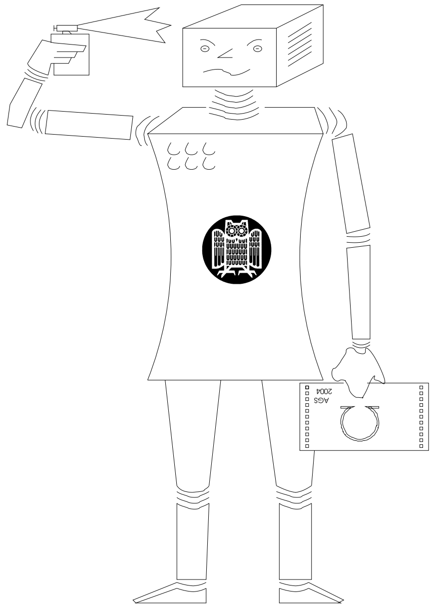
UNIVERSITÄT DES SAARLANDES
FACHRICHTUNG INFORMATIK
D–66123 SAARBRÜCKEN
GERMANY
WWW: http://www.ags.uni-sb.de/

SEKI-REPORT ISSN 1437-4447



The New Standard Tactics of the Inductive Theorem Prover QuodLibet

Tobias Schmidt-Samoa
FB Informatik, TU Kaiserslautern
schmidt@informatik.uni-kl.de

SEKI Report SR–2004–01

This SEKI Report was internally reviewed by:
Claus-Peter Wirth
FR Informatik, Universität des Saarlandes, D–66123 Saarbrücken, Germany
E-mail: `cp@ags.uni-sb.de`
WWW: `http://www.ags.uni-sb.de/~cp/welcome.html`

Editor of SEKI series:
Claus-Peter Wirth
FR Informatik, Universität des Saarlandes, D–66123 Saarbrücken, Germany
E-mail: `cp@ags.uni-sb.de`
WWW: `http://www.ags.uni-sb.de/~cp/welcome.html`

The New Standard Tactics of the Inductive Theorem Prover QuodLibet

Tobias Schmidt-Samoa
FB Informatik, TU Kaiserslautern
schmidt@informatik.uni-kl.de

September 28, 2004

Abstract

QuodLibet is a tactic-based inductive theorem prover for the verification of algebraic specifications of algorithms in the style of abstract data types with positive/negative-conditional equations. Its core system consists of a small inference machine kernel that merely acts as a proof checker. Automation is achieved with tactics written in QML (QuodLibet-Meta-Language), an adapted imperative programming language. In this paper, we describe QuodLibet's new standard tactics, a pool of general purpose tactics provided with the core system that support the user in proving inductive theorems. We aim at clarifying the underlying ideas as well as explaining the parameters with which the user can influence the behavior of the tactics during the proof process. One of the major achievements of this paper is the application of conditional lemmas controlled by obligatory and mandatory literals. This has drastically improved the degree of automation without increasing the runtime significantly as will be illustrated by the case studies. Nevertheless, the degree of automation depends on the specification style used. Thus, we will also give some guidelines how to write specifications and how to use the new tactics efficiently.

1 Introduction

QuodLibet [1, 11] is an inductive theorem prover for the specification and verification of algorithms in the style of abstract data types. It admits *partial* definitions of operators over *free constructors* C using (possibly *non-terminating*) *positive/negative*-conditional equations E as well as *destructor* recursion or *mutual* recursion. There is a well-defined semantics based on the class of so-called *data models*, i.e., models that do not equalize any different constructor ground terms. This class is guaranteed not to be empty if the specification fulfills a simple *admissibility condition* that essentially calls for confluence but not termination of the *axioms* given by E. This is tested by a simple syntactic confluence criterion. A clause is *inductively valid* iff it is valid in all data models of the specification. This semantics is *monotonic*, i.e., an inductively valid clause remains inductively valid in all consistent extensions of a specification. As usual a *clause* is a disjunction of literals, a *literal* is an atom or a negated atom. *Atoms* are built from predefined predicate symbols and terms. We have three kinds of atoms: *equality* atoms (represented by the predicate symbol $=$) for the formulation of equality-based specifications, *definedness* atoms (`def`) to establish the domain of (partial) operators, and *order* atoms ($<$) for the representation of a fixed induction order.

QuodLibet provides the user with an inference system to construct proofs of *lemmas*. Each inference rule transforms a goal into a (possibly empty) sequence of subgoals. A *goal* consists of a clause and a *weight* that determines a measure of the appropriate clause. To perform inductive proofs we do not want to use an induction scheme that has to be fixed at the beginning of the proof like in *explicit induction* [5, 19]. Instead, during the proof process we can apply a lemma *inductively* resulting in an additional *inductive proof obligation*. To establish an inductive proof obligation, it has to be shown that the weight of the induction hypothesis is "smaller" than the weight of the goal it is applied to. At the beginning of the proof process the weight of a lemma is represented by a weight variable. It may be instantiated during the proof by a tuple of terms using the variables of the lemma clause and arbitrary function symbols. This has to be done such that all inductive proof obligations can be fulfilled w.r.t. the fixed induction order which is a lexicographical combination based on the length of constructor ground terms. By the instantiation of the weight we have a degree of freedom to choose the concrete induction order. In this way we can realize *lazy induction* [14] and *descente infinie* [17]. Certainly, we can also simulate explicit induction.

There are inference rules for verifying simple tautologies, decomposing atoms, removing redundant literals, using negated atoms, handling order atoms, performing case analysis, and applying lemmas[1] (inductively or non-inductively) to rewrite or subsume goals. Each inference rule fulfills two local properties — *soundness* and *safeness* — that are used for establishing inductive validity. Informally speaking, under the assumption that all non-inductively applied lemmas are inductively valid, the *soundness* of an inference rule means that there will be a "smaller" counter-example to one of the subgoals if there is a counter-example to the original goal; an inference rule is *safe* if the inductive validity of the original goal implies the inductive validity of all subgoals.

A proof attempt of a lemma is represented by a *proof state tree* which contains goal and inference nodes. A proof state tree is *closed* iff all leaf nodes are inference nodes. In this case the soundness of the inference rules guarantees the inductive validity of the lemma provided that all non-inductively applied lemmas are inductively valid. Contrarily, if a goal with a non-valid clause, like the empty clause, is derived in a proof attempt of a lemma, then this lemma or one of the applied lemmas cannot be inductively valid because of the safeness property of the inference rules. In general it is also possible to start several proof attempts in parallel by applying more than one inference rule to a goal node leading to an AND/OR-tree as proof state tree.

QuodLibet's core system consists of a small inference machine kernel that guarantees soundness by allowing to alter the inference machine state only by a limited number of actions. These actions can be controlled by a text-based or graphical user-interface. The actions enable the user to enter and consistently extend specifications that fulfill the admissibility condition. Furthermore, the user can perform proofs by applying inference rules.

[1] We regard axioms as a subset of all lemmas that are inductively valid by definition. As a consequence we always apply axioms non-inductively.

In this way the system merely acts as a proof checker just allowing sound proofs without any automation.

To achieve semi-automation of the proof process we use a tactic-based approach. The user can formulate tactics in an adapted imperative programming language called QML (QuodLibet-Meta-Language, see [15]). As the tactics can only use the commands of the inference machine to alter the state they cannot produce incorrect proofs. Tactics can be written to support the whole proof process in the form of general purpose tactics as well as specialized for a certain domain. In this paper we will describe the new general purpose standard tactics that are made available within the system. They are mainly inspired by the former standard tactics described in [11] and by ideas developed in [5] and [10]. They are intended to overcome some of the shortcomings of the former standard tactics that will be described in the next section.

The rest of the paper is organized as follows: In Section 2 we will describe the proof process as it is modeled by the new standard tactics in comparison to the old ones. At the end of this section we will also give some comments on explicit induction. One of the major achievements of the new tactics is the automatic application of conditional lemmas that are not *directly applicable*, i.e., whose conditions are not completely fulfilled in the goal clause. QuodLibet provides inference rules to apply such lemmas creating *condition subgoals* that have to be proved for each missing condition. But as these condition subgoals just extend the original goal by some literals, special actions have to be taken to avoid non-terminating computations during the simplification process. This is controlled by *mandatory literals* in the goal clause and *obligatory literals* in the lemma clause as described in Section 3. After this motivation of the inductive proof process as well as a main part of the simplification process we will explain how to write specifications and use these proof processes in Section 4. In Section 5 we will describe the new tactics in detail arranged according to their modular structure. We will illustrate the degree of automation of the new tactics by some case studies in Section 6 and conclude with an outlook on further work in Section 7.

2 The Inductive Proof Process

An inductive proof for a clause φ performed by descente infinie can be divided into the following steps:

1. At first a case analysis is performed that depends on the recursion analysis of the operators in φ and leads to a case splitting. This step is for instance described for explicit induction in [5] or for the *cover set method* in [19].

2. Each case is simplified by applying inference rules and using the given lemmas in order to reduce it to a valid formula (base case) or to apply a smaller instance of φ (induction step) or other lemmas that are used by mutual induction.

3. It has to be shown that only smaller instances are used in the induction step. Therefore an appropriate wellfounded induction order has to be selected and the order constraints have to be proved.

4. If the clause cannot be shown inductively valid by this method additional steps have to be taken into account as e.g. generalization of φ or the speculation of additional lemmas.

This whole proof scheme was already used by the former standard tactics in [11]. The handling of step 1 is inherited from these old tactics if the inductive case analysis is performed fully *automatically*. It merges the *expandable* operator calls of φ, that do not *obstruct* other operator calls in the clause, into one inductive case splitting as described in [11]. Besides, the new tactics enable the user to perform the case analysis *semi-automatically* by selecting the operator calls that should be considered or *manually* by specifying the induction variables themselves.

In contrast to step 1, the simplification process of step 2 is completely reorganized in comparison to the old tactics. While the old tactics use different simplification tactics for different kinds of atoms that perform similar tasks with code sharing only on the level of auxiliary functions, there is essentially only one simplification tactic in the new implementation. This simplification tactic is divided into several passes, each one can be started separately for a special literal or for all literals. Besides, the handling in each pass depends on the kind of atom considered. This leads to a flexible simplification structure with huge code sharing so that the whole simplification process benefits from improvements of the tactic code. Furthermore the new tactics are parameterized so that some performance-relevant settings can be changed at runtime. Last but not least the new tactics tackle the following additional simplification tasks that will be explained in the next sections in more detail:

- *delayed* verification of conditions by the use of condition subgoals when applying conditional lemmas with the possibility to debug abortive attempts;
- rewriting with *permutative* lemmas as e.g. commutativity of plus or times provided the instantiation of the right-hand side is smaller than that of the left-hand side in a fixed wellfounded order (see [5]);
- applying lemmas inductively additional to the one represented by the current proof state tree, enabling semi-automatic handling of mutually recursive functions;
- improved handling of order and negated equality atoms;
- provision of *alternative literal representations* for boolean equalities and inequalities;
- avoidance of repetitions of equal inference steps in one path of a proof state tree to prevent infinite loops and speed up the computation;
- avoidance of top-level repetitions of alternative proof attempts so that different proof attempts will be computed each time.

Step 3 is performed by instantiating the weight variables of all involved induction hypotheses appropriately. In case of mutual induction with different weight variables this step is not automatized until now. In case of simple (non-mutual) induction, every lexicographical combination of the constructor variables in the lemma (possibly modified by monadic operators called *weight modifiers* in the following, see Section 5.4.2) is tried as weight instantiation. To cope with destructor recursion the tactics consider *order lemmas* activated by the user. By this procedure, the weight variable can be instantiated and the order constraints proved automatically in nearly all cases, provided the needed weight modifiers and order lemmas have been activated before. The new achievement in this step is the use of weight modifiers enabling the tactics to set the weight for sorting algorithms to the *length* of the sorted list instead the list itself which leads to simpler proofs of the order constraints.

The last step 4 is supported only rudimentally. So far, the tactics do not provide means to generalize a lemma clause. The user can choose to start a new inductive proof attempt automatically for each goal clause the simplification process gets stuck by generating a new proof state tree. But without generalization of the clause this method does not solve many problems but leads to infinite loops in most cases. Nevertheless there are proofs that profit from this possibility.

The overall proof process is modeled by special tactics called *strategies*. The old tactics offered only two different strategies that both performed an automatic inductive case analysis, simplified the resulting goals possibly using the root of the proof state tree inductively. They only differed in that the `standard-strategy` started a new proof state tree with another inductive proof for every goal that could not be simplified anymore, whereas the `restricted-strategy` stopped the whole proof process in this situation. We call a strategy *recursive* iff it tries another inductive proof when it gets stuck. In this sense, the standard-strategy is a recursive strategy. The new tactics on the other hand are parameterized by three orthogonal decisions:

- the way in which the inductive case splitting is performed: automatically, semi-automatically or manually;
- the lemmas that can be used as induction hypotheses;
- the decision whether the strategy is recursive.

This leads to a number of 12 different proof schemes so that the user can decide very flexible how the proof attempt should be performed. Besides, there are additional parameters to fine-tune the simplification process. By changing these parameters, the user can influence e.g. the handling of conditional and permutative lemmas, of free variables, and of order or negated equational atoms.

With the new tactics it is possible to reduce the manual applications for the examples in [11] from 43 to 5 without increasing the required time significantly. Furthermore we were able to perform some other case studies including sorting algorithms, irrationality of $\sqrt{2}$, properties of the greatest common divisor and the lexicographical path order (`Lpo`) using mutual recursion. These examples were partly out of scope of the old tactics because of the many proof steps that have to be performed manually. Nevertheless the new tactics do not prove complicated lemmas on their own. Thus, the user has to learn how to formulate lemmas, speculate intermediate lemmas, activate the right lemmas for the simplification process, and set the parameters of the tactics efficiently.

In explicit induction, steps 1 to 3 of our inductive proof scheme are performed at once at the beginning of the proof attempt. Therefore, this method depends on a strong analysis process. An induction scheme is introduced consisting of the formulas generated in step 1 as well as the induction hypotheses that solely can be used during the induction steps. Thus, an implication is constructed for each induction step relating induction hypotheses with induction conclusions. The induction schemes can only be generated according to terminating total function definitions that guarantee the wellfoundedness of the associated induction order.

The delay of the wellfoundedness proof in our approach does not cause any major differences because no information about this proof is used for guiding the inductive proof process in explicit induction. But explicit induction can take advantage of the early introduction of the induction hypotheses if they are appropriate:

- The simplification process can be guided in a goal-directed way using heuristics such as rippling techniques (see [7]).
- The application of lemmas, as e.g. transitivity lemmas, can be supported by providing additional information about the instantiation of free variables (see Example 3.3 in [17]).
- The information can be used in step 4 to generalize lemmas or speculate additional lemmas (see [5]).

The first of the above advantages of explicit induction is made independent from eager hypotheses application in [14]: Instead of using rippling techniques to rewrite an induction conclusion to a concrete induction hypothesis it is only important to move the differences of the induction conclusion (w.r.t. the original lemma) to tolerable positions, i.e., to top-level or variable positions. Therefore, rippling techniques can also be used with our approach. Whereas we cannot completely compensate for the other two advantages of explicit induction there are some problems with the explicit induction approach as well: As explained in [14] it is not always possible to compute appropriate induction hypotheses at the beginning of a proof attempt. Thus, the method of explicit induction may fail even for relatively simple examples unless the user provides an appropriate induction scheme himself. Especially, when dealing with mutually recursive functions this seems to raise great difficulties.

Our approach can be used to simulate explicit induction. Therefore, we can also combine both approaches by introducing the most probable induction hypotheses at the beginning of the proof. Whereas we are then able to perform the same proofs as in explicit induction we are not restricted to that method but can also use other induction hypotheses if they are more appropriate. This approach will be studied in the future.

3 Application of Conditional Lemmas

One of the most crucial tasks during the simplification process, and hence in the whole proof process, is the application of lemmas. Lemmas are provided by the user to guide the proof process. On the one hand, they should be checked for applicability intensively to free the user from routine work. On the other hand, heuristics have to control the applications to guarantee the termination of the process within a reasonable amount of time. The situation gets even more complicated as we deal with conditional lemmas. Thus, we have to try to relieve the conditions within a given context provided by the considered goal, possibly using other conditional lemmas. There are some important questions regarding this recursive process:

- In which order should the literals in the goal be considered?
- In which order should the lemmas be checked for applicability?
- What kind of checks have to be performed before we try to apply a lemma?
- Is there a special treatment when trying to relieve conditions of a conditional lemma?

Many authors have dealt with the application of conditional lemmas. An overview can be found in [18] where *contextual rewriting* as a generalization of *conditional rewriting* is described. It pays special attention to the context of the goal which the conditional lemma is applied in. We will compare our treatment to that approach at the end of this section.

We start with some notions to clarify the task. Both, the lemma and the goal it is applied to, consist of clauses. A clause is represented by its set of literals $\{l_1, \ldots, l_n\}$. A clause can be interpreted as an implication $\overline{l_1} \wedge \cdots \wedge \overline{l_{n-1}} \Rightarrow l_n$, where $\bar{l}$ is the negation of l[2]. A clause is called *conditional* iff $n > 1$; otherwise, it is called *unconditional*. We fix one literal in the lemma clause, called the *head literal*, and one literal in the goal clause, called the *focus literal*. The negation of the other literals in the lemma clause and goal clause are called *condition literals* and *context literals*, respectively.

QuodLibet provides the user with two kinds of inference rules to apply a lemma (inductively or non-inductively) to a goal, namely *rewriting* and *subsumption*. For both applications the lemma is instantiated by a substitution σ. Rewriting can only be applied if the head literal of the lemma is an equation $s = t$ (or $t = s$) so that $\sigma(s)$ is equal to a subterm of the focus literal of the goal clause. In that case, the subterm is replaced by $\sigma(t)$ in the *rewrite subgoal*. Besides, the instantiated condition literals have to be "fulfilled" by the context. This is checked lazily: We call a condition literal *directly fulfilled* in a goal iff the instantiated condition literal is present in the set of context literals of the goals. If a condition literal is not directly fulfilled in a goal, a new *condition subgoal* will be created that essentially extends the original goal by the instantiated condition literal. In this way, it is guaranteed that the condition or the original goal is proved if this proof attempt is closed. Furthermore, *definedness subgoals* will be created if σ binds a non-constructor term to a constructor variable. See [11] for details. The subsumption of a goal by a lemma results in the same definedness and condition subgoals as explained for rewriting. We call

[2]The negation of $s = t$, `def` s, and $s < t$ will be represented as $s \neq t$, `~def` s, and `~`$(s < t)$, respectively.

a lemma *directly applicable* iff all condition literals are directly fulfilled in the goal. Thus, every unconditional lemma is directly applicable (if there is a match from the head literal to the focus literal). We will clarify these notions in the following simple example.

Example 3.1 Let the specification consist of two sorts

- `Bool` for the boolean values with constructors `true` and `false`;
- `Nat` representing the natural numbers with the constructors `0` for zero and `s` for the successor function.

We consider two defined operators `less` and `plus` given by the axioms:

(1) $\mathtt{less}(\mathtt{0}, \mathtt{s}(y)) = \mathtt{true}$
(2) $\mathtt{less}(x, \mathtt{0}) = \mathtt{false}$
(3) $\mathtt{less}(\mathtt{s}(x), \mathtt{s}(y)) = \mathtt{less}(x, y)$

(4) $\mathtt{plus}(x, \mathtt{0}) = x$
(5) $\mathtt{plus}(x, \mathtt{s}(y)) = \mathtt{s}(\mathtt{plus}(x, y))$

We want to prove the conjecture

(6) $\{\ \mathtt{less}(x, \mathtt{plus}(y, z)) = \mathtt{true},\ \mathtt{less}(x, y) \neq \mathtt{true}\ \}$

using the following lemmas

(7) $\{\ \mathtt{def}\ \mathtt{plus}(x, y)\ \}$
(8) $\{\ \mathtt{less}(x, \mathtt{plus}(x, y)) = \mathtt{true},\ y = \mathtt{0}\ \}$

(9) $\{\ \mathtt{less}(x, z) = \mathtt{true},\ \mathtt{less}(x, y) \neq \mathtt{true},\ \mathtt{less}(y, z) \neq \mathtt{true}\ \}$

A proof state tree for the conjecture is illustrated in Figure 1. The root goal node consists of the conjecture to be proved and is displayed at the top of the proof state tree. As we do not perform an inductive proof we omit the weights of the goals. We reference goal and inference nodes by their positions in the proof state tree. The root position is denoted by `root` whereas all other positions are sequences of natural numbers separated by colons and enclosed by brackets. The numbers encode the path from the root goal node to the considered node. Thus, the position of a goal/inference node has equal/unequal length. The successor nodes are enumerated from left to right starting with 1.

The root goal node is rewritten by the conditional lemma (9) using the substitution $[z \leftarrow \mathtt{plus}(y, z)]$. The substitution can be determined by using the first literal of lemma (9) as head literal, the first literal of the root goal as focus literal and matching the head literal to the focus literal. The application results in three new subgoals: one definedness subgoal at position $[1:1]$, one condition subgoal at position $[1:2]$ and one rewrite subgoal at position $[1:3]$. Because of the condition subgoal, the lemma is not directly applicable. The definedness subgoal is proved by a direct application of lemma (7) as subsumption lemma. The subsumption of the goal at position $[1:2]$ by lemma (8) leads to another condition subgoal. After replacing the variable z with `0` by the inference rule $\neq$-`unif` the third literal of the resulting subgoal at position $[1:2:1^4]$[3] is directly rewritten by the unconditional axiom (4).

[3]This is an abbreviation for $[1:2:1:1:1:1]$.

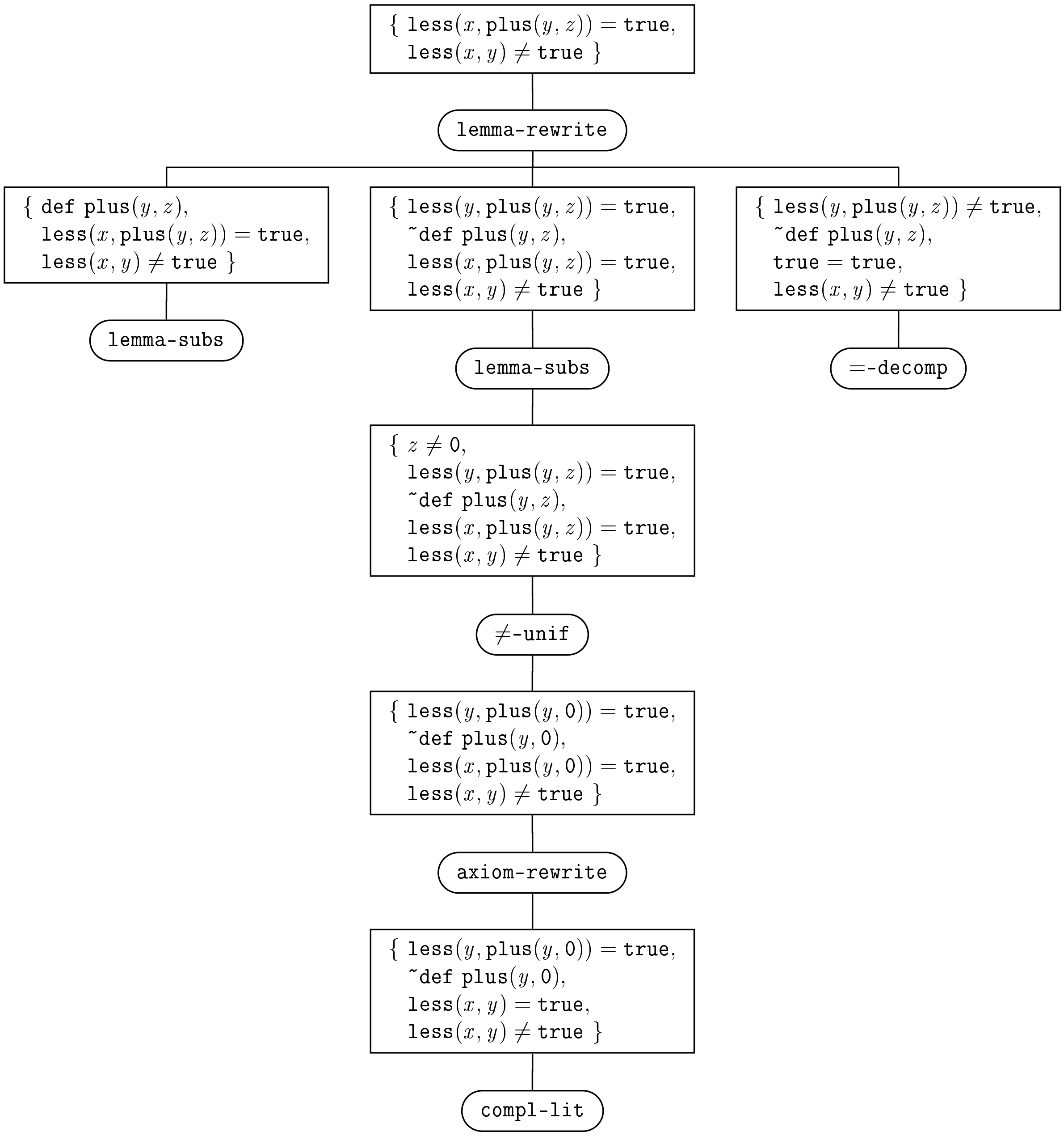


Figure 1: Proof State Tree for Example 3.1

The new tactics are able to perform such a simple proof automatically. In fact, if the needed axioms and lemmas are made available to the tactics they nearly construct the same proof state tree with only two exceptions. Firstly, the tactics use equations for rewriting unless the left-hand side of the equation is a variable. Therefore, the lemma subsumption at position $[1 : 2 : 1]$ is replaced by lemma rewriting and an application of the inference rule =-decomp analogously to the first inference node at position $[1]$. Secondly, not only the third literal but beforehand also the first literal of the goal at position $[1 : 2 : 1^4]$ is rewritten by axiom (4). This results in a slightly more complicated proof attempt but the added complexity is negligible.

We will now present some of the underlying ideas for the lemma application of the new tactics. Thereby, we will answer the initial questions of this section. The whole proof process is controlled by a so-called *database*. The database stores information about the *analysis* of defined operators and the *activated lemmas*. The analysis of a defined operator is used for performing an inductive case splitting automatically whereas the activated lemmas are applied by the tactics to rewrite or subsume a goal. The user has to analyze all operators and activate all lemmas that the tactics should consider during the proof process.

To reduce the search space when trying to apply a lemma, additional information is gathered during the activation. Most importantly, head literal(s) have to be provided. These can be given by the user; otherwise, one head literal will be computed automatically by some heuristics. According to the head literals the lemmas are divided into several sets: rewrite lemmas (REWRITE) or subsumption lemmas either for equations (EQSUBS), negated equations (NEGEQSUBS), definedness atoms (DEFSUBS) or order atoms (ORDERSUBS). When the tactics try to subsume a goal by a lemma, only the head literal of the right kind of subsumption lemma will be matched against the focus literal. In the same manner only the left-hand side of the head literal of a rewrite lemma will be matched against a subterm of the focus literal when trying to rewrite this subterm.

A lemma variable that is not present in the head literal of the lemma is called a *free variable*. As we do not want to guess the instantiation of free variables they have to be bound by matching appropriate lemma literals against other goal literals. If this is not possible the lemma will not be applied. When activating lemma (9) of Example 3.1 with the first literal as head literal, the variable y is free. It can be bound by matching the second or the third literal to a goal literal. When applying the lemma to the root goal node of Figure 1 it will be bound by matching the second lemma literal to the second goal literal.

If such a substitution can be found that binds all lemma variables, the lemma can be applied possibly resulting in condition subgoals. The handling of the condition subgoals may be expensive as the whole simplification process is applied recursively. Therefore, such a treatment should be restricted to applications with sufficient prospects of success. We believe that these prospects are given if the head literal of the lemma is specialized (see below) or if some condition literals are directly fulfilled in the goal as e.g. in the case of free variable handling. We call a term a *general term* iff it consists of an operator call with different variables at all argument positions. A lemma is *specialized* iff it is not a rewrite lemma with a general term as left-hand side of the head literal. To restrict the use of lemmas that are neither specialized nor have free variables, *obligatory literals* can be introduced when activating a lemma. These literals have to be fulfilled directly in the goal.

As an example, consider the lemma

(10) $\{$ `less`$(x, y) =$ `true`,
`leq`$(y, x) =$ `true` $\}$

If we activate this lemma for the first literal without obligatory literals it can be applied in the proof presented in Figure 1 to every subterm starting with symbol `less` without contributing to that proof. This can be prevented by using the second literal as obligatory literal. This reduces drastically the applications that have to be withdrawn but still allows the application of the lemma if it is directly applicable to a goal.

During the proof of definedness or condition subgoals we have to handle a subgoal that extends the original goal by some literals. Therefore, we have to take care that we do not infinitely apply the same conditional lemma to that subgoal. As for every other inference rule, this is achieved at first by preventing repeated applications of the same inference rule within one proof attempt. Additionally, during the relief of definedness or condition subgoals we use heuristics based on so-called *mandatory literals* which guarantee that at least one of the definedness or condition literals contribute to the proof. For this purpose, a new set of mandatory literals is introduced for every definedness and condition subgoal. This set consists of all the literals that have been added to the subgoal by the application of the lemma. When applying an inference rule to a goal that contains mandatory literals the inference rule must handle at least one of the mandatory literals. After the application, the set of mandatory literals for all subgoals (other than definedness or condition subgoals) will be supplemented by those literals that have been changed or added. Note, that a new set of mandatory literals is introduced for definedness and condition subgoals whereas for all other subgoals sets of mandatory literals are modified if they are not empty. Let us consider Example 3.1 once again. The definedness subgoal at position $[1 : 1]$ has one mandatory literal — the first one — that is handled by the following lemma subsumption. The mandatory literals of the condition subgoal at position $[1 : 2]$ are the first two literals. The first one is handled by the following lemma subsumption, that introduces the first literal as the only mandatory literal for the condition subgoal at position $[1 : 2 : 1^2]$. This single mandatory literal is used by the inference rule $\neq$`-unif`. As this inference rule modifies the first three literals of the resulting subgoal, they become the mandatory literals. From these the third is used for proving the subgoal in two further inference steps. Note that the subgoal at position $[1 : 3]$ has no mandatory literals as it is a rewrite subgoal and not a definedness or condition subgoal of a lemma application. This is justified by the fact, that the original goal is not contained in the new subgoal. Thus, preventing an infinite loop of the same lemma application.

The mechanism of mandatory literals is implemented by the tactics. Therefore, a data structure similar to proof state trees is created. This data structure is reconstructed each time the user calls a tactic. This may lead to a different result if a proof attempt is restarted in the middle of a proof state tree because the set of mandatory literals is empty and therefore may be smaller for the first subgoal of the restart.

If the definedness or condition subgoals of a lemma application cannot be shown by the tactics this failed proof attempt will be deleted and an alternative proof attempt will be tried. The user can turn on a *debug mode* that prevents the physical deletion of failed proof attempts. In debug mode the tactics inspect exactly the same proof attempts as in normal operation mode but the user will be able to analyze why a conditional lemma application has failed. Since the maintenance of alternative proof attempts gives rise to a high complexity, the computation slows down if the debug mode is turned on. Thus, the debug mode should be used only if necessary.

During the activation of a lemma, the user can influence the subsequent application of that lemma by providing the head and obligatory literals. Besides, there are some further parameters that restrict the application of lemmas. Their default values can be set in the so-called *default settings* and they can be overridden by *keyword parameters* when calling a tactic. We will represent such parameters by a leading colon. By the parameter *:rule-type-order*, for instance, the user can determine whether axioms or lemmas (inductively or non-inductively) should be applied first. Furthermore, the user can disallow the application of lemmas that are not directly applicable, the invention of new operators in condition subgoals or set the maximal recursion depth when applying conditional lemmas to prove condition subgoals. Further restrictions regarding the application of permutative lemmas and the handling of free variables will be explained in Section 5.2.

We will now summarize our previous description of automatic lemma applications by answering the initial questions of this section. When trying to rewrite or subsume a goal the tactics use each literal successively as focus literal. This is justified by the fact that all inference rules add new literals that may be important for the proof attempt to the front of the goal. At first the subsumption lemmas are checked, after that the subterms of the literal are tested for applicability of a rewrite lemma using an innermost left-to-right strategy. The application of lemmas is checked in two passes. During the first pass, only lemmas that are directly applicable are considered whereas during the second pass all lemmas are tested. In doing so, we hope to find easier proofs.

Having fixed a focus literal, axioms and lemmas (inductively and non-inductively) are considered according to the parameter *:rule-type-order*. Within each group axioms or lemmas will be tested in reverse activation order. Thus, newer activations are prefered to older ones. This order may be changed by the user so that he can influence the proof search. If a lemma can be applied proving all its definedness and condition subgoals it will not be deleted anymore. Thus, no alternative proof attempts for successful applications will be tried during this tactic execution.

As already explained, the head literal has to match against (a subterm of) the focus literal to initiate an applicability check for a lemma. Furthermore, all free variables have to be bound by other matching operations and all obligatory literals have to be directly fulfilled in the goal. This results in a fine-grained control of the lemma application mechanism.

The relief of definedness and condition subgoals is influenced by mandatory literals. This mechanism guarantees that the conditions are used in the proof attempt of that subgoal. This reduces the number of proofs that use unnecessary lemmas although they cannot be avoided completely. The relief of a condition can be further restricted by parameters that e.g. limit the recursion depth of conditional lemmas or prohibit the invention of new operators in these subgoals.

We will conclude this section with a brief comparison of our approach to the ones of [4], [18] and [2]. In [4] rewriting with a conditional lemma is done by a recursive process. During the recursion the relief of the hypotheses (i.e., the conditions) of the lemma is tried assuming the negation of the other goal literals (i.e., the context literals) to be true. The description is rather informal, especially w.r.t. the usage of the context literals during the rewriting process. One possible usage may be given by the *cross-fertilization* process. During this process negated equations in the goal clause can be used for replacing one side of the negated equation by the other side in another goal literal based on some heuristics. A formalization of these ideas coined *contextual rewriting* containing some improvements can be found in [18]. The major improvement is the usage of a constant congruence closure algorithm to decide the equality given by the context literals. But as mentioned in [18], both approaches are unable to prove the formula $\{\ \mathtt{p}(\mathsf{a}, \mathsf{b}) = \mathtt{false}\ \}$ by rewriting given the axioms

(11) $\{\ \mathtt{p}(\mathsf{a}, x) = \mathtt{false},\ \mathtt{q} \neq \mathtt{true}\ \}$

(12) $\{\ \mathtt{p}(y, \mathsf{b}) = \mathtt{false},\ \mathtt{q} = \mathtt{true}\ \}$

since for both axioms the condition, i.e., the second literal, cannot be relieved in an empty context. This limitation is overcome by *case rewriting* in [2]. In this approach, a term is rewritten by a set of n lemmas resulting in $n+1$ new subgoals: For each lemma one rewrite subgoal is created; additionally one *well-coveredness* subgoal is produced. This last subgoal is to guarantee the completeness of the case splitting w.r.t. the given lemmas. In the example above, the well-coveredness subgoal is $\{\ \mathtt{q} \neq \mathtt{true},\ \mathtt{q} = \mathtt{true}\ \}$ which is a tautology. But if the case splitting is not complete, this approach leads to an invalid well-coveredness subgoal.

Our approach is similar to case rewriting. But instead of applying all lemmas in parallel we apply them one after the other when we try to prove the condition subgoals of the former applications. Applying axiom (11) to the above goal formula leads to the condition subgoal $\{\ \mathtt{q} = \mathtt{true},\ \mathtt{p}(\mathsf{a}, \mathsf{b}) = \mathtt{false}\ \}$ which can be proved by axiom (12). If the lemmas do not fulfill the well-coveredness condition our approach does not produce invalid subgoals but leads to specialized subgoals that may be proved by other simplification methods. These specialized subgoals were omitted in [2] on purpose to prevent infinite rewrite loops. In our approach these loops are prevented by the use of mandatory literals and other heuristics that avoid redundant applications of inference rules.

Apart from condition subgoals that do not have to be created because the conditions are directly fulfilled in the context, we do not exploit the context very much. We only use negated equations in the context in the style of the cross-fertilization process of [4] by applying the inference rule `const-rewrite`. This inference rule may also be used for realizing a congruence closure algorithm. This improvement will be investigated for future development.

4 How to Use the New Tactics

Before we will present each public routine in detail in Section 5, we give some advice on their general and efficient use. This section, we hope, is particularly useful for new users of the automation mechanisms of QuodLibet. Some of the hints are special to our inductive theorem prover QuodLibet whereas others are more general and are essentially inspired by [5], chapters 9 and 13, and [10], chapter 9. As those hints are originally given for the inductive theorem provers `NQTHM` and `ACL2` they have to be translated to fit our inductive theorem prover.

To be able to use the theorem prover efficiently the user has to know exactly how the theorem prover works and how it can be influenced. What is even worse is that little changes in the specification or activation style may have significant consequences on the abilities of the system to prove theorems at all as well as on the time consumed to do the proofs. Therefore, the user has to learn how to write good specifications, i.e., specifications that enable the tactics to find small proofs in a short time. We will consider this question in Section 4.1. In addition to some fundamental guidelines about the formulation and activation of axioms and lemmas, we will present a tool that helps users to analyze former specifications and to improve their specification style.

The new tactics provide the user with a lot of different routines. In Section 4.2 we try to clarify when to use which tactic. Therefore, the tactics will be divided into several groups. This classification is reflected in the modular structure that will be used for explaining the tactics in detail in the technical Section 5.

If the proof attempt gets stuck the user has to help the theorem prover manually, e.g. by applying an inference rule, formulating an auxiliary lemma or activating a former lemma in a different way. Another reason for a failed proof attempt may be that the lemma is not inductively valid at all. We will give some hints on how to analyze (failed) proof attempts in Section 4.3. There, we will also discuss the debug mode that helps users to understand why a (conditional) lemma has not been applied.

After a proof attempt has succeeded, the user normally wants to save the result in a succinct way to be able to use the lemmas during further proofs in another session. There are two ways to do this by saving a state of the inference machine or a command script. Whereas states can be read in much faster, command scripts provide more information about the proof process. Therefore, we prefer to save our work in command scripts, possibly supplemented by state files if the command scripts are read in too slowly. QuodLibet provides means to save command scripts automatically but these scripts tend to be very large as they store every command entered. Even worse, these command scripts may fail to be read in again if a command has been aborted due to a non-terminating computation. Hence, we prefer to save successful command applications by copying them to an editor. We will present this approach in Section 4.4. Furthermore, in this section we will adapt *The Method* described in [10], chapter 9. Using this method enables the user to search top-down for a proof of a lemma and to store it in a bottom-up style that is supported by our inductive theorem prover.

We will conclude this introduction with some important technical notes: Before using the tactics the database has to be initialized. The tactics can only generate case splittings for analyzed operators and apply activated lemmas. Therefore, a typical usage of the system is to initialize the inference machine and the database in the beginning, then to specify and analyze the required operators, and after that to specify, prove and activate each lemma in succession.

4.1 How to Write Good Specifications

It is very difficult to give guidelines for writing good specifications, i.e., specifications that support the tactics in proving lemmas within a minimal amount of time, since the tactics are very sensitive w.r.t. little changes of the specification. What is good in one situation may be bad in another. Thus, every guideline that we give in the following is to be understood as a heuristic that often leads to satisfying results but may also fail. Some of the guidelines are implemented as heuristics (e.g. for the activation of lemmas) but many of them are very abstract and have to be concretized to be applicable. There may even be conflicts between different guidelines. Therefore, it is important to make one's own experiences in using the theorem prover and to analyze the resulting specifications. A tool that supports this analysis will be presented at the end of this section.

The first two guidelines deal with the invention and usage of (defined) operators.

Guideline 4.1 We recommend using the properties of the specification language like clause form or built-in predicate symbols instead of user-defined operators. This concerns e.g. boolean operators like `not`, `and`, `or`.

The reason for this guideline is very natural: As the inference rules are defined to reflect the semantics of the specification language and since these rules are fixed, tactics can make much more use of them than applying axioms of defined operators. There is e.g. a special treatment for definedness atoms and equational literals. This results from a fixed arrangement of some inference rules to *macro inference steps*.

Because of the clause form for lemmas, the presence of negated atoms and the implicit conjunction of all lemmas, the boolean operators are not needed in most cases. Nevertheless, it may be convenient to use them in some situations to get specifications that are more problem-oriented or to exploit properties of the operators as the associativity or commutativity of `and`/`or`. To solve this problem, two different representations may be used, one external problem-oriented one and another internal for the reasoning process. A connection should be established by showing the equivalence between the two representations as rewrite rule from the external to the internal representation. Thus, a user can write problem-oriented specifications in the external representation whereas most of the reasoning process will take place in the internal representation after rewriting.

Guideline 4.2 For sorts with exactly two constructors that are constants, we recommend using just one of the constants as far as possible.

This guideline refers for instance to the boolean values `true` and `false` as defined in Example 3.1 on page 8. For each equational literal consisting of a defined term on one side and one of the constants on the other side, there are two different representations. For example $t = \texttt{true}$ is equivalent to $t \neq \texttt{false}$ if t is a defined term, even though they are not identical. Because of our provision for obligatory and mandatory literals during conditional lemma application (see Section 3), it is vital to identify both representations during the test whether a context literal directly fulfills a condition literal. Therefore, it complicates the test if we use both constants. As it is vital for the proof process, our tactics consider both representations as it will be explained in Section 5.1. Thus, this guideline is only good for efficiency reasons and has minor priority.

The next guideline is concerned with the specification of axioms for defined operators.

Guideline 4.3 We recommend using a constructor-based specification style for axioms as far as possible.

Especially, the inductive case analysis is better suited for constructor-based specifications. Besides, these axioms will only be attempted if a syntactic matching criterion succeeds leading to a smaller number of conditions that have to be checked (see Guideline 4.6). This reduces the number of failed axiom applications that have to be withdrawn later on because the condition subgoals could not be proved. On the other hand there are certainly many functions that are to be specified with destructor recursion like the division on natural numbers based on the destructor minus. Therefore, following this guideline should not result in unnatural specifications.

The next guideline deals with the analysis of defined operators.

Guideline 4.4 We recommend analyzing each defined operator after having provided all defining rules. If the analysis shows unexpected results it may be necessary to change the specification or activate auxiliary lemmas.

The analysis of a defined operator (see Section 5.4.2 for details) gives a first hint if there are any flaws in the specification. Besides, the tactics for performing an inductive case splitting depend on the generated definition scheme so that lemmas can only be proved after the analysis of all involved operators. There may be several reasons if the analysis heuristics fail to guess a property, e.g. termination, that the operator is expected to have. Firstly, there may simply be typos in the specification. Secondly, when using destructor recursion, there normally has to be an activated order subsumption lemma that gives reasons for the argument of the recursive operator call to be smaller w.r.t. the induction order than the operator call on the left-hand side of the axiom. Last but not least, the specification style may not be well suited for the analysis procedure. In this situation the specification or the analysis procedure should be changed to be able to perform proofs for the specification.

The next four guidelines give hints for the specification of lemmas.

Guideline 4.5 We recommend formulating (negated) equational lemmas as rewrite lemmas as far as possible.

This means e.g. that a literal $\texttt{op}(x) \neq \texttt{true}$ should be substituted by $\texttt{op}(x) = \texttt{false}$ if the literal is used as the head literal of a lemma for a (defined) operator `op`. The reason for this is that each application of a negated equation as subsumption lemma can be simulated by applying the equation as rewrite rule and proving the rewritten clause containing the literal $\texttt{false} \neq \texttt{true}$ with the inference rule $\neq$`-taut`. On the other hand, the rewrite rule has a much broader range of application since it can also be used for rewriting arbitrary subterms within all kinds of atoms. This guideline may be in conflict with Guideline 4.2 that suggest to use only one of the boolean constants as far as possible. Due to the handling of alternative literal representation using rewrite rules is to be prefered.

Guideline 4.6 We recommend formulating lemmas with as few conditions as possible.

If not disabled by the parameter *:allow-delayed-condition-check* (see Section 5.2) condition subgoals will be created for each condition literal that is not directly fulfilled by the context literals when applying a conditional lemma. This may enlarge the search space enormously.

Therefore, it is a good specification style to reduce the number of conditions, e.g. the clause

(13) { $\texttt{sorted-list-p(append(}k, l\texttt{))} = \texttt{sorted-list-p(}l\texttt{)}$,
$\texttt{sorted-list-p(}k\texttt{)} \neq \texttt{true}$,
$\texttt{listleqlist-p(}k, l\texttt{)} \neq \texttt{true}$ }

is preferable in comparison to the clause

(14) { $\texttt{sorted-list-p(append(}k, l\texttt{))} = \texttt{true}$,
$\texttt{sorted-list-p(}k\texttt{)} \neq \texttt{true}$,
$\texttt{sorted-list-p(}l\texttt{)} \neq \texttt{true}$,
$\texttt{listleqlist-p(}k, l\texttt{)} \neq \texttt{true}$ }

as far as both clauses are inductively valid. In this example, clause (13) is even logically stronger making it applicable more often, since it establishes an equivalence between $\texttt{sorted-list-p(}l\texttt{)}$ and $\texttt{sorted-list-p(append(}k, l\texttt{))}$ whereas clause (14) only establishes an implication.

Guideline 4.7 We recommend orienting rewrite rules uniformly to get closer to "canonical forms".

Whereas the rewrite system given by axioms has to be confluent to fulfill the admissibility condition of QuodLibet, the invention of additional lemmas destroys this property in most cases. Nevertheless, the lemmas should not be oriented arbitrarily since the tactics use the rewrite lemmas only in one direction specified by the user. Thus, they should be oriented in a way that the resulting term is easier than the original term, e.g. by rewriting to constructor terms or subterms, or by pushing easier operators like constructors forward to the front of the term. This can be achieved by using a fixed reduction order to orient rewrite lemmas as far as possible. Following this guideline will increase the probability that the tactics find a proof and that they prevent infinite rewrite loops caused by cyclic rewrite relations.

We want to mention two aspects concerning this guideline: Firstly, there are some equations that are inherently non-terminating even when they are just applied from left to right as they allow for single cyclic rewrite steps. We call a rewrite lemma with head literal $l = r$ *permutative* iff l is a variable renaming of r, i.e., there exists a substitution σ with $\sigma(l) \equiv r$ that only renames variables. The commutativity of `plus` (see lemma (15) in Example 4.8) is an example for a permutative lemma. These lemmas need a special treatment. For this purpose we use a fixed total simplification order on terms that at first depends on the number of variables, the term length and the name of the top-level operator or variable. If all these values are equal for both terms, the first unequal argument terms are considered recursively. This order is inspired by [5]. A permutative lemma is only used for rewriting if the instance of the right-hand side is less than the instance of the left-hand side w.r.t. this order. This prevents infinite loops in most cases but as there is no reduction order that orients permutative lemmas there is no guarantee for this. Secondly, additional lemmas may be needed to rewrite to canonical forms. We do not use a completion algorithm to find these additional lemmas automatically but the user has to state them on his own. To illustrate both aspects, we will consider the commutativity and associativity of the addition on natural numbers.

Example 4.8 Let the operator `plus` over sort `Nat` be defined as in Example 3.1 on page 8. The commutativity and associativity of `plus` are given by the clauses

(15) $\{\ \texttt{plus}(x, y) = \texttt{plus}(y, x)\ \}$
(16) $\{\ \texttt{plus}(\texttt{plus}(x, y), z) = \texttt{plus}(x, \texttt{plus}(y, z))\ \}$

They can be proved automatically by the `auto-strategy` (see Section 5.7). In the first case, the keyword parameter *:recursive-strategy-p* has to be enabled so that two auxiliary lemmas can be proved by induction; otherwise, the proof gets stuck. But activating these lemmas is not enough to prove lemmas like

(17) $\{\ \texttt{plus}(x, \texttt{plus}(y, z)) = \texttt{plus}(y, \texttt{plus}(x, z))\ \}$

automatically by simplification. Because of the total simplification order used for permutative lemmas both terms $\texttt{plus}(x, \texttt{plus}(y, z))$ and $\texttt{plus}(y, \texttt{plus}(x, z))$ are irreducible. It is just this *extended commutativity* (17) that is needed to get a ground confluent rewrite system w.r.t. the simplification order used for permutative rewriting and, hence, to reduce most nested `plus`-terms to a canonical form.

In general, when proving associativity and commutativity of an operator, we recommend to prove also the extended commutativity of the operator. The lemmas should be activated in the order: commutativity, extended commutativity, associativity, so that the last property is tried first. Otherwise, an application of the associativity will always be substituted by two applications of the commutativity and one application of the extended commutativity. The associativity can be proved without induction if the other two properties have been activated before:

$$\begin{aligned} \underline{\texttt{plus}(\texttt{plus}(x, y), z)} &\longrightarrow_{(15)} \underline{\texttt{plus}(z, \texttt{plus}(x, y))} \\ &\longrightarrow_{(17)} \texttt{plus}(x, \underline{\texttt{plus}(z, y)}) \\ &\longrightarrow_{(15)} \texttt{plus}(x, \texttt{plus}(y, z)) \end{aligned}$$

Guideline 4.9 We recommend selecting variable names carefully if permutative operators are present in the lemma clause.

As explained in Guideline 4.7 rewriting with a permutative lemma, is only done if the rewritten term is smaller than the original term w.r.t. a fixed total simplification order. As this fixed order depends on the names of the operators and variables the naming of the variables does matter for the simplification process. Let us illustrate this by a simple example based on relations between the addition and the order on natural numbers.

Example 4.10 We consider the sorts `Bool` and `Nat` with the defined operator `plus` as specified in Example 3.1 on page 8 and operator `leq` given by the axioms:

(18) $\texttt{leq}(\texttt{0}, y) = \texttt{true}$
(19) $\texttt{leq}(\texttt{s}(x), \texttt{0}) = \texttt{false}$
(20) $\texttt{leq}(\texttt{s}(x), \texttt{s}(y)) = \texttt{leq}(x, y)$

As `plus` is defined by recursion on the second argument a lemma with clause

(21) $\{\ \texttt{leq}(\texttt{plus}(x, z), \texttt{plus}(y, z)) = \texttt{leq}(x, y)\ \}$

can be easily shown. It is proved automatically e.g. by the `auto-strategy` that performs a structural induction on z.

When trying to prove a lemma with clause

(22) $\{\ \texttt{leq}(\texttt{plus}(x,y),\texttt{plus}(x,z)) = \texttt{leq}(y,z)\ \}$

directly by induction (on y and z) two auxiliary lemmas are needed with clauses

(23) $\{\ \texttt{leq}(x,\texttt{s}(\texttt{plus}(x,z))) = \texttt{true}\ \}$
(24) $\{\ \texttt{leq}(\texttt{s}(\texttt{plus}(x,y)),x) = \texttt{false}\ \}$

On the other hand, lemma (22) can be proved by applying the commutativity of `plus` twice and then lemma (21). To achieve this permutative rewriting automatically, lemma (22) has to be reformulated so that the first argument of `plus` is less than the second argument of `plus` for both occurrences of `plus` in the left-hand side. This can be done by applying the substitution $[x \leftarrow z, y \leftarrow x, z \leftarrow y]$ resulting in the lemma clause

(25) $\{\ \texttt{leq}(\texttt{plus}(z,x),\texttt{plus}(z,y)) = \texttt{leq}(x,y)\ \}$

Depending on whether one wants to rewrite permutatively, the variables have to be named differently as shown in the last example. Certainly, permutative rewriting can be prevented at all by disabling the parameter *:allow-perm-rewriting-p* (see Section 5.2) or deactivating the permutative lemma (see Section 5.4.3).

The following four guidelines affect the activation of lemmas. These guidelines have been implemented as heuristics in the procedure `activate-lemma` described in Section 5.4.3. In special situations these heuristics can be overridden by using appropriate keyword parameters.

Guideline 4.11 We recommend activating lemmas for at most one head literal.

This guideline is made due to performance reasons. Enabling a lemma for two head literals may result in doubling the work to be done. On the other hand, it may lead to a proof if only an instantiation of the second head literal is directly fulfilled in the goal whereas the instantiation of the first literal can be proved as condition subgoal.

Guideline 4.12 We recommend activating lemmas with definedness or order literals as head literals if this is possible. Otherwise, we recommend selecting equational literals as head literals that bind most of the variables in the lemma clause.

As there are no means for proving negated definedness or order atoms except for using complementary literals, it is sensible that order or definedness atoms of lemmas have to be fulfilled directly in the goal clause to which the lemma is applied. This turns order and definedness atoms into perfect candidates for head literals. Otherwise, we recommend using a head literal that binds most of the variables because we demand that every free variable has to be instantiated by matching other lemma literals to goal literals before the lemma is applied. Choosing a literal with fewer bound variables enlarges the search space for binding the free variables.

Guideline 4.13 We recommend using obligatory literals for rewrite lemmas that have a general term as left-hand side of the head literal.

General terms — i.e., terms that consist of one operator call with different variables at all argument positions — do not provide any specific information if the lemma is useful when applied as rewrite rule. Therefore, these lemmas can be applied very often if the parameter *:allow-delayed-condition-check* is enabled (see Section 5.2). To reduce the search space at least one obligatory literal should be selected that restricts the applications of the lemma appropriately. This obligatory literal has to be fulfilled directly in the goal clause before the lemma is applied. Nevertheless, in some situations this cannot be achieved, i.e., a lemma is needed for a proof but no condition literal of the lemma is directly fulfilled in the goal clause. In this case the lemma should be reactivated temporarily without any obligatory literals. But the old activation should be restored as soon as possible for efficiency reasons.

Guideline 4.14 We recommend activating simplified lemmas only.

Otherwise, the lemma may not be applicable, as the literals, for which it should be applied, may be simplified by other lemmas first. This especially concerns the head literals (for rewrite rules at least the left-hand side) and the obligatory literals that have to be directly fulfilled in the goal clause. For all other literals of the lemma, simplification is to be prefered because of efficiency reasons, to reduce the number of further simplification steps.

As already mentioned we do not believe that a new user will write specifications that are perfectly suited for our inductive theorem prover just because of the guidelines presented above. Instead he will have to learn how to use the system in practice by analyzing the results of real specifications. We have implemented a perl script to help the user in doing this. This perl script extracts statistics about the proofs of a specification from a run of our inductive theorem prover given by a log file. The extracted statistics provide detailed information about the overall proof attempts as well as for each lemma separately. The information consist of the runtimes, the called tactics and procedures as well as the applied inference rules — both manually and automatically applied — and the automatically deleted inference rules due to unproved condition subgoals in conditional rewriting. The applications and deletions are further broken down to the applied inference rules and to the lemmas that have been applied.

The statistics can be used for fine-tuning specifications as well as for improving one's specification style. For instance, if a lemma is applied very often but all except of a few applications are deleted because of unproved condition subgoals, the user should restrict the use of the lemma e.g. by providing obligatory literals or by activating the lemma only for those proofs that use it. This has helped to reduce the proof effort for some specifications drastically.

Besides, the statistics have been used for improving the tactics. If any change is made to the tactics the effects will be studied for a set of specifications comparing the statistics of both versions. The change will only be kept if the results are significantly better for at least a few specifications. If they lead to worse results for other specifications a new parameter will be invented that can be set by the user. This has led e.g. to the invention of the notion *obligatory literals* to reduce the search effort when applying general term rewriting rules.

We conclude this section with a warning w.r.t. the generated statistics: They are essentially useful to identify conditional lemmas that have been attempted but deleted at the end because of unproved condition subgoals but they cannot be used for identifying lemmas that lead to complicated proofs. These can only be detected by looking at the proof state trees.

4.2 How to Choose the Right Tactic

Although many routines are provided only few of them are needed to perform most of the proofs. In this section we will focus on these routines. But let us first start with a warning: Like other inductive theorem provers, QuodLibet is not able to perform proofs even of relative simple lemmas without user interaction. The user has to perform at least one crucial task manually by providing enough auxiliary lemmas that the tactics can use for rewriting and subsumption. Therefore, the user normally has to think about a manual proof that is translated into the logic of QuodLibet later on. Certainly, the user can start a proof attempt, see if it succeeds, and analyze the failed proof attempt, otherwise. For experienced users, this may lead to the needed auxiliary lemmas in a short amount of time, but especially for new users it may be frustrating and confusing since the clauses tend to get large in the middle of a failed proof attempt. Hints for analyzing (failed) proof attempts will be given in Section 4.3.

So for now, let us assume that there is a sketch of a proof plan at hand, that the specification has been entered, all operators have been analyzed and some auxiliary lemmas have been proved and activated. If we want to prove another lemma automatically by using the tactics, we can always start a proof attempt by one of the strategies in the module `Proof-Strategies`. These strategies first try to simplify the goal as far as possible unless the parameter *:allow-cond-simplification-before-induction-p* (see Section 5.2) is disabled. In this case, simplification is restricted to directly applicable lemmas. The whole simplification process may also be activated by calling the tactic `simplify` in the module `Simplification`. If the goal cannot be proved by simplification, an inductive proof attempt is started for each open subgoal. The way in which the inductive case splitting is performed depends on the strategy that is used. For unexperienced users, we recommend starting with the `auto-strategy`. If the lemma mutually depends on other lemmas, these lemmas can be given by the keyword parameter *:ind-lemmas*. The strategy will perform the inductive case analysis automatically which will be beneficial for new users. The resulting case splittings will often lead to successful proofs if the needed auxiliary lemmas have been activated before, but unnecessary case splittings may be introduced resulting in a poorer performance.

The proof attempt for a lemma may lead to three different results:

- the proof process does not stop within a reasonable time;
- the proof process stops with a failure;
- the proof attempt succeeds.

In the first case, the user has to abort the proof process manually resulting in a failed proof attempt comparable to the second case. This abortion should not be done too early in order not to stop a successful proof attempt by mistake. But at least if the user notices in the output of the log window a cyclic behavior or terms that are pumped up, the proof process can be stopped. The reasons for an infinite proof process may be the activation of cyclic rewrite lemmas or the use of a recursive strategy, i.e., a proof strategy with keyword parameter *:recursive-strategy-p* enabled.

After a failed proof attempt the user first has to find out why this has happened by analyzing the failed proof attempt. We will consider this process in Section 4.3. Even in the case of a successful proof attempt the user should have a look at some crucial checkpoints of the proof because of efficiency considerations. One of these checkpoints in an inductive proof is the initial inductive case splitting, another is the result of the simplification process. The inductive case splitting and the result of the simplification process can be analyzed directly in the proof state tree, whereas additional information about the applied lemmas can be retrieved from a dependency graph that manages the applications of lemmas.

There may be several reasons if the tactics fail to prove a lemma:

- The lemma is not inductively valid:
  If the proof process results in a goal clause that is apparently not inductively valid like the empty clause, one of the unproved lemma clauses used in the proof is not inductively valid. In most cases this lemma clause has to be specialized e.g. by some condition literals to get a valid clause and the proof has to be restarted for the changed lemma.

- One lemma has not been applied as expected:
  The reasons for this may be a wrong activation of the lemma or some free variables that cannot be instantiated by the tactics appropriately. In the first case, another activation of the lemma may help. Otherwise, the user has to apply an inference rule manually. After this the original proof can be continued e.g. by one of the tactics in the module `Simplification` for instance by `simplify` to simplify just one goal at a time or by `simplify-open-subgoals` for handling all derived subgoals in succession. Alternatively, the tactic `cont-proof-attempt` in the module `Proof-Strategies` can be used. This last tactic is preferable if there are no other failed proof attempts so that an inductive proof can be closed by setting the weight variable and showing all order constraints.

- Other auxiliary lemmas are needed for the proof:
  If the simplification process fails to prove a goal the derived goals often give hints for speculating an auxiliary lemma by generalizing one of the goal clauses. After having proved this lemma, the original proof can be continued e.g. by one of the tactics in the module `Simplification` or the `cont-proof-attempt` tactic in the module `Proof-Strategies`. Alternatively, the whole proof can be deleted and restarted from the beginning.

- In the case of an inductive proof, a wrong induction scheme has been used:
  This may lead to a failed proof as well as to a bad performance. The user may try different induction schemes by offering the effective operator calls or induction variables that should be used for the case splitting by calling the `operator-strategy` or `variable-strategy` in the module `Proof-Strategies`.

If the simplification process after the inductive case splitting fails (e.g., runs forever) it is possible to introduce just the inductive case splitting automatically and then to stop without simplification. This is achieved by using one of the tactics in the module `Inductive-Case-Analyses`, although these have never been used in our case studies so far. Then the user can provide further directions interactively.

Last but not least it may be necessary to apply the inductive case splitting manually by an inference step and continue the proof attempt e.g. by tactic `cont-proof-attempt` in module `Proof-Strategies`.

In general, we recommend using the routines without any keyword parameters in the beginning as they provide sensible default values in most cases. Only if a proof attempt fails or when the user has become more experienced one should think of using keyword parameters. This concerns the proof strategies and simplification tactics as well as the procedures of the database. But one should keep in mind that the keyword parameters enable the user to reduce the effort during the simplification process e.g. by disallowing the use of permutative lemmas or the use of conditional lemmas which are not directly applicable, or by testing only one instantiation of free variables.

4.3 How to Analyze (Failed) Proof Attempts

As explained in Section 4.2 there may be different reasons for a failed or inefficient proof attempt. Before the proof attempt can be corrected the reason has to be determined by analyzing the proof attempt that is represented by a proof state tree. There are two main checkpoints introduced by the tactics that should be inspected:

- the automatic inductive case splitting;
- the result of the simplification process.

For technical reasons, only the root nodes of proof state trees can be applied as induction hypotheses. Thus, the tactics start an inductive proof with an inductive case splitting only for root nodes of proof state trees. For inner nodes that are to be proved by induction new proof state trees will be created. Therefore, the user has to look at the root of the proof state trees to inspect the first checkpoint. If the user expects another instantiation of the variables for the proof of the given lemma he can try to generate a different induction scheme automatically by using a different tactic or provide it manually by applying one of the inference rules `subst-add` (for constructor recursion) or `lit-add` (for destructor recursion or cut). An inappropriate inductive case analysis may result in too many subgoals and hence an inefficient proof, or even worse in a failed proof attempt if the wrong induction variables have been chosen.

The second checkpoint is particularly important if the proof attempt fails. If the failed proof attempt stops, all open subgoals are simplified as far as possible w.r.t. the implemented simplification process. The user can navigate through all open subgoals with the assistance of the graphical user interface by using the navigation button `Next Open GNode` of the proof state tree. If an open subgoal contains an order atom with a weight variable, this subgoal can be skipped at first after having checked that there is an instantiation of the weight variable that proves the subgoal. These subgoals result from inductive applications and normally do not cause any problems. They just have not been proved yet since the weight variable of the proof state tree is only instantiated automatically after all other subgoals have been shown. Otherwise, the proof of another subgoal may require induction and this may lead to an inductive proof obligation that is in conflict to a former instantiation of the weight variable. The instantiation is done automatically in most cases when calling the tactic `cont-proof-attempt` after all other subgoals have been shown.

For the other subgoals, the user should ask himself whether the subgoal is indeed inductively valid. If this is not the case, the subgoal often contains a hint for a counter-example. Then the user has to reformulate a specialization of the lemma so that these counter-examples are eliminated e.g. by introducing additional conditions. Otherwise, the user should identify literals in the clause of the subgoal that are expected to be inductively valid, thereby achieving a generalization of the subgoal. As the subgoal often consists of many literals it may be a difficult (and the most challenging) task to find out the relevant literals. To reduce the complexity of the clause, as a general rule negated definedness and negated order atoms should not be considered, and shorter literals should be prefered. Sometimes, it may be necessary to regard a predecessor of the open subgoal to get an appropriate auxiliary lemma as the simplification process will try some extraordinary transformations as a last resort to prove the subgoal. These transformations may cause difficulties in finding an auxiliary lemma. Certainly, this whole approach in finding an auxiliary lemma is just a heuristic that tries to help finding the needed relationships between the operators of the specification. Sometimes, this will require a deeper semantic understanding of the specification than described in this primarily syntactic approach.

Having found an auxiliary lemma, the user should introduce this lemma if it is not already present. After that the user has two possibilities to proceed: proving the auxiliary lemma first or continuing the proof of the main lemma. The decision depends essentially on one's individual flavor. Some users want to prove the simpler statement first to close some of the paths of the proof state tree, whereas others first want to check the critical statements. Both approaches are available within QuodLibet. If the user wants to use the auxiliary lemma in the proof of the main lemma automatically, the auxiliary lemma has to be activated. This activation should be removed before the auxiliary lemma is shown to prevent cyclic non-inductive lemma applications that do not establish inductive validity.

The last case we want to consider in this section is that we speculate an auxiliary lemma that is already present and activated in the database. Therefore, we ask ourselves why it has not been applied. This normally results from an inappropriate lemma activation. In this situation, the debug mode may provide a helpful view on the simplification process, at least if the auxiliary lemma is conditional. The debug mode can be enabled by the parameter *:debug-p* as explained in Section 5.2. For efficiency reasons, it should be disabled again as soon as possible. While in debug mode, proof attempts that have failed, e.g. because of unproved condition subgoals, will not be deleted physically from the proof state tree but will not be considered by the tactics. Thus, the same behavior is guaranteed with or without debug mode. This enables the user to find out easily why a certain condition could not be established. To use the debug mode the simplification process should be restarted from a subgoal where the auxiliary lemma is expected to be applied. This may lead to a slightly different proof attempt as explained in Section 3 since the data structure for mandatory literals is reconstructed each time the user calls a tactic. But often this behavior is adequate to analyze a failed proof attempt. Alternatively, the whole proof process can be restarted which, however, may result in a big and unhandy proof state tree. Having understood the reason for the failed lemma application, the failed proof attempts can be deleted by calling the tactic `delete-root-proof-attempts` in the module `Protected-Inference-Machine`, which is described in Section 5.3. The solution to the problem may then be performed by a different lemma activation using keyword parameters or by changing the order of the activated lemmas as described in Section 5.4.3. The result of the lemma activation w.r.t. the selected head literals or obligatory literals may be controlled by displaying the internal database as explained in Section 5.4.4. On the other hand, additional lemmas may have to be inserted to relieve the conditions of the lemma.

4.4 How to Get Good Command Scripts

As already mentioned in the introduction to Section 4, command scripts that have been extracted automatically from the log window of the proof session may fail to be read again since the log window may contain aborted commands that run forever[4]. Furthermore, when trying to prove a lemma there are often unforeseen difficulties leading to further auxiliary lemmas. This results in a top-down proof search whereas the tactics behave more efficiently when used in a bottom-up style. Last but not least, the extracted files tend to be rather large as they contain many unnecessary commands as e.g. all navigation commands in the proof state trees from which only a few are needed. Unfortunately, we do not have any means to extract shorter command scripts automatically. Therefore, we recommend creating a command script manually by inserting the successful commands within a separate editor session to a new command script file, or at least modifying the automatically extracted command script. During this step aborted commands as well as unnecessary commands should be deleted and the lemmas should be rearranged in a bottom-up style.

To guide the user during this last task of rearranging the commands we adopt *The Method* described in [10], chapter 9. *The Method* allows the user to perform a top-down proof search while arranging the lemmas in a bottom-up style. For doing this, *The Method* uses two lists separated by an imaginary line called *barrier*. The first list, the *done* list, contains already proved lemmas, whereas the second list, the *to-do* list, contains a proof plan of what has still to be done. By applying *The Method*, the user has to think about a proof for the first lemma of the *to-do* list. If there are any auxiliary lemmas for the proof missing in the *done* list, the user has to add these lemmas in front of the *to-do* list. Otherwise, the user tries to prove the first lemma of the *to-do* list by calling a simplification tactic or a proof strategy, and aborts the proof attempt if it takes too much time. If the lemma has been proved, the *barrier* is advanced behind the proved lemma. Otherwise, the failed proof attempt has to be analyzed as described in Section 4.3. This may result in a manual intervention leading to a successful proof, or another unforeseen auxiliary lemma that is inserted in front of the *to-do* list. After having proved and activated the auxiliary lemmas, the proof attempt of the original lemma is restarted.

[4]It is possible to abort the (non-terminating) command execution when reading a command script by sending the *keyboard interruption signal* 2 (`SIGINT`) from a unix shell directly to the lisp process. This will cause QuodLibet to enter the debugger from which the computation can be stopped.

5 Organization of the New Tactics

QML is a modularized imperative programming language adapted to the inductive theorem prover QuodLibet. This means that on the one hand there are special internal data structures for the proof objects of QuodLibet as e.g. terms, atoms, literals, clauses, axioms, lemmas, goal and inference nodes, as well as routines as e.g. the inference rules that influence the state of the inference machine, or routines that inform the tactics about the actual specification. On the other hand tactics and procedures of QML-modules can be made *public* so that they can be called within QuodLibet through the text-based or graphical user-interface.

In this section we will describe the organization of the new tactics in great detail from a user's point of view. This means that we will concentrate on the public tactics and explain their approach in proving inductive theorems. We try to give the user a feeling how the tactics work and especially how they can be influenced, but we will not go into detail about the implementation issues of the tactics.

The modularization concept allows for the structuring of the tactics in separate parts by the information hiding principle. We will use this system architecture to structure our description of the tactics. We start with the auxiliary modules `basics`, `complexity`, `alt-lit-rep`, `substitutions`, and `condition-trees` that do not export any public routines. After that we explain the functioning of the public modules `Default-Settings`, `Protected-Inference-Machine`, `Database`, `Simplification`, `Inductive-Case-Analyses` and `Proof-Strategies` that model the inductive proof process as described in Section 2.

Technically, a user has to *call* a public routine to start its execution. At the text-based interface this can be done for routine `rout` by the command `call rout`. Besides, for each public routine a menu item will be created at the graphical user-interface that allows its execution. These menu items are arranged according to the modules where the routines are defined. Menu items for procedures will be placed in the `QML`-menu of the main window whereas menu items for tactics will be placed in the `Tactics`-menu of the proof state tree windows since tactics always apply to the activated goal node of a proof state tree. Routines may need additional information to start their execution in a meaningful way. The procedure `activate-lemma` for instance depends on the lemma which is to be activated (see Section 5.4.3). This information is acquired by *obligatory parameters* that have to be supplied with values. Some routines also provide *optional keyword parameters* that enable the user to override the default behavior of the routines in a controlled way. These parameters will be supplied with a value only if the corresponding keyword is given beforehand. This mechanism with keyword parameters enables us to improve our tactics by a finer-grained control without invalidating former proof scripts. This is achieved by the invention of new keyword parameters that allow to distinguish between different behaviors of the routine making the former behavior the default behavior. At the text-based interface the command `call activate-lemma plus-com :activate-first-lit-p TRUE` will start the execution of routine `activate-lemma` with `plus-com` as the value for an obligatory parameter and `TRUE` as the value for the optional parameter *:activate-first-lit-p*. At the graphical user-interface the parameter values can be entered through special widgets. Widgets for optional parameters will only be displayed if the menu item for the routine is not selected with the left mouse button. This behavior takes into account that optional parameters are rarely used since they have sensible default values. In the following sections the public routines will be presented in QML-style. This means that the parameters will be enclosed in parenthesis and separated by commas. When calling a routine at the text-based interface the parenthesis and commas have to be omitted.

5.1 Auxiliary Modules

The module `basics` provides some basic functions that can be seen as extensions of the library functions provided by the core system. These functions neither depend on new data types nor do they use a state. This module contains on the one hand functions that extend matching routines or the application of substitutions from terms to literals and clauses; on the other hand there are functions that care for a special treatment of goals with weight variables. Furthermore, the simplification order that is used for permutative rewriting (see the explanation after Guideline 4.7 on page 17) is implemented in this module.

The module `complexity` provides a new data type for measuring the complexity of terms, literals and clauses, and for comparing complexities. This measurement can be used for controlling the pruning of proof attempts heuristically when trying to relieve proof obligations in conditional lemma application. At the moment the measures themselves are fixed, e.g. the complexity of a term is its length, whereas the user can decide whether to use complexity measures at all by changing the parameter *:check-condition-complexity* (see Section 5.2).

As already explained for Guideline 4.2 on page 15 equational literals for sorts with exactly two constructors that are constants may be represented in two different ways. Thus, for the boolean values `true` and `false` the literal $t = \texttt{true}$ has an alternative representation $t \neq \texttt{false}$ if t is a defined term. At least when checking the mandatory and obligatory literals in conditional rewriting (see Section 3) the tactics should be able to consider both representations. This will be done if a lemma with clause $\{ \ b = \texttt{true}, \ b = \texttt{false} \ \}$ for a constructor variable b has been activated. This lemma is needed due to technical reasons to convert one representation into the other. The computation of the different representations is implemented in the module `alt-lit-rep`.

The module `substitutions` contains functions for completing a cover set of substitutions as defined in [11] as well as merging two cover sets to compute an appropriate inductive case splitting that results from the expandable calls in a clause to be proved. These functions are inherited from the old tactics.

The module `condition-trees` provides functions to establish a case splitting by addition of literals (cut) introduced to rewrite a given operator call with the axioms of the top-level operator. On the one hand these functions are used for checking for the definedness property of an operator and on the other hand to compute an inductive case splitting at the beginning of an inductive proof attempt. For further details see the description of the old tactics in [11].

5.2 Module `Default-Settings`

There are many decisions that lead to a successful proof in one case but that are just time-consuming in another proof attempt. We tried to parameterize the tactics so that

Table 1: Optional parameters and their default values

Name	Type	Default	Section
:auto-insert-axioms-p	`Boolean`	`TRUE`	5.4.2
:speculate-domain-lemma-p	`Boolean`	`TRUE`	5.4.2
:activate-first-lit-p	`Boolean`	`FALSE`	5.4.3
:activate-left-to-right-p	`Boolean`	`TRUE`	5.4.3
:deactivate-old-p	`Boolean`	`TRUE`	5.4.3
:debug-p	`Boolean`	`FALSE`	5.5
:rule-type-order	`[rule-type_T`[a]`]`	RULES[b]	5.5
:allow-delayed-condition-check	`[appl-type_T`[c]`]`	ALLAPPLS[d]	5.5
:allow-new-operators-in-conditions	`[appl-type_T`[c]`]`	ALLAPPLS[d]	5.5
:check-condition-complexity	`[appl-type_T`[c]`]`	`[]`	5.5
:maximal-condition-depth	`Integer`	`-1`	5.5
:allow-perm-rewriting-p	`Boolean`	`TRUE`	5.5
:allow-free-vars-handling-p	`Boolean`	`TRUE`	5.5
:allow-extended-order-handling-p	`Boolean`	`TRUE`	5.5
:allow-extended-const-rewriting-p	`Boolean`	`TRUE`	5.5
:allow-alternative-free-var-bindings-p	`Boolean`	`TRUE`	5.5
:allow-cond-simplification-before-induction-p	`Boolean`	`TRUE`	5.7
:recursive-strategy-p	`Boolean`	`FALSE`	5.7

[a] `rule-type_T` := `IND | AX | LMA`

[b] RULES := `[IND, AX, LMA]`

[c] `appl-type_T` := `EQSUBS | NEGEQSUBS | DEFSUBS | ORDERSUBS | REWRITE`

[d] ALLAPPLS := `[EQSUBS, NEGEQSUBS, DEFSUBS, ORDERSUBS, REWRITE]`

the user can decide whether to use more elaborated proof search heuristics or to prune the proof state tree earlier. This behavior of the tactics can be influenced by many optional keyword parameters. Setting an optional parameter will change the behavior of the routine for the actual execution. If the optional parameters are not provided by the user default values will be used. The default value for an optional parameter can be changed in the module `Default-Settings` if there is a meaningful static alternative for that parameter. For example, the parameter *:head-litnbs* of procedure `activate-lemma` (see Section 5.4.3), that allows to specify the positions of the literals which should be used as head literals for the lemma, has no meaningful static alternatives since the literal positions depend on the activated lemma. On the other hand, for all boolean parameters `TRUE` or `FALSE` may be chosen as default value.

An overview of all keyword parameters whose default values can be changed in module `Default-Settings` is presented in Table 1. The table contains for each parameter its name, type and initial default setting, i.e. the value that is used after the procedure `reset-default-settings` has been called. These default settings are also restored when the database, that stores information about the analyzed operators and activated lemmas, is initialized (see Section 5.4.1). Furthermore, the table contains for each parameter a reference where the meaning of the parameter is explained in detail. The parameters can be divided into three groups according to the modules where they are used: parameters of module `Database` control the analysis of defined operators and the activation of lemmas, parameters of module `Simplification` influence the simplification process, especially the application of lemmas, whereas the other parameters affect the behavior of the strategies in module `Proof-Strategies`. As the proof strategies use the simplification tactics the optional simplification parameters can be set in the proof strategies as well. In this section we will only describe the procedures that allow to set and display the default settings of the parameters.

The module offers three public procedures to handle the default settings of these optional parameters:

- `reset-default-settings`() restores the predefined values for the optional parameters as presented in Table 1.
- `set-default-settings`(*:auto-insert-axioms-p*, ..., *:recursive-strategy-p*) allows to set the default setting for an arbitrary set of the optional parameters displayed in Table 1. The parameters that should be changed are themselves keyword parameters of this procedure.
- `display-default-settings`() displays the default settings of the optional parameters in the log window.

Note that the default settings for the optional parameters are intended to support unexperienced users. Thus, the computations based on these default values often work on a larger part of the search space that can be restricted by changing some of the optional parameters. This can prune the search space enormously but also lead to strange and unexpected failed proof attempts if the proof process is not understood very well.

For the sake of completeness, we just want to mention that there are two other procedures in this module. These procedures do not have any influence on the proof search but merely control the behavior of some outputs e.g. the internal display of the database (see Section 5.4.4). The output is controlled by the following parameters:

- *:write-level* (`-1`):
  This parameter indicates how deep nested structures, like lists and records, are printed. A value of 0 means that the output will be abbreviated with three dots. A positive value n allows the output of the outermost structure displaying the directly nested structure at level $n-1$. A negative value means that the whole structure is printed. Thus, by default all structures are printed completely.
- *:long-output-p* (`TRUE`):
  This parameter controls whether records are displayed in a long format that attaches to each entry the name of the entry. This setting is enabled by default to clarify the meaning of each record entry.
- *:indent* (`""`):
  The output is prefixed with the content of this parameter.

These write settings can be handled by the following procedures:

- `set-write-settings`(*:write-level*, *:long-output-p*, *:indent*) sets the write settings according to the given values.

- `display-write-settings`() displays the write settings in the log window.

5.3 Module `Protected-Inference-Machine`

The new tactics depend on information that is associated with goal or inference nodes of proof state trees. Whereas some information can be derived directly from the internal data structures of proof state trees provided by the core system, other information has to be stored by the tactics as e.g. the data for mandatory literals (see Section 3). Therefore, the tactics build up a data structure similar to proof state trees in module `Protected-Inference-Machine`. This data structure is more compact since only those nodes are explicitly created for which the stored information has changed in comparison to the predecessor node. For the maintenance of the data structure we have implemented wrappers for each predefined tactic. In the other modules only the wrappers are used instead of the original tactics.

The original tactics change the proof state trees by applying inference rules or deleting inference nodes. The wrappers add further functionality in a uniform way that has the following advantages:

- The repeated application of the same inference rule in one path of a proof state tree or the same top-level application as in an alternative proof path is prevented.

- The constraints for mandatory literals in the proof of definedness and condition subgoals will be checked automatically.

- In debug mode described at the end of Section 4.3 the physical deletion of failed proof attempts is stopped. This is realized in conjunction with another wrapper function that computes the open goal nodes in a proof state tree. This wrapper function will recognize those nodes that have been deleted but not physically removed from the proof state tree to get the same results with as without debug mode.

The module `Protected-Inference-Machine` provides two public tactics to clean up a proof state tree computed in debug mode:

- `delete-goal-proof-attempts`(*goal*) deletes every alternative proof attempt but the last in the subtree starting at the given goal node *goal*.

- `delete-root-proof-attempts`(*goal*) deletes every alternative proof attempt but the last in the whole proof state tree of goal node *goal*, which remains the current goal node in the proof state tree.

5.4 Module `Database`

The database stores all information about analyzed operators and activated lemmas. The information about operators is used for creating the initial case splitting in an inductive proof whereas the activated lemmas are used for rewriting and subsumption during the simplification process. We will divide the description of the public procedures in this module into five sections: the initialization of the database in Section 5.4.1, the analysis of operators in Section 5.4.2, the activation of lemmas in Section 5.4.3, the display operations of the database in Section 5.4.4 and saving the whole state of the database in a file in Section 5.4.5.

5.4.1 Initializing the Database

Before using the tactics the state of the database represented by global variables has to be initialized. This is done by calling the following procedure:

- `initialize-database`() will reset the state of the database as well as the default settings for optional parameters as described in Section 5.2.

Without doing this the tactics will behave unexpectedly, mainly because they cannot initialize the default settings automatically in a meaningful way.

5.4.2 Analyzing Operators

The analysis of a defined operator — more precisely of the axioms that belong to the defined operator — by the old tactics together with the resulting definition scheme of the operator is presented at length in [11]. As we have been satisfied so far with the inductive case splittings stemming from these definition schemes, we have only made minor technical improvements w.r.t. the separation of the axioms and the termination heuristics. Since we are essentially interested in presenting the achievements of the new tactics we will only give a short summary of the definition scheme of a defined operator.

The definition scheme of a defined operator f consists of

- a *normalized representation* of the axioms of f achieved by renaming variables to minimize the number of different variables used.
- a *cover set of substitutions* for the operator that can be used for generating an inductive case splitting for goal clauses like $\{\texttt{def}\ f(x_1, \ldots, x_n)\}$ where n is the arity of f and x_i is a constructor variable for $1 \leq i \leq n$.
- a list of *termination witnesses* if f is recursive and supposed to be terminating by the termination heuristics. These termination witnesses are argument positions that become smaller in a wellfounded order in each recursive call of f. They are used for guessing an instantiation of a weight variable in an inductive proof.
- a list of *induction positions* if f is conjectured to be terminating. The induction positions are those argument positions of f that are changed by the axioms of f. They are used for determining the expandable and obstructed calls in a goal clause with different defined operators to merge different induction schemes into one inductive case splitting.

A major improvement of the new tactics is the application of *weight modifiers* in the termination heuristics. This means that not only lexicographical combinations of constructor variables used in the axioms of f are considered during the termination analysis, but also combinations using constructor variables modified by a sequence of unary defined operators $f_1, \ldots, f_n$ if an appropriate order lemma has been activated before. Such an order lemma has the form

$$\{\ f_1(\ldots(f_n(t_1))\ldots) < c_1(\ldots(c_m(f_1(\ldots(f_n(t_2))\ldots)))\ldots), \quad l_2, \quad \ldots, \quad l_k \quad \},$$

where t_i are terms, c_j are constructors and l_p are literals. We will illustrate the notion of weight modifiers by the following example about a specification of merge sort.

Example 5.1

Let the specification consist of sorts `Bool` and `Nat` as given in Example 3.1 as well as sort `List` for lists over natural numbers with the constructors `nil` for the empty list and `cons` to construct a list consisting of a natural number and a tail list.

The recursive step in the definition of an operator `mergesort` on lists over natural numbers can be defined as follows:

$$(26)\quad \mathtt{mergesort}(\mathtt{cons}(x,\mathtt{cons}(y,l))) = \mathtt{merge}(\mathtt{mergesort}(\mathtt{split1}(\mathtt{cons}(x,\mathtt{cons}(y,l)))),\ \mathtt{mergesort}(\mathtt{split2}(\mathtt{cons}(x,\mathtt{cons}(y,l)))))$$

where `merge` merges two sorted lists and `split1` (respectively `split2`) filters the elements of a list on odd (respectively even) positions.

Provided two order lemmas, using an operator `length` that computes the length of a list, have been activated before with clauses

$$(27)\quad \{\ \mathtt{length}(\mathtt{split1}(l)) < \mathtt{length}(l), l = \mathtt{nil}, \mathtt{length}(l) = \mathtt{s}(\mathtt{0})\ \}$$

$$(28)\quad \{\ \mathtt{length}(\mathtt{split2}(l)) < \mathtt{length}(l), l = \mathtt{nil}\ \}$$

resulting in a weight modifier `length` for sort `List`, the termination heuristics consider `mergesort` terminating with termination witness 1 (the only argument of `mergesort`) and weight modifier `length`. Because of the given lemmas

$$\mathtt{length}(\mathtt{split1}(\mathtt{cons}(x,\mathtt{cons}(y,l)))) < \mathtt{length}(\mathtt{cons}(x,\mathtt{cons}(y,l))) \quad \text{and}$$
$$\mathtt{length}(\mathtt{split2}(\mathtt{cons}(x,\mathtt{cons}(y,l)))) < \mathtt{length}(\mathtt{cons}(x,\mathtt{cons}(y,l))).$$

Therefore, the arguments of the recursive calls of `mergesort` modified by the weight modifier `length` are smaller w.r.t. the induction order than the argument of `mergesort` on the left-hand side of the axiom.

Beside of using weight modifiers, the termination heuristics now also apply a more involved algorithm to compare term tuples by using unconditional activated lemmas to compute normal forms of weight terms and to delete additional constructors on the right-hand side of the order atom before trying to find an order lemma that subsumes the order atom. As this algorithm is rather sophisticated we will not explain it in detail here.

There are two procedures for gathering information about defined operators in the database:

- `analyze-operator`(*op*, *:auto-insert-axioms-p*, *:speculate-domain-lemma-p*) creates the definition scheme for the defined operator *op*. Note that due to technical reasons a defined operator has to be analyzed before a definedness or rewrite lemma can be activated for this operator.

- `deactivate-operator`(*op*) removes the entry about the defined operator *op* from the database. This also removes all activated definedness and rewrite lemmas of the operator.

The first procedure is parameterized by two optional keyword parameters whose default values can be set in module `Default-Settings` (see Section 5.2):

- *:auto-insert-axioms-p*: If this parameter is enabled the axioms of the operator will be activated automatically for the simplification process by calling the procedure `activate-axioms` (see Section 5.4.3). The activation is done in a way that the first axioms specified will be prefered during simplification as we are used to enter the easier axioms (e.g. the base case of a recursive definition) first.

- *:speculate-domain-lemma-p*: If this parameter is enabled the procedure tries to speculate a *definedness* lemma (called *domain* lemma in [11]).

5.4.3 Activating Lemmas

Whereas the information about analyzed operators is used for generating an inductive case splitting, the activated lemmas are applied as rewrite or subsumption lemmas during the simplification process as explained in Section 3. Besides, activated lemmas are used for the computation of alternative literal representations as described in Section 5.1 about the auxiliary module `alt-lit-rep`, and for the termination heuristics as explained in Section 5.4.2.

All these applications are restricted by the head and obligatory literals the lemma is activated for as motivated in Section 3. These literals can be given by the user during the activation of the lemma; otherwise, they are computed by some heuristics. The activation can be influenced by the following keyword parameters:

- *:deactivate-old-p*: If this parameter is enabled all former activations of this lemma will be removed from the database; otherwise, the new activation is added. This enables the user to specify different obligatory literals for each head literal by activating the lemma more than once without removing the old activation.

- *:head-litnbs*: If given, this parameter contains a list of literal positions that are to be used as head literals for the lemma. Otherwise, the head literals will be computed according to parameter *:activate-first-lit-p*.

- *:obl-litnbs-list*: If given, this parameter contains one or more lists of literal positions that are used as obligatory literals for the lemma. This means that at least one of the lists of obligatory literals has to be directly fulfilled in a goal clause before the lemma can be applied to the goal automatically. If this parameter is not given by the user, no obligatory literals will be used with only one exception: If the lemma is activated as a rewrite rule with a general term as left-hand side (i.e., a term that consists of an operator call with different variables at all argument positions), then the next literal is chosen as obligatory literal provided that the clause has more than one literal. This heuristic helps to reduce the search space drastically, as rewrite rules with general terms can be applied very often. On the other hand, it should be checked whether this restriction is too strong if a proof attempt fails.

- *:activate-first-lit-p*: This parameter is only relevant if no head literals have been provided by the user. It distinguishes between two different heuristics to find a head literal automatically. If *:activate-first-lit-p* is enabled, the first equational, negated equational, order or definedness atom in the lemma clause is chosen as head literal. Otherwise, the heuristics proceed as follows: If there is just one order atom or one definedness atom this atom is chosen; if there is more than one order or definedness atom the procedure fails, i.e., the lemma is not activated at all. This failure is justified as there are no means to prove negated definedness or negated order atoms apart from using them as complementary literals, but these kinds of literals will become mandatory literals for a condition subgoal when we apply such a lemma that contains more than one order or definedness atom. If the lemma contains neither order nor definedness atoms, a (negated) equational atom is chosen that binds most of the variables of the lemma. To be more precise, the first equational atom of these literals is chosen if there is one such atom; otherwise the first negated equational atom is taken. In other words, if the procedure does not fail exactly one atom is chosen as head literal prefering order and definedness atoms over equational over negated equational atoms that bind most of the variables in the clause.

- *:activate-left-to-right-p*: If this parameter is enabled equational atoms that can be used as rewrite rules will be activated from left to right. Otherwise, they will be activated from right to left.

For each head literal the lemma is assigned one *application type*. In contrast to the old tactics, where there were special restrictions for rewrite, domain and induction lemmas, we classify the lemmas just by the syntactic form of the head literal:

- If the head literal is a definedness atom the application type is *subsumption lemma for definedness atoms* (`DEFSUBS`);

- if the head literal is an order atom it is a *subsumption lemma for order atoms* (`ORDERSUBS`);

- if the head literal is a negated equational atom it is a *subsumption lemma for negated equational atoms* (`NEGEQSUBS`);

- if the head literal is an equational atom with a variable as left-hand side it is a *subsumption lemma for equational atoms* (`EQSUBS`);

- if the head literal is an equational atom that does not have a variable as left-hand side it is a *rewrite lemma* (`REWRITE`).

Note that there may be rewrite lemmas for defined operators as well as for constructors, i.e., the top-level operator of the left-hand side of the head literal may be a defined operator or a constructor. The latter may occur when proving properties about the interplay of constructors and destructors as demonstrated by lemma

(29) $\{\ \mathtt{cons}(\mathtt{car}(l), \mathtt{cdr}(l)) = l,$
$\quad l = \mathtt{nil}\ \}$

Since the rewrite rules are typically not confluent, the order in which the lemmas are applied does matter not only for efficiency reasons but also for finding a proof at all. As explained in Section 5.5 the user can influence the order in which axioms (`AX`) and lemmas are applied inductively (`IND`) or non-inductively (`LMA`), respectively, by setting the parameter *:rule-type-order*, whereas the order in which the term is tested for rewrite positions is fixed by an innermost strategy left-to-right. Besides, subsumption lemmas are prefered in comparison to rewrite lemmas as they close the proof path, and directly applicable lemmas are prefered, because they normally lead to simpler proofs. Within each group of activated axioms, inductive lemmas and non-inductive lemmas the lemmas are tested recent first, so that newer lemmas are prefered. The user can change the activation order by one of the procedures presented below.

To conclude the description of the activation mechanism we list the procedures for activating lemmas along with their (keyword) parameters:

- `activate-lemma`(*lemma*, *:head-litnbs*, *:obl-litnbs-list*, *:activate-first-lit-p*, *:activate-left-to-right-p*, *:deactivate-old-p*) activates the given *lemma* non-inductively.

- `activate-lemmas`(*lemmas*, *:activate-first-lit-p*, *:activate-left-to-right-p*, *:deactivate-old-p*) activates the list of *lemmas* non-inductively in the given order so that the last lemmas (and thus most recently activated lemmas) will be prefered during simplification.

- `change-lemmas-order`(*lemmas*, *:activate-first-lit-p*, *:activate-left-to-right-p*) changes the order of activated non-inductive lemmas according to the list *lemmas*. Former non-inductive lemma activations that do not appear in *lemmas* are deactivated, whereas new lemmas are activated automatically.

- `deactivate-lemmas`(*lemmas*) deactivates all non-inductive lemma activations in the list *lemmas*.

- `activate-ind-lemma`(*lemma*, *:head-litnbs*, *:obl-litnbs-list*, *:activate-first-lit-p*, *:activate-left-to-right-p*, *:deactivate-old-p*) activates the given *lemmas* inductively.

  The other corresponding procedures for non-inductive lemma activations are omitted for inductive lemmas as they are integrated into the simplification tactics and proof strategies directly.

- `activate-axiom`(*axiom*, *:head-litnbs*, … *:deactivate-old-p*),
  `activate-axioms`(*axioms*, *:activate-first-lit-p*, … *:deactivate-old-p*),
  `change-axioms-order`(*axioms*, *:activate-first-lit-p*, *:activate-left-to-right-p*) and
  `deactivate-axioms`(*axioms*)

correspond to the described procedures for non-inductive lemma activations replacing lemmas by axioms.

Note that directly after an inductive case spitting axioms are applied regardless whether they are activated: Since the inductive case splitting is done to apply certain axioms this will not be prevented by deactivating them.

5.4.4 Displaying the Database

To inform the user about the state of the database the module offers some output procedures. These can be divided into procedures presenting a succinct and more readable view on the database by omitting technical details, and those providing insights into the internal data structures that represent the state of the database.

- `display-database`() displays the contents of the whole database in a succinct and more readable form.
- `display-operator-info`(*op*) displays the information gathered for the defined operator *op* in a succinct form containing the information of the analysis of the operator as well as all definedness and rewrite lemmas activated for the operator.
- `display-ctr-operator-info`(*op*) displays the information gathered for the constructor *op* in a succinct form. This information is non-empty only if a rewrite rule has been activated with the constructor as top-level operator of the left-hand side as e.g. in lemma (29) for constructor `cons`.
- `display-internal-database`() displays the contents of the internal data structures of the whole database.
- `display-internal-operator-info`(*op*) displays the contents of the internal data structure of the defined operator *op* corresponding to the external view as provided by `display-operator-info`.
- `display-internal-ctr-operator-info`(*op*) displays the contents of the internal data structure of the constructor *op* corresponding to the external view as provided by `display-ctr-operator-info`.

The output of the internal view can be influenced by setting the write settings, e.g. the *:write-level*, as explained in Section 5.2.

5.4.5 Saving the Database

As explained in Section 4, there are two ways to automatically extract a file that contains information about the current proof session:

- In a *command script* every user interaction with the prover will be stored. This normally leads to a huge file that contains many irrelevant commands unless the script has been edited manually as described in Section 4.4. A command script documents the proof process in terms of the called tactics. Thus, the whole proof search that may be very time-consuming is redone when the file is read in. On the other hand, this mechanism guarantees that not only the state of the core system but also the state of the QML-modules like `Default-Settings` and `Database` is the same as at the end of the proof session that is stored[5].

- In a *state file* the state of the core system represented by the current specification and the actual proof state trees will be stored. For every proof state tree the applied inference rules that have not been deleted will be stored regardless whether they were applied manually or automatically during the execution of a tactic. Therefore, the process that has led to the proof state tree is lost but a state file will be read in much faster than an equivalent command script since no proof search is necessary. The operation of creating a state file is provided by the core system. Thus, a state file only restores the state of the core system and not the state of the QML-modules.

To overcome this latter limitation of saving a state file, the database offers the following wrapper procedure:

- `output-state`(*filename*) saves the whole state of the core system as well as of the QML-modules in a single file named *filename*.

We did not want to implement procedures just for restoring the state of the QML-modules. Thus, in such a state file we use the procedures described so far. Whereas this does not cause any problems for module `Default-Settings` it may lead to wrong results for module `Database` since e.g. the result of an operator analysis depends on the activated lemmas. We try to overcome this problem by activating all global lemmas first. After that we analyze each operator in the same order as stored in the database, thereby activating all rewrite and definedness lemmas of this operator. Theoretically, this may still lead to wrong results as some lemmas may have been deactivated during the proof session that are needed for the analysis of an operator. But as we have not recognized this problem in practice we did not care for a better book keeping mechanism of lemmas that were activated in the past.

5.5 Module `Simplification`

The simplification process has been completely restructured in comparison to the old tactics dividing it into several passes as already mentioned in Section 2. One of the main improvements is achieved by applying conditional lemmas that are not directly applicable. These applications are controlled by obligatory and mandatory literals as described in Section 3. We will now explain the task of each pass during the simplification process without going into technical details. The implementation of each pass depends on the kind of atom of the considered literal, i.e., whether it is an equational atom, an order atom, a definedness atom or a negation of one of them. The public interface of this module provides tactics to perform one pass for one selected literal or for all literals of a goal as well as a combination of all passes for one or all literals of the given goal or of all open children goals of the given goal, respectively. This process may also be performed recursively as long as any simplification is

[5]This only holds if the command script does not contain any calls of non-terminating tactics.

possible. The public tactics as well as their keyword parameters will be described in more detail at the end of this section.

The whole simplification process is called `simplify` and it is divided into the following five[6] passes:

- `prove-taut`: This pass proves simple tautologies that can be shown by a single application of one inference rule without using any lemmas. To be more precisely, the inference rules for simple tautologies `compl-lit`, $\neq$`-taut` and $<$`-taut`, as well as the inference rules for decomposing atoms $=$`-decomp`, `def-decomp` and $<$`-decomp` are applied as long as they do not produce any new subgoals. Certainly, for each kind of atom only those inference rules are tried which are sensible for that kind.

- `remove-redundant`: During this pass all inference rules that remove redundant literals, i.e., `mult-lit`, $=$`-removal`, $<$`-removal`, $\neq$`-removal` and `~def-removal`, are tried.

- `reduce-pass1`: This pass tries to apply non-permutative lemmas that are directly applicable in the goal clause. If the goal clause can be subsumed by one lemma, the goal is proved; otherwise, a normal form w.r.t. the directly applicable rewrite lemmas is computed testing for simple tautologies and redundant literals after each rewrite step. This pass has been split from `reduce-pass2` to prefer directly applicable lemmas as they contribute to easier proofs. Technically, the different handling of lemmas is achieved by temporarily changing the values of the keyword parameters *:allow-delayed-condition-check* to the empty set `[]` and *:allow-perm-rewriting-p* to `FALSE`.

- `reduce-pass2`: During this pass a great effort is undertaken to prove the goal with the considered literal. Nearly all inference rules (except `const-rewrite`), that have not been applied during the first three passes, are attempted. There are some special *macro inference steps*, i.e., sequences of inference steps that are linked together to simplify goals that contain a special pattern. They are used e.g. for removing constructor prefixes of definedness atoms and equational literals as well as for guessing intermediate weights for order atoms. Above all, the main focus of this pass is on applying subsumption and (possibly permutative) rewrite lemmas even if they are not directly applicable in the goal clause taking into account the obligatory literals of the lemma and the mandatory literals of the goal as explained in Section 3. During this pass some inference rules are tested simplifying their derived subgoals by the whole simplification process recursively. But if some of the conditions, represented by the subgoals, cannot be established, the whole subtree is deleted again. This guarantees that at most one goal (without counting goals with order atoms containing weight variables) results from this pass.

  Note that this pass can be essentially influenced by keyword parameters (see below) and the activation of lemmas in the database as described in Section 5.4.

- `cross-fertilize`: So far, this pass affects only negated equational atoms. They are used for replacing in another literal the occurrence of one side of the negated equation

[6] As there are no means to prove negated definedness or negated order atoms apart from using them as complementary literals, they are not considered by the last three passes at all.

with the other side by applying the inference rule `const-rewrite`. To prevent infinite loops the tactics do not perform applications that undo former ones. Furthermore, the application of this pass is controlled by some heuristics to obtain "simpler" literals. These heuristics apply the following rules until the first succeeds:

- If one of the terms is a subterm of the other, the other term will be replaced with the subterm.
- Otherwise, if one term is a variable, it is replaced with the other term. The other term represents the definition of the variable.
- Otherwise, if one of the terms is "simpler" than the other term w.r.t. a fixed simplification order, the other term is replaced with the "simpler" term. As simplification order we use an LPO on ground terms regarding variables as constants. The precedence is defined by three levels only: constructors are smaller than defined operators which are in turn smaller than variables. This order prefers terms that have less variables and more constructors (at top-level positions).
- Otherwise, i.e., if the two terms are equal with respect to the simplification order, it is tried to arbitrarily replace one term with the other as long as this does not retract a former application. This random choice may help to prove some goal clauses in cases where all other rules do not apply.

The simplification process can be fine-tuned by the following keyword parameters that essentially allow to restrict the application of lemmas and other inference rules:

- *:ind-lemmas*: With this parameter, the user can specify the lemmas that should be used as induction hypotheses during the simplification process. The lemmas will be activated as induction hypotheses analogously to procedure `change-lemmas-order` for non-inductive lemmas (see Section 5.4.3). Thus, every lemma in *:ind-lemmas* that has been activated before will be kept as is, new lemmas will be activated automatically whereas old activations will be removed if the lemmas are not in *:ind-lemmas*. If this parameter is not given, all former inductive lemma activations will be removed. The activation can be controlled further by the keyword parameters *:activate-first-lit-p* and *:activate-left-to-right-p* as explained in Section 5.4.3.

- *:debug-p*: This parameter controls the debug mode as explained at the end of Section 4.3.

- *:rule-type-order*: One of the main steps when simplifying a goal is to apply an axiom (`AX`) or lemma — inductively (`IND`) or non-inductively (`LMA`) — to rewrite or subsume the goal clause. With this parameter the user can influence the order in which axioms and lemmas are searched for.

- *:allow-delayed-condition-check*: This parameter constitutes whether lemmas, that are not directly applicable, should be considered for rewriting or subsumption. The decision can be made separately for each kind of lemma, i.e. subsumption lemmas for equations (`EQSUBS`), for negated equations (`NEGEQSUBS`), for definedness atoms (`DEFSUBS`) and for order atoms (`ORDERSUBS`) as well as for rewrite lemmas (`REWRITE`).

- *:allow-new-operators-in-conditions*: The introduction of new operators while proving conditions of a conditional lemma application can blow up the search space. This parameter enables the user to disallow the invention of new operators for each kind of subsumption or rewrite lemma.

- *:check-condition-complexity*: This is another parameter to reduce the effort used for proving condition subgoals by disallowing these subgoals for literals that are complexer than the complexest literal in the goal clause. At the moment the complexity of a term is measured by its length, the complexity of a literal as the complexity of the largest term in the literal, and the complexity of a clause as the complexity of the largest literal in the clause. It can be enabled for each kind of rewrite or subsumption lemma separately.

- *:maximal-condition-depth*: This parameter restricts the maximal recursion depth for applying conditional lemmas that are not directly applicable in the proof of condition subgoals of (other) conditional lemmas. A value of 0 means that only directly applicable lemmas can be applied. A positive value n allows the application of lemmas whose conditions can be proved by lemma applications with maximal condition depth $n-1$. A negative value stands for an unrestricted use of conditional lemma applications.

- *:allow-perm-rewriting-p*: As already explained after Guideline 4.7 on page 17, a permutative lemma is an equational lemma with a head literal $l = r$, where l is a variable renaming of r, i.e., there exists a substitution σ with $\sigma(l) \equiv r$ that only renames variables. The commutativity of `plus` (see lemma (15)) is an example for a permutative lemma. The tactics use a fixed total simplification order to control the rewriting process with permutative lemmas which prevents infinite loops in most cases. Besides, the parameter *:allow-perm-rewriting-p* allows the user to disable rewriting with permutative lemmas at all.

- *:allow-free-vars-handling-p*: When applying a lemma to a goal clause as rewrite or subsumption lemma each variable in the lemma has to be instantiated. For most of the variables this is done by matching the head literal of the lemma to the focus literal of the goal. Variables of the lemma that are not instantiated by this matching operation are called *free* variables of the lemma. If this parameter is disabled lemmas containing free variables will not be considered for rewriting and subsumption. Otherwise, the tactics try to instantiate these free variables e.g. by matching condition literals of the lemma containing free variables to context literals of the goal.

- *:allow-extended-order-handling-p*: In comparison to the old tactics the new ones use the inference rules `<-mono` and `<-trans` (see [11]) as a last resort when trying to prove order atoms by guessing a new intermediate weight tuple. As this can enlarge the search space the application of these inference rules can be disabled by parameter *:allow-extended-order-handling-p*.

- *:allow-extended-const-rewriting-p*: This parameter controls whether the inference rule `const-rewrite` will be used during pass `cross-fertilize`. In the current state of development, `cross-fertilize` has no effect when this parameter is set to false.

- *:allow-alternative-free-var-bindings-p*: It is possible to get different instantiations of free lemma variables by matching lemma literals to different goal literals. This parameter controls whether alternative bindings of free variables are considered.

This module provides several public tactics that are named by a fixed naming scheme. The first part of the name stands for the pass of the simplification process whereas the last part specifies to which literal(s) the pass is applied. All tactics may be given the keyword parameters listed above. These keyword parameters are represented by dots after the obligatory parameters in the following description of the tactics:

- `simplify`(*goal*, ...) calls the tactic `simplify-all-lits` for *goal* and each derived subgoal as long as there is any change. This produces goals that are stable under simplification.

- `simplify-open-subgoals`(*goal*, ...) simplifies all open subgoals of the last proof attempt for *goal* by calling `simplify`. This tactic is useful after an inference rule has been applied manually to continue the simplification process for all derived subgoals.

- `simplify-all-lits`(*goal*, ...) simplifies the given *goal* by applying each pass to all literals starting with the first pass for all literals. The literals are considered in the same order as they appear in the goal clause beginning with the first literal.

- `simplify-lit`(*goal*, *litnb*, ...) simplifies the literal at position *litnb* of the given *goal* by applying all of the five passes.

- `prove-taut-lit`(*goal*, *litnb*, ...),
  `remove-redundant-lit`(*goal*, *litnb*, ...),
  `reduce-pass1-lit`(*goal*, *litnb*, ...),
  `reduce-pass2-lit`(*goal*, *litnb*, ...) and
  `cross-fertilize-lit`(*goal*, *litnb*, ...)
  perform the corresponding pass for the literal at position *litnb* of the given *goal*.

- `prove-taut-all-lits`(*goal*, ...),
  `remove-redundant-all-lits`(*goal*, ...),
  `reduce-pass1-all-lits`(*goal*, ...) and
  `reduce-pass2-all-lits`(*goal*, ...)
  perform the corresponding pass for all literals of the given *goal*.

- `cross-fertilize-first-lit`(*goal*, ...) performs the corresponding pass for the first literal to which it can be applied. Since cross fertilization is used as a last resort with a high probability of a misleading proof attempt it is stopped after the first literal.

In almost all cases one of the first two tactics is to be applied to simplify the goal clauses as far as possible. Only in special cases, e.g., if it is known that the first passes will fail for example because of an earlier (failed) proof attempt, it is suitable to call one of the tactics that start one pass separately to get a better performance.

5.6 Module `Inductive-Case-Analyses`

The automatic inductive case analysis, which is in principle inherited from the old tactics, is discussed at length in [11]. Therefore, we give only a short summary and concentrate on the new interface to QuodLibet.

Before an inductive case splitting can be generated by the tactics, each defined operator of the goal clause has to be analyzed so that a definition scheme has been created in the database. This information is needed for the inductive case analysis, e.g. the induction positions that indicate those argument positions of a defined operator that are changed by the axioms of that operator (see Section 5.4.2). The inductive case splitting aims at rewriting the resulting subgoal clauses by axioms to simplify them. Hence, the tactic searches for *effective* operator calls, i.e., operator calls that have distinct constructor variables at all induction positions. To be able to apply the axioms of the operator, these variables have to be instantiated according to the induction scheme of the operator. But this is only done if the operator call does not *obstruct* other operator calls, i.e., if one of the induction variables has an occurrence in another operator call that is not in an induction position for this operator. As these obstructing operator calls can cause redundant inference steps they are not taken into account by the inductive case analysis. The induction schemes of the remaining effective operator calls that do not obstruct other operator calls are then merged into one induction scheme, and the resulting subgoals are simplified by the axioms of these operators.

The module provides three tactics to generate an inductive case splitting automatically or by the assistance of the user:

- `ind-case-analysis`(*goal*) produces an inductive case splitting for the given *goal* automatically according to the description above.
- `expand-operators`(*goal*, *litnbs*, *litposl*) generates an inductive case splitting for the given *goal* semi-automatically taking into account only the effective operator calls at the given term positions in the goal. A term position is composed of a number, representing the position of the literal in the goal clause, and a position within the literal. Due to technical reasons the literal numbers and positions within the literals are given by two lists *litnbs* and *litposl*, respectively. These lists must have equal size, and the corresponding positions must be stored in the same order in each lists.
- `expand-variables`(*goal*, *variables*) generates an inductive case splitting for the given *goal* manually using the given *variables* as induction variables. These variables are instantiated by a minimal cover of constructor terms for the sort of the variable. The variable x of sort `Nat` will be e.g. instantiated by the terms `0` and `s`(x). Since no operator calls are specifically marked, no axioms are applied automatically after the inductive case splitting.

5.7 Module `Proof-Strategies`

The whole proof process is implemented by proof strategies that enable the user to prove (simple) inductive lemmas automatically. As already mentioned in Section 2 there are

12 different proof schemes depending on the way the induction analysis is performed, the lemmas that are activated as induction hypotheses, and whether the strategy is recursive.

The proof strategies can only be applied to root goal nodes of proof state trees since only these goal clauses can be applied inductively. First the proof strategies try to simplify the goal clause. If the goal clause is proved during this initial simplification process the proof process stops successfully. Otherwise, if the original goal has been simplified, for each open subgoal a new proof state tree is created (regardless whether the strategy is recursive). All open proof state trees are then processed successively by establishing an inductive case splitting according to the selected method (automatic, semi-automatic, manual) and simplifying the resulting subgoals, possibly applying the given inductive lemmas or creating new proof state trees if the strategy is recursive. If this process succeeds in proving all open subgoals except those containing weight variables due to an inductive application, the weight variables are attempted to be instantiated and the resulting order atoms are proved.

For the instantiation of the weight variables a procedure has been implemented in this module. This procedure is called automatically at the end of the proof process if all other open subgoals have been proved before. Thus, it has to be called manually by the user only if the proof attempt gets stuck e.g. due to some missing lemmas. Even in this case, the user may decide to call the tactic `cont-proof-attempt` instead (see below).

- `set-weights`(*conj*) tries to instantiate the weight variable of the given lemma *conj* so that all resulting order atoms can be proved. This procedure should only be called after all inductive applications have been performed to be able to consider all order constraints.

The tactics of this module allow the user to fine-tune the proof process by the keyword parameters of the simplification process (see Section 5.5). The only difference is that the root of the considered proof state tree will be used as induction hypothesis if no induction hypotheses are given by keyword parameter *:ind-lemmas*. Regardless whether *:ind-lemmas* are provided, each proof state tree that is created during the execution of the tactics will be activated as induction hypothesis.

The proof strategies `auto-strategy`, `operators-strategy` and `variables-strategy` provide two additional keyword parameters:

- *:allow-cond-simplification-before-induction-p*: If this parameter is enabled even conditional lemmas that are not directly applicable are considered for rewriting and subsumption during the initial simplification process before an inductive proof attempt is started.

- *:recursive-strategy-p*: This parameter controls whether the strategy is recursive.

All these optional parameters will be abbreviated by dots in the following description of the tactics.

Due to the different parameters needed to create inductive case splittings automatically, semi-automatically or manually, there are three parameterized strategies that implement the 12 proof schemes.

- `auto-strategy`(*goal*, ...) performs the proof process described above using an automatic case analysis.

- `operators-strategy`(*goal*, *litnbs*, *litposl*, ...) applies the proof process described above performing a semi-automatic case analysis. The parameters *litnbs* and *litposl* are used as positions for the effective operator calls that have to be considered (see tactic `expand-operators` in Section 5.6).

- `variables-strategy`(*goal*, *variables*, ...) applies the above proof process performing a manual case analysis for the given *variables* (see tactic `expand-variables` in Section 5.6).

If not all open subgoals can be simplified automatically the user has to support the tactics e.g. by applying inference rules manually or by providing auxiliary lemmas. After this manual interaction it is useful to continue the proof attempt automatically, possibly instantiating the weight variables at the end of the proof to finalize it. Therefore, we offer one further tactic comparable to the tactic `simplify-open-subgoals` in Section 5.5. In contrast to this tactic the following tactic tries to simplify not only the open successor subgoals of the given goal but all open subgoals of the whole proof state tree. Furthermore, the tactic attempts to instantiate the weight variables if all other subgoals have been proved.

- `cont-proof-attempt`(*goal*, ...) continues the proof attempt by simplifying all open subgoals of the proof state tree which the given *goal* belongs to.

6 Case Studies

We have tested our new tactics with some examples for the old tactics described in [11] and [8]. The examples in [11] range from some easy specifications about arithmetic operators on natural numbers like addition and multiplication, about operations on lists over natural numbers as e.g. a merge sort algorithm, and about search trees, to some specifications that illustrate special features of the specification language as e.g. a partially defined subtraction operation, the computation of the Euclidian quotient defined by destructor recursion and a non-terminating definition of a division operation on natural numbers. These examples have been extended by further sorting algorithms like bubble sort or quick sort in [8]. We will compare the results of these examples with those of our new tactics w.r.t. the degree of automation as well as the time consumed. Besides, we have performed some more challenging case studies that were out of scope of the old tactics because of their weaker simplification process. We have proved properties that involve a more intensive treatment of arithmetic expressions like the irrationality of $\sqrt{2}$ and properties about the greatest common divisor (`gcd`) of two natural numbers as e.g. its commutativity and associativity. Furthermore, we have performed case studies about mutually recursive functions proving that the lexicographical path ordering (`Lpo`) is a simplification order and showing the equivalence of different definitions. Besides, we have proved the `exp-exhelp` example presented in [9] that states the equivalence of call by value and call by name evaluations for simple arithmetic expressions containing function calls.

Version	Lemmas	Manual Appl.	Weight	Automatic Appl.	Deletions	Runtime
Old	79 (+3)	38	5	1154	—	1.53
Improved	78 (+2)	15	3	2478	1155	5.24
New	81 (+3)	4	1	1221	33	2.02

Table 2: Comparison of the different tactic versions for the basic examples in [11]

Version	Lemmas	Manual Appl.	Weight	Automatic Appl.	Deletions	Runtime
Old	93	116	2	2296	—	3.46
Improved	93	49	2	7224	4527	37.56
New	111	1	0	2210	60	3.67

Table 3: Comparison of the different tactic versions for the sorting algorithms in [8]

In [12] a first attempt was made to improve the tactics by providing an extended handling of order atoms and permutative rewriting as well as allowing the application of conditional lemmas that are not directly applicable. This implementation has shown the potential of automation w.r.t. these aspects. But as it is based on the old simplification process that does not provide a general purpose simplification tactic to simplify all kinds of atoms as far as possible or to apply lemmas inductively if needed, the degree of automation is restricted. And — even worse — as it does not provide appropriate restrictions for applications of conditional lemmas, it lacks efficiency. This can be seen below in Table 2 for the examples from [11] and in Table 3 for the sorting algorithms insertion sort, merge sort, quick sort and bubble sort from [8] where the three different versions (i.e., the old tactics of [11], the improved tactics of [12] and our new tactics) are compared w.r.t. the following aspects:

- *Lemmas*: the number of lemmas introduced into the specification, which can be seen as a measure of the complexity of the specification w.r.t. to other examples. If any lemmas have been generated automatically by a recursive proof strategy, they are added in parenthesis.

- *Manual Appl.*: the number of manually applied inference rules to complete the proofs of the lemmas in the specification.

- *Weight*: the number of times the weight variable has to be instantiated manually to get an appropriate induction order.

- *Automatic Appl.*: the number of inference rules successfully automatically applied by the called tactics.

- *Deletions*: the number of inference rules deleted due to a failure in relieving a condition subgoal of a conditional lemma. For the old tactics this parameter is meaningless as they do not apply conditional lemmas that are not directly applicable.

- *Runtime*: the runtime of the tactics in seconds measured by a CMU common lisp system on a machine with a 1330 Mhz AMD processor and 512 MB RAM. Note that this does not include the time for user interaction which has to be added for an overall comparison.

Example	Lemmas	Manual Appl.	Weight	Automatic Appl.	Deletions	Runtime
gcd	84 (+1)	8	2	1087	18	1.98
$\sqrt{2}$	51 (+2)	11	1	1005	28	3.86
Lpo	142 (+6)	5	65	6208	1236	31.35
exp-exhelp	27	0	6	1313	220	7.44

Table 4: Performance of the new tactics for some more challenging examples

First, let us comment on the different numbers of lemmas specified: For the examples in [11] we have tried to use equivalent specifications for all tactic versions. To account for the better automation we have deleted one trivial lemma that was needed by the old tactics and have added three new lemmas for the new tactics. For the sorting algorithms in [8] we have not been so restrictive in changing the specification for the new tactics to reduce manual interventions. Therefore, the second comparison may privilege the new tactics. Nevertheless, the statistics show that the new tactics improve the automation of the proof process drastically without increasing the machine runtime very much. The implemented heuristics prevent many applications of conditional lemmas whose conditions cannot be relieved. This results in a much smaller number of inference rule applications that have to be deleted later in comparison to the implementation in [12]. These checks may be the reason for an additional runtime of 32% in Table 2 in comparison to the old tactics whereas the number of applied inference rules only increases by 3%. But this increment is negligible in comparison to the achieved automation resulting from a bigger proof space that is searched.

The development of the new tactics has greatly benefitted from the automatic extraction of statistical informations from log files created by QuodLibet. By doing so for the examples in [11], we recognized for an earlier version of the new tactics that two lemmas have been activated which were attempted very often but always failed, i.e., their conditions could not be relieved. First, this problem was solved by not activating these lemmas at all, reducing the number of automatic applications and deletions as well as the runtime drastically. In a second review we recognized that these two lemmas were activated as rewrite lemmas with a general term as left-hand side. This has led to the notion of obligatory literals that enable the user to activate these lemmas again. With the automatically extracted statistics it can be shown that the lemmas will not be attempted misleadingly anymore for the specifications in [11] when using obligatory literals. Furthermore, they may help in finding proofs if the context given by the obligatory literals is fulfilled.

We will now discuss some more challenging examples performed with the new tactics. The statistics are presented in Table 4. The `gcd` and $\sqrt{2}$ example contain many arithmetic lemmas that are needed to prove the main theorems. The irrationality of $\sqrt{2}$ is, for instance, essentially specified by the lemma

$$(30)\quad \{\ \mathtt{plus}(\mathtt{times}(y,y),\mathtt{times}(y,y)) \neq \mathtt{times}(x,x),\ y = \mathtt{0}\ \}$$

where `plus` and `times` stand for the addition and multiplication on natural numbers represented by sort `Nat`. The lemma was first proved with QuodLibet by Claus-Peter Wirth using ideas from the ancient Greeks. For the proof a cut is introduced by the literals $\mathtt{leq}(x,y) = \mathtt{true}$ and $\mathtt{leq}(\mathtt{plus}(y,y),x) = \mathtt{true}$. If one of the literals holds, plus(times(y, y), times(y, y)) $\neq$ times(x, x) can be shown by order considerations. Otherwise, if plus(times(y, y), times(y, y)) $=$ times(x, x) would hold for some x and y, then there exist two smaller natural numbers — namely minus(plus(y, y), x) and minus(x, y), — with the same property which is impossible. For these examples the integration of decision procedures for Presburger arithmetic (see [6]) seems to be an important chance to improve the tactics, although the order constraints clearly fall outside of pure Presburger arithmetic due to multiplication. Therefore, the use of the decision procedure has to be enhanced by lemma applications. Besides, some of the rewrite lemmas have to be used in both directions like the distributivity of plus over times. This may be achieved by integrating rippling techniques introduced for explicit induction in [7]. However, the statistics show that even the new tactics alone perform well w.r.t. the degree of automation if the right lemmas have been activated before.

The Lpo and exp-exhelp example are both concerned with mutually recursive functions. An Lpo is a simplification order on terms based on a precedence relation on operator symbols. It is widely used in automatic theorem proving based on rewriting techniques. The example was initiated by Bernd Löchner while trying to prove the correctness of an efficient implementation based on program transformation. His specifications of different versions of the Lpo described in [13] are well suited for QuodLibet. To illustrate the complexity of the example we briefly present some of the specifications. In [13] boolean operators are used in the definition of the different versions of the Lpo. Following Guideline 4.1 we have transformed this specification into an internal representation to support the tactics by using the built-in properties of the specification language. A part of the resulting specification is illustrated in Figures 2 and and 3 whereas an overview about the relationship of the seven mutually recursive operators is given in Figure 4. Terms are represented by a sort Term with constructors V and F for terms consisting just of a variable symbol and for terms consisting of an operator and a (possibly empty) list of terms, respectively. Lists of terms, represented by sort Termlist, are themselves composed of terms by the constructors nil and cons. Therefore, the two sorts mutually depend on each other often leading to a duplication of needed lemmas: If a property has to be specified or shown for terms an appropriate property is also needed for lists of terms. Note that the syntactic specification of terms does not guarantee that the arity of an operator and the length of the appropriate list in the term are the same. Instead, this *wellformedness* property is specified by a boolean operator Well on terms which depends on an operator Well_tl on lists of terms.

Let us give some comments on the specification of the Lpo:[7] The axioms check whether the first term is bigger w.r.t. the Lpo than the second term. Axioms (31) to (33) handle the case that both terms start with an operator symbol. In this case Lpo holds iff one of the Alpha, Beta or Gamma cases holds. This disjunction is split into three clauses. The disjointness of the conditions as well as the negated definedness atoms in the clauses are needed to guarantee the conformance to QuodLibet's admissibility conditions for ensuring consistency of the specification. Axiom (34) addresses the case that the first term starts with an operator symbol whereas the second term is only a variable symbol. In this case the Delta operator is applied which checks whether the variable is contained in the argument list of the first term by using the operator contains_tl (see Axiom (45)). Axiom (35) completes the specification of the Lpo for all wellformed terms. The Alpha operator specified

[7]As we are only interested in properties of the Lpo w.r.t. wellformed terms we will only consider these terms in the following description.

$$
\begin{array}{ll}
(31) & \{\ \mathtt{Lpo}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = \mathtt{true}, \\
 & \quad \mathtt{Alpha}(ts,\mathtt{F}(g,us)) \neq \mathtt{true}\ \} \\
(32) & \{\ \mathtt{Lpo}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = \mathtt{true}, \\
 & \quad \mathtt{Alpha}(ts,\mathtt{F}(g,us)) = \mathtt{true}, \\
 & \quad \mathtt{\tilde{}def\ Alpha}(ts,\mathtt{F}(g,us)), \\
 & \quad \mathtt{Beta}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) \neq \mathtt{true}\ \} \\
(33) & \{\ \mathtt{Lpo}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = \mathtt{Gamma}(\mathtt{F}(f,ts),\mathtt{F}(g,us)), \\
 & \quad \mathtt{Alpha}(ts,\mathtt{F}(g,us)) = \mathtt{true}, \\
 & \quad \mathtt{\tilde{}def\ Alpha}(ts,\mathtt{F}(g,us)), \\
 & \quad \mathtt{Beta}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = \mathtt{true}, \\
 & \quad \mathtt{\tilde{}def\ Beta}(\mathtt{F}(f,ts),\mathtt{F}(g,us))\ \} \\
(34) & \{\ \mathtt{Lpo}(\mathtt{F}(f,ts),\mathtt{V}(y)) = \mathtt{Delta}(\mathtt{F}(f,ts),\mathtt{V}(y))\ \} \\
(35) & \{\ \mathtt{Lpo}(\mathtt{V}(x),u) = \mathtt{false}\ \} \\
\hline
(36) & \{\ \mathtt{Alpha}(\mathtt{nil},u) = \mathtt{false}\ \} \\
(37) & \{\ \mathtt{Alpha}(\mathtt{cons}(t,ts),u) = \mathtt{true}, \\
 & \quad t \neq u\ \} \\
(38) & \{\ \mathtt{Alpha}(\mathtt{cons}(t,ts),u) = \mathtt{true}, \\
 & \quad t = u, \\
 & \quad \mathtt{Lpo}(t,u) \neq \mathtt{true}\ \} \\
(39) & \{\ \mathtt{Alpha}(\mathtt{cons}(t,ts),u) = \mathtt{Alpha}(ts,u), \\
 & \quad t = u, \\
 & \quad \mathtt{Lpo}(t,u) = \mathtt{true}, \\
 & \quad \mathtt{\tilde{}def\ Lpo}(t,u)\ \} \\
\hline
(40) & \{\ \mathtt{Beta}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = \mathtt{Majo}(\mathtt{F}(f,ts),us), \\
 & \quad \mathtt{prec}(f,g) \neq \mathtt{true}\ \} \\
(41) & \{\ \mathtt{Beta}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = \mathtt{false}, \\
 & \quad \mathtt{prec}(f,g) = \mathtt{true}, \\
 & \quad \mathtt{\tilde{}def\ prec}(f,g)\ \} \\
\hline
(42) & \{\ \mathtt{Gamma}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = \mathtt{Majo}(\mathtt{F}(f,ts),us), \\
 & \quad f \neq g, \\
 & \quad \mathtt{Lex}(ts,us) \neq \mathtt{true}\ \} \\
(43) & \{\ \mathtt{Gamma}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = \mathtt{false}, \\
 & \quad f = g\ \} \\
(44) & \{\ \mathtt{Gamma}(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = \mathtt{false}, \\
 & \quad f \neq g, \\
 & \quad \mathtt{Lex}(ts,us) = \mathtt{true}, \\
 & \quad \mathtt{\tilde{}def\ Lex}(ts,us)\ \} \\
\hline
(45) & \{\ \mathtt{Delta}(\mathtt{F}(f,ts),\mathtt{V}(y)) = \mathtt{contains\_tl}(ts,y)\ \}
\end{array}
$$

Figure 2: Specification of the internal representation of the `Lpo`

(46) $\{$ $\texttt{Majo}(t, \texttt{nil}) = \texttt{true}$ $\}$

(47) $\{$ $\texttt{Majo}(t, \texttt{cons}(u, us)) = \texttt{Majo}(t, us)$,
$\texttt{Lpo}(t, u) \neq \texttt{true}$ $\}$

(48) $\{$ $\texttt{Majo}(t, \texttt{cons}(u, us)) = \texttt{false}$,
$\texttt{Lpo}(t, u) = \texttt{true}$,
$\texttt{\~{}def Lpo}(t, u)$ $\}$

---

(49) $\{$ $\texttt{Lex}(\texttt{nil}, \texttt{nil}) = \texttt{false}$ $\}$

(50) $\{$ $\texttt{Lex}(\texttt{cons}(t, ts), \texttt{cons}(u, us)) = \texttt{Lex}(ts, us)$,
$t \neq u$ $\}$

(51) $\{$ $\texttt{Lex}(\texttt{cons}(t, ts), \texttt{cons}(u, us)) = \texttt{Lpo}(t, u)$,
$t = u$ $\}$

Figure 3: Specification of the internal representation of the `Lpo` (continued)

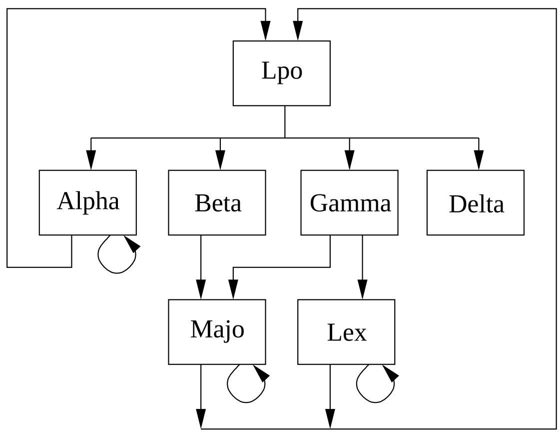


Figure 4: Mutual recursive definition of the `Lpo`

by Axioms (36) to (39) tests whether the first list contains a term that is equal to or bigger than the second term w.r.t. the Lpo. The Beta operator defined by Axioms (40) and (41) holds iff the top-level operator of the first term is bigger than that of the second term w.r.t. the precedence relation given by the operator prec, and the first term is bigger than each argument of the second term w.r.t. the Lpo. This last check is performed by operator Majo. Last but not least, the operator Gamma specified by Axioms (42) to (44) holds iff the top-level operators of the terms are equal, the first argument list is bigger than the second argument list w.r.t. the lexicographic extension of the Lpo defined by operator Lex and the check of Majo holds as for the Beta operator. We have proved that this specification of the Lpo is totally defined for all pairs of wellformed terms and that it really establishes a simplification order. Furthermore, we have shown the equivalence of our specification and the first two versions of the Lpo presented in [13].

If a property has to be proved for Lpo appropriate lemmas for all mutually recursive operators will have to be specified. The new tactics support the proof of properties for mutually recursive operators only rudimentarily. The analysis process does not recognize mutual recursion but inspects each operator separately under the assumption that every other operator is terminating. This suffices to generate an inductive case splitting but other tasks have to be performed manually like

- the specification of auxiliary lemmas for the mutually recursive operators;
- the activation of the auxiliary lemmas for inductive applications;
- the instantiation of the weight variables to get an appropriate induction order.

Furthermore, the simplification process is not directed to use one special inductive instance of one of the mutually dependent lemmas as in explicit induction. This leads to many faulty inductive applications that have to be deleted again resulting in a less efficient simplification process because of the bigger search space. On the other hand, it enables the tactics to find a proof at all with a less precise analysis of the operators. Therefore, the new tactics succeed in finding the proofs for the mutually recursive operators provided that the hints described above are given.

The exp-exhelp example is less problematic w.r.t. mutual recursion since only two operators are mutually recursive. The main task is to find an appropriate induction order to prove the termination of the exp operator that is defined with nested recursion by the axiom $\mathtt{exp}(\mathtt{apply}(e1, x, e2), y, e) = \mathtt{exp}(\mathtt{exp}(e1, x, e2), y, e)$. The induction order has to be chosen so that among other things $\mathtt{w}(e1, x, e2) < \mathtt{w}(\mathtt{apply}(e1, x, e2), y, e)$ and $\mathtt{w}(\mathtt{exp}(e1, x, e2), y, e) < \mathtt{w}(\mathtt{apply}(e1, x, e2), y, e)$ by instantiating the weight variable w. Inspired by the termination arguments in [9], we have chosen a 2-tuple with the number of calls of the operator apply in the expressions as first component and the expressions themselves as second component that is compared lexicographically as usual.

Summing up, we have achieved a high degree of automation in our case studies. Most of the manual interactions are caused by applications of lemmas that

- have free variables which cannot be instantiated by subsumption of other goal literals: Since we do not try to guess the instantiation of these variables they prevent the automatic application of needed lemmas such as the transitivity of order relations.

- are needed but whose head literals do not match any goal literal: As we typically do not have a confluent rewrite relation the application of one lemma may prevent the application of other lemmas in that proof attempt.

- are non-terminating rewrite lemmas: These lemmas cannot be applied automatically without losing termination of the simplification process which we want to avoid. Non-termination may be introduced by lemmas that are inherently non-terminating like the division operation on natural numbers as specified in [11] by the axioms

  (52) $\{\ \mathtt{div1}(x, y, z1, z2) = z1,$
  $x \neq z2\ \}$

  (53) $\{\ \mathtt{div1}(x, y, z1, z2) = \mathtt{div1}(x, y, \mathtt{s}(z1), \mathtt{plus}(z2, y)),$
  $x = z2\ \}$

  Furthermore, non-termination may be caused by the need to apply a rewrite lemma in both directions.

Whereas we cannot imagine a solution for the last case, the first two cases can often be tackled by adding some bridging lemmas to the specification. In the first case a bridging lemma may extend the considered goal by some literals that allow to bind the free variables. This bridging lemma may then be used for proving the original goal. In the second case bridging lemmas may be used for "completing" the rewrite relation. Although these lemmas may increase the degree of automation they also blow up the database which may be confusing for the user and slow down the theorem prover. Thus, it may be sometimes better to apply a lemma manually.

For the first case some approaches will be attempted in the future to increase the degree of automation without using bridging lemmas: We will try to utilize additional information about the induction hypothesis that is most likely to be used. Therefore, we will adopt a similar approach as in explicit induction when analyzing operators and introducing inductive case splittings to prove inductive lemmas, see e.g. Section 16.2 of [16] and Section 3.3 of [17]. This additional information will allow us to instantiate free variables more often. Furthermore, it will enable a simplification process that is more goal directed to the probable induction hypothesis. On the other hand, we are not fixed to use this induction scheme at any prize but may choose to apply other induction hypotheses. This will help us to overcome the problems of explicit induction stated in [14]. Furthermore, we may use special activated lemmas, e.g. lemmas with only one literal to instantiate free variables. Another approach described in [17] defers the instantiation of the free variables by using free γ-variables as place holders until enough information is available. Further experiments will have to be done to see if these approaches are feasible or if they enlarge the search space too much.

7 Conclusion

We have implemented new tactics for the inductive theorem prover QuodLibet that restructure and improve the whole simplification process. They provide one general-purpose simplification tactic that consists of five passes. The simplification process is able to apply permutative and conditional lemmas that are not directly applicable. Furthermore, we have implemented special macro inference steps that are made up of a combination of inference steps to handle special patterns for each kind of atom. Heuristics have been implemented to restrict the proof space that is searched through, avoiding repetitions of equal inference steps. The application of conditional lemmas is e.g. controlled by obligatory literals in the lemma and mandatory literals in the goal clause that prevent many unnecessary computations. The tactics use parameterization to enable the user to control the proof process more flexibly. The options can be set locally by keyword parameters or globally by their default values. Altogether, the user can choose between 12 inductive proof schemes when performing a proof by induction depending on how the inductive case analysis is performed, which lemmas are to be used inductively, and whether the inductive proof process is restarted automatically for subgoals that cannot be proved by simplification.

This paper may serve as a user manual for the new tactics. Therefore, we have described the new tactics in detail from a user's point of view, neglecting issues of the implementation but providing all information that is necessary to instantiate the parameters of the tactics in a reasonable way. Furthermore, we have given some hints to guide users in creating specifications and performing proofs of lemmas about the specification. The case studies show that we have achieved a far better degree of automation than the old tactics. We are now able to tackle some more challenging problems. Besides, these case studies indicate directions for further development.

In the future, we will work toward an improved analysis of mutually recursive operators. We hope that this will enable us to overcome the problems mentioned in Section 6 for lemmas based on mutually recursive operators like choosing an appropriate induction order by instantiating the weight variables of all participating proof state trees. For the automatic speculation of auxiliary lemmas for mutually recursive operators we will try to significantly improve the ideas of the multi-predicate approach in [3]. In this approach a lemma is generated lazily for each mutually recursive operator according to the induction hypothesis that is needed in the proof of another mutually recursive operator. Besides, we will provide our simplification process with more information about probable induction hypotheses as they are computed in explicit induction. This will enable us to use a simplification process that is more goal directed by using rippling techniques described in [7], if this approach is feasible. Last but not least, we will try to integrate decision procedures to QuodLibet starting with Presburger arithmetic along the lines of [6] to improve the efficiency of our inductive theorem prover.

Acknowledgment

I would like to thank Jürgen Avenhaus and Claus-Peter Wirth for many helpful comments on earlier drafts of this paper as well as Bernd Löchner for initiating the `Lpo` example.

References

[1] Jürgen Avenhaus, Ulrich Kühler, Tobias Schmidt-Samoa, and Claus-Peter Wirth. How to prove inductive theorems? QuodLibet! In Franz Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE-19)*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 328–333. Springer, 2003.

[2] A. Bouhoula and M. Rusinowitch. Automatic Case Analysis in Proof by Induction. In R. Bajcsy, editor, *Proceedings 13th International Joint Conference on Artificial Intelligence*, volume 1, pages 88–94, Chambéry (France), August 1993. Morgan Kaufmann.

[3] Richard J. Boulton and Konrad Slind. Automatic derivation and application of induction schemes for mutually recursive functions. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic: First International Conference, CL2000, London, UK, July 2000: Proceedings*, volume 1861, pages 629–643. Springer-Verlag, 2000.

[4] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.

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

[6] Robert S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. In *Machine intelligence 11*, pages 83–124. Oxford University Press, Inc., 1988.

[7] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.

[8] Markus Kaiser. Effizientes Beweisen mit einem formalen Beweissystem. Diplomarbeit (German), Fachbereich Mathematik, Universität Kaiserslautern, Germany, 2002.

[9] Deepak Kapur and Mahadevan Subramaniam. Automating induction over mutually recursive functions. *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology (AMAST'96)*, 1101:117–131, 1996.

[10] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[11] Ulrich Kühler. *A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations*. PhD thesis, Universität Kaiserslautern, 2000.

[12] Nanette Linn. Verbesserung der Beweissteuerung des induktiven Theorembeweisers QuodLibet. Projektarbeit (German), Fachbereich Informatik, Universität Kaiserslautern, Germany, 2003.

[13] Bernd Löchner. Things to know when implementing LPO. In Stephan Schulz, Geoff Sutcliffe, and Tanel Tammet, editors, *IJCAR Workshop on Empirically Successful First Order Reasoning (ESFOR)*, Electronic Notes in Theoretical Computer Science, 2004. To appear.

[14] Martin Protzen. Lazy generation of induction hypotheses. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 42–56. Springer, 1994.

[15] Tobias Schmidt-Samoa. Realisierung einer Taktik-basierten Beweissteuerungskomponente für den induktiven Theorembeweiser QuodLibet. Projektarbeit (German), Fachbereich Informatik, Universität Kaiserslautern, Germany, 1997.

[16] Claus-Peter Wirth. *Positive/Negative-Conditional Equations: A Constructor-Based Framework for Specification and Inductive Theorem Proving*, volume 31 of *Schriftenreihe Forschungsergebnisse zur Informatik*. Verlag Dr. Kovač, Hamburg, Arnoldstr. 49, D–27763 Hamburg, 1997. ISBN 3-86064-551-X, `www.ags.uni-sb.de/~cp/p/diss/welcome.html`.

[17] Claus-Peter Wirth. Descente infinie + Deduction. *Logic Journal of the IGPL*, 12(1):1–96, 2004. `www.ags.uni-sb.de/~cp/p/d/welcome.html`.

[18] Hantao Zhang. Contextual rewriting in automated reasoning. *Fundamenta Informaticae*, 24(1/2):107–123, 1995.

[19] Hantao Zhang, Deepak Kapur, and Mukkai S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. Lush and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction (CADE-9)*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 1988.