrictions and notes:

- There is no distinction between upper-case and lower-case characters.
- A <name> is a sequence of (non-blank) characters not starting with a digit and not containing characters having a special Lisp interpretation (brackets, quotes, colons, etc.)
- Every <specification-name>, <sort-name> and <operator-name> has to be unique in the hierarchy.
- Every <sort-name> in the ops-clause has to be declared in a sorts-clause of the current or of a used specification.
- Every <operator-name> in a constructors-clause has to be declared in the ops-clause of the current specification.
- Every <operator-name> occurring in a term has to be declared in an ops-clause of the current or of a used specification. When parsing terms, every name which is not a declared operator name is interpreted as a variable (this is also the case, if the name is followed by an opening parenthesis; therefore such typing errors often result in unclear error messages).
- All terms, equations and rules have to be well-sorted.
- Any text after the keyword 'endspec' is ignored.

The equations-clause specifies the inductive conjectures, i.e. the equations to be verified by the prover and the rules-clause specifies the set of defining rules.

[1] Here, [...] denotes optional arguments, {...} denotes zero or more repetitions, $< \ldots >$ denotes non-terminals and a bold font indicates grammar terminals.

4 The Checker

The checker tests whether the definition part of a specification conforms to some usual kind of constructor discipline. In particular, the following checks are performed:

Correctness of left-hand sides

Every left-hand side of a defining rule has to be a linear[2] term of the form $f(t_1, \ldots, t_n)$, where f is a defined function symbol, i.e. not a constructor symbol, and $t_1, \ldots, t_n$ are constructor terms.

Termination

Termination of the system of defining rules is checked by the Lexicographic Recursive Path Ordering with Status (RPOS). Hence, an appropriate precedence on the function symbols and a status for every function symbol has to be found. The precedence is generated roughly by the following heuristic method:

- A function symbol f is greater than a function symbol g, if there is a rule $f(t_1, \ldots, t_n) \rightarrow t$, where g occurs in t and there is no rule $g(s_1, \ldots, s_n) \rightarrow s$, where f occurs in s.
- Two function symbols f, g are equivalent, if there are rules $f(t_1, \ldots, t_n) \rightarrow t$ and $g(s_1, \ldots, s_n) \rightarrow s$, where g occurs in t and f occurs in s.

The status information for every function symbol is generated by comparing the sides of the rules using each status and choosing a status for which the system is terminating.

Ground Convergence

Local *(ground) confluence* of the system of defining rules is tried to be established by generating critical pairs and reducing them to identical normal forms. This *(ground) joinability* of all critical pairs together with the termination property implies *(ground) convergence* which is crucial for the correctness of the inductive proof procedure described in the next section.

Totality

For every non-constructor symbol f it is checked whether f is totally defined. This is done by verifying that the tuples $(t_1, \ldots, t_n)$ of left-hand side arguments of definition rules $f(t_1, \ldots, t_n) \rightarrow t$ for f 'cover' all argument tuples $(s_1, \ldots, s_n)$ of constructor ground terms $s_1, \ldots, s_n$. In the case of incompleteness tuples of argument patterns which are not 'covered' by some left-hand side of f-rules are computed and displayed. Moreover, for the case of redundant definition rules for f, the checker is also able to compute different minimal but complete sets of defining rules for f. Hence, one may provide the system with alternative function definitions based on different recursion schemes. This information is exploited and may be crucial when proving inductive conjectures.

[2] The linearity requirement is not very restrictive in practice for this kind of specification discipline. In fact, it could be completely dropped. But this would entail a substantially more complicated totality check for the defined function symbols.

5 The Prover

The core of the prover is a specialized inductive completion algorithm (cf. [Gra89]). We shall describe the prover by specifying the components of a completion (or prover) state, the inference rules which are applied by the prover and the strategy the prover follows in doing that.

The Prover State

The prover operates on a number of independent completion environments and a global data base. Each completion environment (called *hypothesis*) reflects a stage in the process of proving an inductive proposition (called *conjecture*), i.e. an equation or the conjunction of several equations. There are four sources from which hypotheses are produced:

1. The input specification (initial hypotheses): The equations of the specification may be processed independently – then the prover forms a completion environment for each equation – or as the conjunction of equations in one environment.
2. The user creates hypotheses interactively during a call of the prover (command: 'new-hypothesis' at Prover-Toplevel).
3. The prover generates new conjectures by generalizing given ones.
4. The prover tries to prove a conjecture in several different ways so that there are various hypotheses belonging to the same conjecture (variants).

Global w.r.t. the hypotheses, i.e. relevant to all hypotheses, is the base of defining rules and lemmas (proved equations and assumptions), the ordering underlying the completion process (the precedence of the RPOS), the interpretation of the operators as AC- or C-Symbols and other parameters controlling the prover. More precisely, a prover state consists of the following components:

- A set (list) of (open) hypotheses, the proofs of which have not been finished yet.
- A set of hypotheses which have been accepted, i.e. proved.
- A set of hypotheses which have been rejected, i.e. refuted.
- The precedence of the generated RPOS.
- A system of defining rules and reduction lemmas.
- A system of cyclic rules, i.e. equations which are not directable or have not been directed.
- The parameter settings.

The current state can be viewed with the command 'show' at the interaction point Prover-Toplevel and the show-commands in the change-parameter-menu.

Each hypothesis consists of the following components:

- The corresponding conjecture.
- An identification to indicate the proof variant (a number).
- A set of essential pairs. Essential pairs are: the equations of the conjecture, critical pairs resulting from the 'covering' of essential pairs or 'essentialized' inessential pairs (see below). They are, as it were, the equations that remain to be proved.

- A set of inductive rules (called 'inductive rules' due to their correspondence to inductive hypotheses in usual inductive proofs) and a set of cyclic inductive rules, i.e. oriented (essential) pairs which have been covered by critical pairs.
- A set of inessential pairs. Inessential pairs are critical pairs (derived from the conjecture) whose verification is not necessary for the soundness of the proof procedure. In particular, critical pairs between inductive rules are inessential. In some cases, computing inessential pairs leads to interesting auxiliary equations which contribute to the termination of the proof attempt if they are essentialized, i.e. inserted into the set of essential pairs.
- A log component in which the actions of the prover concerning the hypothesis are recorded. The logging mechanism may be switched off (parameter: generate-proof-log(yes | no)).

Furthermore, certain dependencies among the hypotheses are managed using a marking scheme. A conjecture *depends on* the conjecture of a hypothesis, if the former is the result of generalizing an equation belonging to this hypothesis. If the proof attempt for the old conjecture ends or is aborted, this marking scheme allows the removal of all dependent hypotheses.

The Initialization of the Prover State

When applying the prover to a specification, first the check-file of the specification and all prove-files of used subspecifications are read providing the appropriate data base of defining rules and lemmas for the proof attempt.

The initialization of the set of hypotheses is influenced by two parameters: incremental-mode (yes | no) and independent-proving-of-equations(yes | no). When working in incremental mode, the prover determines by inspecting the prove-file of the specification which equations have already been accepted or rejected and removes them from the list of equations to be proved. The parameter independent-proving-of-equations controls whether the (remaining) equations are treated together in one hypothesis or processed independently in distinct hypotheses. In the latter case, the list of hypotheses is arranged in the same order as in the specification (see 'selection of pairs' in the next section).

Additionally, when working in incremental mode, the declarations of operators as AC/C- Symbols are restored from the prove-file. Thus, these parameters cannot be changed in incremental mode.

Having simplified the hypotheses (v. point 4 below) Prover-Toplevel is reached.

The Prover Cycle

When executing the 'body' of the provers main loop (i.e. one *step*), the following operations are carried out (in this order):

1. Selection of the hypothesis and the essential pair to be processed next:

   Relevant parameters: pair-selection (fair | by-user),
   processing-of-hypotheses-in-fixed-succession (yes | no).

   These parameters define three ways of proceeding:

   a) pair-selection = 'by-user': A browser is entered to view the current list of hypotheses and to select the next pair.
   b) pair-selection = 'fair' and p-f-s = 'no': The pair that is composed of the least number of symbols is selected from all (open) hypotheses. This determines the selected hypothesis implicitly.
   c) pair-selection = 'fair' and p-f-s = 'yes': Like (b), but the pair is selected among the hypotheses which belong to the same conjecture as the first hypothesis of the list. In this way one can focus on proving one specific conjecture.

2. Orientation of the selected pair:

   If the two sides of the pair are comparable w.r.t. the current RPOS or it is recognized that the pair cannot be directed in a terminating manner (e.g. for permutative equations), then the corresponding orientation is chosen automatically. Otherwise, the user may extend the precedence or choose an orientation manually. According to its orientation the pair is added to one of the sets of inductive rules.

3. Covering of the selected pair:

   Relevant parameter: choice-of-coverings (one | all).

   For the oriented pair, minimal coverings (sets of critical pairs) are computed by superposing defining rules and lemmas which have the form of defining rules (left-linear, form $f(t_1, \ldots, t_n) \to r$, t_i constructor terms) into the selected pair at an inductively complete position. The critical pairs of a covering are inserted into the set of essential pairs. If choice-of-coverings = 'all', then for each covering a variant of the hypothesis is generated. Otherwise, the user may select one of them manually.

4. Simplification of the essential pairs of the new hypotheses:

   a) Normalization:
      Relevant parameter: normalization-mode (automatic | user-driven).
      The pairs are normalized using the global base of rules and the local set of inductive rules.
   b) Removal of top-level constructors:
      Pairs of the form $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$ where c is a constructor are simplified to the conjunction of the pairs $t_i = s_i$.
   c) Deletion of trivial pairs.
   d) Deletion of subsumed pairs:
      Pairs which are subsumed by a global cyclic lemma or a cyclic inductive rule are deleted.

5. Check for acceptable hypotheses:

   If the set of essential pairs of a hypothesis is empty, the corresponding conjecture is accepted. Accepted conjectures and those conjectures depending on them are removed from the set of open hypotheses. The global base of rules and cyclic rules is enriched by the inductive rules of accepted hypotheses.

6. Check for (obviously) inconsistent hypotheses:

   Hypotheses which contain inconsistent pairs, i.e. pairs of the form $c(t_1, \ldots, t_n) = d(s_1, \ldots, s_m)$, $x = c(t_1, \ldots, t_n)$ or $x = y$ with c, d different constructors and x, y different variables are rejected and the conjecture is removed. (Every sort is assumed to have at least two distinct constructor ground terms.)

7. Computation of inessential pairs for new hypotheses:

   Relevant parameter: generation-of-inessential-pairs(yes | no).

   a) If generation-of-inessential-pairs = 'yes', all critical pairs of the oriented pair with defining rules, reduction lemmas and inductive rules are computed. They are simplified as described in (4 a-d) and additionally, pairs which are subsumed by essential pairs are removed. The resulting pairs are added to the set of inessential pairs.
   b) Check for (obviously) inconsistent hypotheses.

8. Essentialization of inessential pairs:

   Relevant parameter: use-of-inessential-pairs(no | if-reductive).

If use-of-inessential-pairs = 'if-reductive', those inessential pairs which are directable w.r.t. the current RPOS and may be used to reduce an essential pair[3] are inserted into the set of essential pairs (heuristic decision).

9. Generalization:

   Relevant parameter: generation-of-generalizations(yes | no).

   If generation-of-generalizations = 'yes', the prover tries to generalize the oriented pair and the result is used to form new conjectures. Currently some simple generalization techniques like removal of equal topsymbols on both sides of an equation and abstraction of several occurrences of the same subterm (modulo permutative cyclic rules) to a variable are implemented.

Further Remarks concerning Parameters

incremental mode: This option has been implemented to enable the user to extend specifications with additionally needed lemmas without having to treat already proved equations all over again. The option should only be applied carefully (see section 'modifying specifications').

declare-ac/c-symbols: The AC/C assertions for symbols have an effect on nearly all operations of the prover (normalization, subsumption, trivial pairs, critical pairs). The implemented RPOS, however, is *not* an AC-compatible ordering. Thus, when some operators are declared as AC/C-symbols (or pairs are directed to reduction rules manually or the termination of the defining rules could not be established by the RPOS), there is in the end no guarantee that accepted conjectures are correct.

generation-of-prover-file(yes | no): Writing the prove-file at the end of a call of the prover may be switched off (irrelevant for most applications).

trace-depth(no-trace | trace-pairs | trace-steps | trace-all | trace-all-full): This parameter determines the quantity of information displayed by the prover to document actions.

trace-pairs: all processed (essential) pairs are displayed.

trace-steps: all actions of the prover are displayed.

trace-all: additionally, for every reduction step the used rule is displayed together with the depth of the application position.

trace-all-full: additionally, the terms resulting from reduction steps are shown.

The Undo Operation

At the end of each proof step (i.e. processing one essential pair) the current prover state (set of hypotheses, system of rules, precedence) may be saved (cf. the variants of the commands 'next' and 'continue' at Prover-Toplevel). The command 'step-back' (at Prover-Toplevel) can be used to restore one of the former states.

Insertion of Assumptions

At Prover-Toplevel the user may add equations not (yet) proved into the global base of rules. These assumptions may only be removed by using the undo operation. If the global base contains assumptions at the end of a call of the prover, no prove-file is written.[4] A proof log, however, may be produced.

[3] This property of inessential pairs namely to be useful for modifying (i.e. reducing) essential pairs constitutes the heuristic criterion for essentialization.

[4] The main reason for not performing a global update is to maintain consistency of the data base. Whenever there are assumptions left the result of a proof attempt may still depend on the validity of these assumptions.

Modifying Specifications

When the user modifies specifications (esp. by making changes in used subspecifications), corresponding prove-files or input-log-files may become obsolete, i.e. they contain wrong or useless data, e.g. because:

- accepted equations may no longer be valid due to modified definitions,
- adding (or removing) conjectures may influence the proof process so that execution of input-log-files may no longer be possible.

These files, however, will not be deleted and may still be used. Hence, the user is responsible for maintaining consistency of the specification data base.

6 The Printer

Every operation of the prover, i.e. the transformation of a hypothesis, has the same abstract form: a pair is transformed into a number of successor pairs. As pairs may be produced in more than one way, the operations define an acyclic, directed graph, whose nodes are pairs and whose edges are the transformation steps.

When printing a proof, the corresponding proof graph is constructed and optionally reduced (minimized), i.e. in accepting (or stopped) proofs, all actions concerning inessential pairs that are not essentialized are removed and in rejecting proofs, all edges not leading to the inconsistent pair are removed. The resulting graph is cut into segments, i.e. sequences of actions which produce only one successor pair (e.g. reduction, orientation). These segments are sorted, indexed and aligned sequentially. There are three orderings which can be used to sort the segments: depthfirst, breadthfirst and chronological. The latter tries to reproduce the order in which the operations were carried out by the prover.

In addition, when treating disproofs (i.e. proof attempts leading to an inconsistency) the instances of the original conjecture are computed which were needed to infer the inconsistent pair.

A Some More Prover Facilities

Non-Inductive Proofs

One of the secondary insufficiencies of the prover described in section 5 is the way how cyclic (non-directed or non-directable) equations are utilized: Only one equation may be applied once to transform one term of a *normalized* critical pair into the other (subsumption test). Thus, in order to experiment with other patterns of subconnectedness proofs (e.g. $s \stackrel{*}{\Rightarrow} u \stackrel{\leq n}{\Leftrightarrow} v \stackrel{*}{\Leftarrow} t$ or $s \stackrel{*}{\Rightarrow} u \Leftrightarrow v \stackrel{+}{\Rightarrow} w \stackrel{*}{\Leftarrow} t$), we extended the prover with the following two options. In both cases, however, the ordering condition for subconnectedness is not tested, and esp. the second option allows the user to produce nonsensical "proofs".

1. Multiple Subsumption

   Parameter: limit-of-steps-at-subsumption-detection(<number>)

   It may happen that a normalized essential pair can only be joined by more than one rewrite step with cyclic rules. For instance, when proving that the identity of binary trees is commutative, the induction hypothesis (a necessarily non-directed inductive rule) has to be applied twice in parallel or, when proving that the identity of lists is commutative, the induction hypothesis and a lemma stating the commutativity of the identity predicate for the list elements have to be applied in parallel.

   The implemented test for multiple subsumption replaces the test for ordinary subsumption if the value of the parameter above is greater than one. It checks whether one side of a pair can be produced by rewriting the other side using at most <limit.> rewrite steps with cyclic (global) lemmas or cyclic inductive rules in both directions. Cyclic rules with a variable occuring only one side (e.g. $0 = \mathsf{times}(0, x)$ when non-directed) are treated in a special manner: The variables of the 'right-hand' side of the applied equation which are not bound by the 'matching' of the left-hand side become term scheme variables in the result of rewriting. Thus, this process actually rewrites term schemes containing some free variables. The schemes describe sets of terms which can be produced by rewriting the term started with, namely the set of terms which are instances of the scheme. The free variables might be instantiated arbitrarily before applying the next equation. The prover, however, uses a restricted form of instantiation by unifying the left-hand sides of the rules with (subschemes of) a term scheme.

2. User Driven Rewriting

   The command 'equationally-next' at Prover-Toplevel enables the user to rewrite essential pairs manually using the rewrite relation on term schemes described above with the whole rule base (global reduction and cyclic lemmas and inductive rules of the hypothesis, all in both directions).

Reduction Strategies

The standard reduction strategy used for simplifying pairs is a rule priority strategy: Inductive rules are applied before any lemma and lemmas are applied before definition rules. Additionally, lemmas of higher specifications are applied before lemmas of imported specifications. The order in which the rules are actually applied can be viewed with the command 'show/rules' at Prover-Toplevel.

Although this heuristic is successful in most cases, severe efficiency problems may arise, esp. when generating the disjunctive or conjunctive normal form of boolean and/or-expressions because definition rules as $\mathsf{and}(\mathsf{false}, x) \rightarrow \mathsf{false}$ are applied at last. This inefficiency affects space - terms grow very large - and time - many avoidable reduction steps.

To handle this problem a second reduction strategy was implemented which gives higher priority to rules that "delete something" and postpones applications of rules that "duplicate reduction sequences". For this purpose, the rules are divided into three categories: (Let $l \rightarrow r$ be a rule and $V(t)$ (resp. $F(t)$) be the multiset of variables (resp. function symbols) of a term t.)

deletion rules: $V(l) \sqsupset V(r) \vee (V(l) = V(r) \wedge F(l) \sqsupset F(r))$

duplication rules:: $V(l) \not\sqsupseteq V(r) \wedge V(l) \neq V(r)$

transformation rules:: otherwise

($\sqsubset$ denotes strict inclusion of multisets.)

According to the second strategy, deletion rules are applied before transformation and duplication rules, and no duplication rule is applied to a subterm which is reducible below toplevel. The remaining ties are decided with the rule priority described above.

Besides the simple standard *method* of reduction, two other reduction algorithms have been implemented which are based on marking terms and improved structure sharing. The parameter 'normalization-mode' is used to select one of the strategies and methods:

automatic(-unconditional), automatic2a, automatic3a:
standard strategy with standard method, algorithm 2 or 3, resp.
automatic2b, automatic3b:
second strategy with algorithm 2 or 3, resp.

The performance of the methods depend on the considered examples. However, if long sequences of reductions occur, method 3 or 2 should be used . g

B Table of Commands

For the main points of interaction in UNICOM the following tables give an overview of the available commands. The commands (left column) are classified and roughly described (right column).

Unicom Toplevel	
parse, silent-parse check prove parse-check-prove print-proof-logs (compile, run)	commands to call the tools
log-input log-prove execute log-output	I/O logging
change-parameter	menu to preset prover parameters
edit	edit a specification
root-directory quit delete-buffered-prove-files	technical commands

Prover Toplevel	
next, Next continue, Continue stop	continue or terminate the prover cycle
equational-next	user driven rewriting
show change-parameter add-assumption new-hypothesis step-back	information, trial and error
print-proof-logs	command to call the printer (only available if <generation-of-proof-log> = 'yes')

Printer (toplevel and individual parameter)	
selection-of-available-proofs(only-accepted | rejected+open | all) output-to-file(yes | no) choice-of-printing-options(individual | uniform)	printer mode (on toplevel)
reduction-steps(print-sequence | print-rules) equational-steps(print-sequence | print-rules) proof-graph-sequentialization(chronological | breadthfirst | depthfirst) proof-graph-minimization(yes | no) ac-terms-in-flat-form(yes | no) line-length(<number>)	printing options
show-options	information
continue	(on toplevel)
again next	(at individual parameter)

Pair Selection	
up, down next, previous forward, back goto mark, return-to-mark	traversal of the list of hypotheses
select	selection of a pair
print details show-details	information
delete delete-all	deletion of hypotheses

Orientation	
straight reverse cyclic	manual orientation
print-precedence generate-precedence-extensions extend-prededence	precedence extension

Choice of Coverings	
select	selection
details	information

User Driven Rewriting	
show apply	display and execution of equations applications
undo restart	undo operation
insert quit give-up	leave

User Driven Reduction	
reduce continue	application of rules
undo restart	undo operation
stop	leave
details details-mode	information

Prover Parameters	
normalization-mode(automatic {-,2a,2b,3a,3b}, user-driven) generation-of-inessential-pairs(yes|no) use-of-inessential-pairs(yes|no) generation-of-generalizations(yes|no) declare-ac-symbols(<binary symbols>) declare-c-symbols(<binary symbols>) limit-of-steps-at-subsumption-detection(<number>)	prove options
pair-selection(fair|by-user) processing-of-hypotheses-in-fixed-succession(yes|no) choice-of-coverings(one|all) independent-proving-of-equations(yes|no) incremental-mode(yes|no)	prove mode
trace-depth(1 – 5) generation-of-proof-log(yes|no) generation-of-prover-file(yes|no)	output control
show-io show-strategy	information
quit	

C How to Use UNICOM

By means of a detailed example we intend to show in this chapter how UNICOM may be used to work on a specification and verification problem. As a matter of fact, the prover is generally not able to solve (non-trivial) problems automatically. The user plays an indispensable role in supplying lemmas and, unfortunately, in orienting equations and sometimes has to control the proof process down to the last detail by entering a proof step by step. Thus, in order to succeed, the user usually needs most of the interactive facilities (information and control) of the prover.

The example deals with a union-operation on sets that is designed to be associative and commutative. For basic specifications and lemmas cf. appendix (E).

Specifying the Problem

First of all, we have to choose a representation for sets. We shall use lists (of numbers) with constructors 'nil' and 'cons' for this purpose; so we operate on the sort:

set:	nil:		$\rightarrow$ set
	cons:	nat $\times$ set	$\rightarrow$ set

Here the typical problem arises that the identity on constructor ground terms does not correspond to the intended equality on the sort; so, in this example, the 'rewrite' equality is not the desired equality on sets. (The constructor equations $\mathsf{cons}(n,\mathsf{cons}(n,s)) = \mathsf{cons}(n,s)$, $\mathsf{cons}(n,\mathsf{cons}(m,s)) = \mathsf{cons}(m,\mathsf{cons}(n,s))$ (*) would describe the equality on sets if non-freely constructed sorts were permitted. Then 'union' would reduce to an append-operation.) The properties to be proved, however, clearly deal with the equality on sets: $\mathsf{union}(x,y) = \mathsf{union}(y,x)$, $\mathsf{union}(x,\mathsf{union}(y,z)) = \mathsf{union}(\mathsf{union}(x,y),z)$ intentionally state the equality of sets which are built with 'union'. To make these equations valid, the union-operation therefore has to normalize the representation of sets. Normalization means that the identity of normalized lists corresponds to the equality on sets. In the example normalization requires us to sort lists, as we can derive from the desired AC-property. Moreover, we will remove duplicates, too:

insert: nat $\times$ set $\rightarrow$ set

$$
\begin{array}{lcl}
\mathsf{insert}(n,\mathsf{nil}) & \rightarrow & \mathsf{cons}(n,\mathsf{nil}) \\
\mathsf{insert}(n,\mathsf{cons}(m,s)) & \rightarrow & \mathsf{if\text{-}set}(n =_{Nat} m,\ \mathsf{cons}(m,s), \\
 & & \quad \mathsf{if\text{-}set}(n \leq_{Nat} m, \mathsf{cons}(n,\mathsf{cons}(m,s)), \mathsf{cons}(m,\mathsf{insert}(n,s))))
\end{array}
$$

normalize: set $\rightarrow$ set

$$
\begin{array}{lcl}
\mathsf{normalize}(\mathsf{nil}) & \rightarrow & \mathsf{nil} \\
\mathsf{normalize}(\mathsf{cons}(n,s)) & \rightarrow & \mathsf{insert}(n,\mathsf{normalize}(s))
\end{array}
$$

union: set $\times$ set $\rightarrow$ set

$$
\begin{array}{lcl}
\mathsf{union}(\mathsf{nil},s) & \rightarrow & \mathsf{normalize}(s) \\
\mathsf{union}(\mathsf{cons}(n,s),t) & \rightarrow & \mathsf{insert}(n,\mathsf{union}(s,t))
\end{array}
$$

Note that 'insert' transforms normalized lists into normalized ones and as we will see later 'insert' satisfies the properties (*). So 'insert' can be seen as a semantic constructor for sets.

Given these definitions, we have to create some specification files. It is recommendable to separate the definition part form the theorem part and to provide some initially empty specification files to write in lemmas found during the proof attempts. For example, we start with:

union (use nat)	definition
union-ac (use union-ac-lma)	the two equations of the theorem
union-ac-lma (use nat-lma, union-ac-lma-nat, union-if-hom-out, union-if-hom-in, union-if-simpl)	expected lemmas concerning the theorem
union-if-hom-out (use union)	lemmas according to (if.8) in section 'Preliminaries' (see page 38)

union-if-hom-in (use union)	lemmas according to (if.7) in section 'Preliminaries'
union-if-simpl (use union)	lemmas according to (if.1-6) in section 'Preliminaries'
union-ac-lma-nat (use lma-nat)	expected lemmas concerning the nat-specification
lma-nat	theorems concering the nat-specification proved earlier

Finding a Proof of the Theorem

Having parsed and checked these specifications (the definition rules are proved to be terminating by the checker) and generated prover-files for the specifications with empty equations-part by calling the prover, we apply the prover to the specification containing the theorem to find an adequate base of lemmas. To this end, the following configuration of the prover has turned out to be suitable:

- trace-depth = 'trace-all' – to watch all activities of the prover, esp. to determine whether inductive rules are applied
- pair-selection = 'by-user' – to select pairs the user is interested in and to observe how hypotheses are changed
- choice-of-coverings = 'by-user' – In general, the competent user is able to select an appropriate covering, variants may be investigated by using the undo operation.
- normalization-mode = 'automatic()' – one of the implemented reduction strategies, or 'user-driven' – if the user wants to take a closer look at the reduction process. For long reduction sequences, however, the user driven mode is not practicable.
- generation-of-inessential-pairs = use-of-inessential-pairs = generation-of-generalizations = 'no' – these options are to support the prover when working in a more automatic mode and offer little aid to the user.

Reaching Prover-Toplevel we proceed as follows: first we select a pair, orient it if necessary, and choose an inductively complete position for the rule. Then the new set of essential pairs is simplified and we would be well advised to analyse the results. If we know any rules which may be used to reduce the pairs further or which generalize the pairs, we add these rules to the (global) rules base as assumptions. The inserted assumptions are used to simplify pairs and we continue covering left pairs and adding assumptions. Eventually we have collected several new conjectures which we copy into (add to) a specification file and try to prove (possibly using incremental mode). If the conjectures are valid, can be proved and suffice to show the main theorem, we redo the proof of the theorem now using the proved assumptions instead.

In the example, let us first consider the commutativity of 'union': (1): $\mathsf{union}(x,y) \overset{\rightarrow}{=} \mathsf{union}(y,x)$. The equation is oriented automatically and there is only one covering. In the following, the unifier of the superposition corresponding to a critical pair is used to indicate the cases (critical pairs) produced by a covering. We get the pairs:

$x \rightharpoonup \mathsf{nil}$: (1.1): $\mathsf{normalize}(y) \overset{\leftarrow}{=} \mathsf{union}(y,\mathsf{nil})$, which is oriented as shown and covered again:

> $y \rightharpoonup \mathsf{nil}$: $\mathsf{nil} = \mathsf{normalize}(\mathsf{nil}) \Rightarrow \mathsf{nil}$
>
> $y \rightharpoonup \mathsf{cons}(n,y)$: $\mathsf{insert}(n,\mathsf{normalize}(y)) = \mathsf{insert}(n,\mathsf{union}(y,\mathsf{nil})) \Rightarrow \mathsf{insert}(n,\mathsf{normalze}(y))$, applying inductive rule (1.1).
>
> The found property (1.1) may be included in a lemma specification file to facilitate future attempts to prove the theorem.

$x \rightharpoonup \mathsf{cons}(n,x)$: $\mathsf{insert}(n,\mathsf{union}(x,y)) = \mathsf{union}(y,\mathsf{cons}(n,x))$,
leading to the lemma (1.2): $\mathsf{union}(y,\mathsf{cons}(n,x)) \overset{\rightarrow}{=} \mathsf{insert}(n,\mathsf{union}(y,x))$, which is sufficient to finish the proof of theorem (1) if inserted as assumption.

We then turn to conjecture (2): $\mathsf{union}(x,\mathsf{union}(y,z)) = \mathsf{union}(\mathsf{union}(x,y),z)$, which we orient to be cyclic and cover by instantiating the variable x (another orientation or covering may be chosen as well).

$x \rightharpoonup \mathsf{nil}$: (2.1): $\mathsf{normalize}(\mathsf{union}(y,z)) \stackrel{\rightarrow}{=} \mathsf{union}(\mathsf{normalize}(y),z)$

This pair may be generalized to the equations $\mathsf{normalize}(\mathsf{union}(x,y)) \stackrel{\rightarrow}{=} \mathsf{union}(x,y)$ and $\mathsf{union}(\mathsf{normalize}(x),y) \stackrel{\rightarrow}{=} \mathsf{union}(x,y)$. The proofs of these equations, however, involve the same lemmas as used below. We continue:

$y \rightharpoonup \mathsf{nil}$: (2.1.1): $\mathsf{normalize}(\mathsf{normalize}(z)) \stackrel{\rightarrow}{=} \mathsf{normalize}(z)$

$z \rightharpoonup \mathsf{nil}$: trivial

$z \rightharpoonup \mathsf{cons}(n,z)$: $\mathsf{normalize}(\mathsf{insert}(n,\mathsf{normalize}(z))) = \mathsf{insert}(n,\mathsf{normalize}(z))$, leading to lemma (2.1.1.1): $\mathsf{normalize}(\mathsf{insert}(n,z)) \stackrel{\rightarrow}{=} \mathsf{insert}(n,\mathsf{normalize}(z))$, stating that 'insert' preserves normalization.

$y \rightharpoonup \mathsf{cons}(n,y)$: $\mathsf{normalize}(\mathsf{insert}(n,\mathsf{union}(y,z))) = \mathsf{union}(\mathsf{insert}(n,\mathsf{normalize}(y)),z)$, reduces with assumption (2.1.1.1), inductive rule (2.1) and the assumption (2.1.2): $\mathsf{union}(\mathsf{insert}(n,y),z) \stackrel{\rightarrow}{=} \mathsf{insert}(n,\mathsf{union}(y,z))$ to a trivial pair.

$x \rightharpoonup \mathsf{cons}(n,x)$: $\mathsf{insert}(n,\mathsf{union}(x,\mathsf{union}(y,z))) = \mathsf{union}(\mathsf{insert}(n,\mathsf{union}(x,y)),z)$

After applying assumption (2.1.2), the pair is subsumed by the inductive rule (2).

Thus, the first call of the prover leads to the following collection of new conjectures:

(1.1) $\mathsf{union}(y,\mathsf{nil}) \stackrel{\rightarrow}{=} \mathsf{normalize}(y)$

(1.2) $\mathsf{union}(x,\mathsf{cons}(n,y)) \stackrel{\rightarrow}{=} \mathsf{insert}(n,\mathsf{union}(x,y))$

(2.1.1.1) $\mathsf{normalize}(\mathsf{insert}(n,z)) \stackrel{\rightarrow}{=} \mathsf{insert}(n,\mathsf{normalize}(z))$

(2.1.1) $\mathsf{normalize}(\mathsf{normalize}(z)) \stackrel{\rightarrow}{=} \mathsf{normalize}(z)$

(2.1.2) $\mathsf{union}(\mathsf{insert}(n,y),z) \stackrel{\rightarrow}{=} \mathsf{insert}(n,\mathsf{union}(y,z))$

(2.1) $\mathsf{normalize}(\mathsf{union}(y,z)) \stackrel{\rightarrow}{=} \mathsf{union}(\mathsf{normalize}(y),z)$

These equations may be treated as follows:

(1.1) : according to the proof found above

(1.2) : $x \rightharpoonup \mathsf{cons}(m,x)$: $\mathsf{insert}(m,\mathsf{union}(x,\mathsf{cons}(n,y))) = \mathsf{insert}(n,\mathsf{insert}(m,\mathsf{union}(x,y)))$

Applying the inductive rule results in:

$\mathsf{insert}(m,\mathsf{insert}(n,\mathsf{union}(x,y))) = \mathsf{insert}(n,\mathsf{insert}(m,\mathsf{union}(x,y)))$.

A non-trivial generalization of this pair leads to the first fundamental lemma for the considered validation problem: (1.2.1): $\mathsf{insert}(n,\mathsf{insert}(m,x)) \stackrel{\rightarrow}{=} \mathsf{insert}(m,\mathsf{insert}(n,x))$, which in fact holds for non-normalized x, too.

(2.1.1.1) : $\mathsf{normalize}(\mathsf{insert}(n,z)) \stackrel{\rightarrow}{=} \mathsf{insert}(n,\mathsf{normalize}(z))$

$z \rightharpoonup \mathsf{nil}$: trivial

$$z \rightharpoonup \mathsf{cons}(m,z) : \begin{array}{l} \mathsf{if\text{-}set}(n =_{Nat} m, \mathsf{normalize}(\mathsf{cons}(m,z)), \\ \quad \mathsf{if\text{-}set}(n \leq_{Nat} m, \mathsf{normalize}(\mathsf{cons}(n,\mathsf{cons}(m,z))), \\ \qquad \mathsf{normalize}(\mathsf{cons}(m,\mathsf{insert}(n,z))))) = \\ \mathsf{insert}(n,\mathsf{insert}(m,\mathsf{normalize}(z))) \end{array}$$

Applying definition rules and the inductive rule results in:

$$\begin{array}{l} \mathsf{if\text{-}set}(n =_{Nat} m, \mathsf{insert}(m, \mathsf{normalize}(z)), \\ \quad \mathsf{if\text{-}set}(n \leq_{Nat} m, \mathsf{insert}(n, \mathsf{insert}(m, \mathsf{normalize}(z))), \\ \qquad\qquad \mathsf{insert}(m, \mathsf{insert}(n, \mathsf{normalize}(z))))) = \\ \mathsf{insert}(n, \mathsf{insert}(m, \mathsf{normalize}(z))) \ . \end{array}$$

A reformulation of conjecture (1.2.1) reduces the above pair:

(1.2.1b): $\mathsf{if\text{-}set}(b, \mathsf{insert}(n, \mathsf{insert}(m, z)), \mathsf{insert}(m, \mathsf{insert}(n, z))) \ \stackrel{\rightarrow}{=}\ \mathsf{insert}(n, \mathsf{insert}(m, z))$.

We generalize the result:

$$\begin{array}{l} \mathsf{if\text{-}set}(n =_{Nat} m, \mathsf{insert}(m, \mathsf{normalize}(z)), \\ \qquad\qquad \mathsf{insert}(n, \mathsf{insert}(m, \mathsf{normalize}(z)))) = \mathsf{insert}(n, \mathsf{insert}(m, \mathsf{normalize}(z))) \end{array}$$

and obtain the new conjecture (2.1.1.1.1b):

$$\begin{array}{l} \mathsf{if\text{-}set}(n =_{Nat} m, \mathsf{insert}(m, z), \\ \qquad\qquad \mathsf{insert}(n, \mathsf{insert}(m, z))) = \mathsf{insert}(n, \mathsf{insert}(m, z)) \end{array}$$

Conjecture (2.1.2) is proved analogously with (2.1.1.1) and the proofs of (2.1.1) and (2.1) have already been planned. As conjecture (1.2.1b) collapses to equation (1.2.1) when superposing the definition of 'if-set', we are left with the equations (1.2.1) and (2.1.1.1.1b).

Equation (2.1.1.1.1b) combines properties of 'if-set',' $=_{Nat}$ ' and 'insert'. It essentially expresses the property (2.1.1.1.1): $\mathsf{insert}(n, \mathsf{insert}(n, x)) \ \stackrel{\rightarrow}{=}\ \mathsf{insert}(n, x)$, the second fundamental lemma for our problem. It turns out that equation (2.1.1.1.1b) cannot be proved by induction directly whereas equation (2.1.1.1.1) is good-natured enough to be verified. By *use* of lemma (2.1.1.1.1), conjecture (2.1.1.1.1b) can be proved *equationally* with three rewrite steps:

$$\begin{array}{lll} & \mathsf{if\text{-}set}(n =_{Nat} m, \mathsf{insert}(m, x), & \\ & \qquad \mathsf{insert}(n, \mathsf{insert}(m, x))) & \\ = & \mathsf{if\text{-}set}(n =_{Nat} m, \mathsf{insert}(m, \mathsf{insert}(m, x)), & \text{lemma (2.1.1.1.1) reverse} \\ & \qquad \mathsf{insert}(n, \mathsf{insert}(m, x))) & \\ = & \mathsf{insert}(\mathsf{if\text{-}nat}(n =_{Nat} m, m, n), \mathsf{insert}(m, x)) & \text{lemma: } \mathsf{insert}(\mathsf{if\text{-}nat}(b, x, y), z) \ \stackrel{\rightarrow}{=} \\ & & \qquad \mathsf{if\text{-}list}(b, \mathsf{insert}(x, z), \mathsf{insert}(y, z)), \text{ reverse} \\ = & \mathsf{insert}(n, \mathsf{insert}(m, x)) & \text{lemma: } \mathsf{if\text{-}nat}(n =_{Nat} m, m, n) \ \stackrel{\rightarrow}{=}\ n \ \text{, straight} \end{array}$$

This proof may be entered manually using the command 'equationally-next' at Prover-Toplevel provided that the used rules are available.

So the last equation to be treated is lemma (1.2.1), the reversibility of insertion, and this turns out to be more difficult than it appears to be in view of the well-statedness of the property. In the induction step ($x \rightharpoonup \mathsf{cons}(v, x)$), the following has to be proved:

$$\begin{array}{lcl} \mathsf{if\text{-}set}(m =_{Nat} v, \mathsf{insert}(n, \mathsf{cons}(v, x)), & = & \mathsf{if\text{-}set}(n =_{Nat} v, \mathsf{insert}(m, \mathsf{cons}(v, x)), \\ \quad \mathsf{if\text{-}set}(m \leq_{Nat} v, \mathsf{insert}(n, \mathsf{cons}(m, \mathsf{cons}(v, x))), & & \quad \mathsf{if\text{-}set}(n \leq_{Nat} v, \mathsf{insert}(m, \mathsf{cons}(n, \mathsf{cons}(v, x))), \\ \qquad \mathsf{insert}(n, \mathsf{cons}(v, \mathsf{insert}(m, x))))) & & \qquad \mathsf{insert}(m, \mathsf{cons}(v, \mathsf{insert}(n, x))))) \end{array}$$

Exhaustive simplification (about 500 reduction steps with about thirty technical rules) and removal of the leading constructor 'cons' result in the two equations:

$$\begin{array}{lcl} \mathsf{if\text{-}nat}(v \leq_{Nat} m, \mathsf{if\text{-}nat}(v \leq_{Nat} n, v, n), & = & \mathsf{if\text{-}nat}(v \leq_{Nat} n, \mathsf{if\text{-}nat}(v \leq_{Nat} m, v, m), \\ \qquad \mathsf{if\text{-}nat}(m \leq_{Nat} n, m, n)) & & \qquad \mathsf{if\text{-}nat}(n \leq_{Nat} m, n, m)) \end{array}$$

and (*):

$$
\begin{array}{l}
\mathsf{if\text{-}set}(m =_{Nat} v \wedge v =_{Nat} n, x,\\
\quad \mathsf{if\text{-}set}(m \leq_{Nat} v \wedge\\
\qquad\quad m \neq_{Nat} v \wedge m =_{Nat} n, \mathsf{cons}(v,x),\\
\qquad \mathsf{if\text{-}set}(m \neq_{Nat} v \wedge\\
\qquad\qquad v =_{Nat} n \wedge v \leq_{Nat} m, \mathsf{insert}(m,x),\\
\qquad\quad \mathsf{if\text{-}set}((m \leq_{Nat} v \vee n \leq_{Nat} v) \wedge\\
\qquad\qquad\quad (v \neq_{Nat} m \vee n \leq_{Nat} v) \wedge\\
\qquad\qquad\quad (v \leq_{Nat} m \vee n \leq_{Nat} m),\\
\qquad\qquad\quad \mathsf{cons}(\mathsf{if\text{-}nat}(v \leq_{Nat} m, v, m),\\
\qquad\qquad\qquad\quad \mathsf{if\text{-}set}(m =_{Nat} v, x,\\
\qquad\qquad\qquad\qquad \mathsf{if\text{-}set}(m \leq_{Nat} v, \mathsf{cons}(v,x),\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{insert}(m,x)))),\\
\qquad\qquad \mathsf{if\text{-}set}(m =_{Nat} v, \mathsf{insert}(n,x),\\
\qquad\qquad\quad \mathsf{if\text{-}set}(m \leq_{Nat} v,\\
\qquad\qquad\qquad \mathsf{cons}(\mathsf{if\text{-}nat}(v \leq_{Nat} n, v, n),\\
\qquad\qquad\qquad\qquad \mathsf{if\text{-}set}(n =_{Nat} v, x,\\
\qquad\qquad\qquad\qquad\quad \mathsf{if\text{-}set}(n \leq_{Nat} v, \mathsf{cons}(v,x),\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{insert}(n,x)))),\\
\qquad\qquad\qquad \mathsf{insert}(n, \mathsf{insert}(m,x))))))))
\end{array}
=
\begin{array}{l}
\mathsf{if\text{-}set}(v =_{Nat} n \wedge v =_{Nat} m, x,\\
\quad \mathsf{if\text{-}set}(n \leq_{Nat} v \wedge\\
\qquad\quad n \neq_{Nat} v \wedge n =_{Nat} m, \mathsf{cons}(v,x),\\
\qquad \mathsf{if\text{-}set}(n \neq_{Nat} v \wedge\\
\qquad\qquad v =_{Nat} m \wedge v \leq_{Nat} n, \mathsf{insert}(n,x),\\
\qquad\quad \mathsf{if\text{-}set}((n \leq_{Nat} v \vee m \leq_{Nat} v) \wedge\\
\qquad\qquad\quad (v \leq_{Nat} n \vee m \leq_{Nat} n) \wedge\\
\qquad\qquad\quad (v \neq_{Nat} n \vee m \leq_{Nat} v),\\
\qquad\qquad\quad \mathsf{cons}(\mathsf{if\text{-}nat}(v \leq_{Nat} n, v, n),\\
\qquad\qquad\qquad\quad \mathsf{if\text{-}set}(n =_{Nat} v, x,\\
\qquad\qquad\qquad\qquad \mathsf{if\text{-}set}(n \leq_{Nat} v, \mathsf{cons}(v,x),\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{insert}(n,x)))),\\
\qquad\qquad \mathsf{if\text{-}set}(n =_{Nat} v, \mathsf{insert}(m,x),\\
\qquad\qquad\quad \mathsf{if\text{-}set}(n \leq_{Nat} v,\\
\qquad\qquad\qquad \mathsf{cons}(\mathsf{if\text{-}nat}(v \leq_{Nat} m, v, m),\\
\qquad\qquad\qquad\qquad \mathsf{if\text{-}set}(m =_{Nat} v, x,\\
\qquad\qquad\qquad\qquad\quad \mathsf{if\text{-}set}(m \leq_{Nat} v, \mathsf{cons}(v,x),\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{insert}(m,x)))),\\
\qquad\qquad\qquad \mathsf{insert}(m, \mathsf{insert}(n,x))))))))
\end{array}
$$

or the like, depending on the actual set of reduction rules.

Whereas the first pair is rather reasonable, it is unclear how to overcome the second. "Feeling" that, after applying the inductive rule, the remaining problem is to prove the permutational equivalence of the nested if-expressions, we are lead to the following provable generalization:

$$
\begin{array}{l}
\mathsf{if\text{-}set}(m =_{Nat} v \wedge v =_{Nat} n, x,\\
\quad \mathsf{if\text{-}set}(m \leq_{Nat} v \wedge\\
\qquad\quad m \neq_{Nat} v \wedge m =_{Nat} n, cvx,\\
\qquad \mathsf{if\text{-}set}(m \neq_{Nat} v \wedge\\
\qquad\qquad v =_{Nat} n \wedge v \leq_{Nat} m, imx,\\
\qquad\quad \mathsf{if\text{-}set}((m \leq_{Nat} v \vee n \leq_{Nat} v) \wedge\\
\qquad\qquad\quad (v \neq_{Nat} m \vee n \leq_{Nat} v) \wedge\\
\qquad\qquad\quad (v \leq_{Nat} m \vee n \leq_{Nat} m),\\
\qquad\qquad\quad cifvm,\\
\qquad\qquad \mathsf{if\text{-}set}(m =_{Nat} v, inx,\\
\qquad\qquad\quad \mathsf{if\text{-}set}(m \leq_{Nat} v, cifvn, inimx))))))
\end{array}
\;\overset{\sim}{=}\;
\begin{array}{l}
\mathsf{if\text{-}set}(v =_{Nat} n \wedge v =_{Nat} m, x,\\
\quad \mathsf{if\text{-}set}(n \leq_{Nat} v \wedge\\
\qquad\quad n \neq_{Nat} v \wedge n =_{Nat} m, vcx,\\
\qquad \mathsf{if\text{-}set}(n \neq_{Nat} v \wedge\\
\qquad\qquad v =_{Nat} m \wedge v \leq_{Nat} n, inx,\\
\qquad\quad \mathsf{if\text{-}set}((n \leq_{Nat} v \vee m \leq_{Nat} v) \wedge\\
\qquad\qquad\quad (v \leq_{Nat} n \vee m \leq_{Nat} n) \wedge\\
\qquad\qquad\quad (v \neq_{Nat} n \vee m \leq_{Nat} v),\\
\qquad\qquad\quad cifvn,\\
\qquad\qquad \mathsf{if\text{-}set}(n =_{Nat} v, imx,\\
\qquad\qquad\quad \mathsf{if\text{-}set}(n \leq_{Nat} v, cifvm, inimx))))))
\end{array}
$$

Note that the variables in the condition terms of the if-expressions are separated from the variables of the result terms.

Supplying this lemma yields a rewrite proof of (*) of two steps (the lemma and the inductive rule). So the proof of (1.2.1) is completed if the parameter 'limit-of-steps-at-subsumption-detection' is set to '2'.

In summary, we have found the following lemma hierarchy for proving that 'union' is commutative and associative:

Theorem 1: 'Union' is commutative.

$$\mathsf{union}(x,y) \overset{\sim}{=} \mathsf{union}(y,x)$$

(1.1) $\mathsf{union}(y,\mathsf{nil}) \overset{\sim}{=} \mathsf{normalize}(y)$

(1.2) $\mathsf{union}(x,\mathsf{cons}(n,y)) \overset{\sim}{=} \mathsf{insert}(n,\mathsf{union}(x,y))$

(1.2.1) $\mathsf{insert}(n,\mathsf{insert}(m,x)) \overset{\sim}{=} \mathsf{insert}(m,\mathsf{insert}(n,x))$

Theorem 2: 'Union' is associative.

$$\mathsf{union}(x,\mathsf{union}(y,z)) \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{union}(\mathsf{union}(x,y),z)$$

uses 2.1.2

(2.1) $\mathsf{normalize}(\mathsf{union}(y,z)) \;\overset{\rightarrow}{\equiv}\; \mathsf{union}(\mathsf{normalize}(y),z)$

uses 2.1.1.1

(2.1.1) $\mathsf{normalize}(\mathsf{normalize}(z)) \;\overset{\rightarrow}{\equiv}\; \mathsf{normalize}(z)$

(2.1.1.1) $\mathsf{normalize}(\mathsf{insert}(n,z)) \;\overset{\rightarrow}{\equiv}\; \mathsf{insert}(n,\mathsf{normalize}(z))$

(1.2.1b) $\mathsf{if\text{-}set}(b,\mathsf{insert}(n,\mathsf{insert}(m,x)),\mathsf{insert}(m,\mathsf{insert}(n,x))) \;\overset{\rightarrow}{\equiv}\; \mathsf{insert}(n,\mathsf{insert}(m,x))$

(2.1.1.1.1b) $\mathsf{if\text{-}set}(n =_{Nat} m,\mathsf{insert}(m,x),\mathsf{insert}(n,\mathsf{insert}(m,x))) \;\overset{\rightarrow}{\equiv}$
$\mathsf{insert}(n,\mathsf{insert}(m,x))$

(2.1.1.1.1) $\mathsf{insert}(n,\mathsf{insert}(n,x)) \;\overset{\rightarrow}{\equiv}\; \mathsf{insert}(n,x)$

(2.1.2) $\mathsf{union}(\mathsf{insert}(n,y),z) \;\overset{\rightarrow}{\equiv}\; \mathsf{insert}(n,\mathsf{union}(y,z))$

uses 1.2.1b, 2.1.1.1.1b

D An Example Session

The example treated in the following transcript of a UNICOM session is the proof of the property of the function 'reverse' on lists to be self-inverse. The **user's input** is typeset bold-underlined and *comments* are written italic.

u(1)> **root-directory ~/demos**

u(1)> **pcp list** *Parse, check and prove specification 'list'.*

```
*** Parsing Specification  LIST ***
```

The parser echos input while parsing.

```
SPEC list

SORTS  nat, list-nat

OPS o : --> nat
    s : nat --> nat
    empty : --> list-nat
    cons: nat list-nat --> list-nat
    append : list-nat list-nat --> list-nat
    reverse : list-nat --> list-nat

CONSTRUCTORS o, s, empty, cons

EQUATIONS

      reverse(reverse(x)) = x

RULES append(empty,x) --> x
      append(cons(u,x),y) --> cons(u,append(x,y))

      reverse(empty) --> empty
      reverse(cons(u,x)) --> append(reverse(x),cons(u,empty))

ENDSPEC

*** Checking Specification  LIST ***


Checking left hand sides of rules

All left hand sides of rules are correct !
==========================================

Checking termination of rules

System is terminating !
=======================

Checking confluence

System is confluent !
=====================

Checking totality of definitions

Checking totality of APPEND
```

```
1 definition for the function symbol APPEND generated !


Checking totality of REVERSE

1 definition for the function symbol REVERSE generated !


All functions are totally defined !
===================================

*** Proving Specification  LIST ***

Incremental Mode                 : no
Independent Proving of Equations : yes
AC-symbols                       : none
C-symbols                        : none

parse-check-prove>parameter>  dc
```

declare commutative symbols

```
     ****************
     *              *
     *  1   APPEND  *
     *              *
     ****************
```

menu of binary symbols

```
declare-c-symbols>symbols>
```

<return> means empty list

```
parse-check-prove>parameter>  show-strategy

Pair Selection                      : fair
Processing of Hyps in Fixed Succession : no
Choice of Coverings                 : by-user
Normalization Mode                  : automatic-unconditional
Generation of Inessential Pairs     : no
Use of Inessential Pairs            : no
Generation of Generalizations       : no

Independent Proving of Equations    : yes
AC-Symbols                          : none
C-Symbols                           : none
Subsumption Steps                   : 1

parse-check-prove>parameter>  ps
```

parameter 'Pair Selection'

```
     ****************
     *              *
     *  1  fair     *
     *  2  by-user  *
     *              *
     ****************

parameter>pair-selection>  by-user
parse-check-prove>parameter>  quit
parse-check-prove>toplevel>  continue
```

Enter the prover cycle and don't stop after each step. At the end of each step, the prover state is saved. The command Continue continues without saving states.

```
Step: 1
```

```
1. of 1:
```

There is only one hypothesis and this is the first.

```
------------------------------------------------
|                                              |
|  Initial Hypothesis: REVERSE(REVERSE(X)) = X |
|                                              |
|  Variant: 1                                  |
|                                              |
|  Essential Pairs: 1  REVERSE(REVERSE(X)) = X |
|                                              |
------------------------------------------------

continue>choose pair> ??
```

second help command

```
     =============================================================
     >                                                           <
     >  Manual Choice of the Next Pair:                          <
     >                                                           <
     >  By traversing the current list of hypotheses, one can    <
     >     - view the hypotheses (brief or detailed information) <
     >     - delete single variants of a conjecture              <
     >     - select the pair to be processed next                <
     >                                                           <
     =============================================================

continue>choose pair> 1

Processing pair: REVERSE(REVERSE(X)) = X
of hypothesis  REVERSE(REVERSE(X)) = X


Pair inserted as inductive rule: REVERSE(REVERSE(X)) --> X  [LIST 5]


New set of essential critical pairs generated:
```

There is only one covering.

```
     REVERSE(EMPTY) = EMPTY
     REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)

End of step: 1 (state: 1)
```

The current state is saved under name '1'.

```
Step: 2
1. of 1:
------------------------------------------------------------------------------
|                                                                            |
|  Initial Hypothesis: REVERSE(REVERSE(X)) = X                               |
|                                                                            |
|  Variant: 1                                                                |
|                                                                            |
|  Essential Pairs: 1  REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y) |
|                                                                            |
------------------------------------------------------------------------------

continue>choose pair> 1

Processing pair: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)
of hypothesis  REVERSE(REVERSE(X)) = X


Pair inserted as inductive rule:
  REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) --> CONS(U,Y)  [LIST 6]
```

```
New set of essential critical pairs generated:
     REVERSE(APPEND(EMPTY,CONS(U,EMPTY))) = CONS(U,EMPTY)
     REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) = CONS(U,CONS(Z,X))

End of step: 2 (state: 2)

Step: 3
1. of 1:
-----------------------------------------------------------------------------------------
|                                                                                       |
|  Initial Hypothesis: REVERSE(REVERSE(X)) = X                                          |
|                                                                                       |
|  Variant: 1                                                                           |
|                                                                                       |
|  Essential Pairs: 1  REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) = |
|                      CONS(U,CONS(Z,X))                                                |
|                                                                                       |
-----------------------------------------------------------------------------------------

continue>choose pair>  1

Processing pair: REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) =
                 CONS(U,CONS(Z,X))
of hypothesis  REVERSE(REVERSE(X)) = X


Pair inserted as inductive rule:
  REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) -->
  CONS(U,CONS(Z,X))  [LIST 7]


New set of essential critical pairs generated:
     REVERSE(APPEND(APPEND(EMPTY,CONS(Z,EMPTY)),CONS(U,EMPTY))) = CONS(U,CONS(Z,EMPTY))
     REVERSE(APPEND(APPEND(APPEND(REVERSE(Y),CONS(V,EMPTY)),CONS(Z,EMPTY)),CONS(U,EMPTY)))
     =
     CONS(U,CONS(Z,CONS(V,Y)))

End of step: 3 (state: 3)

Step: 4
1. of 1:
----------------------------------------------------------------------------------------
|                                                                                      |
|  Initial Hypothesis: REVERSE(REVERSE(X)) = X                                         |
|                                                                                      |
|  Variant: 1                                                                          |
|                                                                                      |
|  Essential Pairs: 1  REVERSE(                                                        |
|                        APPEND(APPEND(APPEND(REVERSE(Y),CONS(V,EMPTY)),CONS(Z,EMPTY)), |
|                               CONS(U,EMPTY)))                                         |
|                      =                                                                |
|                      CONS(U,CONS(Z,CONS(V,Y)))                                        |
|                                                                                      |
----------------------------------------------------------------------------------------

continue>choose pair>  \\
```

Obviously the process diverges. We abort the command 'continue' and will step back to supply a lemma.

```
Aborting.

parse-check-prove>toplevel>  step-back show-states

*******************************
*                             *
*  (3,3)  (2,2)  (1,1)  (0,0)  *
*                             *
*******************************

step-back>back-to>  1                                    back to state '1' after step '1'
parse-check-prove>toplevel>  show hyps

***********************************************************************************
*                                                                                 *
*  1  ---------------------------------------------------------------------------  *
*     |                                                                       |  *
*     |  Initial Hypothesis: REVERSE(REVERSE(X)) = X                          |  *
*     |                                                                       |  *
*     |  Variant: 1                                                           |  *
*     |                                                                       |  *
*     |  Rules: REVERSE(REVERSE(X)) --> X                                     |  *
*     |           Inductively complete position : LHS (1)                     |  *
*     |                                                                       |  *
*     |  Cyclic Rules: none                                                   |  *
*     |                                                                       |  *
*     |  Inessential Pairs: none                                              |  *
*     |                                                                       |  *
*     |  Essential Pairs: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)  |  *
*     |                                                                       |  *
*     ---------------------------------------------------------------------------  *
*                                                                                 *
***********************************************************************************

parse-check-prove>toplevel>  add-assumptions reverse(append(x,y)) =
 append(reverse(y),reverse(x))  \\
reverse(append(x,y)) =                                            echo of the parser
append(reverse(y),reverse(x))  END

Inserted rules:                               The assumption is oriented automatically.

  REVERSE(APPEND(X,Y)) --> APPEND(REVERSE(Y),REVERSE(X))


Hypothesis accepted:
     REVERSE(REVERSE(X)) = X

#
#   No further hypotheses.
#

parse-check-prove>toplevel>  print-proof-logs          We call the printer to view the proof.

Available logs:
   1   log(s) of accepted hypotheses
   0   log(s) of rejected hypotheses
   0   log(s) of open hypotheses

Selection of Available Proofs            : all
```

```
Output to File                           : yes
Choice of Printing Options for the Proofs : uniform, with

Reductions                               : print used set of rules
Equation Applications                    : print used set of equations
Proof Graph Sequentialization            : depthfirst
Proof Graph Minimization                 : yes
Terms with AC-Symbols in Flat Form       : yes
Line Length                              : 120

print-proof-logs>options> continue

          Accepted conjectures:
```

output of the printer follows

```
======================================================================

The conjecture comprises the equation(s):

   1    REVERSE(REVERSE(X)) = X

======================================================================

1: essential pair: REVERSE(REVERSE(X)) = X

     (conjectured by user)

is oriented to the rule: REVERSE(REVERSE(X)) --> X

The rule is covered at the inductively complete position
     lhs: (1)  by the critical pairs:

with rule: REVERSE(EMPTY) := EMPTY
  1.1:  REVERSE(EMPTY) = EMPTY
with rule: REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))
  1.2:  REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)

======================================================================

1.1: essential pair: REVERSE(EMPTY) = EMPTY

Reduction by the rule(s): (1 steps)

   definition rule   : REVERSE(EMPTY) := EMPTY

results in: EMPTY = EMPTY

  which is trivial.

======================================================================

1.2: essential pair: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)

Reduction by the rule(s): (7 steps)

   assumption        : REVERSE(APPEND(X,Y)) --> APPEND(REVERSE(Y),REVERSE(X))
   inductive rule    : REVERSE(REVERSE(X)) --> X     (see 1)
   definition rule   : REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))
   definition rule   : REVERSE(EMPTY) := EMPTY
   definition rule   : APPEND(EMPTY,X) := X
```

```
   definition rule   : APPEND(CONS(U,X),Y) := CONS(U,APPEND(X,Y))

results in: CONS(U,Y) = CONS(U,Y)

  which is trivial.


This concludes the accepting proof.

======================================================================

#
#   No further hypotheses.
#
```
back at Prover-Toplevel

parse-check-prove>toplevel> <u>**step-back 0**</u>

We undo all and try to prove the theorem under generation of inessential pairs.

parse-check-prove>toplevel> <u>**change-parameter +gip +uip show-strategy**</u>

```
Pair Selection                        : by-user
Processing of Hyps in Fixed Succession : no
Choice of Coverings                   : by-user
Normalization Mode                    : automatic-unconditional
Generation of Inessential Pairs       : yes
Use of Inessential Pairs              : if-reductive
Generation of Generalizations         : no

Independent Proving of Equations      : yes
AC-Symbols                            : none
C-Symbols                             : none
Subsumption Steps                     : 1
```
switched on by '+gip'
switched on by '+uip'

change-parameter>parameter> <u>**td**</u>

To watch the process we choose a deeper trace level.

```
     **********************
     *                    *
     *  1  no-trace       *
     *  2  trace-pairs    *
     *  3  trace-steps    *
     *  4  trace-all      *
     *  5  trace-all-full *
     *                    *
     **********************
```

trace-depth>depth> <u>**trace-all**</u>
change-parameter>parameter> <u>**quit**</u>
parse-check-prove>toplevel> <u>**next**</u>

do one (the next) step

```
Step: 1
1. of 1:
-------------------------------------------------
|                                               |
|  Initial Hypothesis: REVERSE(REVERSE(X)) = X  |
|                                               |
|  Variant: 1                                   |
|                                               |
|  Essential Pairs: 1  REVERSE(REVERSE(X)) = X  |
|                                               |
-------------------------------------------------
```

```
next>choose pair> 1

Processing pair: REVERSE(REVERSE(X)) = X
of hypothesis  REVERSE(REVERSE(X)) = X


Pair inserted as inductive rule: REVERSE(REVERSE(X)) --> X  [LIST 5]


New set of essential critical pairs generated:
     REVERSE(EMPTY) = EMPTY
     REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)

Normalizing essential pair: REVERSE(EMPTY) = EMPTY
LHS (0), definition rule: REVERSE(EMPTY) := EMPTY
Result of normalization: EMPTY = EMPTY

Normalizing essential pair: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)
Result of normalization: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)

Deleting trivial essential pair: EMPTY = EMPTY


New inessential critical pairs:
```

Superpositions into the currently processed pair: reverse(reverse(x)) = x . So the generated essential pairs are built again.

```
with inductive rule: REVERSE(REVERSE(X)) --> X  [LIST 5]
at position lhs (1):
   REVERSE(Y) = REVERSE(Y)
with definition rule: REVERSE(EMPTY) := EMPTY  [LIST 3]
at position lhs (1):
   REVERSE(EMPTY) = EMPTY
with definition rule: REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))  [LIST 4]
at position lhs (1):
   REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)
with inductive rule: REVERSE(REVERSE(X)) --> X  [LIST 5]
at position lhs ():
   Y = Y

Deleting trivial inessential pair: REVERSE(Y) = REVERSE(Y)

Deleting trivial inessential pair: Y = Y

Normalizing inessential pair: REVERSE(EMPTY) = EMPTY
LHS (0), definition rule: REVERSE(EMPTY) := EMPTY
Result of normalization: EMPTY = EMPTY

Normalizing inessential pair: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)
Result of normalization: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)

Deleting trivial inessential pair: EMPTY = EMPTY

Deleting inessential pair: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)
which is subsumed by (1 steps):
  essential pair: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)

End of step: 1 (state: 4)
```

```
parse-check-prove>toplevel>  next ; details

Step: 2
1. of 1:
------------------------------------------------------------------------------
|                                                                            |
|  Initial Hypothesis: REVERSE(REVERSE(X)) = X                               |
|                                                                            |
|  Variant: 1                                                                |
|                                                                            |
|  Rules: REVERSE(REVERSE(X)) --> X                                          |
|           Inductively complete position : LHS (1)                          |
|                                                                            |
|  Cyclic Rules: none                                                        |
|                                                                            |
|  Inessential Pairs: none                                                   |
|                                                                            |
|  Essential Pairs: 1  REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)  |
|                                                                            |
------------------------------------------------------------------------------

next>choose pair>  1

Processing pair: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) = CONS(U,Y)
of hypothesis  REVERSE(REVERSE(X)) = X


Pair inserted as inductive rule:
  REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) --> CONS(U,Y)  [LIST 6]


New set of essential critical pairs generated:                There is only one covering.
     REVERSE(APPEND(EMPTY,CONS(U,EMPTY))) = CONS(U,EMPTY)
     REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) = CONS(U,CONS(Z,X))

Normalizing essential pair: REVERSE(APPEND(EMPTY,CONS(U,EMPTY))) = CONS(U,EMPTY)
LHS (1), definition rule: APPEND(EMPTY,X) := X
LHS (0), definition rule: REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))
LHS (1), definition rule: REVERSE(EMPTY) := EMPTY
LHS (0), definition rule: APPEND(EMPTY,X) := X
Result of normalization: CONS(U,EMPTY) = CONS(U,EMPTY)

Normalizing essential pair:
  REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) = CONS(U,CONS(Z,X))
Result of normalization:
  REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) = CONS(U,CONS(Z,X))

Deleting trivial essential pair: CONS(U,EMPTY) = CONS(U,EMPTY)


New inessential critical pairs:
with inductive rule: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) --> CONS(U,Y)  [LIST 6]
at position lhs (1 1):
   REVERSE(APPEND(CONS(X,Z),CONS(U,EMPTY))) = CONS(U,APPEND(REVERSE(Z),CONS(X,EMPTY)))
with inductive rule: REVERSE(REVERSE(X)) --> X [LIST 5]                 birth of the key rule
at position lhs (1 1):
   REVERSE(APPEND(X,CONS(U,EMPTY))) = CONS(U,REVERSE(X))
with definition rule: REVERSE(EMPTY) := EMPTY  [LIST 3]
at position lhs (1 1):
```

```
   REVERSE(APPEND(EMPTY,CONS(U,EMPTY))) = CONS(U,EMPTY)
with definition rule: REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))  [LIST 4]
at position lhs (1 1):
   REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) = CONS(U,CONS(Z,X))
with inductive rule: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) --> CONS(U,Y)  [LIST 6]
at position lhs ():
   CONS(X,Z) = CONS(X,Z)

Deleting trivial inessential pair: CONS(X,Z) = CONS(X,Z)

Normalizing inessential pair:
  REVERSE(APPEND(CONS(X,Z),CONS(U,EMPTY))) = CONS(U,APPEND(REVERSE(Z),CONS(X,EMPTY)))
LHS (1), definition rule: APPEND(CONS(U,X),Y) := CONS(U,APPEND(X,Y))
LHS (0), definition rule: REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))
Result of normalization: APPEND(REVERSE(APPEND(Z,CONS(U,EMPTY))),CONS(X,EMPTY)) =
                         CONS(U,APPEND(REVERSE(Z),CONS(X,EMPTY)))
```

normalization of the key rule

```
Normalizing inessential pair: REVERSE(APPEND(X,CONS(U,EMPTY))) = CONS(U,REVERSE(X))
Result of normalization: REVERSE(APPEND(X,CONS(U,EMPTY))) = CONS(U,REVERSE(X))

Normalizing inessential pair: REVERSE(APPEND(EMPTY,CONS(U,EMPTY))) = CONS(U,EMPTY)
LHS (1), definition rule: APPEND(EMPTY,X) := X
LHS (0), definition rule: REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))
LHS (1), definition rule: REVERSE(EMPTY) := EMPTY
LHS (0), definition rule: APPEND(EMPTY,X) := X
Result of normalization: CONS(U,EMPTY) = CONS(U,EMPTY)

Normalizing inessential pair:
  REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) = CONS(U,CONS(Z,X))
Result of normalization:
  REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) = CONS(U,CONS(Z,X))

Deleting trivial inessential pair: CONS(U,EMPTY) = CONS(U,EMPTY)

Deleting inessential pair: REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY)))
                           =
                           CONS(U,CONS(Z,X))
which is subsumed by (1 steps):
  essential pair: REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) =
                  CONS(U,CONS(Z,X))

Changing status of inessential critical pair(s):
```

'essentialization' of the key rule

```
     REVERSE(APPEND(X,CONS(U,EMPTY))) = CONS(U,REVERSE(X))

to 'essential', the corresponding rules of which may reduce the
essential pair: REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) =
                CONS(U,CONS(Z,X)).

End of step: 2 (state: 5)

parse-check-prove>toplevel> next ; details

Step: 3
1. of 1:
-----------------------------------------------------------------------------------------
|                                                                                       |
|  Initial Hypothesis: REVERSE(REVERSE(X)) = X                                          |
|                                                                                       |
|  Variant: 1                                                                           |
```

```
|                                                                                           |
|  Rules: REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) --> CONS(U,Y)                           |
|           Inductively complete position : LHS (1 1)                                        |
|         REVERSE(REVERSE(X)) --> X                                                          |
|           Inductively complete position : LHS (1)                                          |
|                                                                                           |
|  Cyclic Rules: none                                                                       |
|                                                                                           |
|  Inessential Pairs: APPEND(REVERSE(APPEND(Z,CONS(U,EMPTY))),CONS(X,EMPTY)) =              |
|                     CONS(U,APPEND(REVERSE(Z),CONS(X,EMPTY)))                               |
|                                                                                           |
|  Essential Pairs: 1  REVERSE(APPEND(X,CONS(U,EMPTY))) = CONS(U,REVERSE(X))                |
|                   2  REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) =    |
|                      CONS(U,CONS(Z,X))                                                     |
|                                                                                           |
---------------------------------------------------------------------------------------------

next>choose pair> 1

Processing pair: REVERSE(APPEND(X,CONS(U,EMPTY))) = CONS(U,REVERSE(X))
of hypothesis  REVERSE(REVERSE(X)) = X


Pair inserted as inductive rule:
  REVERSE(APPEND(X,CONS(U,EMPTY))) --> CONS(U,REVERSE(X))  [LIST 7]
```

Now the key rule is available and used to simplify pairs.

```
Normalizing essential pair:
  REVERSE(APPEND(APPEND(REVERSE(X),CONS(Z,EMPTY)),CONS(U,EMPTY))) = CONS(U,CONS(Z,X))
LHS (0), inductive rule: REVERSE(APPEND(X,CONS(U,EMPTY))) --> CONS(U,REVERSE(X))
LHS (1), inductive rule: REVERSE(APPEND(X,CONS(U,EMPTY))) --> CONS(U,REVERSE(X))
LHS (2), inductive rule: REVERSE(REVERSE(X)) --> X
Result of normalization: CONS(U,CONS(Z,X)) = CONS(U,CONS(Z,X))

Deleting trivial essential pair: CONS(U,CONS(Z,X)) = CONS(U,CONS(Z,X))

Normalizing inessential pair: APPEND(REVERSE(APPEND(Z,CONS(U,EMPTY))),CONS(X,EMPTY)) =
                              CONS(U,APPEND(REVERSE(Z),CONS(X,EMPTY)))
LHS (1), inductive rule: REVERSE(APPEND(X,CONS(U,EMPTY))) --> CONS(U,REVERSE(X))
LHS (0), definition rule: APPEND(CONS(U,X),Y) := CONS(U,APPEND(X,Y))
Result of normalization:
  CONS(U,APPEND(REVERSE(Z),CONS(X,EMPTY))) = CONS(U,APPEND(REVERSE(Z),CONS(X,EMPTY)))

Deleting trivial inessential pair:
  CONS(U,APPEND(REVERSE(Z),CONS(X,EMPTY))) = CONS(U,APPEND(REVERSE(Z),CONS(X,EMPTY)))


New set of essential critical pairs generated:
```

"proof" of the key rule

```
     REVERSE(CONS(U,EMPTY)) = CONS(U,REVERSE(EMPTY))
     REVERSE(CONS(Z,APPEND(V,CONS(U,EMPTY)))) = CONS(U,REVERSE(CONS(Z,V)))

Normalizing essential pair: REVERSE(CONS(U,EMPTY)) = CONS(U,REVERSE(EMPTY))
LHS (0), definition rule: REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))
RHS (1), definition rule: REVERSE(EMPTY) := EMPTY
LHS (1), definition rule: REVERSE(EMPTY) := EMPTY
LHS (0), definition rule: APPEND(EMPTY,X) := X
Result of normalization: CONS(U,EMPTY) = CONS(U,EMPTY)

Normalizing essential pair:
```

```
  REVERSE(CONS(Z,APPEND(V,CONS(U,EMPTY)))) = CONS(U,REVERSE(CONS(Z,V)))
LHS (0), definition rule: REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))
RHS (1), definition rule: REVERSE(CONS(U,X)) := APPEND(REVERSE(X),CONS(U,EMPTY))
LHS (1), inductive rule: REVERSE(APPEND(X,CONS(U,EMPTY))) --> CONS(U,REVERSE(X))
LHS (0), definition rule: APPEND(CONS(U,X),Y) := CONS(U,APPEND(X,Y))
Result of normalization:
  CONS(U,APPEND(REVERSE(V),CONS(Z,EMPTY))) = CONS(U,APPEND(REVERSE(V),CONS(Z,EMPTY)))

Deleting trivial essential pair:
  CONS(U,APPEND(REVERSE(V),CONS(Z,EMPTY))) = CONS(U,APPEND(REVERSE(V),CONS(Z,EMPTY)))

Deleting trivial essential pair: CONS(U,EMPTY) = CONS(U,EMPTY)


Hypothesis accepted:
     REVERSE(REVERSE(X)) = X

End of step: 3 (state: 6)

#
#   No further hypotheses.
#

parse-check-prove>toplevel> show accepted-hyps

****************************************************************************
*                                                                          *
*  1  ------------------------------------------------------------------  *
*     |                                                                |  *
*     |  Initial Hypothesis: REVERSE(REVERSE(X)) = X                   |  *
*     |                                                                |  *
*     |  Variant: 1                                                    |  *
*     |                                                                |  *
*     |  Rules: REVERSE(APPEND(X,CONS(U,EMPTY))) --> CONS(U,REVERSE(X)) |  *
*     |           Inductively complete position : LHS (1)              |  *
*     |         REVERSE(APPEND(REVERSE(Y),CONS(U,EMPTY))) --> CONS(U,Y) |  *
*     |           Inductively complete position : LHS (1 1)            |  *
*     |         REVERSE(REVERSE(X)) --> X                              |  *
*     |           Inductively complete position : LHS (1)              |  *
*     |                                                                |  *
*     |  Cyclic Rules: none                                            |  *
*     |                                                                |  *
*     |  Inessential Pairs: none                                       |  *
*     |                                                                |  *
*     |  Essential Pairs: none                                         |  *
*     |                                                                |  *
*     ------------------------------------------------------------------  *
*                                                                          *
****************************************************************************

#
#   No further hypotheses.
#

parse-check-prove>toplevel> next
Writing prover file
Writing proof log file

u(1)>
```

E A Collection of Examples

The following collection contains example specifications and theorems from the domains 'natural numbers', 'lists' and 'trees' which were treated with the UNICOM system. Although the involved rule systems are not proved to be terminating by the system in general, the proofs found are correct, i.e. the generated essential pairs are subconnected with respect to a terminating ordering[5]. Each example is divided into two parts. The definition part contains the sort declarations (with constructors) and function definitions (arity with definition rules). The theorem part contains a collection of theorems and for each theorem the hierarchy of lemmas used to prove it.

The next section deals with the common basis of all examples.

Preliminaries

Specifications concerning boolean expressions and natural numbers are part of nearly every specification. Boolean operators are involved both in the definition of functions (esp. in the definition of predicates (functions into the sort bool)) and clearly, in the formulation of conjectures. Natural numbers, unless needed directly, often serve as sort parameter, e.g. to 'enumerate' the elements of a sort in parameterized specifications (e.g. elements of lists, function symbols in terms). So this section deals with these fundamental definitions, the lemmas used to normalize boolean expressions, lemmas concerning the basic functions on natural numbers, and the if-functor and the identity predicate which are defined schematically for any sort.

Basic Specifications

The specifications concerning boolean expressions and natural numbers consist of the definition of the usual boolean connectives 'not', 'and', 'or', 'if-bool' and the basic functions on natural numbers: ordering, addition, subtraction and predecessor, as follows:

Sorts:

bool:	true:		$\rightarrow$ bool
	false:		$\rightarrow$ bool
nat:	zero:		$\rightarrow$ nat
	suc:	nat	$\rightarrow$ nat

Function Definitions:

not: bool → bool
and: bool × bool → bool
or: bool × bool → bool
imply: bool × bool → bool
if-bool: bool × bool × bool → bool

$$\begin{array}{lcl} \mathsf{not}(\mathsf{false}) & \rightarrow & \mathsf{true} \\ \mathsf{not}(\mathsf{true}) & \rightarrow & \mathsf{false} \end{array}$$

$$\begin{array}{lcl} \mathsf{and}(\mathsf{false}, x) & \rightarrow & \mathsf{false} \\ \mathsf{and}(\mathsf{true}, x) & \rightarrow & x \end{array} \qquad \begin{array}{lcl} \mathsf{or}(\mathsf{false}, x) & \rightarrow & x \\ \mathsf{or}(\mathsf{true}, x) & \rightarrow & \mathsf{true} \end{array}$$

[5] This can be verified manually using e.g. semantical path orderings (see [KL80]).

imply(false,x)	$\rightarrow$	true	if-bool(true,x,y)	$\rightarrow$	x
imply(true,x)	$\rightarrow$	x	if-bool(false,x,y)	$\rightarrow$	y

$\leq_{Nat}$ **: nat $\times$ nat $\rightarrow$ nat**
$>_{Nat}$ **: nat $\times$ nat $\rightarrow$ nat**

$\leq_{Nat}$ (zero,x)	$\rightarrow$	true	$>_{Nat}$ (zero,x)	$\rightarrow$	false
$\leq_{Nat}$ (suc(x),zero)	$\rightarrow$	false	$>_{Nat}$ (suc(x),zero)	$\rightarrow$	true
$\leq_{Nat}$ (suc(x),suc(y))	$\rightarrow$	$\leq_{Nat}$ (x,y)	$>_{Nat}$ (suc(x),suc(y))	$\rightarrow$	$>_{Nat}$ (x,y)

add: nat $\times$ nat $\rightarrow$ nat
sub: nat $\times$ nat $\rightarrow$ nat
pred: nat $\rightarrow$ nat

pred(zero)	$\rightarrow$	zero
pred(suc(x))	$\rightarrow$	x

			sub(zero,x)	$\rightarrow$	zero
add(zero,x)	$\rightarrow$	x	sub(x,zero)	$\rightarrow$	x
add(suc(x),y)	$\rightarrow$	suc(add(x,y))	sub(suc(x),suc(y))	$\rightarrow$	sub(x,y)

max: nat $\times$ nat $\rightarrow$ nat
min: nat $\times$ nat $\rightarrow$ nat

max(x,zero)	$\rightarrow$	x	min(zero,x)	$\rightarrow$	zero
max(zero,x)	$\rightarrow$	x	min(x,zero)	$\rightarrow$	zero
max(suc(x),suc(y))	$\rightarrow$	suc(max(x,y))	min(suc(x),suc(y))	$\rightarrow$	suc(min(x,y))

If opportune, will use infix notation for the binary operations: and ($\wedge$), or ($\vee$), imply ($\rightarrow$), $>_{Nat}$, $\leq_{Nat}$.

The If-Operator and the Identity Predicate

For each sort s an if-operator and an identity predicate are assumed to be defined according to the following scheme:

if-s: bool $\times$ s $\times$ s $\rightarrow$ s

if-s(true,x,y)	$\rightarrow$	x
if-s(false,x,y)	$\rightarrow$	y

$=_s$**: s $\times$ s $\rightarrow$ bool**

$=_s$(c($x_1,\ldots,x_n$),c($y_1,\ldots,y_n$))	$\rightarrow$	$x_1 =_{s_1} y_1 \wedge \ldots \wedge x_n =_{s_n} y_n$
$=_s$(c($x_1,\ldots,x_n$),d($y_1,\ldots,y_m$))	$\rightarrow$	false

for all constructors c$\neq$d of sort s.

Example:

$=_{Nat}$ (zero,zero)	$\rightarrow$	true
$=_{Nat}$ (suc(x),zero)	$\rightarrow$	false
$=_{Nat}$ (zero,suc(x))	$\rightarrow$	false
$=_{Nat}$ (suc(x),suc(y))	$\rightarrow$	$=_{Nat}$ (x,y)

In addition, a lot of lemmas related to the if-operators are assumed to be available (to be supplied by need), e.g.:

(if.1) if(b,x,x) $\stackrel{\rightarrow}{=}$ x[6]

(if.2) if(b,x,if(a,x,y)) $\stackrel{\rightarrow}{=}$ if($b \vee a$,x,y)

[6] An arrow above the '='-symbol indicates the orientation of the rewrite rule corresponding to the equation.

(if.3) $\mathsf{if}(u,x,\mathsf{if}(v,y,\mathsf{if}(w,z,x))) \ \vec{\equiv}\ \mathsf{if}(u \vee (\neg v \wedge w),x,\mathsf{if}(v,y,z))$

(if.4) $\mathsf{if}(b,x,\mathsf{if}(b,y,z)) \ \vec{\equiv}\ \mathsf{if}(b,x,z)$

(if.5) $\mathsf{if}(b,x,\mathsf{if}(\neg b,y,z)) \ \vec{\equiv}\ \mathsf{if}(b,x,y)$

(if.6) $\mathsf{if}(b,x,\mathsf{if}(\mathsf{b} \wedge a,y,z)) \ \vec{\equiv}\ \mathsf{if}(b,x,z)$

(...)

(if.7) $\mathsf{f}(\ldots,\mathsf{if}\text{-}\mathsf{s}_i(b,x,y),\ldots) \ \vec{\equiv}\ \mathsf{if}\text{-}\mathsf{s}(b,\mathsf{f}(\ldots,x,\ldots),\mathsf{f}(\ldots,y,\ldots))$
where f: $\ldots \times \mathsf{s}_i \times \ldots \to \mathsf{s}$ is a defined symbol.

(if.8) $\mathsf{if}\text{-}\mathsf{s}(b,\mathsf{c}(x_1,\ldots,x_n),\mathsf{c}(y_1,\ldots,y_n)) \ \vec{\equiv}\ \mathsf{c}(\mathsf{if}\text{-}\mathsf{s}_1(b,x_1,y_1),\ldots,\mathsf{if}\text{-}\mathsf{s}_n(b,x_n,y_n))$
where c is a constructor.

Example:

$$\begin{aligned}
&\mathsf{add}(x,\mathsf{if}\text{-}\mathsf{nat}(b,y,z)) \ \vec{\equiv}\ \mathsf{if}\text{-}\mathsf{nat}(b,\mathsf{add}(x,y),\mathsf{add}(x,z))\\
&\mathsf{add}(\mathsf{if}\text{-}\mathsf{nat}(b,x,y),z) \ \vec{\equiv}\ \mathsf{if}\text{-}\mathsf{nat}(b,\mathsf{add}(x,z),\mathsf{add}(y,z))\\
&\mathsf{if}\text{-}\mathsf{nat}(b,\mathsf{suc}(x),\mathsf{suc}(y)) \ \vec{\equiv}\ \mathsf{suc}(\mathsf{if}\text{-}\mathsf{nat}(b,x,y))
\end{aligned}$$

The identity predicates should always be declared to be C-symbols.

Boolean Lemmas

The following is the complete list of lemmas used to transform boolean expressions into their conjunctive normal form. In most examples the disjunctive normal form might be used as well, but it should be noted that in general, lemmas have to be formulated in a different way when using different boolean normal forms. I.e. the lemmas base which has been found to prove a theorem using the rules for generating conjunctive normal form has often to be changed or extended, when the rules for generating conjunctive normal form are replaced by the rules generating disjunctive normal form, in order to prove the theorem using the altered rules base.

(1) $\mathsf{imply}(x,y) \ \vec{\equiv}\ \neg x \vee y$

(2) $\mathsf{if}\text{-}\mathsf{bool}(b,x,y) \ \vec{\equiv}\ \mathsf{imply}(b,x) \wedge \mathsf{imply}(\mathsf{not}(b),y)$

(3) $\mathsf{not}(\mathsf{not}(x)) \ \vec{\equiv}\ x$

(4) $\mathsf{and}(x,x) \ \vec{\equiv}\ x$

(5) $\mathsf{or}(x,x) \ \vec{\equiv}\ x$

(6) $\mathsf{or}(x,\mathsf{not}(x)) \ \vec{\equiv}\ \mathsf{true}$

(7) $\mathsf{and}(x,\mathsf{not}(x)) \ \vec{\equiv}\ \mathsf{false}$

(8) $\mathsf{and}(\mathsf{not}(x),\mathsf{or}(x,y)) \ \vec{\equiv}\ \mathsf{and}(\mathsf{not}(x),y)$

(9) $\mathsf{and}(x,\mathsf{or}(\mathsf{not}(x),y)) \ \vec{\equiv}\ \mathsf{and}(x,y)$

(10) $\mathsf{and}(\mathsf{or}(x,y),\mathsf{or}(x,\mathsf{or}(\mathsf{not}(y),z))) \ \vec{\equiv}\ \mathsf{and}(\mathsf{or}(x,y),\mathsf{or}(x,z))$

(11) $\mathsf{and}(\mathsf{or}(x,y),\mathsf{or}(x,\mathsf{not}(y))) \ \vec{\equiv}\ x$

(12) $\mathsf{not}(\mathsf{and}(x,y)) \ \vec{\equiv}\ \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))$

(13) $\mathsf{not}(\mathsf{or}(x,y)) \ \vec{\equiv}\ \mathsf{and}(\mathsf{not}(x),\mathsf{not}(y))$

(14) $\mathsf{or}(\mathsf{and}(x,y),z) \ \vec{\equiv}\ \mathsf{and}(\mathsf{or}(x,z),\mathsf{or}(y,z))$

The operators 'and', 'or' are declared to be AC-symbols.

Lemmas Concerning Functions on Natural Numbers

The used lemma base for the operators $\leq_{Nat}$, $=_{Nat}$, add, sub etc. contains about fifty equations which state their basic properties. This relatively high number of equations is due to the fact that properties often have to be combined to form applicable reduction rules or properties have to be formulated in several 'versions'. We restrict ourselves to a list of some examples. In general, it is no problem to prove equations of that kind:

(1) $>_{Nat}(x,y) \;\overrightarrow{\equiv}\; \neg \leq_{Nat}(y,x)$

(2) $\mathsf{pred}(x) \;\overrightarrow{\equiv}\; \mathsf{sub}(x,\mathsf{suc}(\mathsf{zero}))$

(3) $x \leq_{Nat} x \;\overrightarrow{\equiv}\; \mathsf{true}$

(4) $x \leq_{Nat} y \wedge y \leq_{Nat} x \;\overrightarrow{\equiv}\; x =_{Nat} y$

(5) $x \leq_{Nat} y \vee y \leq_{Nat} x \;\overrightarrow{\equiv}\; \mathsf{true}$

(6) $x \leq_{Nat} y \vee x =_{Nat} y \;\overrightarrow{\equiv}\; x \leq_{Nat} y$

(7) $x \leq_{Nat} y \wedge x =_{Nat} y \;\overrightarrow{\equiv}\; x =_{Nat} y$

(8) $(x \leq_{Nat} y \vee b) \wedge x =_{Nat} y \;\overrightarrow{\equiv}\; x =_{Nat} y$

(9) $x \leq_{Nat} y \wedge y \leq_{Nat} z \wedge x \leq_{Nat} z \;\overrightarrow{\equiv}\; x \leq_{Nat} y \wedge y \leq_{Nat} z$

(10) $x \leq_{Nat} y \vee y \leq_{Nat} z \vee \neg x \leq_{Nat} z \;\overrightarrow{\equiv}\; \mathsf{true}$

(11) $\mathsf{sub}(\mathsf{sub}(x,y),z) \;\overrightarrow{\equiv}\; \mathsf{sub}(x,\mathsf{add}(y,z))$

(12) $\mathsf{add}(x,y) =_{Nat} \mathsf{zero} \;\overrightarrow{\equiv}\; x =_{Nat} \mathsf{zero} \wedge y =_{Nat} \mathsf{zero}$

(13) $\mathsf{add}(x,y) =_{Nat} \mathsf{add}(x,z) \;\overrightarrow{\equiv}\; y =_{Nat} z$

(14) $\mathsf{if\text{-}nat}(x \leq_{Nat} y,x,y) \;\overrightarrow{\equiv}\; \mathsf{if\text{-}nat}(y \leq_{Nat} x,y,x)$

The operator 'add' is declared to be an AC-Symbol.

Example I: Arithmetic

This example deals with some basic properties of arithmetical functions on natural numbers. The proofs of most of the theorems are produced by UNICOM without user support – lemmas are needed rarely.

Function Definitions:

double: nat $\rightarrow$ nat
half: nat $\rightarrow$ nat

$$\begin{array}{lcl} \mathsf{double}(\mathsf{zero}) & \rightarrow & \mathsf{zero} \\ \mathsf{double}(\mathsf{suc}(x)) & \rightarrow & \mathsf{suc}(\mathsf{suc}(\mathsf{double}(x))) \end{array}$$

$$\begin{array}{lcl} \mathsf{half}(\mathsf{zero}) & \rightarrow & \mathsf{zero} \\ \mathsf{half}(\mathsf{suc}(\mathsf{zero})) & \rightarrow & \mathsf{zero} \\ \mathsf{half}(\mathsf{suc}(\mathsf{suc}(x))) & \rightarrow & \mathsf{suc}(\mathsf{half}(x)) \end{array}$$

even: nat $\rightarrow$ bool

$$\begin{array}{lcl} \mathsf{even}(\mathsf{zero}) & \rightarrow & \mathsf{true} \\ \mathsf{even}(\mathsf{suc}(\mathsf{zero})) & \rightarrow & \mathsf{false} \\ \mathsf{even}(\mathsf{suc}(\mathsf{suc}(x))) & \rightarrow & \mathsf{even}(x) \end{array}$$

sum: nat $\rightarrow$ nat
sum-odd: nat $\rightarrow$ nat
sum-even: nat $\rightarrow$ nat

$$\begin{array}{lcl} \mathsf{sum}(\mathsf{zero}) & \rightarrow & \mathsf{zero} \\ \mathsf{sum}(\mathsf{suc}(x)) & \rightarrow & \mathsf{add}(\mathsf{suc}(x),\mathsf{sum}(x)) \end{array}$$

$$\begin{array}{lcl} \mathsf{sum\text{-}odd}(\mathsf{zero}) & \rightarrow & \mathsf{one} \\ \mathsf{sum\text{-}odd}(\mathsf{suc}(x)) & \rightarrow & \mathsf{add}(\mathsf{suc}(\mathsf{mul}(\mathsf{two},\mathsf{suc}(x))),\mathsf{sum\text{-}odd}(x)) \end{array}$$

$$\begin{array}{lcl} \mathsf{sum\text{-}even}(\mathsf{zero}) & \rightarrow & \mathsf{zero} \\ \mathsf{sum\text{-}even}(\mathsf{suc}(x)) & \rightarrow & \mathsf{add}(\mathsf{mul}(\mathsf{two},\mathsf{suc}(x)),\mathsf{sum\text{-}even}(x)) \end{array}$$

mul: nat $\times$ nat $\rightarrow$ nat
square: nat $\rightarrow$ nat
cube: nat $\rightarrow$ nat
power4: nat $\rightarrow$ nat
power5: nat $\rightarrow$ nat
exp: nat $\times$ nat $\rightarrow$ nat

$$\begin{array}{lcl} \mathsf{mul}(\mathsf{zero},x) & \rightarrow & \mathsf{zero} \\ \mathsf{mul}(\mathsf{suc}(x),y) & \rightarrow & \mathsf{add}(y,\mathsf{mul}(x,y)) \end{array}$$

$$\mathsf{square}(x) \rightarrow \mathsf{mul}(x,x)$$

$$\mathsf{cube}(x) \rightarrow \mathsf{mul}(x,\mathsf{mul}(x,x))$$

$$\mathsf{power4}(x) \rightarrow \mathsf{mul}(\mathsf{square}(x),\mathsf{square}(x))$$

$$\mathsf{power5}(x) \rightarrow \mathsf{mul}(x,\mathsf{mul}(x,\mathsf{mul}(x,\mathsf{mul}(x,x))))$$

$$\begin{array}{lcl} \mathsf{exp}(x,\mathsf{zero}) & \rightarrow & \mathsf{one} \\ \mathsf{exp}(x,\mathsf{suc}(y)) & \rightarrow & \mathsf{mul}(x,\mathsf{exp}(x,y)) \end{array}$$

sum-square: nat $\rightarrow$ nat
sum-cube: nat $\rightarrow$ nat
sum-power4: nat $\rightarrow$ nat

$$\begin{array}{lcl} \mathsf{sum\text{-}square}(\mathsf{zero}) & \rightarrow & \mathsf{zero} \\ \mathsf{sum\text{-}square}(\mathsf{suc}(n)) & \rightarrow & \mathsf{add}(\mathsf{square}(\mathsf{suc}(n)),\mathsf{sum\text{-}square}(n)) \end{array}$$

$$\begin{array}{lcl} \mathsf{sum\text{-}cube(zero)} & \rightarrow & \mathsf{zero} \\ \mathsf{sum\text{-}cube(suc(}x\mathsf{))} & \rightarrow & \mathsf{add(cube(suc(}x\mathsf{)),sum\text{-}cube(}x\mathsf{))} \end{array}$$

$$\begin{array}{lcl} \mathsf{sum\text{-}power4(zero)} & \rightarrow & \mathsf{zero} \\ \mathsf{sum\text{-}power4(suc(}x\mathsf{))} & \rightarrow & \mathsf{add(power4(suc(}x\mathsf{)),sum\text{-}power4(}x\mathsf{))} \end{array}$$

div: nat $\times$ nat $\rightarrow$ nat
mod: nat $\times$ nat $\rightarrow$ nat

$$\begin{array}{lcl} \mathsf{div(zero,}y\mathsf{)} & \rightarrow & \mathsf{zero} \\ \mathsf{div(suc(}x\mathsf{),zero)} & \rightarrow & \mathsf{zero} \\ \mathsf{div(suc(}x\mathsf{),suc(}y\mathsf{))} & \rightarrow & \mathsf{if\text{-}nat(}x \leq_{Nat} y\mathsf{,zero,suc(div(sub(}x,y\mathsf{),suc(}y\mathsf{))))} \end{array}$$

$$\begin{array}{lcl} \mathsf{mod(}x\mathsf{,zero)} & \rightarrow & x \\ \mathsf{mod(zero,suc(}y\mathsf{))} & \rightarrow & \mathsf{zero} \\ \mathsf{mod(suc(}x\mathsf{),suc(}y\mathsf{))} & \rightarrow & \mathsf{if\text{-}nat(}x \leq_{Nat} y\mathsf{,suc(}x\mathsf{),mod(sub(}x,y\mathsf{),suc(}y\mathsf{)))} \end{array}$$

Note that the definitions of the functions div and mod cannot proved to be terminating by UNICOM.

Theorem 1: 'Add is associative and commutative

$$\begin{array}{l} \mathsf{add(add(}x,y\mathsf{),}z\mathsf{)} \stackrel{?}{\equiv} \mathsf{add(}x\mathsf{,add(}y,z\mathsf{))} \\ \mathsf{add(}x,y\mathsf{)} \stackrel{?}{\equiv} \mathsf{add(}y,x\mathsf{)} \end{array}$$

Theorem 2: 'Half' – 'Double' – 'Add'.

$$\begin{array}{l} \mathsf{double(}x\mathsf{))} \stackrel{?}{\equiv} \mathsf{add(}x,x\mathsf{)} \\ \mathsf{half(double(}m\mathsf{))} \stackrel{?}{\equiv} m \\ \mathsf{half(add(add(}m,m\mathsf{),}n\mathsf{))} \stackrel{?}{\equiv} \mathsf{add(}m\mathsf{,half(}n\mathsf{))} \\ \mathsf{add(half(}y\mathsf{),half(}y\mathsf{))} \stackrel{?}{\equiv} \mathsf{ifn(even(}y\mathsf{),}y\mathsf{,dif(}y\mathsf{,one))} \end{array}$$

Theorem 3: 'Even' – 'Add', 'Even' – 'Mul'.

$$\begin{array}{l} \mathsf{even(add(}x,x\mathsf{))} \stackrel{?}{\equiv} \mathsf{true} \\ \mathsf{even(add(}x\mathsf{,add(}y,y\mathsf{)))} \stackrel{?}{\equiv} \mathsf{even(}x\mathsf{)} \\ \mathsf{even(add(}x\mathsf{,mul(}x,x\mathsf{)))} \stackrel{?}{\equiv} \mathsf{true} \end{array}$$

Theorem 4: 'Sub' – 'Add'.

$$\begin{array}{l} \mathsf{sub(}x,x\mathsf{)} \stackrel{?}{\equiv} \mathsf{zero} \\ \mathsf{sub(sub(}x,y\mathsf{),}z\mathsf{)} \stackrel{?}{\equiv} \mathsf{sub(sub(}x,z\mathsf{),}y\mathsf{)} \\ \mathsf{sub(add(}x,y\mathsf{),}x\mathsf{)} \stackrel{?}{\equiv} y \\ \mathsf{sub(}x\mathsf{,add(}x,y\mathsf{))} \stackrel{?}{\equiv} \mathsf{zero} \end{array}$$

Theorem 5: 'Mul' is associative and commutative.

$$\begin{array}{l} \mathsf{mul(}x,y\mathsf{)} \stackrel{?}{\equiv} \mathsf{mul(}y,x\mathsf{)} \\ \mathsf{mul(mul(}x,y\mathsf{),}z\mathsf{)} \stackrel{?}{\equiv} \mathsf{mul(mul(}x,z\mathsf{),}y\mathsf{)} \\ \mathsf{mul(mul(}x,y\mathsf{),}z\mathsf{)} \stackrel{?}{\equiv} \mathsf{mul(mul(}z,y\mathsf{),}x\mathsf{)} \\ \mathsf{mul(mul(}x,y\mathsf{),}z\mathsf{)} \stackrel{?}{\equiv} \mathsf{mul(mul(}y,x\mathsf{),}z\mathsf{)} \end{array}$$

uses 'add' is associative and commutative.

Theorem 6: 'Mul' distributes over 'Add'.

$$\mathsf{mul}(x,\mathsf{add}(y,z)) \;\stackrel{\rightarrow}{=}\; \mathsf{add}(\mathsf{mul}(x,y),\mathsf{mul}(x,z))$$

Theorem 7: Laws of Exponentiation

$$\begin{aligned}
&\mathsf{exp}(\mathsf{suc}(\mathsf{zero}),x) \;\stackrel{\rightarrow}{=}\; \mathsf{one}\\
&\mathsf{exp}(x,\mathsf{add}(y,z)) \;\stackrel{\rightarrow}{=}\; \mathsf{mul}(\mathsf{exp}(x,y),\mathsf{exp}(x,z))\\
&\mathsf{exp}(\mathsf{mul}(x,y),z) \;\stackrel{\rightarrow}{=}\; \mathsf{mul}(\mathsf{exp}(x,z),\mathsf{exp}(y,z))\\
&\mathsf{exp}(\mathsf{exp}(x,y),z) \;\stackrel{\rightleftarrows}{=}\; \mathsf{exp}(\mathsf{exp}(x,z),y)\\
&\mathsf{exp}(\mathsf{exp}(x,y),z) \;\stackrel{\rightarrow}{=}\; \mathsf{exp}(x,\mathsf{mul}(y,z))
\end{aligned}$$

Theorem 8: 'Mod' – 'Div'.

$$\mathsf{add}(\mathsf{mod}(x,y),\mathsf{mul}(\mathsf{div}(x,y),y)) \;\stackrel{\rightarrow}{=}\; x$$

Theorem 9: Sum Formulas

$$\begin{aligned}
&\mathsf{sum}(n) \;\stackrel{\rightarrow}{=}\; \mathsf{half}(\mathsf{mul}(\mathsf{suc}(n),n))\\
&\mathsf{sum\text{-}odd}(n) \;\stackrel{\rightarrow}{=}\; \mathsf{mul}(\mathsf{suc}(n),\mathsf{suc}(n))\\
&\mathsf{sum\text{-}even}(n) \;\stackrel{\rightarrow}{=}\; \mathsf{mul}(\mathsf{two},\mathsf{sum}(n))\\
&\mathsf{add}(\mathsf{sum\text{-}odd}(x),\mathsf{sum\text{-}even}(x)) \;\stackrel{\rightarrow}{=}\; \mathsf{sum}(\mathsf{suc}(2\ x))
\end{aligned}$$

uses theorems (2),(3)

Theorem 10: Binomial Formulas

$$\begin{aligned}
&\mathsf{square}(\mathsf{add}(x,y)) \;\stackrel{\rightarrow}{=}\; \mathsf{add}(\mathsf{square}(x),\mathsf{square}(y),2\ \mathsf{mul}(x,y))\\
&\mathsf{cube}(\mathsf{add}(x,y)) \;\stackrel{\rightarrow}{=}\; \mathsf{add}(\mathsf{cube}(x),\mathsf{cube}(y),\ 3\ \mathsf{mul}(\mathsf{square}(x),y),\ 3\ \mathsf{mul}(x,\mathsf{square}(y)))\\
&\mathsf{power4}(\mathsf{add}(x,y)) \;\stackrel{\rightarrow}{=}\; \mathsf{add}(\mathsf{power4}(x),4\ \mathsf{mul}(\mathsf{cube}(x),y),\mathsf{power4}(y),\\
&\qquad\qquad 6\ \mathsf{mul}(\mathsf{square}(x),\mathsf{square}(y)),\ 6\ \mathsf{mul}(x,\mathsf{cube}(y)))
\end{aligned}$$

uses theorem (6)

Theorem 11: Sum Formulas -2-.

$$\begin{aligned}
&\mathsf{mul}(\mathsf{six},\mathsf{sum\text{-}square}(x)) \;\stackrel{\rightarrow}{=}\; \mathsf{mul}(\mathsf{mul}(x,\mathsf{suc}(x)),\mathsf{suc}(\mathsf{mul}(\mathsf{two},x)))\\
&\mathsf{mul}(\mathsf{four},\mathsf{sum\text{-}cube}(x)) \;\stackrel{\rightarrow}{=}\; \mathsf{mul}(\mathsf{square}(x),\mathsf{square}(\mathsf{suc}(x)))\\
&\mathsf{add}(x,30\ \mathsf{sum\text{-}power4}(x)) \;\stackrel{\rightarrow}{=}\; \mathsf{mul}(\mathsf{cube}(x),\mathsf{add}(6\ \mathsf{square}(x),15\ x,10))
\end{aligned}$$

Example II: Sorting Algorithms

This example deals with the correctness properties of five sorting algorithms: sorting by insertion, minimum-sort, merge-sort, bubble-sort and quick-sort. A more detailed discussion of this example can be found in [Gra90b].

Sorts:

list:	empty:		$\rightarrow$ list
	cons:	nat $\times$ list	$\rightarrow$ list

Function Definitions:

nat≤list: nat $\times$ list $\rightarrow$ bool
list≤nat: list $\times$ nat $\rightarrow$ bool
list≤list: list $\times$ list $\rightarrow$ bool

$$\mathsf{nat{\le}list}(x,\mathsf{empty}) \rightarrow \mathsf{true}$$
$$\mathsf{nat{\le}list}(x,\mathsf{cons}(y,l)) \rightarrow x \le_{Nat} y \wedge \mathsf{nat{\le}list}(x,l)$$

$$\mathsf{list{\le}nat}(\mathsf{empty},x) \rightarrow \mathsf{true}$$
$$\mathsf{list{\le}nat}(\mathsf{cons}(x,l),y) \rightarrow x \le_{Nat} y \wedge \mathsf{list{\le}nat}(l,y)$$

$$\mathsf{list{\le}list}(\mathsf{empty},l) \rightarrow \mathsf{true}$$
$$\mathsf{list{\le}list}(\mathsf{cons}(x,l_1),l_2) \rightarrow \mathsf{nat{\le}list}(x,l_2) \wedge \mathsf{list{\le}list}(l_1,l_2)$$

append: list $\times$ list $\rightarrow$ list

$$\mathsf{append}(\mathsf{empty},l) \rightarrow l$$
$$\mathsf{append}(\mathsf{cons}(x,l_1),l_2) \rightarrow \mathsf{cons}(x,\mathsf{append}(l_1,l_2))$$

member: nat $\times$ list $\rightarrow$ bool
delete: nat $\times$ list $\rightarrow$ list

$$\mathsf{member}(x,\mathsf{empty}) \rightarrow \mathsf{false}$$
$$\mathsf{member}(x,\mathsf{cons}(y,l)) \rightarrow x =_{Nat} y \vee \mathsf{member}(x,l)$$

$$\mathsf{delete}(x,\mathsf{empty}) \rightarrow \mathsf{empty}$$
$$\mathsf{delete}(x,\mathsf{cons}(y,l)) \rightarrow \mathsf{if\text{-}list}(x =_{Nat} y,l,\mathsf{cons}(y,\mathsf{delete}(x,l)))$$

butlast: list $\rightarrow$ list
last: list $\rightarrow$ nat

$$\mathsf{butlast}(\mathsf{empty}) \rightarrow \mathsf{empty}$$
$$\mathsf{butlast}(\mathsf{cons}(n,\mathsf{empty})) \rightarrow \mathsf{empty}$$
$$\mathsf{butlast}(\mathsf{cons}(n,\mathsf{cons}(m,l))) \rightarrow \mathsf{cons}(n,\mathsf{butlast}(\mathsf{cons}(m,l)))$$

$$\mathsf{last}(\mathsf{empty}) \rightarrow \mathsf{zero}$$
$$\mathsf{last}(\mathsf{cons}(n,\mathsf{empty})) \rightarrow n$$
$$\mathsf{last}(\mathsf{cons}(n,\mathsf{cons}(m,l))) \rightarrow \mathsf{last}(\mathsf{cons}(m,l))$$

ordered: list $\rightarrow$ bool
permp: list $\times$ list $\rightarrow$ bool
count: nat $\times$ list $\rightarrow$ nat

$$\mathsf{ordered}(\mathsf{empty}) \rightarrow \mathsf{true}$$
$$\mathsf{ordered}(\mathsf{cons}(x,l)) \rightarrow \mathsf{nat{\le}list}(x,l) \wedge \mathsf{ordered}(l)$$

$$\begin{array}{lcl}
\mathsf{permp}(\mathsf{empty},\mathsf{empty}) & \rightarrow & \mathsf{true} \\
\mathsf{permp}(\mathsf{empty},\mathsf{cons}(x,l)) & \rightarrow & \mathsf{false} \\
\mathsf{permp}(\mathsf{cons}(x,l_1),l_2) & \rightarrow & \mathsf{member}(x,l_2) \wedge \mathsf{permp}(l_1,\mathsf{delete}(x,l_2))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{count}(x,\mathsf{empty}) & \rightarrow & \mathsf{zero} \\
\mathsf{count}(x,\mathsf{cons}(y,l)) & \rightarrow & \mathsf{if\text{-}nat}(x =_{Nat} y,\mathsf{suc}(\mathsf{count}(x,l)),\mathsf{count}(x,l))
\end{array}$$

insert: nat $\times$ list $\rightarrow$ list
ins-sort: list $\rightarrow$ list

$$\begin{array}{lcl}
\mathsf{insert}(x,\mathsf{empty}) & \rightarrow & \mathsf{cons}(x,\mathsf{empty}) \\
\mathsf{insert}(x,\mathsf{cons}(y,l)) & \rightarrow & \mathsf{if\text{-}list}(x \leq_{Nat} y,\mathsf{cons}(x,\mathsf{cons}(y,l)),\mathsf{cons}(y,\mathsf{insert}(x,l)))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{ins\text{-}sort}(\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{ins\text{-}sort}(\mathsf{cons}(x,l)) & \rightarrow & \mathsf{insert}(x,\mathsf{ins\text{-}sort}(l))
\end{array}$$

min-list: list $\rightarrow$ nat
min-sort: list $\rightarrow$ list

$$\begin{array}{lcl}
\mathsf{min\text{-}list}(\mathsf{empty}) & \rightarrow & \mathsf{zero} \\
\mathsf{min\text{-}list}(\mathsf{cons}(n,\mathsf{empty})) & \rightarrow & n \\
\mathsf{min\text{-}list}(\mathsf{cons}(n,\mathsf{cons}(m,l))) & \rightarrow & \mathsf{min}(n,\mathsf{min\text{-}list}(\mathsf{cons}(m,l)))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{min\text{-}sort}(\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{min\text{-}sort}(\mathsf{cons}(n,l)) & \rightarrow & \mathsf{cons}(\mathsf{min\text{-}list}(\mathsf{cons}(n,l)),\mathsf{min\text{-}sort}(\mathsf{delete}(\mathsf{min\text{-}list}(\mathsf{cons}(n,l)),\mathsf{cons}(n,l))))
\end{array}$$

merge: list $\times$ list $\rightarrow$ list
split1: list $\rightarrow$ list
split2: list $\rightarrow$ list
merge-sort: list $\rightarrow$ list

$$\begin{array}{lcl}
\mathsf{merge}(\mathsf{empty},l) & \rightarrow & l \\
\mathsf{merge}(l,\mathsf{empty}) & \rightarrow & l \\
\mathsf{merge}(\mathsf{cons}(x,l_1),\mathsf{cons}(y,l_2)) & \rightarrow & \mathsf{if\text{-}list}(x \leq_{Nat} y, \\
& & \quad \mathsf{cons}(x,\mathsf{merge}(l_1,\mathsf{cons}(y,l_2))),\mathsf{cons}(y,\mathsf{merge}(\mathsf{cons}(x,l_1),l_2)))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{split1}(\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{split1}(\mathsf{cons}(x,\mathsf{empty})) & \rightarrow & \mathsf{cons}(x,\mathsf{empty}) \\
\mathsf{split1}(\mathsf{cons}(x,\mathsf{cons}(y,l))) & \rightarrow & \mathsf{cons}(x,\mathsf{split1}(l))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{split2}(\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{split2}(\mathsf{cons}(x,\mathsf{empty})) & \rightarrow & \mathsf{empty} \\
\mathsf{split2}(\mathsf{cons}(x,\mathsf{cons}(y,l))) & \rightarrow & \mathsf{cons}(y,\mathsf{split2}(l))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{merge\text{-}sort}(\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{merge\text{-}sort}(\mathsf{cons}(x,\mathsf{empty})) & \rightarrow & \mathsf{cons}(x,\mathsf{empty}) \\
\mathsf{merge\text{-}sort}(\mathsf{cons}(x,\mathsf{cons}(y,l))) & \rightarrow & \mathsf{merge}(\mathsf{merge\text{-}sort}(\mathsf{split1}(\mathsf{cons}(x,\mathsf{cons}(y,l)))), \\
& & \qquad \mathsf{merge\text{-}sort}(\mathsf{split2}(\mathsf{cons}(x,\mathsf{cons}(y,l)))))
\end{array}$$

bubble: list $\rightarrow$ list
bubble-sort: list $\rightarrow$ list

$$\begin{array}{lcl}
\mathsf{bubble}(\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{bubble}(\mathsf{cons}(x,\mathsf{empty})) & \rightarrow & \mathsf{cons}(x,\mathsf{empty}) \\
\mathsf{bubble}(\mathsf{cons}(x,\mathsf{cons}(y,l))) & \rightarrow & \mathsf{if\text{-}list}(x \leq_{Nat} y, \\
& & \quad \mathsf{cons}(y,\mathsf{bubble}(\mathsf{cons}(x,l))),\mathsf{cons}(x,\mathsf{bubble}(\mathsf{cons}(y,l))))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{bubble\text{-}sort}(\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{bubble\text{-}sort}(\mathsf{cons}(x,l)) & \rightarrow & \mathsf{cons}(\mathsf{last}(\mathsf{bubble}(\mathsf{cons}(x,l))),\mathsf{bubble\text{-}sort}(\mathsf{butlast}(\mathsf{bubble}(\mathsf{cons}(x,l)))))
\end{array}$$

lowers: nat $\times$ list $\rightarrow$ list
greaters: nat $\times$ list $\rightarrow$ list
quick-sort: list $\rightarrow$ list

$$
\begin{array}{lcl}
\mathsf{lowers}(x,\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{lowers}(x,\mathsf{cons}(y,l)) & \rightarrow & \mathsf{if\text{-}list}(y \leq_{Nat} x,\mathsf{cons}(y,\mathsf{lowers}(x,l)),\mathsf{lowers}(x,l))
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{greaters}(x,\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{greaters}(x,\mathsf{cons}(y,l)) & \rightarrow & \mathsf{if\text{-}list}(y \leq_{Nat} x,\mathsf{greaters}(x,l),\mathsf{cons}(y,\mathsf{greaters}(x,l)))
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{quick\text{-}sort}(\mathsf{empty}) & \rightarrow & \mathsf{empty} \\
\mathsf{quick\text{-}sort}(\mathsf{cons}(x,l)) & \rightarrow & \mathsf{append}(\mathsf{quick\text{-}sort}(\mathsf{lowers}(x,l)),\mathsf{cons}(x,\mathsf{quick\text{-}sort}(\mathsf{greaters}(x,l))))
\end{array}
$$

Note that the definitions of the functions 'min-sort', 'merge-sort', 'bubble-sort' and 'quick-sort' cannot be proved to be terminating by UNICOM.

Theorem 1a: 'Ins-Sort' produces ordered lists

$$\mathsf{ordered}(\mathsf{ins\text{-}sort}(x)) \stackrel{\circ}{\equiv} \mathsf{true}$$

(1a.1) $\mathsf{nat{\leq}list}(n,\mathsf{insert}(m,l)) = n \leq_{Nat} m \wedge \mathsf{nat{\leq}list}(n,l)$

(1a.2) $\mathsf{ordered}(\mathsf{insert}(n,x)) \stackrel{\circ}{\equiv} \mathsf{ordered}(x)$

Theorem 1b: 'Ins-Sort' permutes the list

$$\mathsf{permp}(l,\mathsf{ins\text{-}sort}(l)) \stackrel{\circ}{\equiv} \mathsf{true}$$

(1b.1) $\mathsf{delete}(n,\mathsf{insert}(n,l)) \stackrel{\circ}{\equiv} l$

(1b.2) $\mathsf{member}(n,\mathsf{insert}(n,l)) \stackrel{\circ}{\equiv} \mathsf{true}$

Theorem 1c: 'Ins-Sort' permutes the list (second version)

$$\mathsf{count}(n,\mathsf{ins\text{-}sort}(x)) \stackrel{\circ}{\equiv} \mathsf{count}(n,x)$$

(1c.1) $\mathsf{count}(n,\mathsf{insert}(m,l)) \stackrel{\circ}{\equiv} \mathsf{count}(n,\mathsf{cons}(m,l))$

Theorem 2a: 'Min-Sort' produces ordered lists

$$\mathsf{ordered}(\mathsf{min\text{-}sort}(l)) \stackrel{\circ}{\equiv} \mathsf{true}$$

(2a.1) $\mathsf{nat{\leq}list}(n,\mathsf{min\text{-}sort}(l)) \stackrel{\circ}{\equiv} \mathsf{nat{\leq}list}(n,l)$

  (2a.1.1) $n \leq_{Nat} m \wedge \mathsf{nat{\leq}list}(n,\mathsf{delete}(m,l)) \stackrel{\circ}{\equiv} n \leq_{Nat} m \wedge \mathsf{nat{\leq}list}(n,l)$

(2a.2) $\mathsf{nat{\leq}list}(n,\mathsf{delete}(n,l)) \stackrel{\circ}{\equiv} \mathsf{nat{\leq}list}(n,l)$

(2a.3) $\mathsf{min\text{-}list}(\mathsf{cons}(n,l)) \leq_{Nat} n \stackrel{\circ}{\equiv} \mathsf{true}$

(2a.4) $\mathsf{nat{\leq}list}(\mathsf{min\text{-}list}(\mathsf{cons}(n,l)),l) \stackrel{\circ}{\equiv} \mathsf{true}$

  (2a.4.1) $\mathsf{nat{\leq}list}(\mathsf{min}(n,m),z) \stackrel{\circ}{\equiv} \mathsf{nat{\leq}list}(n,z) \vee \mathsf{nat{\leq}list}(m,z)$

(2a.5) $n \leq_{Nat} \mathsf{min\text{-}list}(\mathsf{cons}(m,l)) \stackrel{\circ}{\equiv} \mathsf{nat{\leq}list}(n,\mathsf{cons}(m,l))$

Theorem 2b: 'Min-Sort' permutes the list

$$\mathsf{permp}(\mathsf{min\text{-}sort}(l),l) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{true}$$

(2b.1) $\mathsf{min}(x,y) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{ifn}(\mathsf{nat}{\leq}(x,y),x,y)$

Theorem 2c: 'Min-Sort' permutes the list (second version)

$$\mathsf{count}(n,\mathsf{min\text{-}sort}(l)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{count}(n,l)$$

Theorem 3a: 'Merge-Sort' produces ordered lists

$$\mathsf{ordered}(\mathsf{merge\text{-}sort}(x)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{true}$$

(3a.1) $\mathsf{ordered}(\mathsf{merge}(x,y)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{ordered}(x) \wedge \mathsf{ordered}(y)$

(3a.1.1) $\mathsf{nat{\leq}list}(z,\mathsf{merge}(x,y)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{nat{\leq}list}(z,x) \wedge \mathsf{nat{\leq}list}(z,y)$

(3a.1.2) $\mathsf{nat{\leq}list}(z,l) \wedge (z =_{Nat} u \rightarrow \mathsf{nat{\leq}list}(u,l)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{nat{\leq}list}(z,l)$

(3a.1.3) $\mathsf{nat{\leq}list}(x,l) \wedge (\ y <_{Nat} x \rightarrow \mathsf{nat{\leq}list}(y,l)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{nat{\leq}list}(x,l)$

Theorem 3c: 'Merge-Sort' permutes the list (second version)

$$\mathsf{count}(m,\mathsf{merge\text{-}sort}(x)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{count}(m,x)$$

(3c.1) $\mathsf{count}(m,\mathsf{merge}(x,y)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{add}(\mathsf{count}(m,x),\mathsf{count}(m,y))$

(3c.2) $\mathsf{add}(\mathsf{count}(m,\mathsf{split1}(x)),\mathsf{count}(m,\mathsf{split2}(x))) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{count}(m,x)$

Theorem 4a: 'Bubble-Sort' produces ordered lists

$$\mathsf{ordered}(\mathsf{bubble\text{-}sort}(l)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{true}$$

uses 2a.3-4

(4a.1) $\mathsf{nat{\leq}list}(n,\mathsf{bubble\text{-}sort}(l)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{nat{\leq}list}(n,l)$

(4a.1.1) $n \leq_{Nat} \mathsf{last}(l) \wedge \mathsf{nat{\leq}list}(n,\mathsf{butlast}(l)) \ \stackrel{\rightarrow}{\equiv}$
$\mathsf{if\text{-}bool}(l =_{List} \mathsf{empty}, n =_{Nat} \mathsf{zero},\mathsf{nat{\leq}list}(n,l))$

(4a.1.2) $\mathsf{bubble}(\mathsf{cons}(x,y)) =_{List} \mathsf{empty} \ \stackrel{\rightarrow}{\equiv}\ \mathsf{false}$

(4a.2) $\mathsf{nat{\leq}list}(\mathsf{last}(l),\mathsf{butlast}(l)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{nat{\leq}list}(\mathsf{last}(l),l)$

(4a.3) $\mathsf{nat{\leq}list}(n,\mathsf{bubble}(l)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{nat{\leq}list}(n,l)$

(4a.4) $\mathsf{last}(\mathsf{bubble}(\mathsf{cons}(n,l))) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{min\text{-}list}(\mathsf{cons}(n,l))$

(4a.4.1) $\mathsf{last}(\mathsf{cons}(x,y)) \ \stackrel{\rightarrow}{\equiv}\ \mathsf{if\text{-}natn}(y =_{List} \mathsf{empty},x,\mathsf{last}(y))$

Note: User driven reduction is needed here to delay the application of lemma (4a.4).

Theorem 4c: 'Bubble-Sort' permutes the list (second version)

$$\mathsf{count}(n,\mathsf{bubble\text{-}sort}(l)) \ \stackrel{\rightarrow}{=}\ \mathsf{count}(n,l)$$

(4c.1) $\mathsf{count}(n,\mathsf{butlast}(l)) \ \stackrel{\rightarrow}{=}\ \mathsf{if\text{-}nat}(n =_{Nat} \mathsf{last}(l),\mathsf{pred}(\mathsf{count}(n,l)),\mathsf{count}(n,l))$

(4c.2) $\mathsf{count}(n,\mathsf{bubble}(l)) \ \stackrel{\rightarrow}{=}\ \mathsf{count}(n,l)$

(4c.3) $\mathsf{count}(n,y) =_{Nat} \mathsf{zero} \wedge n =_{Nat} \mathsf{last}(\mathsf{bubble}(\mathsf{cons}(x,y))) \wedge n \neq_{Nat} x \ \stackrel{\rightarrow}{=}\ \mathsf{false}$

Theorem 5a: 'Quick-Sort' produces ordered lists

$$\mathsf{ordered}(\mathsf{quick\text{-}sort}(l)) \ \stackrel{\rightarrow}{=}\ \mathsf{true}$$

(5a.1) $\mathsf{ordered}(\mathsf{append}(l_1,l_2)) \ \stackrel{\rightarrow}{=}\ \mathsf{ordered}(l_1) \wedge \mathsf{ordered}(l_2) \wedge \mathsf{list{\le}list}(l_1,l_2)$

(5a.1.1) $\mathsf{nat{\le}list}(x,\mathsf{append}(y,z)) \ \stackrel{\rightarrow}{=}\ \mathsf{nat{\le}list}(x,y) \wedge \mathsf{nat{\le}list}(x,z)$

(5a.2) $\mathsf{nat{\le}list}(n,\mathsf{quick\text{-}sort}(l)) \ \stackrel{\rightarrow}{=}\ \mathsf{nat{\le}list}(n,l)$

uses 5a.1.1

(5a.2.1) $\mathsf{list{\le}list}(\mathsf{lowers}(m,l),l_1) \wedge \mathsf{list{\le}list}(\mathsf{greaters}(m,l),l_1) \ \stackrel{\rightarrow}{=}\ \mathsf{list{\le}list}(l,l_1)$

(5a.3) $\mathsf{list{\le}list}(x,\mathsf{cons}(n,y)) \ \stackrel{\rightarrow}{=}\ \mathsf{list{\le}nat}(x,n) \wedge \mathsf{list{\le}list}(x,y)$

(5a.4) $\mathsf{list{\le}list}(l,\mathsf{quick\text{-}sort}(l_1)) \ \stackrel{\rightarrow}{=}\ \mathsf{list{\le}list}(l,l_1)$

uses 5a.3

(5a.4.1) $\mathsf{list{\le}list}(l,\mathsf{append}(x,y)) \ \stackrel{\rightarrow}{=}\ \mathsf{list{\le}list}(l,x) \wedge \mathsf{list{\le}list}(l,y)$

(5a.4.2) $\mathsf{list{\le}list}(l_1,\mathsf{lowers}(m,l)) \wedge \mathsf{list{\le}list}(l_1,\mathsf{greaters}(m,l)) \ \stackrel{\rightarrow}{=}\ \mathsf{list{\le}list}(l_1,l)$

(5a.5) $\mathsf{list{\le}list}(\mathsf{quick\text{-}sort}(l_1),l_2) \ \stackrel{\rightarrow}{=}\ \mathsf{list{\le}list}(l_1,l_2)$

(5a.5.1) $\mathsf{list{\le}list}(\mathsf{append}(x,y),l) \ \stackrel{\rightarrow}{=}\ \mathsf{list{\le}list}(x,l) \wedge \mathsf{list{\le}list}(y,l)$

(5a.5.2) $\mathsf{list{\le}list}(\mathsf{lowers}(m,l),l_1) \wedge \mathsf{list{\le}list}(\mathsf{greaters}(m,l),l_1) \ \stackrel{\rightarrow}{=}\ \mathsf{list{\le}list}(l,l_1)$

(5a.6) $\mathsf{list{\le}nat}(\mathsf{lowers}(x,l),x) \ \stackrel{\rightarrow}{=}\ \mathsf{true}$

(5a.7) $\mathsf{nat{\le}list}(x,\mathsf{greaters}(x,l)) \ \stackrel{\rightarrow}{=}\ \mathsf{true}$

(5a.8) $\mathsf{list{\le}list}(\mathsf{lowers}(x,y),\mathsf{greaters}(x,y)) \ \stackrel{\rightarrow}{=}\ \mathsf{true}$

uses 5a.3

(5a.8.1) $u <_{Nat} z \rightarrow \mathsf{list{\le}nat}(\mathsf{lowers}(u,l),z) \ \stackrel{\rightarrow}{=}\ \mathsf{true}$

(5a.8.2a) $z <_{Nat} u \rightarrow \mathsf{nat{\le}list}(z,\mathsf{greaters}(u,l)) \ \stackrel{\rightarrow}{=}\ \mathsf{true}$

(5a.8.2b) $z =_{Nat} u \rightarrow \mathsf{nat{\le}list}(z,\mathsf{greaters}(u,l)) \ \stackrel{\rightarrow}{=}\ \mathsf{true}$

Theorem 5c: 'Quick-Sort' permutes the list (second version)

$$\mathsf{count}(n,\mathsf{quick\text{-}sort}(l)) \ \stackrel{\rightarrow}{=}\ \mathsf{count}(n,l)$$

(5c.1) $\mathsf{count}(n,\mathsf{append}(x,y)) \ \stackrel{\rightarrow}{=}\ \mathsf{add}(\mathsf{count}(n,x),\mathsf{count}(n,y))$

(5c.2) $\mathsf{add}(\mathsf{count}(n,\mathsf{lowers}(m,l)),\mathsf{count}(n,\mathsf{greaters}(m,l))) \ \stackrel{\rightarrow}{=}\ \mathsf{count}(n,l)$

Example III: Binary Search Trees

This example deals with sorted binary trees of natural numbers, i.e. all numbers in the left-hand child of a node are less or equal to the node's number, and all numbers in the right-hand child are greater or equal to the node's number. The theorems state properties of an insert operation and of two different delete operations.

Sorts:

bintree:	nil-bt:		$\rightarrow$ bintree
	nd:	bintree $\times$ bintree $\times$ nat	$\rightarrow$ bintree

Function Definitions:

greater-all: nat $\times$ bintree $\rightarrow$ bool
less-all: nat $\times$ bintree $\rightarrow$ bool
sorted: bintree $\rightarrow$ bool

$$\begin{array}{lcl} \mathsf{greater\text{-}all}(x,\mathsf{nil\text{-}bt}) & \rightarrow & \mathsf{true} \\ \mathsf{greater\text{-}all}(x,\mathsf{nd}(u,v,w)) & \rightarrow & x \geq_{Nat} w \wedge \mathsf{greater\text{-}all}(x,u) \wedge \mathsf{greater\text{-}all}(x,v) \end{array}$$

$$\begin{array}{lcl} \mathsf{less\text{-}all}(x,\mathsf{nil\text{-}bt}) & \rightarrow & \mathsf{true} \\ \mathsf{less\text{-}all}(x,\mathsf{nd}(u,v,w)) & \rightarrow & x \leq_{Nat} w \wedge \mathsf{less\text{-}all}(x,u) \wedge \mathsf{less\text{-}all}(x,v) \end{array}$$

$$\begin{array}{lcl} \mathsf{sorted}(\mathsf{nil\text{-}bt}) & \rightarrow & \mathsf{true} \\ \mathsf{sorted}(\mathsf{nd}(u,v,w)) & \rightarrow & \mathsf{greater\text{-}all}(w,u) \wedge \mathsf{less\text{-}all}(w,v) \wedge \mathsf{sorted}(u) \wedge \mathsf{sorted}(v) \end{array}$$

contains: bintree $\times$ nat $\rightarrow$ bool
search: bintree $\times$ nat $\rightarrow$ bool

$$\begin{array}{lcl} \mathsf{contains}(\mathsf{nil\text{-}bt},x) & \rightarrow & \mathsf{false} \\ \mathsf{contains}(\mathsf{nd}(u,v,w),x) & \rightarrow & w =_{Nat} x \vee \mathsf{contains}(u,x) \vee \mathsf{contains}(v,x) \end{array}$$

$$\begin{array}{lcl} \mathsf{search}(\mathsf{nil\text{-}bt},x) & \rightarrow & \mathsf{false} \\ \mathsf{search}(\mathsf{nd}(u,v,w),x) & \rightarrow & \mathsf{if\text{-}bool}(w =_{Nat} x,\mathsf{true},\mathsf{if\text{-}bool}(w >_{Nat} x,\mathsf{search}(u,x),\mathsf{search}(v,x)) \end{array}$$

insert: bintree $\times$ nat $\rightarrow$ bintree

$$\begin{array}{lcl} \mathsf{insert}(\mathsf{nil\text{-}bt},x) & \rightarrow & \mathsf{nd}(\mathsf{nil\text{-}bt},\mathsf{nil\text{-}bt},x) \\ \mathsf{insert}(\mathsf{nd}(u,v,w),x) & \rightarrow & \mathsf{if\text{-}bt}(w =_{Nat} x,\mathsf{nd}(u,v,w), \\ & & \quad \mathsf{if\text{-}bt}(w >_{Nat} x,\mathsf{nd}(\mathsf{insert}(u,x),v,w),\mathsf{nd}(u,\mathsf{insert}(v,x),w))) \end{array}$$

combine: bintree $\times$ bintree $\rightarrow$ bintree
delete: bintree $\times$ nat $\rightarrow$ bintree

$$\begin{array}{lcl} \mathsf{combine}(x,\mathsf{nil\text{-}bt}) & \rightarrow & x \\ \mathsf{combine}(x,\mathsf{nd}(u,v,w)) & \rightarrow & \mathsf{nd}(\mathsf{combine}(x,u),v,w) \end{array}$$

$$\begin{array}{lcl} \mathsf{delete}(\mathsf{nil\text{-}bt},x) & \rightarrow & \mathsf{nil\text{-}bt} \\ \mathsf{delete}(\mathsf{nd}(u,v,w),x) & \rightarrow & \mathsf{if\text{-}bt}(w =_{Nat} x,\mathsf{combine}(u,v), \\ & & \quad \mathsf{if\text{-}bt}(w >_{Nat} x,\mathsf{nd}(\mathsf{delete}(u,x),v,w),\mathsf{nd}(u,\mathsf{delete}(v,x),w))) \end{array}$$

leftmost: bintree $\rightarrow$ nat
delete-leftmost: bintree $\rightarrow$ bintree
delete1: bintree $\times$ nat $\rightarrow$ bintree

$$\begin{array}{lcl} \mathsf{leftmost}(\mathsf{nil\text{-}bt}) & \rightarrow & \mathsf{zero} \\ \mathsf{leftmost}(\mathsf{nd}(\mathsf{nil\text{-}bt},x,y)) & \rightarrow & y \\ \mathsf{leftmost}(\mathsf{nd}(\mathsf{nd}(u,v,w),x,y)) & \rightarrow & \mathsf{leftmost}(\mathsf{nd}(u,v,w)) \end{array}$$

$$
\begin{array}{lcl}
\mathsf{delete\text{-}leftmost}(\mathsf{nil\text{-}bt}) & \rightarrow & \mathsf{nil\text{-}bt} \\
\mathsf{delete\text{-}leftmost}(\mathsf{nd}(\mathsf{nil\text{-}bt},x,y)) & \rightarrow & x \\
\mathsf{delete\text{-}leftmost}(\mathsf{nd}(\mathsf{nd}(u,v,w),x,y)) & \rightarrow & \mathsf{nd}(\mathsf{delete\text{-}leftmost}(\mathsf{nd}(u,v,w)),x,y)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{delete1}(\mathsf{nil\text{-}bt},x) & \rightarrow & \mathsf{nil\text{-}bt} \\
\mathsf{delete1}(\mathsf{nd}(u,\mathsf{nil\text{-}bt},w),x) & \rightarrow & \mathsf{if\text{-}bt}(w =_{Nat} x, u, \mathsf{nd}(\mathsf{delete1}(u,x),\mathsf{nil\text{-}bt},w)) \\
\mathsf{delete1}(\mathsf{nd}(x,\mathsf{nd}(u,v,w),y),z) & \rightarrow & \mathsf{if\text{-}bt}(y =_{Nat} z, \mathsf{nd}(x,\mathsf{delete1}(\mathsf{nd}(u,v,w)),\mathsf{leftmost}(\mathsf{nd}(u,v,w))), \\
 & & \quad \mathsf{if\text{-}bt}(y >_{Nat} z, \\
 & & \quad\quad \mathsf{nd}(\mathsf{delete1}(x,z),\mathsf{nd}(u,v,w),y), \\
 & & \quad\quad \mathsf{nd}(x,\mathsf{delete1}(\mathsf{nd}(u,v,w),z),y)))
\end{array}
$$

height: bintree → nat
count: bintree × nat → nat

$$
\begin{array}{lcl}
\mathsf{height}(\mathsf{nil\text{-}bt}) & \rightarrow & \mathsf{zero} \\
\mathsf{height}(\mathsf{nd}(u,v,w)) & \rightarrow & \mathsf{suc}(\mathsf{max}(\mathsf{height}(u),\mathsf{height}(v)))
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{count}(\mathsf{nil\text{-}bt},x) & \rightarrow & \mathsf{zero} \\
\mathsf{count}(\mathsf{nd}(u,v,w),x) & \rightarrow & \mathsf{if\text{-}nat}(w =_{Nat} x, \\
 & & \quad \mathsf{suc}(\mathsf{add}(\mathsf{count}(u,x),\mathsf{count}(v,x))), \\
 & & \quad \mathsf{add}(\mathsf{count}(u,x),\mathsf{count}(v,x)))
\end{array}
$$

Theorem 1a: 'Insert' increases height

$$\mathsf{height}(\mathsf{insert}(x,\mathsf{a})) \geq_{Nat} \mathsf{height}(x) \;\; \overset{\rightarrow}{=} \;\; \mathsf{true}$$

(1a.1) $x \geq_{Nat} x \;\; \overset{\rightarrow}{=} \;\; \mathsf{true}$

(1a.2) $x \geq_{Nat} \mathsf{max}(y,z) \;\; \overset{\rightarrow}{=} \;\; x \geq_{Nat} y \wedge x \geq_{Nat} z$

(1a.3) $\mathsf{max}(x,y) \geq_{Nat} z \;\; \overset{\rightarrow}{=} \;\; x \geq_{Nat} z \vee y \geq_{Nat} z$

The order of reduction steps is crucial. The subterms which may be 'deleted' using the induction hypothesis make it possible to 'delete' other subterms which involve the induction hypothesis in some sense but on which the induction hypothesis cannot be applied directly. Thus, the induction hypothesis has to be applied *after* simplifying the term using lemmas.

Theorem 1b: 'Insert' increases height at most by one

$$\mathsf{suc}(\mathsf{height}(x)) \geq_{Nat} \mathsf{height}(\mathsf{insert}(x,z)) \;\; \overset{\rightarrow}{=} \;\; \mathsf{true}$$

uses 1a.1-3

(1b.1) $x \geq_{Nat} \mathsf{sub}(x,y) \;\; \overset{\rightarrow}{=} \;\; \mathsf{true}$

Theorem 2: 'Insert' inserts a number

$$
\begin{array}{l}
\mathsf{contains}(\mathsf{insert}(x,y),y) \;\; \overset{\rightarrow}{=} \;\; \mathsf{true} \\
\mathsf{contains}(x,y) \rightarrow \mathsf{contains}(\mathsf{insert}(x,z),y) \;\; \overset{\rightarrow}{=} \;\; \mathsf{true} \\
\neg\, \mathsf{contains}(x,y) \wedge y \neq_{Nat} z \rightarrow \neg\, \mathsf{contains}(\mathsf{insert}(x,z),y) \;\; \overset{\rightarrow}{=} \;\; \mathsf{true}
\end{array}
$$

(2.1) $x =_{Nat} x \;\; \overset{\rightarrow}{=} \;\; \mathsf{true}$

Theorem 3a: 'Insert' does not change the number of occurences of other elements

$$y \neq_{Nat} z \rightarrow \mathsf{count}(\mathsf{insert}(x,y),z) =_{Nat} \mathsf{count}(x,z) \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{true}$$

(3a.1) $\mathsf{add}(x,y) =_{Nat} \mathsf{add}(x,z) \;\; \overset{\rightarrow}{\equiv} \;\; y =_{Nat} z$

(3a.2) $\mathsf{add}(x,y) =_{Nat} \mathsf{zero} \;\; \overset{\rightarrow}{\equiv} \;\; x =_{Nat} \mathsf{zero} \wedge y =_{Nat} \mathsf{zero}$

Theorem 3b: 'Insert' inserts one occurrence

$$\neg\, \mathsf{contains}(x,y) \rightarrow \mathsf{count}(\mathsf{insert}(x,y),y) =_{Nat} \mathsf{one} \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{true}$$

(3b.1) $\neg\, \mathsf{contains}(x,y) \rightarrow \mathsf{pred}(\mathsf{count}(\mathsf{insert}(x,y),y)) =_{Nat} \mathsf{zero} \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{true}$

(3b.2) $\neg\, \mathsf{contains}(x,y) \rightarrow \mathsf{count}(\mathsf{insert}(x,y),y) \neq_{Nat} \mathsf{zero} \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{true}$

uses 3a.2

(3b.1-2.1) $\mathsf{add}(x,y) =_{Nat} \mathsf{one} \;\; \overset{\rightarrow}{\equiv} \;\; (x =_{Nat} \mathsf{one} \wedge y =_{Nat} \mathsf{zero}) \vee (x =_{Nat} \mathsf{zero} \wedge x =_{Nat} \mathsf{one})$

Lemmas which state conditional equalities ($\mathsf{s} = \mathsf{t}$ if c) as (3b.1-2) can only be applied by UNICOM if the equation and the condition occur together in the pair to be reduced, i.e. the equation to be proved has to 'contain' the lemma to be applied explicitly. There is no way to *deduce* the application condition from 'contextual' terms in the considered equation. To produce usable condition terms reduction rules often have to introduce several cases (e.g. (3b.1-2.1, 10'.1)). However, not every case distinction can be formulated as a terminating reduction rule and the appropriateness of case distinction rules is more or less restricted to the situation the rule was intended for - the uncontrolled application of case distinction rules is rather destructive than fruitfull (cf. the comment on lemma 10'.1). So UNICOM is only rudimentarily able to realize some kind of nested argumentation.

Theorem 3c: 'Insert' does not add already contained numbers in sorted trees

$$\mathsf{sorted}(x) \wedge \mathsf{contains}(x,y) \rightarrow \mathsf{count}(\mathsf{insert}(x,y),\mathsf{z}) =_{Nat} \mathsf{count}(x,\mathsf{z}) \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{true}$$

(3c.1) $\mathsf{less\text{-}all}(m,u) \wedge n <_{Nat} m \rightarrow \mathsf{count}(u,n) =_{Nat} \mathsf{zero} \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{true}$

(3c.2) $\mathsf{greater\text{-}all}(m,u) \wedge n >_{Nat} m \rightarrow \mathsf{count}(u,n) =_{Nat} \mathsf{zero} \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{true}$

Theorem 4: 'Insert' preserves the sorted property

$$\mathsf{sorted}(x) \rightarrow \mathsf{sorted}(\mathsf{insert}(x,y)) \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{true}$$
$$\mathsf{sorted}(\mathsf{insert}(x,y)) \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{sorted}(x)$$

(4.1) $\mathsf{greater\text{-}all}(x,\mathsf{insert}(y,z)) \;\; \overset{\rightarrow}{\equiv} \;\; x \geq_{Nat} z \wedge \mathsf{greater\text{-}all}(x,y)$

(4.2) $\mathsf{less\text{-}all}(x,\mathsf{insert}(y,z)) \;\; \overset{\rightarrow}{\equiv} \;\; z \geq_{Nat} x \wedge \mathsf{less\text{-}all}(x,y)$

Theorem 5a: 'Delete' removes one occurence of the given number in sorted trees

$$\mathsf{sorted}(x) \rightarrow \mathsf{count}(\mathsf{delete}(x,y),y) \;\; =_{Nat} \mathsf{pred}(\mathsf{cound}(x,y)) \;\; \overset{\rightarrow}{\equiv} \;\; \mathsf{true}$$

uses 3a.1-2, 3c.1-2, 3b.1-2.1

(5a.1) $\mathsf{count}(\mathsf{combine}(x,y),z) \;\stackrel{\rightarrow}{=}\; \mathsf{add}(\mathsf{count}(x,z),\mathsf{count}(y,z))$

Theorem 5b: 'Delete' does not change the number of occurences of other elements

$$y \neq_{Nat} z \rightarrow \mathsf{count}(\mathsf{delete}(x,z),y) =_{Nat} \mathsf{count}(x,y) \quad \stackrel{\rightarrow}{=} \quad \mathsf{true}$$

uses 3a.1-2, 3b.1-2.1, 5a.1

Theorem 6: 'Delete' preserves the sorted property

$$\mathsf{sorted}(x) \rightarrow \mathsf{sorted}(\mathsf{delete}(x,y)) \quad \stackrel{\rightarrow}{=} \quad \mathsf{true}$$

(6.1) $\mathsf{greater\text{-}all}(x,y) \rightarrow \mathsf{greater\text{-}all}(x,\mathsf{delete}(y,z)) \quad \stackrel{\rightarrow}{=} \quad \mathsf{true}$

(6.1.1) $\mathsf{greater\text{-}all}(x,\mathsf{combine}(u,v)) \;\stackrel{\rightarrow}{=}\; \mathsf{greater\text{-}all}(x,u) \wedge \mathsf{greater\text{-}all}(x,v)$

(6.2) $\mathsf{less\text{-}all}(x,y) \rightarrow \mathsf{less\text{-}all}(x,\mathsf{delete}(y,z)) \quad \stackrel{\rightarrow}{=} \quad \mathsf{true}$

(6.1.2) $\mathsf{less\text{-}all}(x,\mathsf{combine}(u,v)) \;\stackrel{\rightarrow}{=}\; \mathsf{less\text{-}all}(x,u) \wedge \mathsf{less\text{-}all}(x,v)$

(6.3) $\mathsf{sorted}(x) \wedge \mathsf{sorted}(y) \wedge \mathsf{greater\text{-}all}(m,x) \wedge \mathsf{less\text{-}all}(m,y) \rightarrow \mathsf{sorted}(\mathsf{combine}(x,y)) \quad \stackrel{\rightarrow}{=} \quad \mathsf{true}$

uses 6.1.1

(6.1.3) $x \geq_{Nat} y \wedge \mathsf{greater\text{-}all}(y,z) \rightarrow \mathsf{greater\text{-}all}(x,z) \quad \stackrel{\rightarrow}{=} \quad \mathsf{true}$

Theorem 7a: 'Delete1' decreases height

$$\mathsf{height}(x) \geq_{Nat} \mathsf{height}(\mathsf{delete1}(x,z)) \quad \stackrel{\rightarrow}{=} \quad \mathsf{true}$$

(This does not hold for 'delete'.)

uses 1a.2-3, 1b.1

(7a.1) $\mathsf{delete\text{-}leftmost}(\mathsf{nd}(u,v,w)) \;\stackrel{\rightarrow}{=}\; \mathsf{if\text{-}bt}(\mathsf{height}(u) =_{Nat} \mathsf{zero}, v, \mathsf{nd}(\mathsf{delete\text{-}leftmost}(u),v,w))$

(7a.2) $\mathsf{delete1}(\mathsf{nd}(u,v,w),x) \;\stackrel{\rightarrow}{=}$
$\quad \mathsf{if\text{-}bt}(w =_{Nat} x,$
$\quad\quad \mathsf{if\text{-}bt}(\mathsf{height}(v) =_{Nat} \mathsf{zero}, u, \mathsf{nd}(u,\mathsf{delete\text{-}leftmost}(v),\mathsf{leftmost}(v))),$
$\quad\quad \mathsf{if\text{-}bt}(w >_{Nat} x \vee \mathsf{height}(v) =_{Nat} \mathsf{zero},\mathsf{nd}(\mathsf{delete1}(u,x),v,w),\mathsf{nd}(u,\mathsf{delete1}(v,x),w)))$

(7a.3) $\mathsf{max}(x,y) =_{Nat} \mathsf{zero} \quad \stackrel{\rightarrow}{=} \quad x =_{Nat} \mathsf{zero} \wedge y =_{Nat} \mathsf{zero}$

(7a.4) $\mathsf{height}(x) \geq_{Nat} \mathsf{height}(\mathsf{delete\text{-}leftmost}(x)) \quad \stackrel{\rightarrow}{=} \quad \mathsf{true}$

Theorem 7b: 'Delete1' decreases height at most by one

$$\mathsf{suc}(\mathsf{height}(\mathsf{delete1}(x,z))) \geq_{Nat} \mathsf{height}(x) \quad \stackrel{\rightarrow}{=} \quad \mathsf{true}$$

uses 7a.1-2, 1a.2-3, 1b.1

(7b.1) $\mathsf{suc(height(delete\text{-}leftmost}(x))) \geq_{Nat} \mathsf{height}(x) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

Theorem 8a: 'Delete1' removes one occurrence of the given number in sorted trees

$$\mathsf{sorted}(x) \to \mathsf{count(delete1}(x,y),y) =_{Nat} \mathsf{pred(cound}(x,y)) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$$

uses 3a.1-2, 3b.1-2.1, 3c.1-2, 7a.2

(8a.1) $\mathsf{height}(u) =_{Nat} \mathsf{zero} \to \mathsf{count}(u,y) =_{Nat} \mathsf{zero} \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

(8a.2) $\mathsf{leftmost}(u) =_{Nat} y \wedge \mathsf{height}(u) \neq_{Nat} \mathsf{zero} \to \mathsf{count}(u,y) \neq_{Nat} \mathsf{zero} \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

(8a.3) $\mathsf{leftmost}(u) =_{Nat} y \wedge \mathsf{height}(u) \neq_{Nat} \mathsf{zero} \to \mathsf{count(delete\text{-}leftmost}(u),y) =_{Nat} \mathsf{pred(count}(u,y))$ $\overset{\rightarrow}{\equiv} \quad \mathsf{true}$

(8a.4) $\mathsf{leftmost}(u) \neq_{Nat} y \to \mathsf{count(delete\text{-}leftmost}(u),y) =_{Nat} \mathsf{count}(u,y) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

uses 7a.1

(8a.1-4.1) $\mathsf{leftmost(nd}(u,v,w)) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{if\text{-}nat(height}(u) =_{Nat} \mathsf{zero}, w, \mathsf{leftmost}(u))$

The 'reformulation' of the recursion rules of the functions 'leftmost', 'delete-leftmost' and 'delete1' (lemmas (7a.1-2, 8a.1-4.1) serves for equalizing cases in the definition of symbols occurring in the theorems (e.g. 'height', 'count' and 'leftmost' in lemma (8a.2)). The rules make it possible to 'expand' e.g. 'leftmost' under the case distinction of 'count'. This, however, does not mean that the induction schemes used in these proofs follow the recursion scheme of 'count'.

Theorem 8b: 'Delete1' does not change the number of occurrences of other elements

$$y \neq_{Nat} z \to \mathsf{count(delete1}(x,z),y) =_{Nat} \mathsf{count}(x,y) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$$

uses 3a.1-2, 3b.1-2.1, 7a.2, 8a.1-4

Theorem 9: 'Delete1' preserves the sorted property

$$\mathsf{sorted}(x) \to \mathsf{sorted(delete1}(x,y)) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$$

uses 3a.1-2, 3b.1-2.1, 7a.2, 8a.1-4

(9.1) $\mathsf{greater\text{-}all}(x,y) \to \mathsf{greater\text{-}all}(x,\mathsf{delete1}(y,z)) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

uses 7a.2

(9.1.1) $\mathsf{greater\text{-}all}(x,y) \to \mathsf{greater\text{-}all}(x,\mathsf{delete\text{-}leftmost}(y)) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

(9.1.2) $\mathsf{greater\text{-}all}(y,u) \to y \geq_{Nat} \mathsf{leftmost}(u) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

(9.2) $\mathsf{less\text{-}all}(x,y) \to \mathsf{less\text{-}all}(x,\mathsf{delete1}(y,z)) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

uses 7a.2

(9.2.1) $\mathsf{less\text{-}all}(x,y) \to \mathsf{less\text{-}all}(x,\mathsf{delete\text{-}leftmost}(y)) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

(9.2.2) $\mathsf{less\text{-}all}(y,\mathsf{nd}(u,v,w)) \to \mathsf{leftmost(nd}(u,v,w)) \geq_{Nat} y \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

(9.3) $\mathsf{sorted}(x) \to \mathsf{sorted(delete\text{-}leftmost}(x)) \quad \overset{\rightarrow}{\equiv} \quad \mathsf{true}$

uses 7a.1, 9.1.1

(9.4) greater-all(x,y) $\wedge$ less-all(x,z) $\wedge$ height$(z) \neq_{Nat}$ zero $\rightarrow$ greater-all(leftmost(z),y) $\overrightarrow{\equiv}$ true

uses 8a.1-4.1, 6.1.3

(9.5) sorted(x) $\rightarrow$ less-all(leftmost(x),delete-leftmost(x)) $\overrightarrow{\equiv}$ true

uses 7a.1, 8a.1-4.1, 9.1.2

(9.5.1) less-all(w,v) $\wedge$ greater-all(w,u) $\rightarrow$ less-all(leftmost(u),v) $\overrightarrow{\equiv}$ true

uses 8a.1-4.1

(9.5.1.1) less-all(zero,y) $\overrightarrow{\equiv}$ true

(9.5.1.2) $w \geq_{Nat} y$ $\wedge$ less-all(w,v) $\rightarrow$ less-all(y,v) $\overrightarrow{\equiv}$ true

Theorem 10: 'Search' is 'contains' in sorted trees

search(x,y) $\rightarrow$ contains(x,y) $\overrightarrow{\equiv}$ true
sorted(x) $\wedge$ contains(x,y) $\rightarrow$ search(x,y) $\overrightarrow{\equiv}$ true

(10.1) less-all(m,y) $\wedge$ contains(y,n) $\rightarrow$ $n \geq_{Nat} m$ $\overrightarrow{\equiv}$ true

(10.2) greater-all(m,x) $\wedge$ contains(x,n) $\rightarrow$ $m \geq_{Nat} n$ $\overrightarrow{\equiv}$ true

Theorem 10': 'Search' is 'contains' in sorted trees (second version)

sorted(x) $\rightarrow$ contains(x,y) $\overrightarrow{\equiv}$ sorted(x) $\rightarrow$ search(x,y)

(10'.1) $\neg$ greater-all(w,u) $\vee$ search(u,x) $\vee$ $w =_{Nat} x$ $\overrightarrow{\equiv}$
($w \geq_{Nat} x$ $\vee$ $\neg$ greater-all(w,u)) $\wedge$ ($x \geq_{Nat} w$ $\vee$ $\neg$ greater-all(w,u) $\vee$ search(u,x))

(10'.2) $\neg$ less-all(w,v) $\vee$ search(v,z) $\vee$ $z \geq_{Nat} w$ $\overrightarrow{\equiv}$ $\neg$ less-all(w,v) $\vee$ $z \geq_{Nat} w$

Lemma (10'.1) has both a case generating and a case simplifying effect. A more readable proof would be obtained with the rule $w =_{Nat} x \overrightarrow{\equiv} w \leq_{Nat} x \wedge w \geq_{Nat} x$ that generates the cases and with rule (10'.2) and its pendant for 'greater-all' that simplify the cases. In general, however, the application of the above rule above cannot be desirable (cf. theorem (3b)).

Example IV: Two-Three-Trees

Two-three-trees are sorted trees where nodes have two or three children. Such trees can always be balanced in the sense that all leaves in a tree have the same depth. The example is taken from [Pad88].

Sorts:

tree:	nil:		$\rightarrow$ tree
	nd2:	tree $\times$ nat $\times$ tree	$\rightarrow$ tree
	nd3:	tree $\times$ nat $\times$ tree $\times$ nat $\times$ tree	$\rightarrow$ tree
frozen-tree:	freeze1:	tree	$\rightarrow$ frozen-tree
	freeze2:	tree $\times$ nat $\times$ tree	$\rightarrow$ frozen-tree
testobject:	tobj:	nat bool	$\rightarrow$ testobject

Function Definitions:

insert: nat $\times$ tree $\rightarrow$ frozen-tree
thaw: frozen-tree $\rightarrow$ tree
insert-and-balance: nat $\times$ tree $\rightarrow$ tree

insert(a,nil) $\rightarrow$ freeze2(nil,a,nil)
insert(a,nd2(x,n,y)) $\rightarrow$ if-fttt($a =_{Nat} n$,freeze1(nd2(x,n,y)),
  if-fttt($a >_{Nat} n$, nd2-2(x,n,insert(a,y)), nd2-1(insert(a,x),n,y)))
insert(a,nd3(x,m,y,n,z)) $\rightarrow$ if-fttt($a =_{Nat} n \vee a =_{Nat} m$,freeze1(nd3($x,m,y,n,z$)),
  if-fttt($a >_{Nat} n$,nd3-3(x,m,y,n,insert(a,z)),
   if-fttt($m >_{Nat} a$,
    nd3-1(insert(a,x),m,y,n,z),
    nd3-2(x,m,insert(a,y),n,z))))

thaw(freeze1(x)) $\rightarrow$ x
thaw(freeze2(x,n,y)) $\rightarrow$ nd2(x,n,y)

insert-and-balance(a,x) $\rightarrow$ thaw(insert(a,x))

nd2-1: frozen-tree $\times$ nat $\times$ tree $\rightarrow$ frozen-tree
nd2-2: tree $\times$ nat $\times$ frozen-tree $\rightarrow$ frozen-tree
nd3-1: frozen-tree $\times$ nat $\times$ tree $\times$ nat $\times$ tree $\rightarrow$ frozen-tree
nd3-2: tree $\times$ nat $\times$ frozen-tree $\times$ nat $\times$ tree $\rightarrow$ frozen-tree
nd3-3: tree $\times$ nat $\times$ tree $\times$ nat $\times$ frozen-tree $\rightarrow$ frozen-tree

nd2-1(freeze1(x),n,y) $\rightarrow$ freeze1(nd2(x,n,y))
nd2-1(freeze2(x,m,y),n,z) $\rightarrow$ freeze1(nd3(x,m,y,n,z))

nd2-2(x,n,freeze1(y)) $\rightarrow$ freeze1(nd2(x,n,y))
nd2-2(x,m,freeze2(y,n,z)) $\rightarrow$ freeze1(nd3(x,m,y,n,z))

nd3-1(freeze1(x),m,y,n,z) $\rightarrow$ freeze1(nd3(x,m,y,n,z))
nd3-1(freeze2(x,l,y),m,z,n,u) $\rightarrow$ freeze2(nd2(x,l,y),m,nd2(z,n,u))

nd3-2(x,m,freeze1(y),n,z) $\rightarrow$ freeze1(nd3(x,m,y,n,z))
nd3-2(x,l,freeze2(y,m,z),n,u) $\rightarrow$ freeze2(nd2(x,l,y),m,nd2(z,n,u))

nd3-3(x,m,y,n,freeze1(z)) $\rightarrow$ freeze1(nd3(x,m,y,n,z))
nd3-3(x,l,y,m,freeze2(z,n,u)) $\rightarrow$ freeze2(nd2(x,l,y),m,nd2(z,n,u))

height-left: tree → nat
balanced: tree → bool

$$\begin{array}{lll}
\mathsf{height\text{-}left}(\mathsf{nil}) & \rightarrow & \mathsf{zero} \\
\mathsf{height\text{-}left}(\mathsf{nd2}(x,n,y)) & \rightarrow & \mathsf{suc}(\mathsf{height\text{-}left}(x)) \\
\mathsf{height\text{-}left}(\mathsf{nd3}(x,m,y,n,z)) & \rightarrow & \mathsf{suc}(\mathsf{height\text{-}left}(x))
\end{array}$$

$$\begin{array}{lll}
\mathsf{balanced}(\mathsf{nil}) & \rightarrow & \mathsf{true} \\
\mathsf{balanced}(\mathsf{nd2}(x,n,y)) & \rightarrow & \mathsf{balanced}(x) \wedge \mathsf{balanced}(y) \wedge \mathsf{height\text{-}left}(x) =_{Nat} \mathsf{height\text{-}left}(y) \\
\mathsf{balanced}(\mathsf{nd3}(x,m,y,n,z)) & \rightarrow & \mathsf{balanced}(x) \wedge \mathsf{balanced}(y) \wedge \mathsf{balanced}(z) \wedge \\
 & & \mathsf{height\text{-}left}(x) =_{Nat} \mathsf{height\text{-}left}(y)) \wedge \\
 & & \mathsf{height\text{-}left}(y) =_{Nat} \mathsf{height\text{-}left}(z)
\end{array}$$

balanced1: tree → bool
balanced1-get: testobject → bool
balanced1-in: tree → testobject
balanced1-out2: testobject × testobject → testobject
balanced1-out3: testobject × testobject × testobject → testobject

$$\mathsf{balanced1}(x) \rightarrow \mathsf{balanced1\text{-}get}(\mathsf{balanced1\text{-}in}(x))$$

$$\mathsf{balanced1\text{-}get}(\mathsf{tobj}(x,y)) \rightarrow y$$

$$\begin{array}{lll}
\mathsf{balanced1\text{-}in}(\mathsf{nil}) & \rightarrow & \mathsf{tobj}(\mathsf{zero},\mathsf{true}) \\
\mathsf{balanced1\text{-}in}(\mathsf{nd2}(x,y,z)) & \rightarrow & \mathsf{balanced1\text{-}out2}(\mathsf{balanced1\text{-}in}(x),\mathsf{balanced1\text{-}in}(z)) \\
\mathsf{balanced1\text{-}in}(\mathsf{nd3}(x,y,z,u,v)) & \rightarrow & \mathsf{balanced1\text{-}out3}(\\
 & & \quad \mathsf{balanced1\text{-}in}(x),\mathsf{balanced1\text{-}in}(z),\mathsf{balanced1\text{-}in}(v))
\end{array}$$

$$\mathsf{balanced1\text{-}out2}(\mathsf{tobj}(x,y),\mathsf{tobj}(u,v)) \rightarrow \mathsf{tobj}(\mathsf{suc}(x), y \wedge v \wedge x =_{Nat} u)$$

$$\begin{array}{lll}
\mathsf{balanced1\text{-}out3}(\mathsf{tobj}(x,y),\mathsf{tobj}(u,v),\mathsf{tobj}(z,w)) & \rightarrow & \mathsf{tobj}(\mathsf{suc}(x), \\
 & & \quad y \wedge v \wedge w \wedge x =_{Nat} u \wedge u =_{Nat} z)
\end{array}$$

freeze1p: frozen-tree → bool

$$\begin{array}{lll}
\mathsf{freeze1p}(\mathsf{freeze1}(x)) & \rightarrow & \mathsf{true} \\
\mathsf{freeze1p}(\mathsf{freeze2}(x,y,z)) & \rightarrow & \mathsf{false}
\end{array}$$

Theorem 1: 'Insert' preserves the balanced property

$$\mathsf{balanced}(x) \rightarrow \mathsf{balanced}(\mathsf{insert\text{-}and\text{-}balance}(a,x))$$

$$\begin{array}{lll}
(1.1a)\ \mathsf{balanced}(\mathsf{thaw}(\mathsf{nd2\text{-}1}(x,y,z))) & \stackrel{\rightarrow}{=} & \mathsf{balanced}(\mathsf{thaw}(x)) \wedge \mathsf{balanced}(z) \wedge \\
 & & \mathsf{if\text{-}bool}(\mathsf{freeze1p}(x), \\
 & & \quad \mathsf{height\text{-}left}(\mathsf{thaw}(x)) =_{Nat} \mathsf{height\text{-}left}(z), \\
 & & \quad \mathsf{height\text{-}left}(\mathsf{thaw}(x)) =_{Nat} \mathsf{suc}(\mathsf{height\text{-}left}(z)))
\end{array}$$

$$\begin{array}{lll}
(1.1b)\ \mathsf{balanced}(\mathsf{thaw}(\mathsf{nd2\text{-}2}(x,y,z))) & \stackrel{\rightarrow}{=} & \mathsf{balanced}(x) \wedge \mathsf{balanced}(\mathsf{thaw}(z)) \wedge \\
 & & \mathsf{if\text{-}bool}(\mathsf{freeze1p}(z), \\
 & & \quad \mathsf{height\text{-}left}(\mathsf{thaw}(z)) =_{Nat} \mathsf{height\text{-}left}(x), \\
 & & \quad \mathsf{height\text{-}left}(\mathsf{thaw}(z)) =_{Nat} \mathsf{suc}(\mathsf{height\text{-}left}(x)))
\end{array}$$

$$\begin{array}{lll}
(1.1c)\ \mathsf{balanced}(\mathsf{thaw}(\mathsf{nd3\text{-}1}(x,y,z,u,v))) & \stackrel{\rightarrow}{=} & \mathsf{balanced}(\mathsf{thaw}(x)) \wedge \mathsf{balanced}(z) \wedge \mathsf{balanced}(v) \wedge \\
 & & \mathsf{height\text{-}left}(z) =_{Nat} \mathsf{height\text{-}left}(v) \wedge \\
 & & \mathsf{if\text{-}bool}(\mathsf{freeze1p}(x), \\
 & & \quad \mathsf{height\text{-}left}(\mathsf{thaw}(x)) =_{Nat} \mathsf{height\text{-}left}(z), \\
 & & \quad \mathsf{height\text{-}left}(\mathsf{thaw}(x)) =_{Nat} \mathsf{suc}(\mathsf{height\text{-}left}(z)))
\end{array}$$

$(1.1d)$ balanced(thaw(nd3-2(x,y,z,u,v))) $\stackrel{\to}{=}$ balanced(thaw(z)) $\wedge$ balanced(x) $\wedge$ balanced(v) $\wedge$
height-left(x) $=_{Nat}$ height-left(v) $\wedge$
if-bool(freeze1p(z),
height-left(thaw(z)) $=_{Nat}$ height-left(x),
height-left(thaw(z)) $=_{Nat}$ suc(height-left(x)))

$(1.1e)$ balanced(thaw(nd3-3(x,y,z,u,v))) $\stackrel{\to}{=}$ balanced(thaw(v)) $\wedge$ balanced(z) $\wedge$ balanced(x) $\wedge$
height-left(x) $=_{Nat}$ height-left(z) $\wedge$
if-bool(freeze1p(v),
height-left(thaw(v)) $=_{Nat}$ height-left(z),
height-left(thaw(v)) $=_{Nat}$ suc(height-left(z)))

(1.2) height-left(x) $=_{Nat}$ y $\wedge$ $\neg$ freeze1p(insert(u,x)) $\rightarrow$ height-left(thaw(insert(u,x))) $=_{Nat}$ suc(y) $\stackrel{\to}{=}$ true

(1.2.1a) freeze1p(nd2-1(x,y,z)) $\stackrel{\to}{=}$ true

(1.2.1b) freeze1p(nd2-2(x,y,z)) $\stackrel{\to}{=}$ true

(1.2.1c) freeze1p(nd3-1(x,y,z,u,v)) $\stackrel{\to}{=}$ freeze1p(x)

(1.2.2a) height-left(thaw(nd2-1(x,y,z))) $\stackrel{\to}{=}$ if-nat(freeze1p(x),
suc(height-left(thaw(x))),height-left(thaw(x)))

(1.2.2b) height-left(thaw(nd2-2(x,y,z))) $\stackrel{\to}{=}$ suc(height-left(x))

(1.2.2c) height-left(thaw(nd3-1(x,y,z,u,v))) $\stackrel{\to}{=}$ suc(height-left(thaw(x)))

(1.2.2d) height-left(thaw(nd3-2(x,y,z,u,v))) $\stackrel{\to}{=}$ if-nat(freeze1p(nd3-2(x,y,z,u,v)),
suc(height-left(x)),suc(suc(height-left(x))))

(1.2.2e) height-left(thaw(nd3-3(x,y,z,u,v))) $\stackrel{\to}{=}$ if-nat(freeze1p(nd3-3(x,y,z,u,v)),
suc(height-left(x)),suc(suc(height-left(x))))

(1.3) height-left(x) $=_{Nat}$ y $\wedge$ freeze1p(insert(u,x)) $\rightarrow$ height-left(thaw(insert(u,x))) $=_{Nat}$ y $\stackrel{\to}{=}$ true

uses 1.2, 1.2.1a-2e

Theorem 2: 'Balanced1' implements 'balanced'

balanced(x) $\stackrel{\to}{=}$ balanced1(x)

(2.1) balanced1-in(x) $\stackrel{\to}{=}$ tobj(height-left(x),balanced(x))

Example V: $\alpha - \beta -$Pruning

The Apha-Beta procedure is a well-known algorithm for searching game trees concerning two-person games. A game trees is explored to determine the next move by analysing possible future situations (leaves of the tree) that may be reached with legal moves (edges of the tree). Each leave is assigned its estimated *worth*. The worths for the next moves are computed by propagating the 'values' of the leaves to the root assuming that both players do their best according to the estimation (minimax). This is realized by the mutually recursive functions 'minimax-max' and 'minimax-min' (see below) which operate on arbitrarily expanded (game) trees with variable arity. The leaves of the trees contain natural numbers – the estimated worth of the situation corresponding to a leave.

The Alpha-Beta procedure can be used to compute the minimax-value (worth of the next move) more efficiently. The efficiency is gained by introducing some memory in the computation. The parameters a,b constitute the lower (a) and the upper (b) bound of the minimax-value to be computed, as known at each stage of the process. Knowing these bounds it is possible to prevent whole subtrees of a game tree from being explored if the lower bound exceeds the upper bound or vice versa. (For details on the Alpha-Beta procedure cf. e.g. [Nil80], [KM75].)

Sorts:

nat-omega:	omega:		→ nat-omega
	ino:	nat	→ nat-omega
node:	entry:	nat-omega	→ node
	nd:	nodes-list	→ node
nodes-list:	nl-nil:		→ nodes-list
	nl-cons:	node × nodes-list	→ nodes-list

Function Definitions:

$\leq_\omega$: nat-omega × nat-omega → bool
$\max_\omega$: nat-omega × nat-omega → nat-omega
$\min_\omega$: nat-omega × nat-omega → nat-omega

$\leq_\omega(x,\text{omega}) \rightarrow \text{true}$
$\leq_\omega(\text{omega},\text{ino}(x)) \rightarrow \text{false}$
$\leq_\omega(\text{ino}(x),\text{ino}(y)) \rightarrow x \leq_{Nat} y$

$\max_\omega(x,y) \rightarrow \text{if}_\omega(x \leq_\omega y, y, x)$

$\min_\omega(x,y) \rightarrow \text{if}_\omega(x \leq_\omega y, x, y)$

minimax-max: node → nat-omega
minimax-min: node → nat-omega

minimax-max(entry(x)) → x
minimax-max(nd(nl-nil)) → ino(zero)
minimax-max(nd(nl-cons(x,y))) → $\max_\omega$(minimax-min(x),minimax-max(nd(y)))

minimax-min(entry(x)) → x
minimax-min(nd(nl-nil)) → omega
minimax-min(nd(nl-cons(x,y))) → $\min_\omega$(minimax-max(x),minimax-min(nd(y)))

a-b-max: node × nat-omega × nat-omega → nat-omega
a-b-min: node × nat-omega × nat-omega → nat-omega

$$\begin{array}{lcl}
\mathsf{a\text{-}b\text{-}max}(\mathsf{entry}(x),a,b) & \to & \mathsf{if}_\omega(b \leq_\omega a \vee b \leq_\omega x, b, \mathsf{if}_\omega(x \leq_\omega a, a, x)) \\
\mathsf{a\text{-}b\text{-}max}(\mathsf{nd}(\mathsf{nl\text{-}nil}),a,b) & \to & \mathsf{if}_\omega(b \leq_\omega a, b, a) \\
\mathsf{a\text{-}b\text{-}max}(\mathsf{nd}(\mathsf{nl\text{-}cons}(x,y)),a,b) & \to & \mathsf{if}_\omega(b \leq_\omega a, b, \mathsf{a\text{-}b\text{-}max}(\mathsf{nd}(y), \mathsf{max}_\omega(a, \mathsf{a\text{-}b\text{-}min}(x,a,b)), b))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{a\text{-}b\text{-}min}(\mathsf{entry}(x),a,b) & \to & \mathsf{if}_\omega(b \leq_\omega a \vee x \leq_\omega a, a, \mathsf{if}_\omega(b \leq_\omega x, b, x)) \\
\mathsf{a\text{-}b\text{-}min}(\mathsf{nd}(\mathsf{nl\text{-}nil}),a,b) & \to & \mathsf{if}_\omega(b \leq_\omega a, a, b) \\
\mathsf{a\text{-}b\text{-}min}(\mathsf{nd}(\mathsf{nl\text{-}cons}(x,y)),a,b) & \to & \mathsf{if}_\omega(b \leq_\omega a, a, \mathsf{a\text{-}b\text{-}min}(\mathsf{nd}(y), a, \mathsf{min}_\omega(b, \mathsf{a\text{-}b\text{-}max}(x,a,b))))
\end{array}$$

Theorem 1: Alpha-Beta implements Minimax

$$\begin{array}{l}
\mathsf{a\text{-}b\text{-}max}(x, \mathsf{ino}(\mathsf{zero}), \mathsf{omega}) = \mathsf{minimax\text{-}max}(x) \\
\mathsf{a\text{-}b\text{-}min}(x, \mathsf{ino}(\mathsf{zero}), \mathsf{omega}) = \mathsf{minimax\text{-}min}(x)
\end{array}$$

(1.1) $\mathsf{a\text{-}b\text{-}max}(x,a,b) \;\overset{=}{\equiv}\; \mathsf{if}_\omega(b \leq_\omega a \vee b \leq_\omega \mathsf{minimax\text{-}max}(x), b, \mathsf{if}_\omega(\mathsf{minimax\text{-}max}(x) \leq_\omega, a, a, \mathsf{minimax\text{-}max}(x)))$

(1.2) $\mathsf{a\text{-}b\text{-}min}(x,a,b) \;\overset{=}{\equiv}\; \mathsf{if}_\omega(b \leq_\omega a \vee \mathsf{minimax\text{-}min}(x) \leq_\omega a, a, \mathsf{if}_\omega(b \leq_\omega \mathsf{minimax\text{-}min}(x), b, \mathsf{minimax\text{-}min}(x)))$

(1.3) $\mathsf{if}_\omega(\mathsf{omega} \leq_\omega z, \mathsf{omega}, z) \;\overset{=}{\equiv}\; z$

$\mathsf{if}_\omega(\mathsf{z} \leq_\omega \mathsf{zero}, \mathsf{zero}, z) \;\overset{=}{\equiv}\; z$

The equations of the theorem are far too specialized to be adequate for a proof by induction. The lemmas (1.1-2) generalize the theorem in that they characterize the roles of the parameters a and b in the recursion. Note that the base cases of the definition of 'a-b-max/min' had to be redefined to make these lemmas possible. The original rules were: $\mathsf{a\text{-}b\text{-}max}(\mathsf{entry}(x),a,b) \;\overset{=}{\equiv}\; x$, $\mathsf{a\text{-}b\text{-}max}(\mathsf{nd}(\mathsf{nl\text{-}nil}),a,b) \;\overset{=}{\equiv}\; a$ (analogous: 'a-b-min'). These rules, however, make it hard to specify exactly what 'a-b-max/min' compute in the unintended cases and to find an expressible and usable generalization of the theorem.

The proof of the lemmas (1.1-2) involves about thirty (more or less technical) lemmas concering properties of '$\leq_\omega$' and 'if_ω' which are verified easily.

Example VI: A More Complex Induction Scheme

The function 'sum' computes the sum of the elements of a tree by use of a stack.

Sorts:

tree:	null:		$\rightarrow$ tree
	node:	tree $\times$ nat $\times$ tree	$\rightarrow$ tree
stack:	empty:		$\rightarrow$ stack
	push:	tree $\times$ stack	$\rightarrow$ stack

Function Definitions:

sum: tree $\times$ nat $\times$ stack $\rightarrow$ nat

$$
\begin{array}{lcl}
\mathsf{sum}(\mathsf{null},n,\mathsf{empty}) & \rightarrow & n \\
\mathsf{sum}(\mathsf{null},n,\mathsf{push}(t,s)) & \rightarrow & \mathsf{sum}(t,n,s) \\
\mathsf{sum}(\mathsf{node}(l,n,t),m,s) & \rightarrow & \mathsf{sum}(l,\mathsf{add}(m,n),\mathsf{push}(t,s))
\end{array}
$$

Note that the definition of the function sum is not proved to be terminating by UNICOM. Instead of the function 'add' any other binary function might be used as well.

Theorem 1: Recursion of 'sum' over the tree argument.

$$\mathsf{sum}(x,n,\mathsf{push}(y,s)) \overset{\rightarrow}{=} \mathsf{sum}(y,\mathsf{sum}(x,n,\mathsf{empty}),s)$$

Whereas the proof of this theorem is based on structural induction over x, the complete induction scheme used is more complex and not derivable from the recursion scheme of 'sum': The inductive hypothesis is applied three times with different instances. To show this, we display the induction step section of the proof record:

```
======================================================================

1.2: essential pair: SUM(L,ADD(M,Z),PUSH(R,PUSH(Y,S))) =
                     SUM(Y,SUM(NODE(L,Z,R),M,EMPTY),S)

Reduction of the pair:
lefthand side:
--> SUM(R,SUM(L,ADD(M,Z),EMPTY),PUSH(Y,S))
by means of inductive rule: SUM(X,N,PUSH(Y,S)) --> SUM(Y,SUM(X,N,EMPTY),S)     (see 1)
--> SUM(Y,SUM(R,SUM(L,ADD(M,Z),EMPTY),EMPTY),S)
by means of inductive rule: SUM(X,N,PUSH(Y,S)) --> SUM(Y,SUM(X,N,EMPTY),S)     (see 1)

righthand side:
--> SUM(Y,SUM(L,ADD(M,Z),PUSH(R,EMPTY)),S)
by means of definition rule: SUM(NODE(L,N,R),M,S) := SUM(L,ADD(M,N),PUSH(R,S))
--> SUM(Y,SUM(R,SUM(L,ADD(M,Z),EMPTY),EMPTY),S)
by means of inductive rule: SUM(X,N,PUSH(Y,S)) --> SUM(Y,SUM(X,N,EMPTY),S)     (see 1)

Result: SUM(Y,SUM(R,SUM(L,ADD(M,Z),EMPTY),EMPTY),S) =
        SUM(Y,SUM(R,SUM(L,ADD(M,Z),EMPTY),EMPTY),S)

  which is trivial.

======================================================================
```

F Installation guide

UNICOM is implemented and currently running on Apollo/Sun Workstations under Lucid Common Lisp (version 3/4). It should be adaptable to other Common Lisp environments without much effort. Especially it is prepared to work on Symbolics Machines (Genera 8.0; context: CLtL) and under AKCL (Austin Kyoto Common Lisp).

An executable Lisp dump may be produced from the UNICOM file tree as follows:

1. Start Lisp at directory 'unicom':

   cd ...../unicom
   <lisp-call>

2. Load the 'install'-file:

   (load "install")

3. Compile Unicom:

   (compile-unicom "system/source/")
   (quit)

   The output of the compiler can be viewed in the file '.../unicom/compiler-output'.

4. Load UNICOM and save the dump:

   <lisp-call>
   (load "install")
   (load-unicom-bin "system/source/")
   (make-unicom "unicom") % or any other file name
   (quit)

5. Invoke UNICOM:

unicom	% starts the executable Lisp dump
(u)	% enters the top level of UNICOM
...	% computing with UNICOM
quit	% leave UNICOM
(quit)	% leave the Lisp environment

If the edit command (Unicom-Toplevel) does not work correctly, adapt the function 'cm=editor-call' (file: system/source/unicom-commands.lisp) to your local requirements.

Acknowledgements

We would like to thank Ulrich Kühler and Inger Sonntag for carefully reading earlier versions of this paper.

References

[AGG+87] J. Avenhaus, R. Göbel, B. Gramlich, K. Madlener, and J. Steinbach. TRSPEC: A term rewriting based system for algebraic specifications. In S. Kaplan and J.-P. Jouannaud, editors, *Proc. of the 1st Int. Workshop on Conditional Term Rewriting Systems*, number 308 in Lecture Notes in Computer Science, pages 245–248. Springer, 1987.

[Bac88] L. Bachmair. Proof by consistency in equational theories. In *Proc. 3rd IEEE Symposium on Logic in Computer Science*, pages 228–233, 1988.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Acadaemic Press, 1979.

[BM88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing.* Acadaemic Press, 1988. Formerly: Notes and Reports in Computer Science and Applied Mathematics.

[Fri86] L. Fribourg. A strong restriction of the inductive completion procedure. In E. Kott, editor, *Proc. of the 13th Int. Conf. on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 105–116. Springer, 1986.

[Göb87] R. Göbel. Ground confluence. In P. Lescanne, editor, *Proc. of the 2nd Int. Conf. on Rewriting Techniques and Applications*, volume 256 of *Lecture Notes in Computer Science*, pages 156–167. Springer, 1987.

[Gra89] B. Gramlich. Inductive theorem proving using refined unfailing completion techniques. SEKI-report SR-89-14, Dept. of Comp. Science, Univ. of Kaiserslautern, 1989.

[Gra90a] B. Gramlich. Completion based inductive theorem proving. In L.C. Aiello, editor, *Proc. of the 9th European Conf. on Artificial Intelligence*, pages 314–319. Pitman Publishing, London, 1990.

[Gra90b] B. Gramlich. Completion based inductive theorem proving: A case study in verifying sorting algorithms. SEKI Report SR-90-04, Dept. of Comp. Science, Univ. of Kaiserslautern, 1990.

[Gra90c] B. Gramlich. Unicom: A refined completion based inductive theorem prover. In M.E. Stickel, editor, *Proc. of the 1oth Int. Conf. on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 655–656. Springer, 1990.

[HH80] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. In *Proc. of the 21st Conf. on Foundations of Computer Science*, pages 96–107, 1980. also in JCSS 25(2), pp. 239-266, 1982.

[HO80] G. Huet and D.C. Oppen. Equations and rewrite rules: A survey. In Ronald V. Book, editor, *Formal Languages, Perspectives And Open Problems*, pages 349–405. Academic Press, 1980.

[JK86] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in equational theories without constructors. In *Proc. Symposium on Logic in Computer Science*, pages 358–366. IEEE, 1986. also in Information and Computation, vol. 82(1), pp. 1-33, 1989.

[KL80] S. Kamin and J.-J. Levy. Two generalizations of recursive path orderings. unpublished note, Dept. of Computer Science, Univ. of Illinois, Urbana, Illinois, 1980.

[KM75] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[Küc87] W. Küchlin. Inductive completion by ground proof transformation. In *Proc. of Coll. on the Resolution of Equations in Algebraic Structures*, 1987.

[KZ89] D. Kapur and H. Zhang. An overview of RRL: Rewrite rule laboratory. In N. Dershowitz, editor, *Proc. of the 3rd Int. Conf. on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 513–529. Springer, 1989.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., 1980.

[Pad88] Peter Padawitz. *Computing in Horn Clause Theories*, volume 16 of *Monographs on Theoretical Computer Science.* Springer-Verlag, 1988.

[Pla85] D. Plaisted. Semantic confluence tests and completion methods. *Information and Control*, 65:182–215, 1985.

[Red90] U.S. Reddy. Term rewriting induction. In M.E. Stickel, editor, *Proc. of the 1oth Int. Conf. on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 162–177. Springer, 1990.

[Sch88] R. Scherer. UNICOM: Ein verfeinerter Rewrite-basierter Beweiser für induktive Theoreme. Master's thesis, Dept. of Comp. Science, Univ. of Kaiserslautern, 1988.