JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Symbolic Computation Prover with Induction[1)]

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Computermathematik

Eingereicht von:
Günther Mayrhofer

Angefertigt am:
Research Institute for Symbolic Computation (RISC)

Beurteilung:
O.Univ.Prof. Dr. Dr.h.c.mult. Bruno Buchberger

Linz, September 2009

[1)] Diese Arbeit wurde durch ein Stipendium des SFB 013 des FWF Wissenschaftsfonds unterstützt.
This thesis was supported by a grant of SFB 013 of the FWF Austrian Science Fund.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 7. September 2009,

Günther Mayrhofer

Zusammenfassung

Eine wichtige Aufgabe beim automatischen Beweisen stellt Rechnen mit "beliebig, aber festen" Konstanten dar. Diese Art des Mittelschulbeweisens tritt im Inneren der meisten Beweise auf. Die vorliegende Arbeit stellt einen automatischen Beweiser vor, der auf diese Berechnungen mit Symbolen spezialisiert ist. Für das Rewriting der Zielformel verwendet der Beweiser Gleichungen und Äquivalenzen aus der Wissensbasis. In allen Formeln sind Allquantoren und ausgewählte logische Operatoren erlaubt. Spezielle Syntaxelemente unterstützen Fallunterscheidungen und Sequenz-Variablen. Der Beweiser verwendet eine spezialisierte Methode für das Beweisen von Gleichungen und eine wichtige Funktion ist das Beweisen mit expliziter Fallunterscheidung. Eine Erweiterung erlaubt auch Induktionsbeweise für einige vordefinierte Strukturen.

Zusätzlich zur Implementierung des Beweisers in *Mathematica* gibt es ein Programm, das den Ablauf des Beweisers verfolgt und ein Protokoll des Beweises erzeugt. Da der Quellcode dieses Überwachungsprogramms getrennt vom Quellcode des Beweisers ist, sind auch unterschiedliche Ausgabeformate möglich. Wichtiger jedoch ist, dass ein Benutzer den Quellcode des Beweisers inspizieren kann ohne dass technische Details zur formatierten Ausgabe die Logik des Beweisers verschleiern.

Die Motivation für diesen Beweiser liegt im Ausbau des *Theorema*–Systems. Das Ziel ist eine Umgebung, in der neue Beweiser in der gleichen Sprache wie Theoreme formuliert werden können. Der Vorteil liegt auf der Hand, denn mit bereits bestehenden Beweisern können Fakten über neue Beweiser bewiesen werden. Damit ist es möglich eine ganze Hierachie von formal geprüften Beweisern aufzubauen. Für so ein Beweisen von Beweisern ist es notwendig struturelle Induktion über Terme und Formeln an den Anfang zu stellen. Der Ausgangsbeweiser in einer Hierachie braucht aber Berechnungen in vielen Beweisteilen. Diese Aufgabe kann der vorliegende Beweiser für Symbolisches Rechnen übernehmen.

Abstract

An important task in automated theorem proving is computing with "arbitrary but fixed" constants. This kind of highschool proving occurs in the main part of most proofs. The current master's thesis presents an automated prover that focuses on such computations with symbols. The prover uses equalities and equivalences in the knowledge base to rewrite a goal formula. In all formulae there may be universal quantifiers and some selected logical connectives. Special syntax elements support case distinctions and sequence variables. The prover uses a specialized method for proving equalities and an important feature is proving by cases. An extension allows induction over some predefined domains.

Additionally to the prover implementation in *Mathematica*, there is a tracer that prints a protocol of the proof flow. Since the code for this tracer is separated from the prover, there may be more than one tracer with different output. But more important is that a user can inspect the code of prover without being confused by technical details for generating some nice output.

The main motivation for this prover is a new extension of the *Theorema* system. The aim is an environment for defining new prover in the same language as theorems. The advantage is clear, existing prover may prove facts of a new one, for example the correctness. Using this it is possible to build up a hierarchy of formally checked provers. For such reasoning about reasoners a starting point needs induction on the structure of terms and formulae. A first prover in the hierarchy will need computations with symbols in many proof branches. This may be done by the current Symbolic Computation Prover.

Acknowledgment

I would like to thank Prof. Bruno Buchberger for the opportunity to write my master's thesis at RISC under his advice. His visions for new ways of doing mathematics and reasoning contain interesting and powerful concepts. Even outside mathematics many principles are very useful. It was a great experience for me to work with him.

Thanks also to the *Theorema* group for support and discussion. A special thank to Wolfgang Windsteiger. His help with the internals of *Mathematica* and *Theorema* made it possible for me to finish the research for this thesis.

During my studies Prof. Peter Paule always had an open ear for students and he is committed to improve the quality of an already high quality curriculum. I am very thankful for his efforts.

An important step forward in formal reasoning for me was the invitation of Prof. Günter Pilz to join a research project for mathematical education using formal reasoning. This opened for me the doors to the subject of my thesis. Cordial thanks to him!

Beyond the university I would like to thank my family for their valuable support over the years. In particular I would like to thank my wife Patricia for her understanding and help in proofreading.

Table Of Contents

1 Introduction

1.1 Motivation

In (formal) reasoning about formulae there are some computations with symbols in nearly every proof [Buch04]. Computation means applying an algorithm on some input data. That input data may be numbers or other objects, depending on the algorithm. We call computation "Symbolic Computation" if the input may contain uninterpreted constants for example "arbitrary but fixed" constants used in the proof of universally quantified formulae. Computations of that kind are necessary in most formal proofs. In automated theorem proving it is useful to have a program which is able to do that. The current master thesis presents such a program for symbolic computation (see *section 2*).

Bruno Buchberger [Buch04] describes in his "proving by intermediate principles" that starting with a simple prover a hierarchy of provers can be built up in a formal proven way. For that "Reasoning about Reasoners" [Buch08] it is necessary to do inductive proofs over the structure of logical formulae. In each proving branch again symbolic computation is necessary to prove that branch. In the *Theorema* project it is planed to incorporate such a prover in the next version. Reasoning about the structure of logical terms needs a distinction between terms in the language and terms that are the objective of reasoning. An approach for incorporating this is "Reflection" [GiBu07].

1.2 Theorema

The *Theorema* project [Tma97, Tma98, Tma99a, Tma99b, Tma00a, Tma00b, Tma00c] founded and headed by Bruno Buchberger has the objective to develop a system for doing mathematics including reasoning. It supports input in traditional mathematical form and output in natural language as well as in an interactive way (see [Buch00]) . Proving and computing is combined in one single system. Automated provers for different specific domains support natural style deduction.

The system is implemented in *Mathematica* [Wolf03] and it is called *Theorema*, too. It uses for the command language *Mathematica* and user language is a subset of higher order predicate logic. The user language is independent of *Mathematica* syntax and supports different input forms especially for quantifiers. A quantifier may be entered in multiline style with multiple variables, different ranges and additional constraints.

Currently all automated theorem provers of *Theorema* are directly implemented in *Mathematica*. In this programming language it is difficult to inspect an implementation automatically. In the future the provers will be written in the *Theorema* user language as algorithms on formulae. That will allow the user to add new reasoners. But more important is the fact, that it will be possible to prove theorems about these reasoners. With that reasoning about reasoners all provers can be proven correct and a hierarchy of proven provers can be constructed. Symbolic Computation together with structural induction as metaprover is able to define and verify all higher level prover like full predicate logic

prover (also alternating quantifiers), solvers (correspond to existential quantifier) and special domain prover for Groebner Basis, set theory, etc. [Buch08].

The Symbolic Computation prover will be embedded in such a structural induction prover in one of the next *Theorema* versions. The implementation of the Symbolic Computation prover is independent of the current *Theorema* to be adaptable enough for future versions. Nevertheless the prover is harmonized with *Theorema* and has a similar user language.

1.3 Details of the Problem

The scope of this thesis contains the Symbolic Computation prover (SC prover), a first extension with induction and a possibility of printing some intermediate results. Due to experiences from case studies some of the requirements for the prover changed during the phase of development. The listed demands in this section describe the final version.

The SC prover has to derive that a given formula is a logical consequence of a given knowledge base. All occurring formulae are in a subset of predicate logic. The language to formalize these formulae is a subset of the user language of *Theorema* [Tma00a]. The language is restricted to equalities, equivalences, conjunctions, implications, case distinctions, and universal quantifiers. It is sufficient to support quantifiers with a single variable and neither with condition nor with a range restriction. But the quantifiers may include sequence variables (see [Kuts02, KuBu05]).

The prover should handle the language constructs in an appropriate way and simplify the goal formula as far as possible. The innermost kernel of the SC prover should use rewriting to derive a true goal formula. The proofs should be done in a natural (human like) way. The range of the proving power of the SC prover corresponds to high school proving. Computations with "arbitrary but fixed" constants and knowledge bases with equalities and equivalences are applied already in high school.

As implementation platform *Mathematica* [Wolf03] is chosen, because *Theorema* is based on it [Tma97] and *Mathematica* has a powerful rewriting engine [Buch96a]. Important for the implementation is that the source code is coded as simple as possible and easy to understand. The reason for that is that the prover does not implement a logic, but the source code of the prover describes the logic [Buch04]. A consequence is that the prover should do the proofs but there should be no output of intermediate results, because that would cause a more complicated source code and it is difficult for a reader to distinguish code for the proving algorithm and the output generating. But this distinction is essential because a user has to trust the prover (like he trusts a logic) and the correctness of the prover – in an informal sense – can be checked only by inspection. On the other hand it is important for the practical use of a proving algorithm to have a possibility to inspect a proof. Especially for analyzing failed proof attempts that is necessary. That leads to a separated program for output generation.

Even if the prover is part of the *Theorema* project the implementation is done independently from the current version of *Theorema*, to prevent dependences to provers of the old generation. The integration in the new version of *Theorema* is future work [Buch08].

1.4 Structure of the Thesis

Section 2 describes the Symbolic Computation prover in detail. It defines an input language and it explains how different kinds of formulae are handled by the prover. The commented source is located in the appendix and describes the implementation details.

A first extension of the SC prover is the induction prover which is treated in *section 3*. This prover shows how to extend the SC prover with additional features.

The prover is not able to produce output of intermediated results. *Section 4* shows how such output may be generated. A special feature of *Mathematica* allows doing this without adapting the source code of the prover. The source code is again in the appendix.

Some case studies in *section 5* show how the prover may be used in application. *Section 6* summarizes the thesis and *section 7* lists the references.

The appendix contains the source code of the two packages: the Prover and the ProofTracer.

1.5 Notation

Mathematica commands and names of variables and options and symbols of the user language are written in bold font:

```
SCProver
```

Output created by *Mathematica* commands is written in a light weight style:

```
Proved
```

Input in *Mathematica* and output of the execution is presented in this form:

```
SCProver[ThmPlus1, {DefPlus1, DefPlus2}]
Failed
```

2 Symbolic Computation Prover

2.1 Design

The Symbolic Computation prover (SC prover) proves formulae mainly by rewriting technics combined with logical transformations from natural deduction. Formulae used as goal or in the knowledge base are predicate logic formulae. They may be quantified by universal quantifiers, but existential quantifiers are not supported. Quantified variables may be single variables or sequence variables [KuBu05]. In the knowledge base there is a special feature for supporting case distinctions. For proving in inductive domains some special proving rules may be embedded in the SC prover. This is done for natural numbers and related inductive domains in *section 3*.

The general proving strategy of SC prover is to reduce the goal by straight forward reductions first. Then on the reduced goal rewriting is applied. If case distinctions appear in the rewritten goal, the prover tries to verify a case or eliminate invalid cases. If rewriting fails and there are case distinctions left a proof by cases is performed. The rewriting process handles equalities separately.

The prover is called from the external point of view with one goal formulae and a knowledge base which is a list of formulae. Some options may control the execution flow of the prover. There are four options. Two for proving by cases: **`ProveByCases`** and **`PBCLevel`**; and one for rewriting: **`MaxNumberOfRewritingSteps`**. The last option is used by the induction prover. It may enable or disable proving by induction. For further details on the options see the description below in the corresponding subsections or in the commented source code (see *section A.2*).

Internally the prover allows handling a list of goal formulae for rewriting. Only the goal reduction is restricted to a single goal formula. The goal formulae are treated as alternative goals. Hence, it is sufficient if one of these formula is true. In case of an equality there are lists of formulae allowed on both sides of the equality. They are also considered as alternatives: The equality is true, if a formula of the left hand side is equal to a formula of the right hand side. This feature is important for the rewriting engine of the prover. It allows the engine to apply the rewriting rules in a arbitrary order without loss of information.

2.1.1 User Language for Terms and Formulae

The terms and formulae which are input for the SC prover are *Mathematica* terms. *Mathematica* includes some knowledge about various symbols, e.g. **Plus**. The prover must not use this knowledge if for instance somebody wants to prove the commutativity of addition over natural numbers. In order to prevent formulae from applying *Mathematica* knowledge onto them it is necessary to use new symbols which have no knowledge within *Mathematica*. A possible way to do this is to add a prefix to the symbols name. In Theorema [BuWi98] this is done by adding the prefix "™", e.g. **™Plus**. This is a new symbol and there is no meaning within *Mathematica.* The SC prover will be embedded into Theorema in future; therefore the approach of Theorema is used for this prover. Theorema also supports input and output in traditional mathematical style [Tma98]; this will not be used within the SC prover.

Most symbols appearing within terms can be chosen by the user with the restriction above. But there are also some symbols which have already some meaning within the prover. The symbol **™Equal** represents an equality. **™Iff** is an equivalence. **™Not** means negation. **™Implies** stands for an implication. **™And** means conjunction. **™ForAll** represents the universal quantifier. **™CaseDistinction** indicates a case distinction. The *Mathematica* symbol **True** is used as the boolean value true. For more details on the usage of these symbols e.g. the arity see the source code (see *section A.2*).

A case distinction allows definitions for several cases. It is a list of terms with conditions. One term together with a condition is called a case. A condition is a formula. A term will be used, if the corresponding condition is true and the conditions of all previous cases are false. A case distinction may occur as outermost symbol or on the right hand side (= second argument) of an equality (**™Equal**) or an equivalence (**™Iff**). But it must not appear in other subexpressions or as outermost symbol. It is possible to built up If–Then–Else by case distinction.

Some additional symbols are indicated by the prefix "•". Symbols with this prefix are reserved for special use only and must not appear as user symbols in formulae. There is **•case** which represents a case within a **™CaseDistinction**. Internally the symbol **•fix** is used for "arbitrary but fixed" constants. The induction prover uses the symbol **•ind**.

Sequence variables [KuBu05] allows defining functions with variable arity. The symbol **•seq** indicates such sequence variables.

2.1.2 Goal Reduction

Goal reduction tries to simplify the goal as far as possible. Some goal formulae may be reduced to true immediately: **True**, **™Equal[expr,expr]**, **™Iff[expr,expr]** and **™Implies[™Not[True], expr]** for arbitrary expressions **expr**. Further reductions are related to implications, equivalences and universal quantifiers.

Instead of proving an implication it is equivalent to assume the premise and prove the conclusion. Hence the prover adds the premise to the knowledge base and continues proving with the conclusion as new goal formula (see Deduction Theorem [Mend66]).

A conjunction of formulae is proved by proving each formula separately. If the proof of a formula fails then the whole proof fails immediately. Any formula that is already proven is added to the knowledge base for the proof of the next formulae.

An universal quantifier in the goal formula will be reduced by applying an Arbitrary–But–Fixed rule. Therefore the bound variable in the quantified formula will be replaced by a new constant ("arbitrary but fixed") and the prover tries to prove this simpler goal. For bound variables over an inductive domain the prover tries proving by induction. For further details on induction see *section 3*.

Instead of proving an equivalence both corresponding implications will be proved: The prover assumes the left hand side formula and proves the right hand side formula and vice versa.

For all other kinds of formulae in the goal rewriting will be started. Equalities will be proved by rewriting each side of the equality individually. The equality is true if both sides rewrite to identical terms.

If there is a case distinction in the goal then the prover tries to prove one of the cases before rewriting. If this is not successful a proof by cases is done. This is not affected by the prover option **`ProveByCases`**.

2.1.3 Transformation of a Knowledge Base to Rewriting Rules

Before starting a rewriting process the rewriting rules will be generated. Therefore the knowledge base will be transformed into a list of rules. In order to gain more rules out of the available knowledge the formulae will be simplified first. Simpler formulae generate simpler patterns and simpler patterns match easier in the rewriting process. From each of these simplified formulae exact one rule is generated.

The simplification process handles universal quantifiers in combination with conjunctions and implications. The conjunctions are moved out of the quantifiers and implications are moved under the quantifiers, if this is possible. Implications are moved into conjunctions too. Formulae in the knowledge base that have a case distinction as outmost symbol are split into implications.

In the prover the set of formulae in knowledge base used as a conjunction of these formulae, therefore it is possible to split a formula with a conjunction as outermost symbol into several formulae. More formulae allows generating more rules, therefore splitting is used. Implications will be translated into a special kind of case distinctions. Moving implications under a quantifier or a conjunction simplifies the conclusion of the implication. As a consequence the generated rewriting rule gets simpler.

The process of generating rewriting rules treats one formula after the other. Equalities and equivalences result in rules which replace the left hand side by the right hand side. For universal quantified

formulae the bound variable will be replaced by a variable pattern, which will be instantiated later in the rewriting process. Sequence variables result in a special pattern.

If a formula contains a case distinction then nothing special has to be done for this formula. The formula is translated like a formula without a case distinction. The special treatment will be done later in the rewriting process.

Implications are transformed into a case distinction with only one case. The case condition is the premise of the implication and the conclusion is used as the case formula itself.

Negated formulae are translated into rules which rewrite the positive formula into **™Not[True]**. Formulae with double negations will be reduced first.

An arbitrary formula which does not match one of the rules above will be used as a rule that rewrites the entire formula into **True**.

2.1.4 Rewriting a Formula

The rewriting process starts with only one goal formula, but the process is designed for handling a list of goal formulae. This feature is used internally to handle several branches of rewriting in one rewriting step and by Proof–By–Cases (see *subsection 2.1.7*). In addition to a list of rewriting rules the process needs a knowledge base. This knowledge base is necessary for recursive prover calls in case of conditional rewriting. The process is done within a big loop which terminates if the formula **True** is found, no new formula could be derived or a proof by cases was done. Additionally the loop is limited by a counter to avoid infinitely many rewriting steps. This is controlled by the prover option **MaxNumberOfRewritingSteps**.

Within the loop first all, new formulae which can be derived from the old formulae by the rules in one stroke will be generated. If one of these formulae is already the formula **True**, then the goal is proved. Otherwise, if there were applied conditional rules, the algorithm tries to prove the conditions in order to eliminate cases (see *subsection 2.1.6*) and derive new formulae. Again if one of these formulae is **True** then the proof (or the subproof, if the proof was split) is successfully done. If there are some conditions that are neither proved nor disproved, a proof by cases is started and the loop is aborted. Proving by cases may be disabled or limited by the user. After that all formulae with case conditions that have conditions with failed proof attempts are removed from the list of goal formulae.

In future versions of the prover the strategy of applying a proof by cases immediately may be replaced by a more sophisticated strategy.

If the proof is not finished, the next rewriting step is done by the next loop iteration. Only the new formulae of the previous iteration are used to derive more new formulae. This avoids multiple applications of a rule on a certain formula. So every formula gets exactly one chance to get rewritten by a present rule.

Technically the rewriting step itself is done by applying all matching rules on each of the current formulae. For every formula and every rule all matching positions are located and the rule is applied on each of this positions. The application is done by instantiating all variables of the rule and replac-

ing the old term by the new term. If the new term contains a case distinction then in every case the replacement is done separately. Hence a case distinction would never occur in a subterm.

2.1.5 Rewriting an Equality

For rewriting an equality a specialized loop of rewriting steps is used. Instead of rewriting the entire formula the terms of each side of the equality will be rewritten separately. Therefore the exit condition of the rewriting loop has to be different. The equality is true if a term occurs on both sides of the equality.

In each loop iteration rewriting is applied on the terms of the left hand side first to derive new terms. Then the new terms are compared with the terms on the right hand side. If a term appears on both sides the loop is aborted and "Proved" is returned. Otherwise if there were conditional rules used to rewrite terms on the left hand side the conditions of the cases are tried to prove or disprove. This elimination of cases (see *subsection 2.1.6*) may find new unconditional terms. Again the truth of the equality is checked by comparison of the terms on both sides of the equation. Next the right hand side is treated in the same way as it was done with the left hand side.

If there are unproved conditional rewriting steps left at the end of an iteration, a proof by cases is done like in the general case for arbitrary formulae. Otherwise the next iteration is started. Similar to the general case only the new terms of each side will be used for rewriting in the next iteration.

2.1.6 Proving a Case or Eliminating Cases

If conditional rules are used for rewriting a formula, the conditions of one case has to be proved in order to get an unconditional formula. Unconditional formulae or terms can be used for simple rewriting. To use a certain case it is necessary to prove the condition of that case and to disprove the conditions of all cases above (see REF CODE: Syntax). Therefore a loop over all conditions is performed.

For each condition the prover is called recursively to prove or to disprove the condition formula. Disproving is done by proving the negation of a goal. If the condition can be proved then the loop aborts because no other case can be valid. If the condition is disproved then this case can be removed from the list of cases and the next condition will be handled. If the prover is neither able to prove nor to disprove the condition of a case, then this case remains for a possible proof by cases.

The elimination of cases returns the formula of a case which condition was proved. If no case could be determined, then the remaining cases will be return as a valid formula with case distinction. Cases with disproved conditions are removed from this case distinction.

2.1.7 Proof By Cases

A proof by cases is done, if in the rewriting process after applying a conditional rule no case can be verified by the prover. The rewriting process chooses the first formula where some cases are left after a elimination of cases. This strategy may be extended by a more sophisticated algorithm in a future version.

The prover option **`ProveByCases`** allows disabling the mechanism of proving by cases for a prover call. With the option **`PBCLevel`** the number of proofs by cases can be limited in depth. For example, if **`PBCLevel = 2`** then proving by cases is deactivated after the proof is split into cases for the second time.

A requirement for a proof by cases is that the defined cases together cover all possible cases. This is guaranteed by claiming that the condition of the last case is the formula **`True`**. In other words, it is necessary that case distinction has an "otherwise" or default case.

Technically the proof is split into branches. Each case is proved independently. The conditions for the current case are added to the knowledge base and the rewriting process is started again. The additional knowledge causes some new rewrite rule. With this rules it is possible to prove at least the conditions for the current case. The rewriting process will apply the conditional rule which induced the proof by cases again on the goal. But now the prover is able to prove the condition and gets an unconditional formula.

If for a case the goal could not be rewritten to **`True`**, then the proof by cases fails immediately. If the goal can be proved in all cases, then the goal is proved.

2.2 Implementation

The implementation in *Mathematica* is done within a package. This creates a new namespace and allows using private functions and symbols. Only the external interface is reachable by the user. Using the feature of packages it is possible to ensure that the internal implementation is not affected by other computations in the current user session.

For executing the code of the prover no other packages are necessary. Currently there are no dependecies of the implementation to the code of *Theorema*. In later versions the code may be embedded into *Theorema* in order to use some facilities provided there.

As usual in *Mathematica* a package is loaded by the command:

```
Needs["SC`ProverKernel`"];
```

Note that the package file must be in a subdirectory named 'SC' of a directory listed in the *Mathematica* variable **$Path**.

For the source code with implementation notes see appendix: package SC`TrustedKernel. (REF source code)

2.3 Limitations and Possible Extensions

Currently there are some limitations. The prover can only handle formulae with a single case distinction. Nested case distinctions and combined case distinctions need some preparation. In general it is possible to extend the prover for formulae of this kind. But it would be better to use a preprocessor to do this before. Otherwise the code of the prover gets more difficult to understand.

The algorithm to choose a case distinction for splitting into branches is currently very simple. It chooses just the first occurring candidate. The prover will get more powerful but also much slower if every possible set of branches is tried. Some future research may find a better algorithm for combining power and low runtime complexity.

While proving case branches the prover starts rewriting without using the fact that there is already a rewriting rule which matches for sure: The rule that induced the latest split. Currently the prover searches again for all matching rules and finds surprisingly at least the said rule. It would be better to apply this rule immediately. From a theoretical point of view there is no difference because the rule gets applied in any case. But it would decrease the runtime because a subproof is necessary to find the appropriate case of the rule again. The reason why this feature is not implemented yet is that it would either need some more variables to transfer the rule to the function where it is needed or use some stack managing functions from *Mathematica* (e.g. **Sow** and **Reap**) which are difficult to understand for people who are not so well trained in *Mathematica*.

For the integration into *Theorema* it would be necessary to use a preprocessor to simplify the quantifier generated by the *Theorema* input parser. There are already some internal routines to do at least

parts of this job (see **SimplifyQuantifier** in package **Theorema`Provers`PredicateLogic`Auxiliary`**). Especially the syntax elements for induction (see *section 3*) needs some extra regards. All other elements of the user language are already in compatible with *Theorema*.

2.3.1 Technical Details for the Integration in *Theorema*

There are some slight differences to *Theorema* syntax. The universal quantifier **™ForAll** has only two arguments: the symbol of the bound variable and the bound formula. In *Theorema* quantifier has three arguments: a list containing variables with ranges, a condition formula, and the bound formula. A translation from the SC prover syntax to *Theorema* syntax may be like this:

```
™ForAll[x_, formula_] →
 ™ForAll[•Range[•simpleRange[x]], True, formula]
```

The reverse translation does not work for the general case, but most universal quantified formulae may be translated after simplifying the formula. The condition has to be moved into the formula as an implication and some special ranges have to be converted to simple ranges. Theorema has already a function to do this: **SimplifyQuantifier** (see package **Theorema`Provers`PredicateLogic`Auxiliary`**). If there is more than one variable then the formula has to be converted into some nested quantified formulae with only one variable each.

Another difference is related to the induction. In *Theorema* there is no special syntax for inductive domains yet. This may be added as a new range object for inductive variables or determined from some special domains as range.

3 Induction Prover

3.1 Inductive Domains

For inductive defined domains it is a great enhancement to use the possibility of proving by the induction principle. It allows using more knowledge if some formula should be proven for all elements in the inductive domain. Hence a bigger set of formulae is provable by the prover [Buch96b].

Inductive domains are defined by one or more base elements and one ore more constructors. In some literature, like in [GoJu98], the basis elements are called *blocks* and a constructor *operator*. In general, inductive domains satisfy the induction law [GoJu98, 1.1.9.]:

If

every base element satisfies the property *P*, and
every constructor preserves the property *P*,

then

every element of the inductive domain satisfies the property *P*.

To keep the prover simple, only two kinds of inductive domains are currently implemented. Other kinds may be added in the future in the same way. The first type is restricted to domains with only one base element and one constructor. A well known example for this kind is the domain of natural numbers with base element "0" and constructor "succ", the successor function. The second type is for all inductive domains with a strict partial order. The set of natural numbers is also an example for this kind (with usual "less than" ordering relation). Another example is the domain of lists with "less than" order on the list length.

Proving by induction is an additional technique for proving formulae with universal quantifiers (in addition to the "arbitrary but fixed" technique of predicate logic), which is valid only for universally quantified formulae over inductive domains. Hence it is natural to embed this as a special case into the SC prover. The application of the induction principle is controlled by the prover option **`Induction`**.

Proving by induction is a special case for proving formulae with universal quantifier. Hence it is natural to embed this as a special case into the SC prover. The application of the induction principle is controlled by the prover option **`Induction`**.

The language used in the prover is an untyped version of predicate logic. It is the similar to the version in Theorema [Tma00a]. Hence, the prover can not check types on the level of syntax. So in the rewriting algorithm does not perform checks on the domain of the variable before rewriting. If there are different kinds of inductive domains in a theory or if inductive domains are mixed with non–inductive domains then the user has to make sure that invalid applications of some knowledge are not possible. This can be done by introducing additional condition in the formulae. That is why the prover treats bound variables in the knowledge whether they are over inductive domains or not.

3.2 Additional Syntax Elements For Formulae

It is necessary for the prover to know which bound variables belong to an inductive domain. Since there are different kinds of inductive domains the prover also needs some more information about the domain itself. Both is satisfied by adding a new syntax element for inductive domains. The syntax element **`•ind`** indicates that a variable is bound over an inductive domain. To distinguish the different kinds of inductive domains the prover expects in one case the base element and the unary constructor as formula symbols. In the other case it expects only the symbol for the ordering relation as a binary function.

Warning: Note that the prover uses this symbol as proving hint in goal formulae only. If it occurs in the knowledge base it is ignored by the prover.

For more details on this symbol see the source code (see *section A.2*).

3.3 Implementation

The implementation of the induction prover is completely embedded into the SC prover (REF source code). There are two definitions for the special cases of universal quantified formulae together with some auxiliary functions for the induction base and the induction step. Some other auxiliary functions are used from the SC prover. The coded version covers complete induction [BuLi81, p. 180] and course of value induction [BuLi81, p. 184].

The new syntax element for the variables over inductive domains forces may also occur in the knowledge base. Hence it is necessary to extend the process of generating of the rewrite rules. This is done by removing the **`•ind`** indicator and use it as an usual bound variable.

3.3.1 How to Add Further Domains?

It is possible to add further inductive domains with different structure. This can not be done in the user language. It is necessary to program some lines of *Mathematica* code. Four steps have to be done:

- Define a syntax pattern **`•ind[symbol,...]`** different from existing inductive domains.
- Add a definition in the prover for this syntax pattern.
- In this definition call the prover for all sub goals.
- Extend the existing proof tracer definitions (see *section 4*)

4 Practical Aspects: Proof Tracer

4.1 Motivation and Design

The SC prover contains the implementation of the necessary logical transformation steps. It is able to return "`Proved`" if a proof was successful or "`Failed`" otherwise. But there is no additional output, why a proof succeeds or fails. For the practical use of a prover it is necessary to have some tools to inspect the failed proofs. On the other hand the implementation of a prover with additional output gets more difficult to read, because some parts of the code are only for producing a more or less nice output. Since the code of the prover describes the built in logic of the prover, it is important to have a well structured and as simple as possible source code.

One way to bring these oppositional demands together is to have two different source codes: One without output generation and another with it. By the way this makes it more difficult to maintain the code. All changes of the prover source code have to be done twice. A new problem occurs: How one can ensure that both implementations behave in the same way?

Another way to solve this problem is very natural: If there is a machine which should be checked why no product is generated, everybody would try to look at the machine while it is working. The same strategy may be used in software development: Let the program run within an environment which allows monitoring the execution flow and protocolling the executed steps.

Mathematica is an interpreted programming language and it has such an environment to monitor the execution. Therefore the proof finding can be done by the SC prover and the output generation is done by a monitoring algorithm. So it is possible to use the SC prover without any changes and there is a facility to inspect the failed proofs.

The use of the original SC prover has the big advantage that if the SC prover works correctly then the proof tracer at least returns the correct result, even if there are bugs in the proof tracer. Such bugs can affect the proof output only, but not the result of the proof.

There are still dependencies between the prover and the proof tracer, but they are only in one direction. Modifications of the prover implementation forces usually changes in the proof tracer, because it depends on the source code of the prover. On the other hand modifications of the proof tracer can be done independently of the prover. It is easily conceivable that there are different proof tracers for one prover. E.g. one produces just simple text output, another generates a full *Theorema* proof object which can be displayed as a written proof or view by focus windows [Buch00].

In the current version of ProofTracer the output is done only as simple text output and formulae and terms are displayed in the internal user language. Using the features of *Theorema* allows printing formulae in traditional mathematical style. E.g. the variable is printed below the quantifier and the ususal symbols like ∀ and ⇒ are used [Tma98].

4.2 Implementation

In *Mathematica* a program is nothing else then a *Mathematica* expression which will be evaluated. The command **Trace** allows to get all intermediate steps of the evaluation. For the proof tracer the related command **TraceScan** is used. This command evaluates two functions on every intermediate expression of certain pattern. One function before the expression is evaluated, the other after the evaluation. This gives the opportunity generate some kind of a proof while the prover is running.

Like the implementation of the prover the proof tracer is implemented within its own package. The package consists of two main parts. First there is the interface for external caller. The main function calls the original SC prover wrapped by **TraceScan**. It handles also the options for the prover and the tracer. The parameter of TraceScan define which expressions should be monitored and which functions will be called while the evaluation.

The second part reflects the implementation of the SC prover. For most definitions in the SC prover there are corresponding definitions in the proof tracer. The same order of the definitions simplifies the maintenance of the source code.

The proof tracer prints immediately an output to the current notebook. For better readability of the output the lines have an indent depending on the depth of the proof. At some points, e.g. if a subproof is done, the proof tracer goes one level deeper in the proof. This is done in the proof tracer only and has nothing to do with the SC prover itself.

In the appendix there is the source code with comments on some technical issues (see *section A.3*).

5 Case Studies

In this section the functionality of the SC prover with induction is present by some examples. The definitions and details on theory exploration in this theory can be found in [Tma99a]. An example with automated lemmata generation in Theorema is given in [Tma99b].

Before the SC prover and the proof tracer are available in *Mathematica* the corresponding packages have to be loaded:

```
Needs["SC`ProverKernel`"];
Needs["SC`Tracer`"];
```

5.1 Natural Numbers: Commutativity of Addition

5.1.1 Definition of Plus

The usual addition of natural numbers is defined in this way [Shoe73, p.22]:

$$\forall_{x} \; x + 0 = 0$$

$$\forall_{x,y} \; x + y^{+} = (x + y)^{+}$$

where x and y are natural numbers and x^{+}denotes the successor of x. The infix notation $x + y$ denotes the application of the binary function Plus to x and y: Plus[x, y]

This function Plus has some interesting properties like commutativity:

$$\forall_{x,y} \; x + y = y + x$$

This property can be shown by the SC prover with induction. First translate the formulae into the syntax (see) of the SC prover:

```
DefPlus1 = ™ForAll[x, ™Equal[™Plus[x, 0], x]];
DefPlus2 =
  ™ForAll[x, ™ForAll[y, ™Equal[™Plus[x, s[y]], s[™Plus[x, y]]]]];
```

5.1.2 Theorem: Commutativity of Plus

In the formula of the theorem **•ind** indicates that the variables x and y are from an inductive domain with base element **0** and constructor **s** (successor):

```
ThmPlus1 = ™ForAll[•ind[x, 0, s],
   ™ForAll[•ind[y, 0, s], ™Equal[™Plus[x, y], ™Plus[y, x]]]];
```

Now the SC prover is called with the theorem:

```
SCProver[ThmPlus1, {DefPlus1, DefPlus2}]
```

```
Failed
```

5.1.3 Proving with SC Prover and Proof Tracer

Since the proof fails we have to find out why. Therefore we can use the proof tracer; the call is similar to the call of the SC prover:

```
SCProof[ThmPlus1, {DefPlus1, DefPlus2}]
```

```
{0}: Proof of:
 ™ForAll[•ind[x, 0, s], ™ForAll[•ind[y, 0, s], ™Equal[™Plus[x, y], ™Plus[y, x]]]]

{0}: Prove by induction:
 ™ForAll[•ind[x, 0, s], ™ForAll[•ind[y, 0, s], ™Equal[™Plus[x, y], ™Plus[y, x]]]]

{0}: Prove base case: ™ForAll[•ind[y, 0, s], ™Equal[™Plus[0, y], ™Plus[y, 0]]]

 {1}: Prove by induction: ™ForAll[•ind[y, 0, s], ™Equal[™Plus[0, y], ™Plus[y, 0]]]

 {1}: Prove base case: ™Equal[™Plus[0, 0], ™Plus[0, 0]]

  {2}: This formula is true: ™Equal[™Plus[0, 0], ™Plus[0, 0]]

 {1}: Assume (induction hypotheses): ™Equal[™Plus[0, •fix[y, 1]], ™Plus[•fix[y, 1], 0]]
and prove (induction step): ™Equal[™Plus[0, s[•fix[y, 1]]], ™Plus[s[•fix[y, 1]], 0]]

  {2}: Rewrite both sides of the equality:
 {™Plus[0, s[•fix[y, 1]]]}=={™Plus[s[•fix[y, 1]], 0]}

  {2}: Rewrite (lhs): {™Plus[0, s[•fix[y, 1]]]} to {s[™Plus[0, •fix[y, 1]]]}

  {2}: Rewrite (rhs): {™Plus[s[•fix[y, 1]], 0]} to {s[•fix[y, 1]]}

  {2}: Rewrite (lhs): {s[™Plus[0, •fix[y, 1]]]} to {s[™Plus[•fix[y, 1], 0]]}

  {2}: Rewrite (lhs): {s[™Plus[•fix[y, 1], 0]]} to {s[•fix[y, 1]]}

  {2}: The term s[•fix[y, 1]]
  appears on both sides of the equality, hence the equality is true.

{0}: Assume (induction hypotheses):
 ™ForAll[•ind[y, 0, s], ™Equal[™Plus[•fix[x, 1], y], ™Plus[y, •fix[x, 1]]]]
and prove (induction step):
 ™ForAll[•ind[y, 0, s], ™Equal[™Plus[s[•fix[x, 1]], y], ™Plus[y, s[•fix[x, 1]]]]]

 {1}: Prove by induction:
 ™ForAll[•ind[y, 0, s], ™Equal[™Plus[s[•fix[x, 1]], y], ™Plus[y, s[•fix[x, 1]]]]]

 {1}: Prove base case: ™Equal[™Plus[s[•fix[x, 1]], 0], ™Plus[0, s[•fix[x, 1]]]]

  {2}: Rewrite both sides of the equality:
 {™Plus[s[•fix[x, 1]], 0]}=={™Plus[0, s[•fix[x, 1]]]}

  {2}: Rewrite (lhs): {™Plus[s[•fix[x, 1]], 0]} to {s[•fix[x, 1]]}

  {2}: Rewrite (rhs): {™Plus[0, s[•fix[x, 1]]]} to {s[™Plus[0, •fix[x, 1]]]}
```

```
Failed
```

Now we can see what happens. First the induction on x begins. In the base case $x = 0$ the induction on y is done. The base case $y = 0$ is simply true because both sides of the equation are identical. Next is the induction step: Assume the property for y and prove it for $s[y]$. After some rewriting steps of the left and the right hand side again identical formulae are found. This finishes the base $x = 0$.

The induction step for $s[x]$ starts also with an induction on y. But the base case $y = 0$ fails. The prover is unable to rewrite the formula:

$$(0 + x_1)^+ \quad \text{to} \quad {x_1}^+$$

This leads to the idea to introduce a lemma which just allows this rewriting step:

$$\underset{x}{\forall}\,(0 + x) = x$$

```
LemmaPlus1 = ™ForAll[•ind[x, 0, s], ™Equal[™Plus[0, x], x]];
```

Try to prove this lemma:

```
SCProver[LemmaPlus1, {DefPlus1, DefPlus2}]
Proved
```

The lemma is valid; hence we can add it to the knowledge base and try the proof of the theorem again:

```
SCProof[ThmPlus1, {DefPlus1, DefPlus2, LemmaPlus1}]
```

The beginning of the proof is the same as before, let's starting with the induction step x to $s[x]$:

```
{0}: Assume (induction hypotheses):
 ™ForAll[•ind[y, 0, s], ™Equal[™Plus[•fix[x, 1], y], ™Plus[y, •fix[x, 1]]]]
and prove (induction step):
 ™ForAll[•ind[y, 0, s], ™Equal[™Plus[s[•fix[x, 1]], y], ™Plus[y, s[•fix[x, 1]]]]]

 {1}: Prove by induction:
 ™ForAll[•ind[y, 0, s], ™Equal[™Plus[s[•fix[x, 1]], y], ™Plus[y, s[•fix[x, 1]]]]]

 {1}: Prove base case: ™Equal[™Plus[s[•fix[x, 1]], 0], ™Plus[0, s[•fix[x, 1]]]]

  {2}: Rewrite both sides of the equality:
 {™Plus[s[•fix[x, 1]], 0]}={™Plus[0, s[•fix[x, 1]]]}

  {2}: Rewrite (lhs): {™Plus[s[•fix[x, 1]], 0]} to {s[•fix[x, 1]]}

  {2}: Rewrite (rhs): {™Plus[0, s[•fix[x, 1]]]}
  to {s[•fix[x, 1]], s[™Plus[0, •fix[x, 1]]]}

  {2}: The term s[•fix[x, 1]]
  appears on both sides of the equality, hence the equality is true.

 {1}: Assume (induction hypotheses):
 ™Equal[™Plus[s[•fix[x, 1]], •fix[y, 1]], ™Plus[•fix[y, 1], s[•fix[x, 1]]]]
and prove (induction step):
 ™Equal[™Plus[s[•fix[x, 1]], s[•fix[y, 1]]], ™Plus[s[•fix[y, 1]], s[•fix[x, 1]]]]

  {2}: Rewrite both sides of the equality:
 {™Plus[s[•fix[x, 1]], s[•fix[y, 1]]]}={™Plus[s[•fix[y, 1]], s[•fix[x, 1]]]}
```

```
{2}: Rewrite (lhs): {™Plus[s[•fix[x, 1]], s[•fix[y, 1]]]}
to {s[™Plus[s[•fix[x, 1]], •fix[y, 1]]]}

{2}: Rewrite (rhs): {™Plus[s[•fix[y, 1]], s[•fix[x, 1]]]}
to {s[™Plus[s[•fix[y, 1]], •fix[x, 1]]]}

{2}: Rewrite (lhs): {s[™Plus[s[•fix[x, 1]], •fix[y, 1]]]}
to {s[™Plus[•fix[y, 1], s[•fix[x, 1]]]]}

{2}: Rewrite (lhs): {s[™Plus[•fix[y, 1], s[•fix[x, 1]]]]}
to {s[s[™Plus[•fix[y, 1], •fix[x, 1]]]]}
```

```
Failed
```

Now the induction step on x fails again. The induction base $y = 0$ works fine with the lemma, but the induction step of y fails. The prover is unable to rewrite this equation:

$$((y_1 + x_1)^+)^+ = (y_1{}^+ + x_1)^+$$

A second lemma will help:

$$\underset{x,y}{\forall}\ x^+ + y = (x + y)^+$$

```
LemmaPlus2 = ™ForAll[x, ™ForAll[•ind[y, 0, s],
    ™Equal[™Plus[s[x], y], s[™Plus[x, y]]]]];
```

Note that the variable x is not declared as a inductive variable although it is one. In this case it is sufficient to do induction on y only. But induction on x is also possible.

```
SCProver[LemmaPlus2, {DefPlus1, DefPlus2}]
```

```
Proved
```

The lemma is true, so add it to the knowledge base:

```
SCProver[ThmPlus1, {DefPlus1, DefPlus2, LemmaPlus1, LemmaPlus2}]
```

```
Proved
```

... and the proof is done.

5.2 Natural Numbers: Multiplication

This case study presents some theorems about multiplication of natural numbers. They are all proven by the SC prover with induction. The focus of this case study is on building up a theory without logical steps. Failed proof attempts and details on theory exploration are omitted here. We present only the theorems and lemmata in traditional mathematical style and translated into user language. The prover is called with these formulae, but no output with ProofTracer is generated. The proofs have similar style as in the case study above, but the occurring formulae are longer.

5.2.1 Definition of Times

First we define the operation Times as the usual multiplication on the natural numbers [Shoe73,p.22]:

$$\forall_{x} \; x * 0 = 0$$

$$\forall_{x,y} \; x * y^{+} = x * y + x$$

```
DefTimes1 = ™ForAll[x, ™Equal[™Times[x, 0], 0]];
DefTimes2 = ™ForAll[x,
   ™ForAll[y, ™Equal[™Times[x, s[y]], ™Plus[™Times[x, y], x]]]];
```

5.2.2 Associativity of Plus

For some properties of the multiplication the associativity of the addition is necessary.

$$\forall_{x,y,x} \; (x + y) + z = x + (y + z)$$

```
ThmPlus2 = ™ForAll[x, ™ForAll[•ind[y, 0, s], ™ForAll[z,
     ™Equal[™Plus[™Plus[x, y], z], ™Plus[x, ™Plus[y, z]]]]]];

SCProver[ThmPlus2, {DefPlus1, DefPlus2, ThmPlus1}]

Proved
```

Let's have a look on the proving flow:

```
SCProof[ThmPlus2, {DefPlus1, DefPlus2, ThmPlus1}]

{0}: Proof of: ™ForAll[x, ™ForAll[•ind[y, 0, s],
   ™ForAll[z, ™Equal[™Plus[™Plus[x, y], z], ™Plus[x, ™Plus[y, z]]]]]]

{0}: Take •fix[x, 1] arbitrary but fixed and prove: ™ForAll[•ind[y, 0, s],
  ™ForAll[z, ™Equal[™Plus[™Plus[•fix[x, 1], y], z], ™Plus[•fix[x, 1], ™Plus[y, z]]]]]

{0}: Prove by induction: ™ForAll[•ind[y, 0, s],
  ™ForAll[z, ™Equal[™Plus[™Plus[•fix[x, 1], y], z], ™Plus[•fix[x, 1], ™Plus[y, z]]]]]

{0}: Prove base case:
 ™ForAll[z, ™Equal[™Plus[™Plus[•fix[x, 1], 0], z], ™Plus[•fix[x, 1], ™Plus[0, z]]]]

 {1}: Take •fix[z, 1] arbitrary but fixed and prove:
 ™Equal[™Plus[™Plus[•fix[x, 1], 0], •fix[z, 1]], ™Plus[•fix[x, 1], ™Plus[0, •fix[z, 1]]]]

 {1}: Rewrite both sides of the equality:
 {™Plus[™Plus[•fix[x, 1], 0], •fix[z, 1]]}=={™Plus[•fix[x, 1], ™Plus[0, •fix[z, 1]]]}

 {1}: Rewrite (lhs): {™Plus[™Plus[•fix[x, 1], 0], •fix[z, 1]]} to
 {™Plus[•fix[x, 1], •fix[z, 1]], ™Plus[•fix[z, 1], ™Plus[•fix[x, 1], 0]],
  ™Plus[™Plus[0, •fix[x, 1]], •fix[z, 1]]}

 {1}: Rewrite (rhs): {™Plus[•fix[x, 1], ™Plus[0, •fix[z, 1]]]} to
 {™Plus[•fix[x, 1], ™Plus[•fix[z, 1], 0]], ™Plus[™Plus[0, •fix[z, 1]], •fix[x, 1]]}

 {1}: Rewrite (lhs): {™Plus[•fix[x, 1], •fix[z, 1]],
  ™Plus[•fix[z, 1], ™Plus[•fix[x, 1], 0]], ™Plus[™Plus[0, •fix[x, 1]], •fix[z, 1]]}
  to {™Plus[•fix[z, 1], •fix[x, 1]], ™Plus[•fix[z, 1], ™Plus[0, •fix[x, 1]]]}
```

```
 {1}: Rewrite (rhs):
 {™Plus[•fix[x, 1], ™Plus[•fix[z, 1], 0]], ™Plus[™Plus[0, •fix[z, 1]], •fix[x, 1]]}
  to {™Plus[•fix[x, 1], •fix[z, 1]], ™Plus[™Plus[•fix[z, 1], 0], •fix[x, 1]]}

 {1}: The term ™Plus[•fix[x, 1], •fix[z, 1]]
  appears on both sides of the equality, hence the equality is true.

{0}: Assume (induction hypotheses):
 ™ForAll[z, ™Equal[™Plus[™Plus[•fix[x, 1], •fix[y, 1]], z],
   ™Plus[•fix[x, 1], ™Plus[•fix[y, 1], z]]]]
and prove (induction step): ™ForAll[z, ™Equal[™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], z],
   ™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], z]]]]

 {1}: Take •fix[z, 1] arbitrary but fixed and prove:
 ™Equal[™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], •fix[z, 1]],
  ™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], •fix[z, 1]]]]

 {1}: Rewrite both sides of the equality:
 {™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], •fix[z, 1]]}
 =={™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], •fix[z, 1]]]}

 {1}: Rewrite (lhs): {™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], •fix[z, 1]]}
  to {™Plus[s[™Plus[•fix[x, 1], •fix[y, 1]]], •fix[z, 1]],
  ™Plus[•fix[z, 1], ™Plus[•fix[x, 1], s[•fix[y, 1]]]],
  ™Plus[™Plus[s[•fix[y, 1]], •fix[x, 1]], •fix[z, 1]]}

 {1}: Rewrite (rhs): {™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], •fix[z, 1]]]}
  to {™Plus[•fix[x, 1], ™Plus[•fix[z, 1], s[•fix[y, 1]]]],
  ™Plus[™Plus[s[•fix[y, 1]], •fix[z, 1]], •fix[x, 1]]}

 {1}: Rewrite (lhs): {™Plus[s[™Plus[•fix[x, 1], •fix[y, 1]]], •fix[z, 1]],
  ™Plus[•fix[z, 1], ™Plus[•fix[x, 1], s[•fix[y, 1]]]],
  ™Plus[™Plus[s[•fix[y, 1]], •fix[x, 1]], •fix[z, 1]]}
  to {™Plus[s[™Plus[•fix[y, 1], •fix[x, 1]]], •fix[z, 1]],
  ™Plus[•fix[z, 1], s[™Plus[•fix[x, 1], •fix[y, 1]]]],
  ™Plus[•fix[z, 1], ™Plus[s[•fix[y, 1]], •fix[x, 1]]]}

 {1}: Rewrite (rhs): {™Plus[•fix[x, 1], ™Plus[•fix[z, 1], s[•fix[y, 1]]]],
  ™Plus[™Plus[s[•fix[y, 1]], •fix[z, 1]], •fix[x, 1]]}
  to {™Plus[•fix[x, 1], s[™Plus[•fix[z, 1], •fix[y, 1]]]],
  ™Plus[™Plus[•fix[z, 1], s[•fix[y, 1]]], •fix[x, 1]]}

 {1}: Rewrite (lhs): {™Plus[s[™Plus[•fix[y, 1], •fix[x, 1]]], •fix[z, 1]],
  ™Plus[•fix[z, 1], s[™Plus[•fix[x, 1], •fix[y, 1]]]],
  ™Plus[•fix[z, 1], ™Plus[s[•fix[y, 1]], •fix[x, 1]]]}
  to {s[™Plus[•fix[z, 1], ™Plus[•fix[x, 1], •fix[y, 1]]]],
  ™Plus[•fix[z, 1], s[™Plus[•fix[y, 1], •fix[x, 1]]]]}

 {1}: Rewrite (rhs): {™Plus[•fix[x, 1], s[™Plus[•fix[z, 1], •fix[y, 1]]]],
  ™Plus[™Plus[•fix[z, 1], s[•fix[y, 1]]], •fix[x, 1]]}
  to {s[™Plus[•fix[x, 1], ™Plus[•fix[z, 1], •fix[y, 1]]]],
  ™Plus[s[™Plus[•fix[z, 1], •fix[y, 1]]], •fix[x, 1]],
  ™Plus[•fix[x, 1], s[™Plus[•fix[y, 1], •fix[z, 1]]]]}

 {1}: Rewrite (lhs): {s[™Plus[•fix[z, 1], ™Plus[•fix[x, 1], •fix[y, 1]]]],
  ™Plus[•fix[z, 1], s[™Plus[•fix[y, 1], •fix[x, 1]]]]}
  to {s[™Plus[•fix[z, 1], ™Plus[•fix[y, 1], •fix[x, 1]]]],
  s[™Plus[™Plus[•fix[x, 1], •fix[y, 1]], •fix[z, 1]]]}

 {1}: Rewrite (rhs): {s[™Plus[•fix[x, 1], ™Plus[•fix[z, 1], •fix[y, 1]]]],
  ™Plus[s[™Plus[•fix[z, 1], •fix[y, 1]]], •fix[x, 1]],
  ™Plus[•fix[x, 1], s[™Plus[•fix[y, 1], •fix[z, 1]]]]}
  to {s[™Plus[•fix[x, 1], ™Plus[•fix[y, 1], •fix[z, 1]]]],
  s[™Plus[™Plus[•fix[z, 1], •fix[y, 1]], •fix[x, 1]]],
  ™Plus[s[™Plus[•fix[y, 1], •fix[z, 1]]], •fix[x, 1]]}
```

```
{1}: Rewrite (lhs): {s[™Plus[•fix[z, 1], ™Plus[•fix[y, 1], •fix[x, 1]]]],
 s[™Plus[™Plus[•fix[x, 1], •fix[y, 1]], •fix[z, 1]]]}
 to {s[™Plus[•fix[x, 1], ™Plus[•fix[y, 1], •fix[z, 1]]]],
 s[™Plus[™Plus[•fix[y, 1], •fix[x, 1]], •fix[z, 1]]]}

{1}: The term s[™Plus[•fix[x, 1], ™Plus[•fix[y, 1], •fix[z, 1]]]]
 appears on both sides of the equality, hence the equality is true.
```

Proved

We see that the proof is done in a way like humans will do. First induction is applied and then in the base case as well as in the induction step case the rewriting engine of the SC prover finds identical terms on both sides of the equality.

5.2.3 Associativity of Plus - Alternative Proof with Double Induction

The proof of associativity in the section above uses the commutativity of ™Plus. It is possible to proof associativity without further knowledge. But in this case nested induction is necessary.

```
ThmPlus2a =
  ™ForAll[x, ™ForAll[•ind[y, 0, s], ™ForAll[•ind[z, 0, s],
     ™Equal[™Plus[™Plus[x, y], z], ™Plus[x, ™Plus[y, z]]]]]];
```

We define the theorem again, but this time we declare induction for the variables **y** and **z**.

```
SCProver[ThmPlus2a, {DefPlus1, DefPlus2}]
```

Proved

The proof was successful. We look at the proof:

```
SCProof[ThmPlus2a, {DefPlus1, DefPlus2}]
```

```
{0}: Proof of: ™ForAll[x, ™ForAll[•ind[y, 0, s],
   ™ForAll[•ind[z, 0, s], ™Equal[™Plus[™Plus[x, y], z], ™Plus[x, ™Plus[y, z]]]]]]

{0}: Take •fix[x, 1] arbitrary but fixed and prove:
 ™ForAll[•ind[y, 0, s], ™ForAll[•ind[z, 0, s],
   ™Equal[™Plus[™Plus[•fix[x, 1], y], z], ™Plus[•fix[x, 1], ™Plus[y, z]]]]]

{0}: Prove by induction: ™ForAll[•ind[y, 0, s], ™ForAll[•ind[z, 0, s],
   ™Equal[™Plus[™Plus[•fix[x, 1], y], z], ™Plus[•fix[x, 1], ™Plus[y, z]]]]]

{0}: Prove base case: ™ForAll[•ind[z, 0, s],
  ™Equal[™Plus[™Plus[•fix[x, 1], 0], z], ™Plus[•fix[x, 1], ™Plus[0, z]]]]

 {1}: Prove by induction: ™ForAll[•ind[z, 0, s],
  ™Equal[™Plus[™Plus[•fix[x, 1], 0], z], ™Plus[•fix[x, 1], ™Plus[0, z]]]]

 {1}: Prove base case:
 ™Equal[™Plus[™Plus[•fix[x, 1], 0], 0], ™Plus[•fix[x, 1], ™Plus[0, 0]]]

  {2}: Rewrite both sides of the equality:
 {™Plus[™Plus[•fix[x, 1], 0], 0]}=={™Plus[•fix[x, 1], ™Plus[0, 0]]}

  {2}: Rewrite (lhs): {™Plus[™Plus[•fix[x, 1], 0], 0]} to {™Plus[•fix[x, 1], 0]}

  {2}: Rewrite (rhs): {™Plus[•fix[x, 1], ™Plus[0, 0]]} to {™Plus[•fix[x, 1], 0]}
```

```
  {2}: The term ™Plus[•fix[x, 1], 0]
  appears on both sides of the equality, hence the equality is true.

 {1}: Assume (induction hypotheses):
 ™Equal[™Plus[™Plus[•fix[x, 1], 0], •fix[z, 1]], ™Plus[•fix[x, 1], ™Plus[0, •fix[z, 1]]]]
and prove (induction step): ™Equal[™Plus[™Plus[•fix[x, 1], 0], s[•fix[z, 1]]],
  ™Plus[•fix[x, 1], ™Plus[0, s[•fix[z, 1]]]]]

  {2}: Rewrite both sides of the equality: {™Plus[™Plus[•fix[x, 1], 0], s[•fix[z, 1]]]}
 =={™Plus[•fix[x, 1], ™Plus[0, s[•fix[z, 1]]]]}

  {2}: Rewrite (lhs): {™Plus[™Plus[•fix[x, 1], 0], s[•fix[z, 1]]]} to
 {s[™Plus[™Plus[•fix[x, 1], 0], •fix[z, 1]]], ™Plus[•fix[x, 1], s[•fix[z, 1]]]}

  {2}: Rewrite (rhs): {™Plus[•fix[x, 1], ™Plus[0, s[•fix[z, 1]]]]}
  to {™Plus[•fix[x, 1], s[™Plus[0, •fix[z, 1]]]]}

  {2}: Rewrite (lhs):
 {s[™Plus[™Plus[•fix[x, 1], 0], •fix[z, 1]]], ™Plus[•fix[x, 1], s[•fix[z, 1]]]} to
 {s[™Plus[•fix[x, 1], •fix[z, 1]]], s[™Plus[•fix[x, 1], ™Plus[0, •fix[z, 1]]]]}

  {2}: Rewrite (rhs): {™Plus[•fix[x, 1], s[™Plus[0, •fix[z, 1]]]]}
  to {s[™Plus[•fix[x, 1], ™Plus[0, •fix[z, 1]]]]}

  {2}: The term s[™Plus[•fix[x, 1], ™Plus[0, •fix[z, 1]]]]
  appears on both sides of the equality, hence the equality is true.

{0}: Assume (induction hypotheses):
 ™ForAll[•ind[z, 0, s], ™Equal[™Plus[™Plus[•fix[x, 1], •fix[y, 1]], z],
   ™Plus[•fix[x, 1], ™Plus[•fix[y, 1], z]]]]
and prove (induction step): ™ForAll[•ind[z, 0, s],
  ™Equal[™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], z],
   ™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], z]]]]

 {1}: Prove by induction:
 ™ForAll[•ind[z, 0, s], ™Equal[™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], z],
   ™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], z]]]]

 {1}: Prove base case: ™Equal[
  ™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], 0], ™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], 0]]]

  {2}: Rewrite both sides of the equality: {™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], 0]}
 =={™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], 0]]}

  {2}: Rewrite (lhs): {™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], 0]} to
 {™Plus[s[™Plus[•fix[x, 1], •fix[y, 1]]], 0], ™Plus[•fix[x, 1], s[•fix[y, 1]]]}

  {2}: Rewrite (rhs): {™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], 0]]}
  to {™Plus[•fix[x, 1], s[•fix[y, 1]]]}

  {2}: The term ™Plus[•fix[x, 1], s[•fix[y, 1]]]
  appears on both sides of the equality, hence the equality is true.

 {1}: Assume (induction hypotheses):
 ™Equal[™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], •fix[z, 1]],
  ™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], •fix[z, 1]]]]
and prove (induction step): ™Equal[™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]],
   s[•fix[z, 1]]], ™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], s[•fix[z, 1]]]]]

  {2}: Rewrite both sides of the equality:
 {™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], s[•fix[z, 1]]]}
 =={™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], s[•fix[z, 1]]]]}

  {2}: Rewrite (lhs): {™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], s[•fix[z, 1]]]}
  to {s[™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], •fix[z, 1]]],
  ™Plus[s[™Plus[•fix[x, 1], •fix[y, 1]]], s[•fix[z, 1]]]}

  {2}: Rewrite (rhs): {™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], s[•fix[z, 1]]]]}
  to {™Plus[•fix[x, 1], s[™Plus[s[•fix[y, 1]], •fix[z, 1]]]]}
```

```
{2}: Rewrite (lhs): {s[™Plus[™Plus[•fix[x, 1], s[•fix[y, 1]]], •fix[z, 1]]],
™Plus[s[™Plus[•fix[x, 1], •fix[y, 1]]], s[•fix[z, 1]]]}
to {s[™Plus[s[™Plus[•fix[x, 1], •fix[y, 1]]], •fix[z, 1]]],
s[™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], •fix[z, 1]]]]}

{2}: Rewrite (rhs): {™Plus[•fix[x, 1], s[™Plus[s[•fix[y, 1]], •fix[z, 1]]]]}
to {s[™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], •fix[z, 1]]]]}

{2}: The term s[™Plus[•fix[x, 1], ™Plus[s[•fix[y, 1]], •fix[z, 1]]]]
appears on both sides of the equality, hence the equality is true.
```

```
Proved
```

The prover takes variable **x** arbitrary but fixed. First induction is applied for the variable **y** and in each case – base case and induction step – induction is applied for the variable **z**.

5.2.4 Commutativity of Times

$$\underset{x,y}{\forall}\ x * y = y * x$$

```
ThmTimes1 = ™ForAll[•ind[x, 0, s],
   ™ForAll[y, ™Equal[™Times[x, y], ™Times[y, x]]]];
```

For the proof of commutativity we will need some lemmata, similar to the lemmata for the addition. They can be found again by inspection the failed proofs. How they are found is described in [Tma99a].

$$\underset{x}{\forall}\ 0 * x = 0$$

$$\underset{x,y}{\forall}\ x^{+} * y = x * y + y$$

```
LemmaTimes1 = ™ForAll[•ind[x, 0, s], ™Equal[™Times[0, x], 0]];
LemmaTimes2 = ™ForAll[x, ™ForAll[•ind[y, 0, s],
    ™Equal[™Times[s[x], y], ™Plus[™Times[x, y], y]]]];

SCProver[LemmaTimes1, {DefTimes1, DefTimes2, DefPlus1}]
```

```
Proved
```

```
SCProver[LemmaTimes2,
 {DefTimes1, DefTimes2, DefPlus1, DefPlus2, ThmPlus1, ThmPlus2}]
```

```
Proved
```

```
SCProver[ThmTimes1,
 {DefTimes1, DefTimes2, LemmaTimes1, LemmaTimes2}]
```

```
Proved
```

Since the prover was able to prove the theorems we do not look on the details of the proofs. Nevertheless it is possible to do so.

5.2.5 Distributivity of Plus and Times

$$\forall_{x,y,z} \; x * (y + z) = x * y + x * z$$

$$\forall_{x,y,z} \; (x + y) * z = x * z + y * z$$

```
ThmDistLeft = ™ForAll[x, ™ForAll[y, ™ForAll[•ind[z, 0, s],
     ™Equal[™Times[x, ™Plus[y, z]],
      ™Plus[™Times[x, y], ™Times[x, z]]]]]];
ThmDistRight = ™ForAll[x, ™ForAll[y, ™ForAll[z,
     ™Equal[™Times[™Plus[x, y], z],
      ™Plus[™Times[x, z], ™Times[y, z]]]]]];
ThmDist = ThmDistLeft;
```

(Left) Distributivity is provable by using the definitions and the associativity of ™Plus:

```
SCProver[ThmDistLeft,
 {DefTimes1, DefTimes2, DefPlus1, DefPlus2, ThmPlus2}]
```

```
Proved
```

Right distributivity is provable by using left distributivity and commutativity of ™Times:

```
SCProver[ThmDistRight, {ThmTimes1, ThmDistLeft}]
```

```
Proved
```

Here is the proof:

```
SCProof[ThmDistRight, {ThmTimes1, ThmDistLeft}]
```

```
{0}: Proof of: ™ForAll[x, ™ForAll[y,
   ™ForAll[z, ™Equal[™Times[™Plus[x, y], z], ™Plus[™Times[x, z], ™Times[y, z]]]]]]

{0}: Take •fix[x, 1] arbitrary but fixed and prove: ™ForAll[y, ™ForAll[z,
   ™Equal[™Times[™Plus[•fix[x, 1], y], z], ™Plus[™Times[•fix[x, 1], z], ™Times[y, z]]]]]

{0}: Take •fix[y, 1] arbitrary but fixed and prove:
 ™ForAll[z, ™Equal[™Times[™Plus[•fix[x, 1], •fix[y, 1]], z],
   ™Plus[™Times[•fix[x, 1], z], ™Times[•fix[y, 1], z]]]]

{0}: Take •fix[z, 1] arbitrary but fixed and prove:
 ™Equal[™Times[™Plus[•fix[x, 1], •fix[y, 1]], •fix[z, 1]],
  ™Plus[™Times[•fix[x, 1], •fix[z, 1]], ™Times[•fix[y, 1], •fix[z, 1]]]]

{0}: Rewrite both sides of the equality:
 {™Times[™Plus[•fix[x, 1], •fix[y, 1]], •fix[z, 1]]}==
 {™Plus[™Times[•fix[x, 1], •fix[z, 1]], ™Times[•fix[y, 1], •fix[z, 1]]]}

{0}: Rewrite (lhs): {™Times[™Plus[•fix[x, 1], •fix[y, 1]], •fix[z, 1]]}
  to {™Times[•fix[z, 1], ™Plus[•fix[x, 1], •fix[y, 1]]]}

{0}: Rewrite (rhs):
 {™Plus[™Times[•fix[x, 1], •fix[z, 1]], ™Times[•fix[y, 1], •fix[z, 1]]]}
  to {™Plus[™Times[•fix[x, 1], •fix[z, 1]], ™Times[•fix[z, 1], •fix[y, 1]]],
  ™Plus[™Times[•fix[z, 1], •fix[x, 1]], ™Times[•fix[y, 1], •fix[z, 1]]]}

{0}: Rewrite (lhs): {™Times[•fix[z, 1], ™Plus[•fix[x, 1], •fix[y, 1]]]}
  to {™Plus[™Times[•fix[z, 1], •fix[x, 1]], ™Times[•fix[z, 1], •fix[y, 1]]]}

{0}: Rewrite (rhs):
 {™Plus[™Times[•fix[x, 1], •fix[z, 1]], ™Times[•fix[z, 1], •fix[y, 1]]],
  ™Plus[™Times[•fix[z, 1], •fix[x, 1]], ™Times[•fix[y, 1], •fix[z, 1]]]}
  to {™Plus[™Times[•fix[z, 1], •fix[x, 1]], ™Times[•fix[z, 1], •fix[y, 1]]]}

{0}: The term ™Plus[™Times[•fix[z, 1], •fix[x, 1]], ™Times[•fix[z, 1], •fix[y, 1]]]
  appears on both sides of the equality, hence the equality is true.

Proved
```

5.2.6 Associativity of Times

$$\forall_{x,y,x} \; (x * y) * z = x * (y * z)$$

```
ThmTimes2 = ™ForAll[x, ™ForAll[y, ™ForAll[•ind[z, 0, s],
     ™Equal[™Times[™Times[x, y], z], ™Times[x, ™Times[y, z]]]]]];

SCProver[ThmTimes2, {DefTimes1, DefTimes2, ThmDistLeft}]

Proved
```

As we can see associativity of ™Times is a consequence of the distributivity.

5.3 Correctness of Square and Multiply Algorithm

The exponentiation of natural numbers is defined in a recursive way. The runtime of computing x^n needs in the native way n multiplications. In application, mainly in cryptography, this might be to long especially if there are big numbers involved. Fortunately there exists more efficient ways to do this. One basic method is the well known "Square and Multiply" algorithm which gets along with approximately $2 * log(n)$ multiplications [Gord98, Knut81].

In this case study we will give a proof of the fact that the algorithm gives correct results. We will use some facts which we will not prove from definitions; we will restrict our work to the main proof which is based on some lemmata. This proof shows the capabilities of the prover in relation to case distinctions.

5.3.1 Definitions in Theorema User Language

First we give definitions, the theorem and the necessary knowledge in *Theorema* user language [Tma98]. This is not necessary for proving with SC prover; nevertheless the syntax is more natural and hence easier to read. It may also help to formalize the formulae in the user language of SC prover.

For executing the following definitions in *Mathematica* it is necessary to load *Theorema*:

```
Needs["Theorema`"];
```

The first definition defines the usual exponentiation in a recursive way and it uses a case distinction. The function for exponentiation is called **E**:

```
Definition["Def E",
 ∀x,n (E[x, n] = { 1          ⇐ n = 0
                  { x * E[x, pre[n]]  ⇐ otherwise )
]
```

It uses some implicit knowledge which was not used in the previous case studies about natural numbers. We assume that there are already definitions for the constant **1**,which is equal to **s[0]**,and the function symbol **pre[n]**,which should denote the predecessor. There is no need for an explicit definition in the later proof, because the implicit knowledge about this symbols given in the definitions is sufficient.

Next we define the new algorithm. The essential difference is that in case of an even exponent the computation can be speeded up by squaring the basis **x** and halving the exponent **n**. **Square**, **Half** and **is-even** are again implicit defined functions respective a predicate. There knowledge is given by later defined lemmata. The function for the square and multiply algorithm is called **exp**:

```
Definition["Def exp",
  ∀x,n (exp[x, n] = { 1                          ⇐ n = 0
                    { exp[Square[x], Half[n]]    ⇐ is-even[n]
                    { x * exp[x, pre[n]]         ⇐ otherwise )
]
```

The correctness theorem for this algorithm is obvious. We have to check if the result of both functions **E** and **exp** are equal for every input. We do not care about termination in this case study; hence the theorem is actually about partial correctness.

```
Theorem["exp=E",
  ∀x,n (exp[x, n] = E[x, n])
];
```

For proving this theorem there is some knowledge about the occurring symbols necessary. We will add this knowledge by defining three lemmata, which we will not prove in the scope of this case study. There validity is nearly obvious and can be proven more or less from the definitions. Assuming that a suitable axiomatization is used.

```
Lemma["1", ∀n (pre[n] < n ⇔ n ≠ 0)]

Lemma["2", ∀n (Half[n] < n ⇔ n ≠ 0)]

Lemma["3", ∀x,n (is-even[n] ⇒ (E[x, n] = E[Square[x], Half[n]]))]
```

As one might recognize, the third lemma provides the essential fact for the correctness of the new algorithm.

Theorema allows showing the stored data by calling the definition again without formula. For example:

```
Lemma["1"]

•lma[1, •range[], True, •flist[•lf[, ∀n (pre[n] < n ⇔ n ≠ 0)]]]
```

Now we can look at the last part – the formula – in so called input form. This is provided by *Mathematica* and it shows an alternative syntax for manual input:

```
Lemma["1"][[4, 1, 2]] // InputForm

™ForAll[•range[•simpleRange[•var[n]]], True,
 ™Iff[™Less[pre[•var[n]], •var[n]],
  ™Not[™Equal[•var[n], 0]]]]
```

This input form is similar to the user language of SC prover and may be used as basis for translation. The universal quantifier has a simpler syntax and bound variables are not wrapped by the symbol **•var** in the user language.

5.3.2 Translation into SC Prover User Language

Now we give all formulae in the user language of SC prover. For details on the syntax see *user language*. The meaning of the formulae was already described in the section above.

```
defE = ™ForAll[x, ™ForAll[n, ™Equal[™E[x, n], ™CaseDistinction[
       •case[1, ™Equal[n, 0]],
       •case[™Times[x, ™E[x, pre[n]]], True]
      ]]]];

defExp =
  ™ForAll[x, ™ForAll[n, ™Equal[exp[x, n], ™CaseDistinction[
       •case[1, ™Equal[n, 0]],
       •case[exp[™Square[x], ™Half[n]], is–even[n]],
       •case[™Times[x, exp[x, pre[n]]], True]
      ]]]];

theorem = ™ForAll[•ind[n, ™Less],
   ™ForAll[x, ™Equal[exp[x, n], ™E[x, n]]]];

Lemma1 = ™ForAll[n, ™Iff[™Less[pre[n], n], ™Not[™Equal[n, 0]]]];

Lemma2 =
  ™ForAll[n, ™Iff[™Less[™Half[n], n], ™Not[™Equal[n, 0]]]];

Lemma3 = ™ForAll[x, ™ForAll[n, ™Implies[is–even[n],
     ™Equal[™E[x, n], ™E[™Square[x], ™Half[n]]]]]];
```

We collect the formulae of the knowledge base. This makes it easier to use later.

```
KB = {defE, defExp, Lemma1, Lemma2, Lemma3};
```

5.3.3 Proof

The prover is called with the goal formula and the knowledge.

```
SCProver[theorem, KB]
```

```
Failed
```

Since there are case distinctions in the goal, a proof by case might be helpful. So we allow the prover to do this by activating the option **ProveByCases**:

```
SCProver[theorem, KB, ProveByCases → True]
```

```
Proved
```

This was successful. We are interested in the proof flow. So we call the prover wrapped by the proof tracer. The proof is a little bit long, but it shows well how the prover works.

Note: In front of each line there is a number which gives to the "depth" in the proof. This is not used by the prover, but it is generated by the proof tracer to recognize the structure of proof easier. Sub proofs always increase the depth.

SCProof[theorem, KB, ProveByCases → True]

```
{0}: Proof of: ™ForAll[•ind[n, ™Less], ™ForAll[x, ™Equal[exp[x, n], ™E[x, n]]]]

{0}: Prove by Course-of-Values-Induction:
 ™ForAll[•ind[n, ™Less], ™ForAll[x, ™Equal[exp[x, n], ™E[x, n]]]]

{0}: Assume (induction hypotheses):
 ™ForAll[n, ™Implies[™Less[n, •fix[n, 1]], ™ForAll[x, ™Equal[exp[x, n], ™E[x, n]]]]]

{0}: And prove (induction step):
 ™ForAll[x, ™Equal[exp[x, •fix[n, 1]], ™E[x, •fix[n, 1]]]]

{0}: Take •fix[x, 1] arbitrary but fixed and prove:
 ™Equal[exp[•fix[x, 1], •fix[n, 1]], ™E[•fix[x, 1], •fix[n, 1]]]

{0}: Rewrite both sides of the equality:
 {exp[•fix[x, 1], •fix[n, 1]]}={™E[•fix[x, 1], •fix[n, 1]]}

{0}: Rewrite (lhs): {exp[•fix[x, 1], •fix[n, 1]]} to
 {™CaseDistinction[•case[™E[•fix[x, 1], •fix[n, 1]], ™Less[•fix[n, 1], •fix[n, 1]]]],
  ™CaseDistinction[•case[1, ™Equal[•fix[n, 1], 0]],
   •case[exp[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]], is-even[•fix[n, 1]]],
   •case[™Times[•fix[x, 1], exp[•fix[x, 1], pre[•fix[n, 1]]]], True]]}

 {1}: Try to prove these case conditions: {™Less[•fix[n, 1], •fix[n, 1]]}

 {1}: Prove/Disprove: ™Less[•fix[n, 1], •fix[n, 1]]

  {2}: Rewrite this formula: {™Less[•fix[n, 1], •fix[n, 1]]}

  {2}: Rewrite this formula: {™Not[™Less[•fix[n, 1], •fix[n, 1]]]}

 {1}: Proof of sub goal failed: ™Less[•fix[n, 1], •fix[n, 1]]

 {1}: No case was proven.

 {1}: Try to prove these case conditions:
 {™Equal[•fix[n, 1], 0], is-even[•fix[n, 1]], True}

 {1}: Prove/Disprove: ™Equal[•fix[n, 1], 0]

  {2}: Rewrite both sides of the equality: {•fix[n, 1]}={0}

  {2}: Rewrite this formula: {™Not[™Equal[•fix[n, 1], 0]]}

 {1}: Proof of sub goal failed: ™Equal[•fix[n, 1], 0]

 {1}: Prove/Disprove: is-even[•fix[n, 1]]

  {2}: Rewrite this formula: {is-even[•fix[n, 1]]}

  {2}: Rewrite this formula: {™Not[is-even[•fix[n, 1]]]}

 {1}: Proof of sub goal failed: is-even[•fix[n, 1]]

 {1}: Prove/Disprove: True

  {2}: This formula is true: True

 {1}: Sub goal proved: True
```

```
 {1}: No case was proven completely, the remaining case conditions may be used
   for proving by cases: {™Equal[•fix[n, 1], 0], is-even[•fix[n, 1]], True}

{0}: Rewrite (rhs): {™E[•fix[x, 1], •fix[n, 1]]} to
 {™CaseDistinction[•case[™E[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]],
    is-even[•fix[n, 1]]]], ™CaseDistinction[•case[1, ™Equal[•fix[n, 1], 0]],
   •case[™Times[•fix[x, 1], ™E[•fix[x, 1], pre[•fix[n, 1]]]], True]]}

 {1}: Try to prove these case conditions: {is-even[•fix[n, 1]]}

 {1}: Prove/Disprove: is-even[•fix[n, 1]]

  {2}: Rewrite this formula: {is-even[•fix[n, 1]]}

  {2}: Rewrite this formula: {™Not[is-even[•fix[n, 1]]]}

 {1}: Proof of sub goal failed: is-even[•fix[n, 1]]

 {1}: No case was proven.

 {1}: Try to prove these case conditions: {™Equal[•fix[n, 1], 0], True}

 {1}: Prove/Disprove: ™Equal[•fix[n, 1], 0]

  {2}: Rewrite both sides of the equality: {•fix[n, 1]}={0}

  {2}: Rewrite this formula: {™Not[™Equal[•fix[n, 1], 0]]}

 {1}: Proof of sub goal failed: ™Equal[•fix[n, 1], 0]

 {1}: Prove/Disprove: True

  {2}: This formula is true: True

 {1}: Sub goal proved: True

 {1}: No case was proven completely, the remaining case
   conditions may be used for proving by cases: {™Equal[•fix[n, 1], 0], True}

{0}: Proof by cases of: {exp[•fix[x, 1], •fix[n, 1]]} = {™E[•fix[x, 1], •fix[n, 1]]}

{0}:  Using these cases:
 {{™Equal[•fix[n, 1], 0]}, {™Not[™Equal[•fix[n, 1], 0]], is-even[•fix[n, 1]]},
  {™Not[™Equal[•fix[n, 1], 0]], ™Not[is-even[•fix[n, 1]]]]}}

{0}: Proving case: {™Equal[•fix[n, 1], 0]}

 {1}: Rewrite (lhs): {exp[•fix[x, 1], •fix[n, 1]]} to {exp[•fix[x, 1], 0],
  ™CaseDistinction[•case[™E[•fix[x, 1], •fix[n, 1]], ™Less[•fix[n, 1], •fix[n, 1]]]],
  ™CaseDistinction[•case[1, ™Equal[•fix[n, 1], 0]],
   •case[exp[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]], is-even[•fix[n, 1]]],
   •case[™Times[•fix[x, 1], exp[•fix[x, 1], pre[•fix[n, 1]]]], True]]}

  {2}: Try to prove these case conditions: {™Less[•fix[n, 1], •fix[n, 1]]}

  {2}: Prove/Disprove: ™Less[•fix[n, 1], •fix[n, 1]]

   {3}: Rewrite this formula: {™Less[•fix[n, 1], •fix[n, 1]]}

   {3}: Rewrite formula: {™Less[•fix[n, 1], •fix[n, 1]]}
  to {™Less[0, •fix[n, 1]], ™Less[•fix[n, 1], 0]}

   {3}: Rewrite formula: {™Less[0, •fix[n, 1]], ™Less[•fix[n, 1], 0]} to {™Less[0, 0]}

   {3}: Rewrite this formula: {™Not[™Less[•fix[n, 1], •fix[n, 1]]]}

   {3}: Rewrite formula: {™Not[™Less[•fix[n, 1], •fix[n, 1]]]}
  to {™Not[™Less[0, •fix[n, 1]]], ™Not[™Less[•fix[n, 1], 0]]}
```

```
   {3}: Rewrite formula:
 {™Not[™Less[0, •fix[n, 1]]], ™Not[™Less[•fix[n, 1], 0]]} to {™Not[™Less[0, 0]]}

  {2}: Proof of sub goal failed: ™Less[•fix[n, 1], •fix[n, 1]]

  {2}: No case was proven.

  {2}: Try to prove these case conditions:
 {™Equal[•fix[n, 1], 0], is-even[•fix[n, 1]], True}

  {2}: Prove/Disprove: ™Equal[•fix[n, 1], 0]

   {3}: Rewrite both sides of the equality: {•fix[n, 1]}={0}

   {3}: Rewrite (lhs): {•fix[n, 1]} to {0}

   {3}: The term 0 appears on both sides of the equality, hence the equality is true.

  {2}: Sub goal proved: ™Equal[•fix[n, 1], 0]

  {2}: This term will be used: 1

 {1}: Rewrite (rhs): {™E[•fix[x, 1], •fix[n, 1]]} to
 {™CaseDistinction[•case[™E[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]],
    is-even[•fix[n, 1]]]], ™CaseDistinction[•case[1, ™Equal[•fix[n, 1], 0]],
   •case[™Times[•fix[x, 1], ™E[•fix[x, 1], pre[•fix[n, 1]]]], True]], ™E[•fix[x, 1], 0]}

  {2}: Try to prove these case conditions: {is-even[•fix[n, 1]]}

  {2}: Prove/Disprove: is-even[•fix[n, 1]]

   {3}: Rewrite this formula: {is-even[•fix[n, 1]]}

   {3}: Rewrite formula: {is-even[•fix[n, 1]]} to {is-even[0]}

   {3}: Rewrite this formula: {™Not[is-even[•fix[n, 1]]]}

   {3}: Rewrite formula: {™Not[is-even[•fix[n, 1]]]} to {™Not[is-even[0]]}

  {2}: Proof of sub goal failed: is-even[•fix[n, 1]]

  {2}: No case was proven.

  {2}: Try to prove these case conditions: {™Equal[•fix[n, 1], 0], True}

  {2}: Prove/Disprove: ™Equal[•fix[n, 1], 0]

   {3}: Rewrite both sides of the equality: {•fix[n, 1]}={0}

   {3}: Rewrite (lhs): {•fix[n, 1]} to {0}

   {3}: The term 0 appears on both sides of the equality, hence the equality is true.

  {2}: Sub goal proved: ™Equal[•fix[n, 1], 0]

  {2}: This term will be used: 1

 {1}: The term 1 appears on both sides of the equality, hence the equality is true.

 {1}: Case proved.

{0}: Proving case: {™Not[™Equal[•fix[n, 1], 0]], is-even[•fix[n, 1]]}

 {1}: Rewrite (lhs): {exp[•fix[x, 1], •fix[n, 1]]} to
 {™CaseDistinction[•case[™E[•fix[x, 1], •fix[n, 1]], ™Less[•fix[n, 1], •fix[n, 1]]]],
  ™CaseDistinction[•case[1, ™Equal[•fix[n, 1], 0]],
   •case[exp[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]], is-even[•fix[n, 1]]],
   •case[™Times[•fix[x, 1], exp[•fix[x, 1], pre[•fix[n, 1]]]], True]]}
```

```
 {2}: Try to prove these case conditions: {™Less[•fix[n, 1], •fix[n, 1]]}

 {2}: Prove/Disprove: ™Less[•fix[n, 1], •fix[n, 1]]

  {3}: Rewrite this formula: {™Less[•fix[n, 1], •fix[n, 1]]}

  {3}: Rewrite this formula: {™Not[™Less[•fix[n, 1], •fix[n, 1]]]}

 {2}: Proof of sub goal failed: ™Less[•fix[n, 1], •fix[n, 1]]

 {2}: No case was proven.

 {2}: Try to prove these case conditions:
{™Equal[•fix[n, 1], 0], is-even[•fix[n, 1]], True}

 {2}: Prove/Disprove: ™Equal[•fix[n, 1], 0]

  {3}: Rewrite both sides of the equality: {•fix[n, 1]}={0}

  {3}: Rewrite this formula: {™Not[™Equal[•fix[n, 1], 0]]}

  {3}: Rewrite formula: {™Not[™Equal[•fix[n, 1], 0]]} to {™Not[™Not[True]]}

  {3}: Rewrite formula: {™Not[™Not[True]]} to {True}

 {2}: Sub goal disproved: ™Equal[•fix[n, 1], 0]

 {2}: Prove/Disprove: is-even[•fix[n, 1]]

  {3}: Rewrite this formula: {is-even[•fix[n, 1]]}

  {3}: Rewrite formula: {is-even[•fix[n, 1]]} to {True}

 {2}: Sub goal proved: is-even[•fix[n, 1]]

 {2}: This term will be used: exp[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]]

{1}: Rewrite (rhs): {™E[•fix[x, 1], •fix[n, 1]]} to
{™CaseDistinction[•case[™E[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]],
   is-even[•fix[n, 1]]]], ™CaseDistinction[•case[1, ™Equal[•fix[n, 1], 0]],
  •case[™Times[•fix[x, 1], ™E[•fix[x, 1], pre[•fix[n, 1]]]], True]]}

 {2}: Try to prove these case conditions: {is-even[•fix[n, 1]]}

 {2}: Prove/Disprove: is-even[•fix[n, 1]]

  {3}: Rewrite this formula: {is-even[•fix[n, 1]]}

  {3}: Rewrite formula: {is-even[•fix[n, 1]]} to {True}

 {2}: Sub goal proved: is-even[•fix[n, 1]]

 {2}: This term will be used: ™E[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]]

 {2}: Try to prove these case conditions: {™Equal[•fix[n, 1], 0], True}

 {2}: Prove/Disprove: ™Equal[•fix[n, 1], 0]

  {3}: Rewrite both sides of the equality: {•fix[n, 1]}={0}

  {3}: Rewrite this formula: {™Not[™Equal[•fix[n, 1], 0]]}

  {3}: Rewrite formula: {™Not[™Equal[•fix[n, 1], 0]]} to {™Not[™Not[True]]}

  {3}: Rewrite formula: {™Not[™Not[True]]} to {True}

 {2}: Sub goal disproved: ™Equal[•fix[n, 1], 0]

 {2}: Prove/Disprove: True
```

```
    {3}: This formula is true: True

   {2}: Sub goal proved: True

   {2}: This term will be used: ™Times[•fix[x, 1], ™E[•fix[x, 1], pre[•fix[n, 1]]]]

  {1}: Rewrite (lhs):
 {exp[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]]} to {™CaseDistinction[•case[
    ™E[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]], ™Less[™Half[•fix[n, 1]], •fix[n, 1]]]],
  ™CaseDistinction[•case[1, ™Equal[™Half[•fix[n, 1]], 0]],
   •case[exp[™Square[™Square[•fix[x, 1]]], ™Half[™Half[•fix[n, 1]]]],
    is-even[™Half[•fix[n, 1]]], •case[™Times[™Square[•fix[x, 1]],
     exp[™Square[•fix[x, 1]], pre[™Half[•fix[n, 1]]]]], True]]}

   {2}: Try to prove these case conditions: {™Less[™Half[•fix[n, 1]], •fix[n, 1]]}

   {2}: Prove/Disprove: ™Less[™Half[•fix[n, 1]], •fix[n, 1]]

    {3}: Rewrite this formula: {™Less[™Half[•fix[n, 1]], •fix[n, 1]]}

    {3}: Rewrite formula:
  {™Less[™Half[•fix[n, 1]], •fix[n, 1]]} to {™Not[™Equal[•fix[n, 1], 0]]}

    {3}: Rewrite formula: {™Not[™Equal[•fix[n, 1], 0]]} to {™Not[™Not[True]]}

    {3}: Rewrite formula: {™Not[™Not[True]]} to {True}

   {2}: Sub goal proved: ™Less[™Half[•fix[n, 1]], •fix[n, 1]]

   {2}: This term will be used: ™E[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]]

   {2}: Try to prove these case conditions:
  {™Equal[™Half[•fix[n, 1]], 0], is-even[™Half[•fix[n, 1]]], True}

   {2}: Prove/Disprove: ™Equal[™Half[•fix[n, 1]], 0]

    {3}: Rewrite both sides of the equality: {™Half[•fix[n, 1]]}={0}

    {3}: Rewrite this formula: {™Not[™Equal[™Half[•fix[n, 1]], 0]]}

   {2}: Proof of sub goal failed: ™Equal[™Half[•fix[n, 1]], 0]

   {2}: Prove/Disprove: is-even[™Half[•fix[n, 1]]]

    {3}: Rewrite this formula: {is-even[™Half[•fix[n, 1]]]}

    {3}: Rewrite this formula: {™Not[is-even[™Half[•fix[n, 1]]]]}

   {2}: Proof of sub goal failed: is-even[™Half[•fix[n, 1]]]

   {2}: Prove/Disprove: True

    {3}: This formula is true: True

   {2}: Sub goal proved: True

   {2}: No case was proven completely, the
    remaining case conditions may be used for proving by cases:
  {™Equal[™Half[•fix[n, 1]], 0], is-even[™Half[•fix[n, 1]]], True}

  {1}: The term ™E[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]]
   appears on both sides of the equality, hence the equality is true.

  {1}: Case proved.

 {0}: Proving case: {™Not[™Equal[•fix[n, 1], 0]], ™Not[is-even[•fix[n, 1]]]}
```

```
{1}: Rewrite (lhs): {exp[•fix[x, 1], •fix[n, 1]]} to
{™CaseDistinction[•case[™E[•fix[x, 1], •fix[n, 1]], ™Less[•fix[n, 1], •fix[n, 1]]]],
 ™CaseDistinction[•case[1, ™Equal[•fix[n, 1], 0]],
  •case[exp[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]], is-even[•fix[n, 1]]],
  •case[™Times[•fix[x, 1], exp[•fix[x, 1], pre[•fix[n, 1]]]], True]]}

 {2}: Try to prove these case conditions: {™Less[•fix[n, 1], •fix[n, 1]]}

 {2}: Prove/Disprove: ™Less[•fix[n, 1], •fix[n, 1]]

  {3}: Rewrite this formula: {™Less[•fix[n, 1], •fix[n, 1]]}

  {3}: Rewrite this formula: {™Not[™Less[•fix[n, 1], •fix[n, 1]]]}

 {2}: Proof of sub goal failed: ™Less[•fix[n, 1], •fix[n, 1]]

 {2}: No case was proven.

 {2}: Try to prove these case conditions:
{™Equal[•fix[n, 1], 0], is-even[•fix[n, 1]], True}

 {2}: Prove/Disprove: ™Equal[•fix[n, 1], 0]

  {3}: Rewrite both sides of the equality: {•fix[n, 1]}={0}

  {3}: Rewrite this formula: {™Not[™Equal[•fix[n, 1], 0]]}

  {3}: Rewrite formula: {™Not[™Equal[•fix[n, 1], 0]]} to {™Not[™Not[True]]}

  {3}: Rewrite formula: {™Not[™Not[True]]} to {True}

 {2}: Sub goal disproved: ™Equal[•fix[n, 1], 0]

 {2}: Prove/Disprove: is-even[•fix[n, 1]]

  {3}: Rewrite this formula: {is-even[•fix[n, 1]]}

  {3}: Rewrite formula: {is-even[•fix[n, 1]]} to {™Not[True]}

  {3}: Rewrite this formula: {™Not[is-even[•fix[n, 1]]]}

  {3}: Rewrite formula: {™Not[is-even[•fix[n, 1]]]} to {™Not[™Not[True]]}

  {3}: Rewrite formula: {™Not[™Not[True]]} to {True}

 {2}: Sub goal disproved: is-even[•fix[n, 1]]

 {2}: Prove/Disprove: True

  {3}: This formula is true: True

 {2}: Sub goal proved: True

 {2}: This term will be used: ™Times[•fix[x, 1], exp[•fix[x, 1], pre[•fix[n, 1]]]]

{1}: Rewrite (rhs): {™E[•fix[x, 1], •fix[n, 1]]} to
{™CaseDistinction[•case[™E[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]],
   is-even[•fix[n, 1]]]], ™CaseDistinction[•case[1, ™Equal[•fix[n, 1], 0]],
  •case[™Times[•fix[x, 1], ™E[•fix[x, 1], pre[•fix[n, 1]]]], True]]}

 {2}: Try to prove these case conditions: {is-even[•fix[n, 1]]}

 {2}: Prove/Disprove: is-even[•fix[n, 1]]

  {3}: Rewrite this formula: {is-even[•fix[n, 1]]}

  {3}: Rewrite formula: {is-even[•fix[n, 1]]} to {™Not[True]}

  {3}: Rewrite this formula: {™Not[is-even[•fix[n, 1]]]}
```

```
  {3}: Rewrite formula: {™Not[is-even[•fix[n, 1]]]} to {™Not[™Not[True]]}

  {3}: Rewrite formula: {™Not[™Not[True]]} to {True}

 {2}: Sub goal disproved: is-even[•fix[n, 1]]

 {2}: This term will be used: ™E[™Square[•fix[x, 1]], ™Half[•fix[n, 1]]]

 {2}: Try to prove these case conditions: {™Equal[•fix[n, 1], 0], True}

 {2}: Prove/Disprove: ™Equal[•fix[n, 1], 0]

  {3}: Rewrite both sides of the equality: {•fix[n, 1]}={0}

  {3}: Rewrite this formula: {™Not[™Equal[•fix[n, 1], 0]]}

  {3}: Rewrite formula: {™Not[™Equal[•fix[n, 1], 0]]} to {™Not[™Not[True]]}

  {3}: Rewrite formula: {™Not[™Not[True]]} to {True}

 {2}: Sub goal disproved: ™Equal[•fix[n, 1], 0]

 {2}: Prove/Disprove: True

  {3}: This formula is true: True

 {2}: Sub goal proved: True

 {2}: This term will be used: ™Times[•fix[x, 1], ™E[•fix[x, 1], pre[•fix[n, 1]]]]

{1}: Rewrite (lhs): {™Times[•fix[x, 1], exp[•fix[x, 1], pre[•fix[n, 1]]]]} to
{™CaseDistinction[•case[™Times[•fix[x, 1], ™E[•fix[x, 1], pre[•fix[n, 1]]]],
   ™Less[pre[•fix[n, 1]], •fix[n, 1]]]],
 ™CaseDistinction[•case[™Times[•fix[x, 1], 1], ™Equal[pre[•fix[n, 1]], 0]],
  •case[™Times[•fix[x, 1], exp[™Square[•fix[x, 1]], ™Half[pre[•fix[n, 1]]]]],
   is-even[pre[•fix[n, 1]]], •case[
   ™Times[•fix[x, 1], ™Times[•fix[x, 1], exp[•fix[x, 1], pre[pre[•fix[n, 1]]]]], True]]}

 {2}: Try to prove these case conditions: {™Less[pre[•fix[n, 1]], •fix[n, 1]]}

 {2}: Prove/Disprove: ™Less[pre[•fix[n, 1]], •fix[n, 1]]

  {3}: Rewrite this formula: {™Less[pre[•fix[n, 1]], •fix[n, 1]]}

  {3}: Rewrite formula:
{™Less[pre[•fix[n, 1]], •fix[n, 1]]} to {™Not[™Equal[•fix[n, 1], 0]]}

  {3}: Rewrite formula: {™Not[™Equal[•fix[n, 1], 0]]} to {™Not[™Not[True]]}

  {3}: Rewrite formula: {™Not[™Not[True]]} to {True}

 {2}: Sub goal proved: ™Less[pre[•fix[n, 1]], •fix[n, 1]]

 {2}: This term will be used: ™Times[•fix[x, 1], ™E[•fix[x, 1], pre[•fix[n, 1]]]]

 {2}: Try to prove these case conditions:
{™Equal[pre[•fix[n, 1]], 0], is-even[pre[•fix[n, 1]]], True}

 {2}: Prove/Disprove: ™Equal[pre[•fix[n, 1]], 0]

  {3}: Rewrite both sides of the equality: {pre[•fix[n, 1]]}={0}

  {3}: Rewrite this formula: {™Not[™Equal[pre[•fix[n, 1]], 0]]}

 {2}: Proof of sub goal failed: ™Equal[pre[•fix[n, 1]], 0]

 {2}: Prove/Disprove: is-even[pre[•fix[n, 1]]]

  {3}: Rewrite this formula: {is-even[pre[•fix[n, 1]]]}
```

```
  {3}: Rewrite this formula: {™Not[is–even[pre[•fix[n, 1]]]]}

 {2}: Proof of sub goal failed: is–even[pre[•fix[n, 1]]]

 {2}: Prove/Disprove: True

  {3}: This formula is true: True

 {2}: Sub goal proved: True

 {2}: No case was proven completely, the
  remaining case conditions may be used for proving by cases:
{™Equal[pre[•fix[n, 1]], 0], is–even[pre[•fix[n, 1]]], True}

{1}: The term ™Times[•fix[x, 1], ™E[•fix[x, 1], pre[•fix[n, 1]]]]
 appears on both sides of the equality, hence the equality is true.

{1}: Case proved.

{0}: All cases proved.
```

```
Proved
```

At depth {0} we can see that there is first a course of values induction. While rewriting the equality by the definitions no valid case can be found. Hence a proof by cases is tried. Three cases have to be proven individually:

$n = 0$
$n \neq 0 \wedge$ is–even[n]
$n \neq 0 \wedge \neg$ is–even[n]

In each case rewriting of both sides of the equality is used in order to obtain identical terms on both sides.

Each time a case distinction occurs in a term the prover tried to prove one of the case conditions. This leads to several failing supervise. Nevertheless in the end the prover is able to prove each case successfully.

Aside the failing sub proofs – they are usually omitted by humans – the proof is very natural and similar to proofs made by humans.

6 Summary and Outlook

6.1 Other Systems

In the project HOL Light, a system implemented in ML especially OCaml, there are some powerful proof tools [Harr09]. The user language is a simple typed lambda calculus. Similar to our system there are some built–in inference rule, but additionally there is a big set of knowledge built in. A tactics language is for the proof assistant. There is also a tactics for induction.

The SPIKE system uses an inductionless induction for proving in equational theories as well as in conditional theories [BKR95]. They use a mechanism to generated lemmata and reduce the problem of proving and disproving to algebraic simplification. There is no need for explicit induction. The algorithm finds the necessary cases to prove implicitly. This approach is claimed to be fast and reduces the need of lemmata. The main difference to our project is that the proofs are not found in a human like style.

ACL2 supports some kind of meta reasoning, which is similar to the reasoning about resoners [HKK+05]. It uses first-order logic with first-order reasoning and it includes conditional rewriting. Some extensions allows mathematical induction and meta reasoning. The logic in our work is stronger because it is not restricted to first-order.

6.2 Summary and Future Work

The Symbolic Computation prover is already able to prove a set of theorems. The most important logical connectives are built–in and the user language offers some powerful language constructs like case distinctions and sequence variables. The first extension with induction for natural numbers – and related inductive domains – shows the capability to extend the prover for reaching new domains. At this link an induction on logical terms will start to be able to do reasoning about reasoners.

Printing a log of the proof flow is a first approach for inspection a proof generated by SC prover. The idea of monitoring the execution of the program allows a genuine view on the work of the prover. The implementation of the prover stays clear and transparent. There is a strict distinction between proving and generating an output of the proof.

Both, the prover as well as the proof tracer are a basis for further research. The next steps may be embedding the prover into the new *Theorema* system and adding another proof tracer that is able to generate an entire *Theorema* proof object. This will allow printing the proof in a natural language. A pretty printer of *Theorema* ensures a traditional style for formulae and all other tools of *Theorema* will be accessible like interactive proof notebooks [Tma99a] and focus windows for proof inspection [Buch00].

As further work for introducing reasoning about reasoners the prover needs the implementation of an induction rule for terms and formulae. This requires already the existence of reflection in the user language. Therefore some theoretical and practical problems need to be solved.

For embedding the user language of the prover in user language of *Theorema* there are only some smaller changes to do. Some syntax translations with respect to quantifiers are already described in *section 2.3*, for details see the next subsction. Others related to induction need more attention. A concept for tagging variables over inductive domains is necessary for applying the induction rule only on these variables. Especially if there are different types of inductive domains a distinction is essential.

For improving the proving power and a speed up a more sophisticated choice of cases may help. This enhanced "proving by cases" is described also in *section 2.3*.

Appendix

A.1 Some *Mathematica* commands

For an implementation in *Mathematica* there are many built in commands in use. In the case of the SC prover the source code describes the logic that is used for proving. Therefore it is necessary for a reader to understand at least some parts of the underlying programming language. Hence we give a short overview on some *Mathematica* commands. A complete description of all *Mathematica* commands is given in [Wolf03] or online in [Wolf09]. At the end of the implementation of the SC prover there is a list with all *Mathematica* commands used by the prover.

In general *Mathematica* evaluates expressions in cells after pressing *Ctrl+Return* or *Enter (Numpad)*. Expressions are build up by a head symbol, for example a function name, and a sequence of arguments wrapped by square brackets: **`f[1,2]`**

For some expressions there are so-called traditional form representations which are more natural. For example **`Plus[5,7]`** and **`5+7`** represent the same expression. If *Mathematica* has some definitions for the symbols in an evaluated expression it applies them and tries to simplify it. **`5+7`** gets evaluated to **`12`**.

For declaring new functions this is done by delayed evaluation. The syntax for this is:

```
functionname[arg1_, arg2__] := expr;
```

This defines a function called **`functionname`** with some arguments. Arguments are marked by blanks. The name before the blank gives the name of the argument. A single blank stands for a single value. A double blank matches to one or more arguments and a triple blank may also be empty. If an expression matches this pattern of arguments then the expression **`expr`** will be evaluated. More sophisticated patterns are also possible. The pattern **`arg1_Integer`** matches only values with head symbol **`Integer`**. It is also helpful to use combined patterns like **`{arg1_Integer, arg2_Complex}`**. This matches to a list containing exactly two arguments, the first needs to be an integer, the second a complex number.

Additional constraints for the function may be entered like this:

```
functionname[arg1_, arg2__] /; constraint := expr;
```

This function definition matches only if the constraint is fulfilled (returns **`True`**).

To use compound expressions there are different block elements available. Two expressions are separated by a semicolon (**`;`**). Two often used possibilities are the commands **`Module`** and **`Block`**. **`Module[{vars}, expr]`** allows having local variables. All variables declared in **`vars`** gets replaced by an unique symbol while an evaluation of **`expr`**. This allows calling functions recursive without overwriting values in local variables. By contrast **`Block[{vars}, expr]`** has also local

variables, but they have the values of global variables (with the same name). They are just saved at the beginning of an evaluation and restored at the end. This allows the use of a local version of global variable.

Important constructs in *Mathematica* are lists and sequences. A list has the head symbol **`List`** and may be entered with curly brackets. The elements are separated by colons (**`,`**). Lists are not sorted and may contain duplicates and nested lists. Sequences are similar, but they are flattened automatically. A **`Sequence`** is usually used in an argument list.

There are many functions for handling lists. Most of them are self-explanatory like **`Reverse`** which returns a list in reversed order. **`DeleteCases`** removes elements from a list which match the pattern in the second argument. **`Map[f, list]`** applies the function **`f`** at each element in the list and returns a list with all results. **`Apply[f, {a,b,c}]`** returns **`f[a,b,c]`**, the list elements are the arguments of the function **`f`**. **`Fold[f,x,{a,b,c}]`** applies the function **`f`** recursively: **`f[f[f[x,a],b],c]`**. Most of the commands for lists work also for arbitrary expressions in an analogous way.

Another important category of commands is related with replacing. **`Replace[expr, a→b]`** replaces in the expression **`expr`** the symbol **`a`** by **`b`** at the first occurring position. **`Replace[expr, a→b]`** replaces all occurrences of **`a`** in **`expr`**. This command may also be entered as **`expr/.a→b`**.

For further commands and more details we refer to the *Mathematica* Documentation [Wolf09].

A.2 Commented Source Code of Package

Kernel for Symbolic Computation Prover

Package Description

This package contains the implementation of the prover kernel for **SCProver** (Symbolic Computation).

```
BeginPackage["SC`ProverKernel`"];
```

Other packages are not required for this prover. For details about the used functions of *Mathematica's* **System`** package see *Used Commands and Symbols of Mathematica*..

Exported Symbols

SCProver: Usage and Options

```
Clear[SCProver];
SCProver::usage =
  "SCProver[G,K] applies rewriting for proving of the
    goal `G` w.r.t. the knowledge base `K`, but does
    not produce a proof object.\nNote that a slightly
    different syntax than in Theorema is used.";

Options[SCProver] = {ProveByCases → False, PBCLevel → ∞,
   MaxNumberOfRewritingSteps → 7, Induction → True};

ProveByCases::usage = "ProveByCases activates
    or deactivates proving by cases for rewriting.";
PBCLevel::usage = "PBCLevel controls the depth
    of recursive Prove-By-Cases branches.";
MaxNumberOfRewritingSteps::usage = "MaxNumberOfRewritingSteps
    bounds the number of rewriting steps.";
Induction::usage = "Induction enables the
   proving-by-induction feature."
```

Syntax Elements for Terms and Formulae

All **terms** in the user language that is given to the prover have to be *Mathematica* expressions and fulfil these two properties:

1) Each (*Mathematica*) symbol is a term, if and only if there are no definitions (knowledge) about it within *Mathematica*.

   Note: Symbol names starting with "•" are reserved for symbols, which are not terms, but they may be part of terms (see below).

2) **n** be an integer. If **f, t_1, …, t_n** are terms then this is a term:
   f[t_1, …, t_n]

A **formula** is a term that is either true or not true.

Note, that it is possible to use the *Mathematica* symbols **True** and **False**. They do not have definitions, because they are just symbols. They represent the Boolean truth values in many Mathematica functions, but this depends only on the definition of this functions. Therefore they can be used.

In order to ensure *Mathematica* symbols without any *Mathematica* knowledge it is recommended to use symbol names with prefix "™" like this is done in *Theorema*. Symbols with prefix "•" should be avoided by the user since some of them are used by the prover internally.

The following symbols are used within this prover and they have already some semantics. They must not be used in other ways than defined in this section:

```
Clear[™Equal, ™Iff, ™Not, ™Implies, ™ForAll,
  ™And, ™CaseDistinction, •case, •ind, •seq, •fix];
```

Terms and formulae with predefined semantics

Let **n** be an integer, let **f_1, f_2, …, f_n** be formulae, let **t_1, t_2** be terms and let **s** be a symbol. Then these are formulae:

True	**"boolean truth value true"**
™Not[f_1]	**"negation of f_1"**
™Implies[f_1, f_2]	**"f_1 implies f_2"**
™Equal[t_1, t_2]	**"t_1 is equal to t_2"**
™Iff[f_1, f_2]	**"f_1 is equivalent to f_2"**
™And[f_1, …, f_n]	**"conjunction of f_1, …, f_n"**
™ForAll[s, f_1]	**"universal quantifier with bound variable s"**

Universal quantifiers are allowed with one variable only. There is no predefined syntax for existential quantifier.

Case Distinction

Let **n** be an integer, let $\mathbf{t_1, \ldots, t_n}$ be terms and let $\mathbf{c_1, \ldots, c_n}$ be formulae. Then this is a term:

$$\mathbf{™CaseDistinction[•case[t_1, c_1], •case[t_2, c_2], \ldots, •case[t_n, c_n]]}$$

with the semantics:

If $\mathbf{c_1}$ is true, then the whole term is equal to $\mathbf{t_1}$.
If $\mathbf{™And[™Not[c_1], c_2]}$ is true, then the whole term is equal to $\mathbf{t_2}$.
…
If $\mathbf{™And[™Not[c_1], \ldots, ™Not[c_{n-1}], c_n]}$ is true, then the whole term is equal to $\mathbf{t_n}$.

Case distinctions may occur as formula or as one sub term of **™Equal** or **™Iff**. But they are not allowed in other sub terms. Nested case distinctions are not permitted.

Induction

Let **s** be a symbol, **f** be a formula, and **base, succ** be terms. Then this is a formula:

™ForAll[•ind[s, base, succ], f]

with the semantics:

s is a bound variable over an inductive domain with base element **base** and successor function **succ**.

Course of Values Induction

Let **s** be a symbol, **f** a formula, and **ordering** a term. Then this is a formula:

™ForAll[•ind[s, ordering], f]

with the semantics:

s is a bound variable over an inductive domain ordered by **ordering**.

Sequence variable

Let **s** be a symbol and **f** a formula, then this is a symbol

•seg[s]

and this is a formula:

™ForAll[•seg[s], f]

with the semantics:

- `seg[s]` is a bound sequence variable and it is different from `s`. A sequence variable may be instatiated by some values (a sequence of values) or even by no value.

Fixed constant

Let `s` be a symbol and `n` an integer, then this is a symbol

- `fix[s, n]`

This symbol is used for internally created constants ("arbitrary but fixed") and should not be used by the user. Integers are used here as set of symbols only.

Note that integers are not part of the user language. They are used only as symbols for description.

Further Symbols

Return Values

```
¢Proved = "Proved";
¢Disproved = "Disproved";
¢Failed = "Failed";
```

Implementation

Begin

All definitions from now on are private. This means they are (usually) not visible to the user.

```
Begin["`Private`"];
```

Symbols used for global variables in the package. They contain the prover options and they are used for each prover call individually.

```
Clear[$ProveByCases, $PBCLevel,
  $MaxNumberOfRewritingSteps, $Induction];
```

Prover for External Use: SCProver

External Call with Options

This handles the external call from *Mathematica* by the user. The options given by the user are stored in global variables. If an option is not declared, the program will take the predefined value.

```
SCProver[goal_, KB_List, options___] :=
 Block[{$ProveByCases, $PBCLevel,
   $MaxNumberOfRewritingSteps, $Induction},
  {$ProveByCases, $PBCLevel,
    $MaxNumberOfRewritingSteps, $Induction} =
   {ProveByCases, PBCLevel, MaxNumberOfRewritingSteps,
      Induction} /. {options} /. Options[SCProver];
  Prover[goal, KB]
 ]
```

Implementation note:

Defining the variables for the options within a **Block[]** environment make them visible globally but they are restored with the old values after executing the block. This would allow recursive calls with different options.

Prover

Implementation

Internally the prover is called **Prover**. It assumes that the options are stored in the global variables.

Final Goals

Some cases of trivial goals:

```
Prover[™Equal[expr_, expr_], KB_] := ¢Proved

Prover[™Iff[expr_, expr_], KB_] := ¢Proved

Prover[True, KB_] := ¢Proved
```

Implication

Implications are usually proved by assuming the premise and proving the conclusion. This is done by appending the premise to the knowledge base and calling the prover recursively with the conclusion as new goal.
In order to prevent some trivial cases of contradictory knowledge bases the proof is finished successfully if the premise is **™Not[True]**. Note, that the prover is not able to find a contradiction in the knowledge base because all proving steps are performed on the goal only.

$$\frac{KB,\ A \vdash B}{KB \vdash A \Rightarrow B} \qquad \frac{\text{Proved}}{KB \vdash \text{™Not}[\text{True}] \Rightarrow B}$$

```
Prover[™Implies[™Not[True], B_], KB_] := ¢Proved;
```

```
Prover[™Implies[A_, B_], KB_] := Prover[B, Append[KB, A]]
```

Conjunction

The proof of a conjunction is done by proving each sub formulae separately. If a sub formula was proved, this formula can be used as knowledge for proving further formulae. If the proof of a sub goal fails then the whole proof fails immediately.

$$\frac{KB \vdash f1 \quad KB, f1 \vdash f2 \quad \ldots \quad KB, f1, \ldots fn-1 \vdash fn}{KB \vdash f1 \wedge f2 \wedge \ldots \wedge fn}$$

```
Prover[™And[forms__], KB_] := Module[{kb = KB, ret},
   Scan[(
      ret = ProveSubgoal[#, kb];
      If[ret =!= ¢Proved, Return[¢Failed]];
      AppendTo[kb, #];
      (* Add formula of this branch to knowledge base*)
     ) &, {forms}];
   ret
  ];
```

Implementation note:

Scan[f, l] calls the function **f** with each element of the list **l** separately one after the other. In contrast to **Map**, no result list will be generated by **Scan**. This is more efficient in this application. The execution of **Scan** can be broken immediately by calling **Return**.
(...)& defines a (pure) function with one parameter **#**. The advantage of this construction is that the scope of the variables used inside the function is the same as the scope of the function.

Proving a sub goal is just a recursive call of the prover.

```
Clear[ProveSubgoal];
ProveSubgoal[form_, KB_] := Prover[form, KB];
```

Induction

A proof by induction is performed by proving the base case and by proving the induction step. If the proof of the induction base fails then the whole proof fails immediately. The proof of the induction step contains a universal quantified goal. This is be proven by using an arbitrary but fixed constant. See also the next section.

> Note: The following code for induction is taken from the package *Theorema`Provers`Induction`NIP`*, original version by Bruno Buchberger [Buch96a], and modified.

```
Prover[™ForAll[•ind[v_, base_, succ_], formula_], KB_] /;
  $Induction := Module[{vFix, ret},
  ret = InductionBase[ formula, v, base, KB];
  If[ret =!= ¢Proved, Return[ret]];
  (* Do induction step only if induction base was proved *)
  vFix = ArbitraryButFixedConstant[v, formula, KB];
  InductionStep[formula, v, succ, vFix, KB]
 ]
```

Implementation note:

Module ensures local variables even if the prover is called recursively.

Proving the induction base is done by replacing the variable in the goal formula by the base element.

```
Clear[InductionBase];
InductionBase[formula_, v_, base_, KB_] :=
  Prover[ ReplaceVariable[formula, v → base], KB];
```

The induction step is performed by assuming the formula for an arbitrary value and proving the formula with the successor of the value.

```
Clear[InductionStep];
InductionStep[formula_, v_, succ_, vFix_, KB_] :=
  Prover[ReplaceVariable[formula, v → succ[vFix]],
   Prepend[KB, ReplaceVariable[formula, v → vFix]]];
```

Course of values induction assumes the formula for all values less than a certain value and proves the formula for that value.

```
Prover[™ForAll[•ind[v_, ordering_], formula_], KB_] /;
  $Induction := Module[{vFix},
  vFix = ArbitraryButFixedConstant[v, formula, KB];
  Prover[ ReplaceVariable[formula, v → vFix], Prepend[KB,
    ™ForAll[v, ™Implies[ordering[v, vFix], formula]] ]]
 ]
```

If proving by induction is disabled then the proving hint **•ind** is removed.

```
Prover[™ForAll[•ind[v_, ___], formula_], KB_] /;
  ¬ TrueQ[$Induction] := Prover[™ForAll[v, formula], KB]
```

Universal Quantifier - Arbitrary But Fixed

The bound variable is replaced by a new constant which is arbitrary but fixed.

```
Prover[™ForAll[v_, f_], KB_] := Prover[ ReplaceVariable[
   f, v → ArbitraryButFixedConstant[v, f, KB]], KB]
```

For creating a new symbol, that repesents the new constant, a special syntax element is used. Like in *Theorema* **•fix[s,n]** is used, where **s** is the symbol of the variable which is replaced by the new constant and **n** is an integer to make sure that the symbol neither occurs in the goal nor in the knowledge base. The integer **n** is used as a symbol in the term and has no further properties.

```
Clear[ArbitraryButFixedConstant];
ArbitraryButFixedConstant[variable_, formulae_, KB_] :=
  •fix[variable, Max[0,
     Cases[{formulae, KB}, •fix[variable, i_] ↦ i, {1, ∞}]] + 1];
```

Implementation note:

Cases searches for other fixed constants in the goal formula and the knowledge base. Only fixed constants of the same variable name are interesting, they are replaced by the corresponding integer. The new constant gets the successor of the greatest integer. If there is no fixed constant, the integer will be 1. In this case, **Cases** returns an empty list and **Max** returns 0.

Proving Equivalence

Proving equivalence is done by proving both corresponding implications: "from left to right" and "from right to left". If the first proof fails, the second branch is skipped because the whole proof fails.

$$\frac{KB \vdash A \Rightarrow B \qquad KB \vdash B \Rightarrow A}{KB \vdash A \Leftrightarrow B}$$

```
Prover[™Iff[lhs_, rhs_], KB_] :=
 Module[{ret},
  ret = ProveIffLeftToRight[lhs, rhs, KB];
  If[ret ≠ ¢Proved, Return[ret]];
  ProveIffRightToLeft[lhs, rhs, KB]
 ]
```

Implementation note:

Module ensures local variables even if the prover is called recursively.

Each branch is proven by a recursive call of the prover with the corresponding impilcation in the goal.

```
Clear[ProveIffLeftToRight];
ProveIffLeftToRight[lhs_, rhs_, KB_] :=
 Prover[™Implies[lhs, rhs], KB]

Clear[ProveIffRightToLeft];
ProveIffRightToLeft[lhs_, rhs_, KB_] :=
 Prover[™Implies[rhs, lhs], KB]
```

Proving an Equality by Rewriting

For proving an equality both sides will be rewritten until both are identical or no rules can be applied. If there is a case distinction in the goal then the prover tries to prove a case. If this fails then a proof by case is done. This is done also if proving by case is disabled for rewriting by the prover variable **ProveByCases**. For further details see *Rewriting an Equality*.

If the case distinction is on the left hand side of the equality then sides are swapped. If there are case distinctions on both sides the proof fails. This case is not allowed by syntax.

```
Prover[™Equal[lhs_™CaseDistinction, rhs_™CaseDistinction],
    KB_] /; lhs =!= rhs := ¢Failed;
Prover[™Equal[cd_™CaseDistinction, rhs_], KB_] :=
  Prover[™Equal[rhs, cd], KB];

Prover[™Equal[lhs_, cd_™CaseDistinction], KB_] :=
 Module[{rhs = {cd}, termPBC},
  rhs = ProveAllCaseConditions[rhs, KB];
  If[Length[rhs] == 0, Return[¢Failed]];
  If[¬ FreeQ[rhs, ™CaseDistinction],
   termPBC = ChooseCaseDistinction[rhs];
   If[Length[termPBC] == 0, Return[¢Failed]];
   ProveGoalByCases[™Equal[{lhs}, rhs], KB, termPBC, 0],
   (* else *)
   RewriteEquality[{lhs}, rhs, GenerateRWRules[KB], KB]
  ]
 ]

Prover[™Equal[lhs_, rhs_], KB_] :=
 RewriteEquality[{lhs}, {rhs}, GenerateRWRules[KB], KB]
```

Proving a Case Distinction by Rewriting

For proving a case distinction is similar to rewriting by a conditional rule. First the prover tries to prove a case. If this fails a proof by cases is done (independently of the prover option **ProveByCases**). For further details see *Rewriting a Predicate*.

```
Prover[formula_™CaseDistinction, KB_] :=
 Module[{form = {formula}, formPBC},
  form = ProveAllCaseConditions[form, KB];
  If[Length[form] == 0, Return[¢Failed]];
  If[¬ FreeQ[form, ™CaseDistinction],
   formPBC = ChooseCaseDistinction[form];
   If[Length[formPBC] == 0, Return[¢Failed]];
   ProveGoalByCases[form, KB, formPBC, 0],
   (* else prove the unconditional predicate *)
   Prover[form // First, KB]
  ]
 ]
```

Proving a Predicate by Rewriting

In case of an arbitrary predicate – no previous definitions matched – also rewriting will be used to receive **True**.

```
Prover[formula_, KB_] /;
  ¬ MemberQ[{™Equal, ™Iff, ™Implies, ™ForAll}, Head[formula]] :=
 RewriteFormula[{formula}, GenerateRWRules[KB], KB]
```

Rewriting

Rewriting

Rewriting assumes a list of formulae to rewrite, a list of rewrite rules and the current knowledge base for some sub proofs. The optional parameter **`step`** is used only for limiting the total number of rewriting steps.

> Note: Goal formulae are expected as list of formulae. It is sufficient to prove only one of these formulae:
> $\{G_1, G_2, \ldots, G_n\} \equiv G_1 \vee G_2 \vee \ldots \vee G_n$
>
> For equalities two lists of terms are used instead of a list of equalities. The equality is true if a term occurs in both lists:
> $(\{L_1, L_2 \ldots, L_n\} = \{R_1, R_2 \ldots, R_m\}) \equiv$
> $((L_1 = R_1) \vee (L_1 = R_2) \vee \ldots \vee (L_1 = R_m) \vee (L_2 = R_1) \vee (L_2 = R_2) \vee \ldots \vee (L_n = R_m))$

Rewriting a Predicate - an Arbitrary Formula

The next function applies the rewriting steps on the list of goal formulae and compares all new formulae with **`True`**. Within a loop the following steps are done:

- Creating new formulae by rewriting the current goal formulae. Only formulae different from the previous ones are treated further. Therefore, the variable **`allForm`** contains a list of all formulae which occurred already in the rewriting process.
- If no new formula was found then the loop breaks and the proof fails.
- Checking if the formula **`True`** is found. In this case the loop breaks and the proof succeeds.
- If conditional rules were used in the last rewriting step, then proving of the case conditions is tried. After that, a search for the formula **`True`** is done again.
- If some conditions could not be proven in the last step then a proof by cases is tried. This is controlled by proving options stored in global variables. If there are possible candidates for proving by cases, proofs by cases are enabled by the options, and the maximum depth of proof branches is not reached then the rewriting is continued in each branch independently. Otherwise all formulae with unproven conditions are removed from the list of goals.
- At last the all remaining new formulae are appended to the list **`allForm`**.

```
RewriteFormula[form0_List, rules_, KB_, step_: 1] :=
  Module[{allForm = form0, form = form0, ret, formPBC},
   ret = Do[

     (* transform form and compare with True *)
     form = RewriteAllNew[form, allForm, rules, rules];

     (* break if no new formula was found *)
     If[Length[form] == 0, Return[Null]];

     If[IsTrue[form],  Return[¢Proved]];

     (* check for conditional
       rewriting steps (™CaseDistinction appears) *)
     If[¬ FreeQ[form, ™CaseDistinction],
      form = ProveAllCaseConditions[form, KB];
      If[IsTrue[form],    Return[¢Proved]];
     ];

     If[¬ FreeQ[form, ™CaseDistinction],
      (* try proving by cases *)
      formPBC = ChooseCaseDistinction[form];
      If[(formPBC =!= {}) && ($ProveByCases) && ($PBCLevel > 0),
       Return[ProveGoalByCases[
          RemoveDuplicates[Join[allForm,
            DeleteCases[form, _™CaseDistinction]]],
          KB, formPBC, i]
        ];
      ];
      (* remove all remaining ™CaseDistinction *)
      form = DeleteCases[form, _™CaseDistinction];
     ];

     allForm = Join[allForm, form];

     , {i, step, $MaxNumberOfRewritingSteps}];

    If[ret === Null, ¢Failed, ret]
   ];
```

Implementation note:

The **Do**-loop is controlled by the global variable **$MaxNumberOfRewritingSteps** and the optional parameter **step**. The loop variable **i** runs from **step** to the upper bound. The parameter **step** is used to pass the number of already done steps along to each case of a proof by cases. This ensures that the maximal number of rewrite steps cannot be exceeded even if a proof by cases is done.
A **Do**-loop returns **Null** after the upper bound is reached. Hence, if the variable **ret** is identical to **Null** the proof has failed. If the loop is aborted (with a **Return** command) then the argument of the **Return** command is returned.

The auxiliary function **IsTrue** checks if a goal formula is already **True**. This is done on the level of list elements by the *Mathematica* function **MemberQ**.

```
Clear[IsTrue];
IsTrue[form_List] := MemberQ[form, True];
```

Rewriting an Equality

For rewriting equalities, it is more powerful to rewrite both sides of the equation separately until the term on the left hand side is identical to the term on the right hand side. By supporting lists of terms on both sides of the equation instead of just two terms it is possible to handle many rewriting branches in parallel. In addition, the order of application becomes irrelevant because all kinds of order will be tried and if the result is independent of the order the branches will coincide after some steps. The equation is proven if a term occurs on both sides of the equality.

The algorithm is adapted from the case of arbitrary predicates, but the main strategy is the same. The main difference is that both sides of the equation are rewritten alternately and independently. To avoid unnecessary steps there are checks whether the equation is already true after each step. If after a round of rewriting some conditional terms are remaining, then a proof by case is tried like in the general case for rewriting arbitrary predicates. That depends also on prover options.

```
Clear[RewriteEquality];
RewriteEquality[LHS_List, RHS_List, rules_, KB_, step_: 1] :=
  Block[{$PBCLevel = $PBCLevel},
   Module[{allLHS = LHS,
     allRHS = RHS, lhs = LHS, rhs = RHS, ret, termPBC},
    ret = Do[
      (* transform lhs and compare with rhs *)
      lhs = RewriteAllNew[lhs, allLHS, rules, rules];
      If[Length[lhs] > 0,
       If[IsEqual[lhs, allRHS],    Return[¢Proved]];

       (* check for conditional
         rewriting steps (™CaseDistinction appears) *)
       If[¬ FreeQ[lhs, ™CaseDistinction],
```

```
  lhs = ProveAllCaseConditions[lhs, KB];
  If[IsEqual[lhs, allRHS],    Return[¢Proved]];
 ];
];

(* transform rhs and compare with lhs *)
rhs = RewriteAllNew[rhs, allRHS, rules, rules];
If[Length[rhs] > 0,
 If[IsEqual[Join[allLHS, lhs], rhs],    Return[¢Proved]];

 (* check for conditional
   rewriting steps (™CaseDistinction appears) *)
 If[¬ FreeQ[rhs, ™CaseDistinction],
  rhs = ProveAllCaseConditions[rhs, KB];
  If[
   IsEqual[Join[allLHS, lhs], rhs],    Return[¢Proved]];
 ];
];

If[¬ FreeQ[{lhs, rhs}, ™CaseDistinction],
 (* try proving by cases *)
 termPBC = ChooseCaseDistinction[Join[lhs, rhs]];
 If[(termPBC =!= {}) && ($ProveByCases) && ($PBCLevel > 0),
  Return[ProveGoalByCases[
     ™Equal[
      RemoveDuplicates[Join[allLHS,
        DeleteCases[lhs, _™CaseDistinction]]],
      RemoveDuplicates[Join[allRHS,
        DeleteCases[rhs, _™CaseDistinction]]]
     ],
     KB, termPBC, i]
   ];
 ];

 (* remove all remaining ™CaseDistinction *)
 lhs = DeleteCases[lhs, _™CaseDistinction];
 rhs = DeleteCases[rhs, _™CaseDistinction];
];

(* break if no new term was found *)
If[Length[lhs] + Length[rhs] == 0,
 Return[Null];
];

allLHS = RemoveDuplicates[Join[allLHS, lhs]];
```

```
      allRHS = RemoveDuplicates[Join[allRHS, rhs]];

      , {i, step, $MaxNumberOfRewritingSteps}];

    If[ret === Null, ¢Failed, ret]
   ]
  ];
```

The auxiliary function **IsEqual** checks if at least one term of the left-hand-side is identical to the right-hand-side. This is done by intersecting the corresponding lists of formulae.

```
Clear[IsEqual];
IsEqual[lhs_List, rhs_List] :=
  Length[Intersection[lhs, rhs]] > 0;
```

Proving of Case Conditions

Proving case conditions is the first attempt to handle conditional rewriting rules or rules that are split into cases. After applying such a rule a **™CaseDistinction** term is generated. The functions in this subsection try to prove the condition of one case in order to simplify this term to this case. The proofs are done by a recursive call of the prover. This means that the whole power of the prover is used in the sub proofs and proofs of conditions may occur in these sub proofs again.

Proving Case Conditions of All Case Distinctions

This function applies proving of case conditions at all formulae with **™CaseDistinction**.

```
Clear[ProveAllCaseConditions];
ProveAllCaseConditions[term_List, KB_] := Module[{cd},
  cd = Map[ProveCaseConditions[KB],
    Cases[term, _™CaseDistinction]];
  cd = DeleteCases[cd, {}];
  RemoveDuplicates[
   Join[DeleteCases[term, _™CaseDistinction], cd]]
 ]
```

Implementation note:

Defining local variables within a **Module** is necessary due to the possibility of a recursive call of the algorithm within a sub proof.

Cases selects all terms with head symbol **™CaseDistinction**. Map applies the unary function **ProveCaseConditions[KB][_]** on each term selected by **Cases**. Note, that *Mathematica* allows defining a function where the function name is not a single symbol but a function with parameters. Map only allows unary functions, but a proof of cases conditions depends on the condition itself and the available knowledge base. Therefore the expression

`ProveCaseConditions[KB]` is a unary function, but it depends on the **`KB`**. This expression can be seen as a prover that has included a certain knowledge base and this prover is applied to some terms.

The second line deletes terms if they have neither a proven case nor there is a default case, which is necessary for proving by cases. In the last line the lists of terms are joined and duplicates are removed.

Proving a Case Condition or Eliminating Some

This function tries to prove one of the cases for a given case distinction. Since the semantics of a case distinction depends on the order of the cases (see *Syntax Elements for Terms and Formulae*) one condition after the other is handled. The algorithm tries to prove or disprove each condition formula. If it was possible to prove a condition formula then the following cases are not relevant any more and the loop can be aborted. If a condition formula was disproved then this case can be removed and the iteration for the next case follows. Removing a case is implemented by not appending it to the list of remaining cases.

After all necessary condition formulae are proved or disproved, the result can be generated. It depends on the remaining cases. If the list of remaining cases is empty then for one case the condition formula was proven and all previous condition formulae were disproved. Hence, this case can be used and the term of this case is returned.

If at least one condition formula was neither proved nor disproved then there are remaining cases. If the last handled condition formula was proved (or simply **`True`** for a default case), then the last proven case is added with condition **`True`**. This means that this is the new default case for the new case distinction. The term with a case distinction is returned.

If the last handled condition formulae was not proven – no matter it was disproved or not – then there is no default case. Hence a proof by cases is not possible because the given cases do not cover all possible cases. Therefore, an empty term is returned.

```
Clear[ProveCaseConditions];
ProveCaseConditions[KB_][cases_™CaseDistinction] :=
  Module[{ret, remainingCases = {}, caseNo = 0},
   While[caseNo < Length[cases],
    caseNo++;
    ret = SCProveDisprove[cases⟦caseNo, 2⟧, KB];
    If[ret == ¢Proved, Break[]];
    If[ret =!= ¢Disproved, AppendTo[remainingCases, caseNo]];
   ];

   If[Length[remainingCases] > 0,
    remainingCases = cases⟦remainingCases⟧;
    (* add the first proved case with condition True *)
    If[ret == ¢Proved,
     AppendTo[remainingCases, •case[cases⟦caseNo, 1⟧, True]],
     (* else:
        The last condition was neither proven nor True,
      hence proving by cases does not work,
      because there is no default case! *)
     Return[{}];
    ];
    (* return the list of remaining cases,
     e.g. for proving by cases *)
    Return[remainingCases],
    (* else: return term of the proved case *)
    Return[cases⟦caseNo, 1⟧]
   ];
  ];
```

Implementation note:

The **While**-loop effectively iterates over all cases. The condition formula of a case is assumed to be the second element in a **•case** object. **cases⟦caseNo,2⟧** gives access to the second element of the case with the number **caseNo**. **Break[]** aborts a loop immediately.

If the result is an empty term this is indicated by returning an empty list **{}**. The calling function (**ProveAllCaseConditions**) will delete those empty lists later.

Recursive Prover Call (Extended by Disproving)

SCProveDisprove proves or disproves a goal. It is used for proving case conditions. A goal is disproved, if the negation of the goal is proven. Proving the negation is tried only if the positive goal was not proven.

```
Clear[SCProveDisprove];
SCProveDisprove[goal_, KB_] := Module[{ret},
   ret = Prove[goal, KB];
   If[ret =!= ¢Proved,
    ret = Disprove[goal, KB];
    If[ret == ¢Proved, ret = ¢Disproved];
   ];
   ret
  ];
```

Proving a goal simply means calling the prover with this goal.

```
Clear[Prove];
Prove[goal_, KB_] := Prover[goal, KB];
```

Disproving is done by calling the prover with the negation of the goal. If the goal is already a negated formula then the outermost negation is just removed in order to avoid unnecessary double negation.

```
Clear[Disprove];
Disprove[™Not[goal_], KB_] := Prover[goal, KB];
Disprove[goal_, KB_] := Prover[™Not[goal], KB];
```

Proving By Cases

Proving by cases splits a proof into several branches. The branches depend on the case conditions of a chosen case distinction. **ChooseCaseDistinction** selects a case distinction; currently the first possible case distinction is chosen. **ProveGoalByCases** performs the actual split and **ProveCase** proves a branch by rewriting. It is sufficient to do rewriting because other logical simplifications were already done before a proof by cases is started.

Choosing a Case Distinction

ChooseCaseDistinction selects the first possible case distinction. This may be extended in future versions to a more sophisticated selection algorithm. Case distinctions without a default case (syntax pattern: **•case[_,True]**) are excluded because the prover cannot ensure that all possible cases are covered.

```
Clear[ChooseCaseDistinction];
ChooseCaseDistinction[forms_List] := Block[{cand},
   cand =
    Cases[forms, ™CaseDistinction[___, •case[_, True]], 1, 1];
   If[Length[cand] > 0, First[cand], {}]
  ];
```

Proving a Goal by Cases

ProveGoalByCases splits a proof into several branches. For each case in the given case distinction the algorithm generates a branch with all assumptions and proves the goal formula. If no case distinction is given, then the proof fails. If a branch cannot be proven then the whole proof fails and the following branches will not be executed.

```
Clear[ProveGoalByCases];
ProveGoalByCases[formula:_™Equal|_List, KB_, {}, step_] :=
 ¢Failed; ProveGoalByCases[formula:_™Equal|_List,
  KB_, cases_™CaseDistinction, step_] :=
 Block[{$PBCLevel = $PBCLevel - 1},
  Module[{asml, ret = ¢Failed},
   asml = GenerateAssumptions[cases];
   Scan[(
      ret = ProveCase[formula, KB, #, step];
      If[ret =!= ¢Proved, Return[¢Failed]];
     ) &, asml];
   ret
  ]
 ];
```

Implementation note:

The **Block** environment is used to change a global variable and reset it afterwards. The variable **$PBCLevel** controls the depth of nested proofs by cases. If the value is zero or less then proving by cases will be disabled. The **Module** environment defines local even for recursive calls. For implementation notes on **Scan** and **(...)&** see *Conjunction.*

Proving a Case

For proving a case first the assumptions for this case will be joined to the knowledge base and the rewriting algorithm will be started. For equalities the specialized version of rewriting is used.

```
Clear[ProveCase];
ProveCase[™Equal[lhs_List, rhs_List], KB_, case_, step_] :=
  Module[{curKB},
   curKB = Join[case, KB];
   RewriteEquality[lhs,
    rhs, GenerateRWRules[curKB], curKB, step]
  ];
ProveCase[form_List, KB_, case_, step_] :=
  Module[{curKB},
   curKB = Join[case, KB];
   RewriteFormula[form, GenerateRWRules[curKB], curKB, step]
  ];
```

Generating Assumptions

This algorithm takes a case distinction and creates lists of assumptions. For each case a list of assumptions is generated. As described in *Syntax Elements for Formulae* the full condition to choose a certain case is that the case condition of this case is fulfilled and for all all previous cases the negated case conditions are fulfilled.

```
Clear[GenerateAssumptions];
GenerateAssumptions[cases_™CaseDistinction] :=
  Block[{conds, neg = {}, asml = {}},
   conds = Most[Apply[List, Map[Last, cases]]];
   (* list of conditions, last one is 'True' *)
   Scan[(
      AppendTo[asml, Append[neg, #]];
      AppendTo[neg, NegateFormula[#]];
    ) &, conds];
   AppendTo[asml, neg]; (* add the last case *)
   asml
  ];
```

Implementation note:

cases is a **™CaseDistinction** structure of **•cases[formula, condtion]**. **conds** contains a list of formulae, the case conditions. This list is generated by taking the last (= second) element of each **•case** pair: **Map[Last, cases]**. The **Apply** command changes the **™CaseDistinction** structure to a *Mathematica* list. The **Most** command removes the last condition, which has to be the formula **True** anyway. This was checked earlier by **ChooseCaseDistinction**. The variable **neg** collects the negations of the conditions and the variable **asml** collects the assumption lists for all cases. The assumption list for one case contains the condition for this case and the negated conditions of the previous cases.

For impementation notes on **Scan** and **(...)&** see *Conjunction*.

For the last case the full condition is the negation of all case conditions except the last condition which has to be **True**.

Negating a Formula

NegateFormula returns the negation of a formula and avoids double negation.

```
Clear[NegateFormula];
NegateFormula[™Not[formula_]] := formula;
NegateFormula[formula_] := ™Not[formula];
NegateFormula[{}] := {};
(* a special case for RewriteTerm: an empty term *)
```

Rewriting Tools

In this section the core functions for the actual rewriting are defined. They apply usually on formulae in the goal. For the special case of rewriting an equality the functions of this section are appleid to the terms of both sides. So they apply on terms too.

Transforming a Set of Terms by Application of All Matching Rewriting Rules

RewriteAllNew rewrites all terms and formulae in **goals** but returns only terms that do not occur in **oldgoals**. Hence, the returned terms are new ones. There is an exception for case distinctions. If a case distinction is in the goal formula then this formula is returned also like a new formula. Each new term is generated by applying one rule. There are no recursive applications. The rules from **rulesOutermost** are applied to an entire term and rules from **rulesSubterms** rewrite sub terms only. Each rule is tried on each term on each matching position. Note that currently both lists of rules have identical content.

```
Clear[RewriteAllNew];
RewriteAllNew[{}, ___] := {};
RewriteAllNew[_, _, {}, {}, ___] := {};
RewriteAllNew[goals_, oldgoals_,
   rulesOutermost_, rulesSubterms_] := Module[{res},
   res = Map[RewriteGoal[rulesOutermost, rulesSubterms], goals];

   res = Flatten[res, 2];
   (* remove unnecessary layers of lists *)
   res = DeleteCases[res, {}];  (* remove empty terms *)
   res = RemoveDuplicates[res]; (* remove duplicates *)
   Complement[res, DeleteCases[oldgoals, _™CaseDistinction]]
   (* return only new terms *)
  ];
```

Implementation note:

Map applies the function **RewriteGoal** on each term in the goal. For further notes on **Map** and functions of functions see *Prove All Case Conditions*.

RewriteGoal applies each rule on a goal and uses the corresponding function for rewriting. But for case distinctions in the goal formula no rewriting is applied.

```
Clear[RewriteGoal];
RewriteGoal[__][goal_™CaseDistinction] := goal;
RewriteGoal[rulesOutermost_, rulesSubterms_][goal_] := Join[
  Map[RewriteSubterm[goal], rulesSubterms],
  Map[RewriteTerm[goal], rulesOutermost]
 ]
```

RewriteSubterm searches for all possible sub term matches of the left hand side of the rule in the given goal formula or term. For each matching position a new term is generated by rewriting.

```
Clear[RewriteSubterm];
RewriteSubterm[goal_][rule_] := Module[{pos},
  pos =
   Position[goal /. ™CaseDistinction[___] → ™CaseDistinction,
    rule[[1]], {0, ∞}];
  pos = DeleteCases[pos, {}];
  Map[RewriteByRule[goal, rule, #] &, pos]
 ]
```

RewriteTerm checks whether the rule matches the entire term. If it is so, the term is rewritten by the rule. For a negated goal formula the check is performed on the negated formula as well as on the positive formula. If there is a double negation at the outermost position of a goal formula then this is removed instead of applying the rule. The rule will get a chance in the next round of rewriting.

```
Clear[RewriteTerm];
RewriteTerm[™Not[™Not[goal_]]][rule_] := goal;
RewriteTerm[™Not[goal_]][rule_] :=
  If[MatchQ[™Not[goal], rule[[1]]],
   RewriteByRule[™Not[goal], rule, EntireExpression],
   (* else rewrite within the negation *)
   If[MatchQ[goal, rule[[1]]],
    RewriteByRule[™Not[goal], rule, {1}],
    (* else: no matching *)
    {}
   ]
  ];
RewriteTerm[goal_][rule_] :=
 If[MatchQ[goal, rule[[1]]],
  RewriteByRule[goal, rule, EntireExpression],
  {}
 ]
```

Rewriting a Term by a Conditional Rule

RewriteByRule applies a certain rule on a term at a specified position. In case of rewriting the entire term this behaves like an unconditional rule. For conditional rules a new case distinction is generated and in each case the rule is applied at the given position.

```
Clear[RewriteByRule];
RewriteByRule[term_, rule : (lhs_ → ™CaseDistinction[__•case]) ,
   EntireExpression] := ReplaceAt[term, rule, EntireExpression];

RewriteByRule[term_, rule : (lhs_ → ™CaseDistinction[__•case]) ,
   pos_] := Module[{cases, t, c},
   cases = Replace[Extract[term, pos], rule];
   cases = ReplaceAll[cases,
     •case[t_, c_] → •case[ReplacePart[term, t, pos], c]];
   cases
  ];
```

Implementation note:

Implementation note:
The *Mathematica* environment **Module** is used to get unique symbols for **t** and **c**. The first **Replace** command applies the rule on the sub term at position **pos**. After that the local variable **cases** contains a **™CaseDistinction** structure with all patterns are filled properly. In the second line the entire term is wrapped around each case term. This is done by **ReplacePart**. **ReplaceAll** transforms one **•case** after the other.

Rewriting a Term by an Unconditional Rule

In case of an unconditional rule **RewriteByRule** is more or less a call of the *Mathematica* **Replace** command.

```
RewriteByRule[term_, rule_, pos_] := ReplaceAt[term, rule, pos];
```

Low-level Rewriting Tools

These auxiliary functions work directly on the *Mathematica* expressions and extend some *Mathematica* commands.

Removing Duplicates

This function removes all duplicates without sorting the elements, e.g. like *Mathematica* **Union**.
The Implementation is taken of *Mathematica* Help / Command **Reap**. The original function is called **UnsortedUnion**.

```
Clear[RemoveDuplicates];
RemoveDuplicates[x_] := Reap[Sow[1, x], _, #1 &][[2]];
```

Applying a Rule at a Certain Position

ReplaceAt works like *Mathematica* **Replace** but it replaces at a certain position. **Replace** only applies a rule at the top position.

```
Clear[ReplaceAt];
ReplaceAt::usage = "ReplaceAt[expr, rule,
    pos] applies rule on expr only at position pos.";
ReplaceAt[expr_, r : (_Rule | _RuleDelayed), EntireExpression] :=
  Replace[expr, r];
ReplaceAt[expr_, r : (_Rule | _RuleDelayed), pos_] :=
  ReplacePart[expr, Replace[Extract[expr, pos], r], pos];
```

Symbol EntireExpression (Extends Mathematica Extract and ReplacePart)

The *Mathematica* functions **Extract** and **ReplacePart** are extended to be able to process the parameter **EntireExpression** instead of a position.

```
Clear[EntireExpression];
EntireExpression::usage = "EntireExpression is a
    symbol used by ReplaceAt, Extract, ReplacePart";
Extract[expr_, EntireExpression] ^:= expr;
ReplacePart[expr_, new_, EntireExpression] ^:= new;
```

Implementation note:

UpSetDelayed (**^:=**) defines a so called upvalue. It allows assigning a definition to another symbol than the head symbol. In this case the functions **Extract** and **ReplacePart** are protected from *Mathematica*. Therefore the definitions are assigned to the symbol **EntireExpression**. Every time **Extract** is called with **EntireExpression**, this definition will be used by *Mathematica*. In *Mathematica* upvalue definitions have a higher priority than usual (downvalue) definitions.

Generating Rules

Generating Rewrite Rules

GenerateRWRules transforms a knowledge base (a list of formulae) into a list of rewrite rules. These rules are *Mathematica* rules. Before each formula is translated into a formula the induction variables are removed (**RemoveInduction**) and the knowledge base is simplified (**SimplifyKnowledgeBase**). After the rule generation all invalid rules are deleted and the valid ones are returned to the caller.

```
Clear[GenerateRWRules];
GenerateRWRules::usage =
 "GenerateRWRules[KB] returns a list of rewrite rules."
GenerateRWRules[KB_List] := Block[{rules},
   rules = Map[GenerateRule,
     SimplifyKnowledgeBase[RemoveInduction[KB]]];
   DeleteCases[rules, {}]
  ];
```

Removing Induction

RemoveInduction replaces the syntax construct for induction variables by a simple variable with the same name. This simplifies the following formula handling.

```
Clear[RemoveInduction];
RemoveInduction[KB_List] :=
  ReplaceAll[KB, {•ind[x_, ___] → x}];
```

Simplifying a Knowledge Base

SimplifyKnowledgeBase simplifies one formula after the other by **SimpFormula**.

```
Clear[SimplifyKnowledgeBase];
SimplifyKnowledgeBase[l_] := Map[SimpFormula, l];
```

Quantifiers, Implications and Conjunctions

Quantifiers are moved out from an implication, but they are moved into conjunctions. If the conclusion of an implication is a conjunction then the implication is spilt into a conjunction of implications. Conjunctions at outermost position are spilt into a list of formulae. This is possible because the knowledge base is treated as a conjunction of all contained formulae.

Examples of some simplifications:

$$\forall_x C[x] \Rightarrow \left(\forall_y P[x, y]\right) \longrightarrow \forall_x \forall_y (C[x] \Rightarrow P[x, y])$$

$$C \Rightarrow \left(\forall_y P[y]\right) \longrightarrow \forall_y (C \Rightarrow P[y])$$

$$C \Rightarrow (A \wedge B) \longrightarrow (C \Rightarrow A) \wedge (C \Rightarrow B)$$

$$\forall_x (P[x] \wedge Q \wedge R[x]) \longrightarrow \forall_x P[x] \;,\; \forall_x Q \;,\; \forall_x R[x]$$

```
Clear[SimpFormula];
SimpFormula[™ForAll[x_, formula_], vars___] :=
  SimpFormula[formula, vars, x];
SimpFormula[formula : ™And[___], vars___] :=
  Apply[Sequence, Map[(SimpFormula[#1, vars]) &, formula]];
SimpFormula[™Implies[condition_, True], vars___] := Sequence[];
SimpFormula[
   ™Implies[condition_, ™ForAll[x_, formula_]], vars___] :=
  SimpFormula[™Implies[condition, formula], vars, x];
SimpFormula[™Implies[condition_, formula : ™And[___]],
   vars___] :=
  SimpFormula[Map[(™Implies[condition, #1]) &, formula], vars];
SimpFormula[formula_, vars___] :=
  Fold[(™ForAll[#2, #1]) &, formula, Reverse[{vars}]];
```

Implementation note:

The function **SimpFormula** is called with one parameter, which is the formula that should be simplified. Internally optional parameters at the end are used to collect bound variables. The function will be called recursively to go through the whole depth of the formula.

If the formula is a universal quantifier then the quantifier is removed and in the recursive function call the bound variable is stored as an additional parameter at the end of the parameter list. This allows collecting all bound variable.

If the formula is a conjunction then the formula is split into a sequence of formulae. For each formula **SimpFormula** is called recursively with the collected bound variables. Note that the *Mathematica* syntax **(..)&** defines a (pure) function with parameter **#1**, **#2** and so on. **Map** would map this function on each formula inside **formula** and **Apply** transforms the head symbol **™And** into **Sequence**.

If the formula is an implication and the conclusion is a universal quantified formula then the implication is moved under the quantifier. This is done by a recursive call of **`SimpFormula`** on the implication without the quantifier and appending the bound variable to the collected list of variables.

If the formula is an implication and the conclusion is a conjunction then the implication is applied to each part of the conjunction. After that the conjunction is split by a recursive call of **`SimpFormula`**.

If **`SimpFormula`** gets an arbitrary formula then the innermost formula is reached and the quantifiers are reconstructed in the same order as before. Unused bound variables may occur in this formula, they will be ignored later in the rule generation process. Note that later the function **`GenerateRule`** will collect all bound variables again. But to avoid a dependency between these two implementations **`SimpFormula`** returns a syntactical correct formula.

Case Distinction

If a case distinction appears in the knowledge base as outermost symbol, it is translated into some implications. Each case leads to an implication. Note that later each implication is transformed into a case distinction, which is different to the original case distintion. This allows conditional rewriting.

Examples:

$$\begin{cases} f_1 \Leftarrow c_1 \end{cases} \quad \mapsto \quad c_1 \Rightarrow f_1$$

$$\begin{cases} f_1 \Longleftarrow c_1 \\ f_2 \Longleftarrow c_2 \\ \quad \dots \\ f_n \Longleftarrow \text{True} \end{cases} \quad \mapsto \quad \begin{array}{l} c_1 \Rightarrow f_1 \\ ((\neg\, c_1) \wedge c_2) \Rightarrow f_2 \\ \dots \\ ((\neg\, c_1) \wedge \dots \wedge (\neg\, c_{n-1})) \Rightarrow f_n \end{array}$$

```
SimpFormula[™CaseDistinction[•case[form_, cond_]], vars___] :=
  SimpFormula[™Implies[cond, form], vars];

SimpFormula[cd_™CaseDistinction, vars___] :=
  Apply[Sequence, MapThread[
    (SimpFormula[™Implies[If[Length[#1] > 1,
         Apply[™And, #1], First[#1]], First[#2]], vars]) &,
    {GenerateAssumptions[cd], Apply[List, cd]}]
  ];
```

Implementation note:

If there is only one case then the translation is simple. In all other cases it is a little bit tricky. The premises of the implications are built by **`GenerateAssumptions[cd]`**. For each case an implication is generated and **`SimpFormula`** is called recursively. Depending on the number of conditions a conjunction of the conditions is generated.

In more details: **MapThread** maps a function on some lists of arguments. It runs over all lists simultaneously and takes one element of each list. The first call takes the first element of each list, the second call the second elements and so on. Therefore all lists must have equal length. The first parameter list is a list of all assumptions generated from the case distinction (**GenerateAssumptions[cd]**). The second list contains all cases. The expression **Apply[List,cd]** just changes the head symbol from **™CaseDistinction** to **List** because **MapThread** needs this. Then for each case the pure function defined in **(...)&** is called with two parameters. The first parameter **#1** is taken from the first list and contains a list of conditions. The second parameter **#2** contains a **•case** object with two elements: The formula for this case and the condition. If the first parameter (= condition formulae) contains more than one element then a conjunction of this formulae is generated by **Apply[™And,#1]**. Otherwise just the first element (it is the only one) is taken by **First[#1]**. The result of this **If** statement is the premise for the implication. From the second parameter **#2** only the first element (= case formula) is used for the conclusion: **First[#2]**.

Generating a Rule

GenerateRule contains the main definitions for the rule generation process. Depending on the pattern of the given formula a specific rewriting rule is generated.

```
Clear[GenerateRule];
GenerateRule::usage =
 "GenerateRule[formula] returns a rule which performs
   the replacement defined by the given formula.";
```

Equality and Equivalence

Equalities and equivalences are used to rewrite the term on the left hand side by the term on the right hand side. If such formulae are universal quantified, then the bound variables are replaced by patterns. In order to be able to fill in all blanks of the pattern later in the rewrite process the function **CheckVariableOccurrence** checks if all variables of the right hand side term occur also on the left hand side. Otherwise this formula is skipped.

```
GenerateRule[™Equal[P_, Q_]] := (P → Q);
GenerateRule[™Iff[P_, Q_]] := (P → Q);

GenerateRule[™ForAll[x_, (eq : ™Iff | ™Equal)[P_, Q_]]] /;
   CheckVariableOccurrence[P, Q, x] :=
  ReplaceVariableByPattern[P → Q, x];
```

Implementation note:

(eq:™Iff|™Equal)[P_,Q_] defines a pattern with alternative head. The head symbol may be either **™Iff** or **™Equal**. For a certain call the symbol is stored in the local variable **eq** and may be used in the function body. This allows handling two syntactical similar formulae with only one definition.

The next definition handles multiple quantified variables like this:

$$\forall_x \forall_y f[x, y] = g[x, y]$$

```
™ForAll[x, ™ForAll[y, ™Equal[f[x, y], g[x, y]]]]
```

Therefore a formula with more than one quantifier in a row is translated internally to a formula with only one quantifier, but this quantifier supports a list of variables.

```
(* collect all bound variables: *)
GenerateRule[™ForAll[x_List, ™ForAll[y_, f_]]] :=
  GenerateRule[™ForAll[Append[x, y], f]] ;
GenerateRule[™ForAll[x_, ™ForAll[y_, f_]]] :=
  GenerateRule[™ForAll[{x, y}, f]] ;

GenerateRule[™ForAll[v_List, (eq : ™Iff | ™Equal)[P_, Q_]]] /;
   CheckVariableOccurrence[P, Q, v] :=
  ReplaceVariableByPattern[P → Q, v];
```

Implementation note:

The first two definitions collect the variables. The third one generates the rule for the formula after checking the occurrence of the variables. **P** is rewritten by **Q**.

Case Distinction

Most formulae containing **™CaseDistinction** need no special handling. They are treated like unconditional formulae. Some simplifications are done previously in **SimplifyKnowledgeBase**, especially if a case distinction appears as outermost symbol.

If a case distinction appears on the left hand side of an equality or an equivalence the terms are swapped to the other side. But this makes no sense if both sides contain case distinctions. In this case the formula is used exceptionally for rewriting from right to left, because the other way is much more difficult to match.

```
GenerateRule[(eq : ™Iff | ™Equal)[
    lhs_™CaseDistinction, rhs : Except[™CaseDistinction]]] :=
  GenerateRule[eq[rhs, lhs]];
```

Implementation note:

The pattern **rhs:Except[™CaseDistinction]** matches all expressions except these ones with head symbol **™CaseDistinction**. The content of the pattern is stored in **rhs**.

Implication

If a formula contains an implication then this implication is translated into an equivalence with **™CaseDistinction** formula with only one case. Examples:

$$C \Rightarrow A = B \quad \mapsto \quad A = \left\{ B \Leftarrow C \right.$$

$$C \Rightarrow A \quad \mapsto \quad A \Leftrightarrow \left\{ \text{True} \Leftarrow C \right.$$

So **A** is rewritten by **B** resp. **True** only if the prover is able to prove **C**.

```
GenerateRule[™Implies[C_, (eq:™Iff|™Equal)[A_, B_]] ] :=
  GenerateRule[eq[A, ™CaseDistinction[•case[B, C]] ] ];

GenerateRule[™Implies[C_, A_]] :=
  GenerateRule[™Iff[A, ™CaseDistinction[•case[True, C]] ] ];

GenerateRule[
   ™ForAll[v_, ™Implies[C_, (eq:™Iff|™Equal)[A_, B_]]]] :=
  GenerateRule[™ForAll[v,
    eq[A, ™CaseDistinction[•case[B, C]] ]] ];

GenerateRule[™ForAll[v_, ™Implies[C_, A_] ]] :=
  GenerateRule[
   ™ForAll[v, ™Iff[A, ™CaseDistinction[•case[True, C]] ]] ];
```

Negation

Transformation of some negated formulae to logically equivalent formulae:

$$\neg \neg A \quad \mapsto \quad A$$

$$\neg A \quad \mapsto \quad A \Leftrightarrow \neg \text{True}$$

```
GenerateRule[™Not[™Not[F_]] ] := GenerateRule[F];

GenerateRule[™Not[F_] ] := GenerateRule[™Iff[F, ™Not[True]] ];
```

Other Formulae

… are rewritten by **True**:

$$\underset{x}{\forall} A \quad \mapsto \quad \underset{x}{\forall} (A \Leftrightarrow \text{True})$$

$$A \quad \mapsto \quad A \Leftrightarrow \text{True}$$

```
GenerateRule[™ForAll[x_, F_] ] :=
  GenerateRule[™ForAll[x, ™Iff[F, True] ]];
```

```
GenerateRule[f_] := GenerateRule[™Iff[f, True]];
```

The last defintion is a fallback for formulae which do not match any other definition. This ensures that for each formula a rule is generated.

Pattern for Variables

ReplaceVariableByPattern returns a rule where variables are replaced by labelled blanks on the left hand side and by the labels on the right hand side. **VariablePattern** returns a pattern for a simple variable or a sequence variable. The pattern contains conditions to prevent invalid instantiations (see *ValidInstance*).

```
Clear[ReplaceVariableByPattern];
ReplaceVariableByPattern::usage =
 "ReplaceVariableByPattern[rule,x]
   replaces x in rule by a labelled blank.";
ReplaceVariableByPattern[P_ → Q_, vars_List] :=
  Fold[ReplaceVariableByPattern, P → Q, vars];
ReplaceVariableByPattern[P_ → Q_, x_] := Module[{t, rule},
   rule = x → VariablePattern[x, t];
   ReplaceVariable[P, rule] → ReplaceVariable[Q, x → t]
  ];
```

Implementation note:

Fold applies a binary function (**ReplaceVariableByPattern**) on the expression **P→Q** recursively and adds as second parameter an element from the list **vars**.
Example:

```
Fold[f,expr,{x,y,z}] = f[f[f[expr,x],y],z]
```

The **Module** environment in **ReplaceVariableByPattern** ensures unique symbols for **t**. So each variable can be replaced by a unique symbol.

```
Off[RuleDelayed::"rhs"] ;
(* This prevents some unnecessary warnings. *)

Clear[VariablePattern];
VariablePattern::usage =
 "VariablePattern defines the pattern for
   a simple variable or a sequence variable.";
VariablePattern[x_, t_] := If[IsSequenceVar[x],
  Pattern[t, ___?ValidInstance],
  Pattern[t, _?ValidInstanceSeqFree]
 ];

On[RuleDelayed::"rhs"] ; (* It enables warnings again. *)
```

Implementation note:

Off[…**]** suppresses some warnings while loading the package. Otherwise warnings are printed because **Blank**s appear within a function body which is not usual and in most cases an input error. But in this case **Blank**s are necessary to generate patterns for rewrite rules. **On[**…**]** restores the warning mechanism again.

IsSequenceVar returns True if a variable is a sequence variable.
The code of **IsSequenceVar** is taken from ***Theorema`Language`Syntax`Core`IsSequenceFree*** and modified.

```
IsSequenceVar[x___] := ¬ FreeQ[{x}, •seq[_], 1];
```

ValidInstance

ValidInstance and **ValidInstanceSeqFree** check the validity of an instantiation for a pattern.
The first check prevents some matches of internal structure to function terms.

Example:
An arbitrary but fixed constant **•fix[x,0]** occurs in a formula **P[•fix[x,0]]**.
Without a check the pattern **P[f_[t___]]** would match, but this pattern should only match to formulae like **P[f[t,s]]**.

A second check prevents sequence variables from matching to a simple pattern (only for **ValidInstanceSeqFree**).

```
Clear[ValidInstance];
ValidInstance::usage =
  "ValidInstance checks the validity of a pattern.";
ValidInstance[•seq] = False;
ValidInstance[•fix] = False;
ValidInstance[•ind] = False;
ValidInstance[_] = True;

Clear[ValidInstanceSeqFree];
ValidInstanceSeqFree::usage = "ValidInstanceSeqFree
    checks the validity of a non-sequence pattern.";
ValidInstanceSeqFree[•seq[___]] := False;
ValidInstanceSeqFree[x_] := ValidInstance[x];
```

ReplaceVariable

ReplaceVariable performs the actual replacement of a variable by a pattern. The algorithm ensures that constants **•fix** and sequence variables **•seq** are not replaced by a simple variable.

```
Clear[ReplaceVariable];
ReplaceVariable[formula_, rule : (•seq[_] → _) ] :=
  ReplaceAll[formula, {(f : •fix[__]) → f, rule}];
ReplaceVariable[formula_, rule : (_ → _) ] := ReplaceAll[
   formula, {(f : •fix[__]) → f, (f : •seq[__]) → f, rule}];
```

Implementation note:

The *Mathematica* **ReplaceAll** applies only one rule to an expression. Therefore the additional rules, which do effectively nothing, prevent the replacement in sub expression of **•fix** and **•seq**.

CheckVariableOccurrence

CheckVariableOccurrence checks if all variables in **v**, which appear in the term on the right hand side also appear in the left hand side term. **•fix** constants are removed because only variables are interesting for this check. Each variable is tested separately by **CheckOneVariableOccurrence**. Simple variables and sequence variables may have a common variable symbol but they are different as variables. Therefore in case of a simple variable it is necessary to remove all sequence variables with the same before the test, because the check is done by pattern matching with **FreeQ**. For sequence variables such a removal is not necessary because simple variables would not match the pattern of a sequence variable.

```
Clear[CheckVariableOccurrence];
CheckVariableOccurrence[lhs_, rhs_, v_] :=
  CheckOneVariableOccurrence[
    ReplaceAll[{lhs, rhs}, •fix[__] → {}]][v];
CheckVariableOccurrence[lhs_, rhs_, v_List] :=
  Apply[And, Map[CheckOneVariableOccurrence[
     ReplaceAll[{lhs, rhs}, •fix[__] → {}]], v]];

Clear[CheckOneVariableOccurrence];
CheckOneVariableOccurrence[{lhs_, rhs_}][•seq[v_]] :=
 ¬ FreeQ[lhs, v] ∨ FreeQ[rhs, v];
CheckOneVariableOccurrence[{lhs_, rhs_}][v_] :=
 ¬ FreeQ[lhs /. (•seq[v] → {}), v] ∨ FreeQ[rhs /. (•seq[v] → {}), v];
```

Implementation note:

CheckOneVariableOccurrence[…**][**…**]** has a function as head symbol. For details see *Prove All Case Conditions*.

End

```
End[];

EndPackage[];
```

This ends the definition of the package. The code in the next line allows loading the package immediately:

```
Get["SC`ProverKernel`"]
```

Used Commands and Symbols of *Mathematica*

This section lists alphabetically ordered all commands and symbols from the *Mathematica* package **System`**. This package is loaded automatically by *Mathematica* and enables the main functionality of *Mathematica*. For more details on these commands and symbols see *Mathematica* help.

Commands and Symbols with Short Form

These commands are used mainly in prefix, post fix and infix notation. In most cases these notations are more natural and hence easier to read.

Alternatives	**|**	**Pattern**
And	**&&**	**PatternTest**
Blank	**_**	**Plus**
BlankNullSequence	**___**	**ReplaceAll**
BlankSequence	**__**	**Rule**
CompoundExpression	**;**	**RuleDelayed**
Condition	**/;**	**SameQ**
Equal	**==**	**Sequence**
Function	**&**	**Set**
Greater	**>**	**SetDelayed**
Increment	**++**	**Slot**
Infinity	**∞**	**String**
List	**{}**	**Subtract**
MessageName	**::**	**Unequal**
Not	**¬**	**UnsameQ**
Part	**[[]]**	**UpSetDelayed**

Symbols

False, True

Commands

AppendTo, Apply, Begin, BeginPackage, Block, Break, Cases, Clear, Clear, DeleteCases, Do, End, EndPackage, Extract, First, Flatten, Fold, FreeQ, Head, If, Intersection, Join, Length, Map, MapThread, MatchQ, Max, MemberQ, Module, Most, Needs, Off, On, Options, Prepend, Reap, Replace, ReplacePart, Return, Reverse, Scan, Sow, While

A.3 Commented Source Code of Package

Proof Tracer for Symbolic Computation Prover

Package Description

This package contains the implementation of a tracer for **SCProver** (Symbolic Computation Prover).

```
BeginPackage["SC`Tracer`",
  {"SC`ProverKernel`", "Utilities`FilterOptions`"}];
```

Utilities`FilterOptions` is needed for **FilterOptions**.

The package **SC`ProverKernel`** contains the implementation of the corresponding prover. This tracer allows printing some output while the prover is running. This documents the proof flow and allows inspecting a proof even if the proof fails.

Exported Symbols

SCProof: Usage and Options

```
Clear[SCProof];
SCProof::usage =
  "SCProof[G,K] calls SCProver[G,K] and generates
    additionally a simple output of the proof flow.";

Options[SCProof] = {trPrintRules → False};
trPrintRules::usage =
 "trPrintRules controls the output of rewriting rules."
```

Implementation

Begin

```
Begin["`Private`"];
```

The private variable **$curDepth** stores the current depth of the proof. Some proof steps may increase the depth. It will be used for a better structured output of the proof flow. **$trPrintRules** stores the value of the corresponding tracer option. **$trSKIP** is used only for developing and debugging the tracer.

```
$curDepth = 0;
Clear[$trPrintRules];

$trSKIP = False; (* a variable used for debugging purpose *)
```

Implementation note:

The initialization of the global variable has no effect on the tracer; they will get valid values in each call. Except the variable **$trSKIP,** which is for debugging only: the value of this variable will not be changed later.

Tools

Print command

This function generates an output with *Mathematica* **Print**. For better imagination of the proof flow the variable **$curDepth** controls the indent and the value is printed in front of the output.

```
Clear[trPrint];
trPrint[out___] := Print[StringJoin[Table[" ", {$curDepth}]],
   {$curDepth}, ": ", Sequence @@ DeleteCases[{out}, Null]];
```

Implementation note:

StringJoin[Table[" ",{$curDepth}]] generates the indent.
Sequence@@DeleteCases[{out},Null] removes **Null** values from the output.

Proof Tracer: SCProof

Implementation

The main function handles the options and calls the prover within a **TraceScan** command. This command calls the functions **tr** before an expression of the prover is evaluated and **trRes** after the evaluation. The list **TracedSymbols** filters the expressions for which the functions **tr** and **trRes** are called. This is not mandatory but it simplifies the development and maintenance of the tracer.

```
SCProof[goal_, KB_, opts___] :=
 Block[{$curDepth = 0, $trPrintRules},
  Module[{ret},
   {$trPrintRules} = {trPrintRules} /.
     Join[{FilterOptions[SCProof, opts]}, Options[SCProof]];

   trPrint["Proof of: ", goal];
   ret = TraceScan[ tr,
     SCProver[goal, KB, FilterOptions[SCProver, opts]],
     Evaluate[
      Alternatives @@ (#[___] & /@ TracedSymbols)], trRes];
   ret
  ]
 ]
```

Implementation note:

The **Block** command allows defining values for variables. The old value will be restored after the execution. Option handling uses **FilterOptions** and the default values for the options.

TracedSymbols contains a list of symbols, **#[___]&/@** generates a function pattern for each symbol. E.g. the pattern generated from symbol **s** is **s[___]**. **Alternatives@@** converts the list of patterns into alternative patterns. The function **Evaluate** executes the code immediately, even if a function has the attribute **HoldFirst**, **HoldRest**, or **HoldAll**. **TraceScan** has the attribute **HoldAll** to suppress the excution of all parameters; therefore **Evaluate** is necessary to compute the alternative patterns.

Example:
If **TracedSymbols** contains **{s1,s2,s3}** then the third parameter of **TraceScan** looks like this:

```
s1[___] | s2[___] | s3[___]
```

Such a pattern matches function calls like of **s1[p1,p2]** or **s3[]**.

```
TracedSymbols = {
   SC`ProverKernel`Private`Prover,
   SC`ProverKernel`Private`RewriteEquality,
   SC`ProverKernel`Private`RewriteFormula,
   SC`ProverKernel`Private`RewriteAllNew,
   SC`ProverKernel`Private`IsEqual,
   SC`ProverKernel`Private`ProveSubgoal,
   SC`ProverKernel`Private`InductionBase,
   SC`ProverKernel`Private`InductionStep,
   SC`ProverKernel`Private`ProveIffLeftToRight,
   SC`ProverKernel`Private`ProveIffRightToLeft,

   SC`ProverKernel`Private`ProveAllCaseConditions,
   SC`ProverKernel`Private`ProveCaseConditions[___],
   SC`ProverKernel`Private`SCProveDisprove,
   SC`ProverKernel`Private`Prove,
   SC`ProverKernel`Private`Disprove,
   SC`ProverKernel`Private`ProveGoalByCases,
   SC`ProverKernel`Private`ProveCase
  };
```

The function **tr** and **trRes** are called for a certain pattern only if there is an entry in **TracedSymbols** for this pattern. Note that each symbol will be extended to a function pattern. The function **ProveCaseConditions[___][___]** has two parameter sequences, therefore it is necessary to append one already in this list. Most of the symbols in **TracedSymbols** are in the **Private`** context of the package for **SCProver**. Hence, they are not visible outside this context. Therefore it is necessary to specify the entire symbol name including the context. For more information on contexts see *Mathematica Help*.

Tracer

Implementation of tr and trRes

tr and **trRes** are called for every expression which occurs while executing **SCProver**. Each expression is wrapped by **HoldForm** in order to prevent the expression from being simplified and executed by *Mathematica*. Private symbols from other packages need the full symbol name including the context. For an expression **expr** with the result **res** the definitions of **tr** and **trRes** look like this:

```
tr[HoldForm[expr]] := ...
trRes[HoldForm[expr], res] := ...
```

expr and **res** are usually pattern with blanks.

The implementation of both functions is done analogous to the definitions in the package **ProverKernel**. In most cases the function body is only a call of **trPrint** with a certain output text.

```
Clear[tr, trRes];
```

Final Goals

```
tr[HoldForm[SC`ProverKernel`Private`Prover[
     ™Equal[expr_, expr_], KB_List]]] :=
  trPrint["This formula is true: ", ™Equal[expr, expr]];

tr[HoldForm[SC`ProverKernel`Private`Prover[
     ™Iff[expr_, expr_], KB_List]]] :=
  trPrint["This formula is true: ", ™Iff[expr, expr]];

tr[HoldForm[SC`ProverKernel`Private`Prover[ True, KB_List]]] :=
  trPrint["This formula is true: ", True];
```

Implication

```
tr[HoldForm[SC`ProverKernel`Private`Prover[
     ™Implies[™Not[True], B_], KB_]]] :=
  trPrint["From 'False' everything can be concluded,
     hence this formula is true: ", ™Implies[False, B]];

tr[HoldForm[
    SC`ProverKernel`Private`Prover[ ™Implies[A_, B_], KB_]]] :=
  trPrint["Assume: ", A, "\n and prove: ", B];
```

No output for the next **Prover** call (This expression occurs before parameter **KB** is evaluated and without this definition the line before would be called twice):

```
tr[HoldForm[SC`ProverKernel`Private`Prover[_, Append[__]]]] :=
  Null;
```

Conjunction

```
tr[HoldForm[
    SC`ProverKernel`Private`Prover[ ™And[forms__], KB_]]] :=
  trPrint["Prove the conjunction by proving
     each part separately: ", {forms}];

trRes[HoldForm[
    SC`ProverKernel`Private`Prover[ ™And[forms__], KB_]],
   ¢Proved] := trPrint["All sub goals proved."];
```

```
tr[HoldForm[SC`ProverKernel`Private`ProveSubgoal[
     form_, _Symbol]]] := Null;

tr[HoldForm[
    SC`ProverKernel`Private`ProveSubgoal[form_, KB_]]] :=
  (trPrint["Prove sub goal: ", form]; $curDepth++);

trRes[HoldForm[SC`ProverKernel`Private`ProveSubgoal[
     form_, _Symbol]], _] := Null;

trRes[
   HoldForm[SC`ProverKernel`Private`ProveSubgoal[form_, KB_]],
   ¢Proved] := ($curDepth--; trPrint["Sub goal proved.
      Add the formula to the knowledge base: ", form]);

trRes[
   HoldForm[SC`ProverKernel`Private`ProveSubgoal[form_, KB_]],
   ¢Failed] := ($curDepth--;
   trPrint["Proof of sub goal failed. Abort the proof."]);
```

Implementation note:

If the second parameter of **ProveSubgoal** is a symbol, then this expression contains an unevaluated variable for **KB**. This expression occurs before the parameter **KB** is evaluated. After evaluation of **KB** the expression matches another definition hence only one output is generated.

Induction

```
tr[HoldForm[SC`ProverKernel`Private`Prover[
      ™ForAll[•ind[v_, base_, succ_], formula_], KB_]]] /;
   SC`ProverKernel`Private`$Induction :=
  trPrint["Prove by induction: ",
   ™ForAll[•ind[v, base, succ], formula]];

tr[HoldForm[SC`ProverKernel`Private`InductionBase[
     formula_, v_, base_, KB_]]] :=
  (trPrint["Prove base case: ", formula /. v → base];
   $curDepth++);
trRes[HoldForm[SC`ProverKernel`Private`InductionBase[
     formula_, v_, base_, KB_]], result_] := ($curDepth--);
```

```
tr[HoldForm[SC`ProverKernel`Private`InductionStep[
     formula_, v_, succ_, vFix_•fix, KB_]]] :=
  (trPrint["Assume (induction hypotheses): ",
    formula /. v → vFix,
    "\nand prove (induction step): ", formula /. v → succ[vFix]];
   $curDepth++);
trRes[HoldForm[SC`ProverKernel`Private`InductionStep[formula_,
     v_, succ_, vFix_•fix, KB_]], result_] := ($curDepth--);
```

Implementation note:

The prover uses induction only if the prover variable **SC`ProverKernel`Private`$Induction** is True. Hence the tracer has to check this condition before printing an output.

Course of values induction

```
tr[HoldForm[SC`ProverKernel`Private`Prover[
      ™ForAll[•ind[v_, ordering_], formula_], KB_]]] /;
   SC`ProverKernel`Private`$Induction := Module[{vFix},
   trPrint["Prove by Course-of-Values-Induction: ",
    ™ForAll[•ind[v, ordering], formula]];
   vFix = SC`ProverKernel`Private`ArbitraryButFixedConstant[
     v, formula, KB];
   trPrint["Assume (induction hypotheses): ",
    ™ForAll[v, ™Implies[ordering[v, vFix], formula]]];
   trPrint["And prove (induction step): ", formula /. v → vFix]
  ];
```

Implementation note:

The tracer has to check the prover variable **SC`ProverKernel`Private`$Induction** again. The assignment of the local variable **vFix** uses the same function call as the prover. Therefore it gets the same value, but it is computed a second time without being monitored by **TraceScan**.

If proving by induction is disabled then no output should be generated.

```
tr[HoldForm[SC`ProverKernel`Private`Prover[
     ™ForAll[•ind[v_, ___], formula_], KB_]]] /;
  ¬ TrueQ[SC`ProverKernel`Private`$Induction] := Null
```

Universal Quantifier - Arbitrary But Fixed

```
tr[HoldForm[SC`ProverKernel`Private`Prover[
     ™ForAll[v_, f_], KB_]]] := Module[{const},
   const = SC`ProverKernel`Private`ArbitraryButFixedConstant[
     v, f, KB];
   trPrint["Take ", const , " arbitrary but fixed and prove: ",
    f /. v → const];
  ];
```

No output for the next prover call:

```
tr[HoldForm[SC`ProverKernel`Private`Prover[
     _ /. _ → SC`ProverKernel`Private`ArbitraryButFixedConstant[
        __], _]]] := Null;
```

Proving Equivalence

```
tr[HoldForm[
   SC`ProverKernel`Private`Prover[™Iff[lhs_, rhs_], KB_]]] :=
 trPrint["For proving equivalence, it is
    sufficient to prove both directions: "]

tr[HoldForm[SC`ProverKernel`Private`ProveIffLeftToRight[
     lhs_, rhs_, KB_]]] :=
  (trPrint["Prove from left to right (=>): ",
    ™Implies[lhs, rhs]]; $curDepth++);

tr[HoldForm[SC`ProverKernel`Private`ProveIffRightToLeft[lhs_,
     rhs_, KB_]]] := (trPrint["Prove from right to left (<=): ",
    ™Implies[rhs, lhs]]; $curDepth++);

trRes[HoldForm[SC`ProverKernel`Private`ProveIffLeftToRight[
     lhs_, rhs_, KB_]], result_] :=
  ($curDepth--);

trRes[HoldForm[SC`ProverKernel`Private`ProveIffRightToLeft[
     lhs_, rhs_, KB_]], result_] :=
  ($curDepth--);
```

Rewriting a Predicate

```
tr[HoldForm[SC`ProverKernel`Private`RewriteFormula[
     formula_List, rules_List, kb_]]] :=
  (trPrint["Rewrite this formula: ", formula,
    If[$trPrintRules === True,
     "\nUsing these rules: " <> ToStringPrivate[rules] ]
   ]);
```

Rewriting an Equality

```
tr[HoldForm[SC`ProverKernel`Private`Prover[™Equal[
       lhs_™CaseDistinction, rhs_™CaseDistinction], KB_]]] /;
   lhs =!= rhs := trPrint["Equalities with case distinctions
     on both sides are not supported. The proof fails."];

tr[HoldForm[SC`ProverKernel`Private`Prover[
     ™Equal[cd_™CaseDistinction, rhs_], KB_]]] :=
  trPrint["Using the symmetry of equality: Swap
     left hand side and right hand side."];

tr[HoldForm[SC`ProverKernel`Private`Prover[
     ™Equal[lhs_, cd_™CaseDistinction], KB_]]] := Null;

tr[HoldForm[SC`ProverKernel`Private`RewriteEquality[
     lhs_, rhs_, rules_List, kb_]]] :=
  (trPrint["Rewrite both sides of the equality: ",
    lhs, "==", rhs,
    If[$trPrintRules === True,
     "\nUsing these rules: " <> ToStringPrivate[rules] ]
   ]);

tr[HoldForm[SC`ProverKernel`Private`IsEqual[
     lhs_List, rhs_List]]] := Block[{i},
   i = Intersection[lhs, rhs];
   If[Length[i] > 0,
    trPrint["The term ", First[i], " appears on both sides
       of the equality, hence the equality is true."]];
  ];
```

Proving of Case Conditions

```
tr[HoldForm[SC`ProverKernel`Private`ProveCaseConditions[KB_][
     cases_™CaseDistinction]]] := ($curDepth++;
   trPrint["Try to prove these case conditions: ",
    (#[[2]]) & /@ (List @@ cases)];);

trRes[
   HoldForm[SC`ProverKernel`Private`ProveCaseConditions[KB_][
     cases_™CaseDistinction]], result_™CaseDistinction] :=
  (trPrint["No case was proven completely, the remaining
      case conditions may be used for proving by cases: ",
    (#[[2]]) & /@ (List @@ result)];
   $curDepth--);

trRes[
   HoldForm[SC`ProverKernel`Private`ProveCaseConditions[KB_][
     cases_™CaseDistinction]], {}] :=
  (trPrint["No case was proven."]; $curDepth--);

trRes[
   HoldForm[SC`ProverKernel`Private`ProveCaseConditions[KB_][
     cases_™CaseDistinction]], result_] :=
  (trPrint["This term will be used: ", result]; $curDepth--);
```

Recursive Prover Call (Extended by Disproving)

```
tr[HoldForm[SC`ProverKernel`Private`SCProveDisprove[
     ™CaseDistinction[___][[__]], KB_]]] := Null;
tr[HoldForm[SC`ProverKernel`Private`SCProveDisprove[
     goal_, KB_]]] :=
  (trPrint["Prove/Disprove: ", goal]; $curDepth++;);

trRes[HoldForm[SC`ProverKernel`Private`SCProveDisprove[
     goal : ™CaseDistinction[___][[__]], KB_]], result_] := Null;
trRes[HoldForm[SC`ProverKernel`Private`SCProveDisprove[
     goal_, KB_]], ¢Proved] :=
  ($curDepth--; trPrint["Sub goal proved: " , goal];);
trRes[HoldForm[SC`ProverKernel`Private`SCProveDisprove[
     goal_, KB_]], ¢Disproved] :=
  ($curDepth--; trPrint["Sub goal disproved: " , goal];);
trRes[HoldForm[SC`ProverKernel`Private`SCProveDisprove[
     goal_, KB_]], result_] :=
  ($curDepth--; trPrint["Proof of sub goal failed: " , goal];);
```

Proving By Cases

Proving a Goal by Cases

```
tr[HoldForm[SC`ProverKernel`Private`ProveGoalByCases[
     ™Equal[lhs_List, rhs_List], KB_,
     cases_™CaseDistinction, step_]]] :=
  (trPrint["Proof by cases of: ", lhs, " == ", rhs];
   trPrint[" Using these cases: ",
    SC`ProverKernel`Private`GenerateAssumptions[cases]]);
tr[HoldForm[SC`ProverKernel`Private`ProveGoalByCases[formula:
      _™Equal | _List, KB_, cases_™CaseDistinction, step_]]] :=
  (trPrint["Proof by cases of: ", formula,
    "\nUsing these cases: ",
    SC`ProverKernel`Private`GenerateAssumptions[cases]]);

trRes[HoldForm[SC`ProverKernel`Private`ProveGoalByCases[
     formula_, KB_, cases_™CaseDistinction, step_]],
   ¢Proved] := trPrint["All cases proved."];
trRes[HoldForm[SC`ProverKernel`Private`ProveGoalByCases[
    formula_, KB_, cases_™CaseDistinction, step_]], ¢Failed] :=
 trPrint["Failed to prove at least one case."];
trRes[HoldForm[SC`ProverKernel`Private`ProveGoalByCases[
    formula_, KB_, cases_™CaseDistinction, step_]], result_] :=
 trPrint["Proving all cases: ", result];
```

Proving a Case

```
tr[HoldForm[SC`ProverKernel`Private`ProveCase[
     {goal_™CaseDistinction}, KB_, case_, step_]]] :=
  (trPrint["Proving case: ", case]; $curDepth++;
   trPrint["Goal formula: ", goal];);
```

The first case appears only if a case distinction was in the goal formula already before rewriting.

```
tr[HoldForm[SC`ProverKernel`Private`ProveCase[
     _, KB_, case_, step_]]] :=
  (trPrint["Proving case: ", case]; $curDepth++);
```

```
trRes[HoldForm[SC`ProverKernel`Private`ProveCase[
     _, KB_, case_, step_]], ¢Proved] :=
  (trPrint["Case proved."]; $curDepth--);
trRes[HoldForm[SC`ProverKernel`Private`ProveCase[
     _, KB_, case_, step_]], ¢Failed] :=
  (trPrint["Proof of this case failed."]; $curDepth--);
trRes[HoldForm[SC`ProverKernel`Private`ProveCase[
     _, KB_, case_, step_]], result_] :=
  (trPrint["Proof result: ", result]; $curDepth--);
```

Rewriting Tools

The output generation for **RewriteAllNew** is a little bit tricky. The function is called at three different positions, but the evaluated parameters have identical patterns. So the output depends on the name of the variable used in the parameter. Therefore the parameter **goals** needs to be a **Symbol**. In this case the unevaluated variable **goals** contains the symbol name of the variable used in the original function call. The function **GetRWSide** generates a text depending on the symbol name.

For the output the value of **goals** is needed. To get this value, a special effect of tracing is used. While the function **trRes** is evaluated, the execution of the traced function **SCProver** is interrupted. Evaluating the symbol stored in **goals** returns the value of the symbol. Using the variable **goals** will immediately return the value even if in the function pattern, which contains **HoldForm**, a **Symbol** is claimed. Hence, for getting the symbol name in **goals** it is necessary to wrap it by **HoldForm**.

```
trRes[HoldForm[SC`ProverKernel`Private`RewriteAllNew[goals_,
     oldgoals_, rulesOutermost_, rulesSubterms_]], {}] := Null;

trRes[
   HoldForm[SC`ProverKernel`Private`RewriteAllNew[goals_Symbol,
     oldgoals_, rulesOutermost_, rulesSubterms_]], new_List] :=
  If[goals =!= new,
   trPrint["Rewrite " <> GetRWSide[LocalName[HoldForm[goals]]] <>
     ": ", goals, " to ", new]
  ];
```

The function **GetRWSide** returns a text depending on the symbol name. If the symbol name is "lhs" or "rhs" then the function RewriteAllNew was called while rewriting an equality. In this case the text should denote the side of the equation that was rewritten. In all other cases a formula was rewritten.

```
Clear[GetRWSide];
GetRWSide["lhs"] = "(lhs)";
GetRWSide["rhs"] = "(rhs)";
GetRWSide[_] = "formula";
```

The auxiliary function **LocalName** removes the context of the symbol and the identifier for uniqueness. The returned string contains only the symbol name like it is used in the package for **SCProver**.

```
Clear[LocalName];
LocalName[s_HoldForm] :=
  StringReplace[ToStringPrivate[s], "$" ~~ __ → ""];
```

Default Case for tr

This prints an output only if the debugging variable **$trSKIP** is true. This may be used for debugging the tracer.

```
tr[t___] := If[$trSKIP, trPrint["SKIP: ", t]];
```

Tools

Symbol names are handier if they do not contain the full context. Therefore this function removes the private context from a given string.

```
Clear[ToStringPrivate];
ToStringPrivate[s_String] :=
  StringReplace[s, "SC`ProverKernel`Private`" → ""];
ToStringPrivate[expr_] := ToStringPrivate[ToString[expr]];
```

End

```
End[];

EndPackage[];
```

This ends the definition of the package.

The code in the next line allows loading the package immediately:

```
Get["SC`ProverKernel`"]
```

Example for TraceScan

This gives a simple example for using **`TraceScan`**. It shows the order of the function calls.

```
Clear[trBefore, trAfter];
trBefore[expr_] := Print["Before: ", expr];
trAfter[expr_, result_] :=
  Print["After:  ", expr, " -> ", result];

TraceScan[trBefore, f[f[1 + 2]], f[___], trAfter]
```

```
Before: f[f[1 + 2]]

Before: f[1 + 2]

Before: f[3]

After:  f[3] -> f[3]

After:  f[1 + 2] -> f[3]

Before: f[f[3]]

After:  f[f[3]] -> f[f[3]]

After:  f[f[1 + 2]] -> f[f[3]]

f[f[3]]
```

Note, that **`Print`** removes the wrapped **`HoldForm`** before printing the expression without evaluating it.

A.4 References

[BKR95]
A.Bouhoula, E.Kounalis, and M.Rusinowitch.
Automated Mathematical Induction.
in *Journal of Logic and Computation*, Vol. 5, pp. 631-668, 1995.

[Buch96a]
B.Buchberger.
Mathematica as a Rewrite Language.
In T.Ida, A.Ohori, and M.Takeichi, editors, Functional and Logic Programming (Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming, November 1-4,1996, Shonan Village Center), pages 1–13. World Scientific, Singapore-New Jersey-London-Hong Kong, 1996.

[Buch96b]
B.Buchberger.
Symbolic Computation: Computer Algebra and Logic.
In: Frontiers of Combining Systems, Proceedings of FROCOS 1996 (1st International Workshop on Frontiers of Combining Systems), March 26-28, 1996, Munich, F. Bader, K.U. Schulz (ed.), Applied Logic Series Vol.3, pp. 193-220. Kluwer Academic Publisher, Dordrecht - Boston - London, The Netherlands. 1996.

[Buch00]
B.Buchberger.
Focus Windows: A New Technique for Presenting Mathematical Proofs (in Automated Theorem Proving Systems).
Technical Report 2000-01-30, Research Institute for Symbolic Computation, Johannes Kepler University Linz, 2000.

[Buch04]
B.Buchberger.
Proving by First and Intermediate Principles, November 1-2 2004.
Invited talk at Workshop on Types for ”Mathematics / Libraries of Formal Mathematics”, University of Nijmegen, The Netherlands.

[Buch08]
B.Buchberger.
Various talks at Theorema seminar, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2008.

[BuLi81]
B.Buchberger, and F.Lichtenberger.
Mathematik für Informatik I - Die Methode der Mathematik (Mathematics for Computer Science I - The Method of Mathematics), German.
Springer-Verlag, Berlin - Heidelberg - New York, second edition, 1981.

[BuWi98]
B.Buchberger, and W.Windsteiger
The Theorema Language: Implementing Object- and Meta-Level Usage of Symbols.
SFB F013 Numerical and Symbolic Scientific Computing. Technical report no. 98-07, 1998.

[GiBu07]
M.Giese, and B.Buchberger.
Towards Practical Reflection for Formal Mathematics.
Technical report, RISC Report Series, University of Linz, Austria, 2007.

[GoJu98]
M.Goldstern, and H.Judah.
The Incompleteness Phenomenon.
A K Peters, Natick, Massachusetts, 1998.

[Gord98]
D. M. Gordon.
A survey of fast exponentiation methods.
Journal of Algorithms, 27(1):129–146, April 1998.

[Harr09]
J. Harrison.
HOL Light: An Overview
Proceedings of TPHOLs 2009, the 22nd International Conference on Theorem Proving in Higher Order Logics. Springer LNCS 5674, pp. 60-66.

[HKK+05]
W.Hunt, M.Kaufmann, R.Krug, J.Moore, and E.Smith.
Meta Reasoning in ACL2.
in *Theorem Proving in Higher Order Logics*, LNCS, Verlag Springer Berlin / Heidelberg, 2005.

[Knut81]
D.E.Knuth.
Seminumerical Algorithms, volume 2 of *The Art of Computer Programming*.
Addison-Wesley, Reading, Massachusetts, second edition, 1981

[Kuts02]
T.Kutsia
Unification with Sequence Variables and Flexible Arity Symbols and its Extension with Pattern-Terms. In: Artificial Intelligence, Automated Reasoning and Symbolic Computation. Proceedings of Joint AISC'2002 - Calculemus'2002 conference, Jacques Calmet and Belaid Benhamou and Olga Caprotti and Laurent Henocque and Volker Sorge (ed.), Lecture Notes in Artificial Intelligence 2385, pp. 290-304. July 1-5 2002. Springer Verlag, Marseille, France, ISBN 3-540-43865-3.

[KuBu05]
T.Kutsia, and B.Buchberger.
Predicate Logic with Sequence Variables and Sequence Function Symbols.
Technical Report 05-17, RISC Report Series, University of Linz, Austria, 2005.

[Mend66]
E.Mendelson.
Introduction to Mathematical Logic.
Van Nostrand Company, New York, reprinting, 1966.

[Shoe73]
J.Shoenfield.
Mathematical Logic.
Addison-Wesely, Reading, Massachusetts, second printing, 1973.

[Tma97]
B.Buchberger, T.Jebelean, F.Kriftner, M.Marin, E.Tomuta, and D.Vasaru.
A Survey of the Theorema Project.
Technical report no.97-15 in RISC Report Series, University of Linz, Austria, March 1997.

[Tma98]
B.Buchberger, K.Aigner, C.Dupre, T.Jebelean, F.Kriftner, M.Marin, K.Nakagawa, O.Podisor, E.Tomuta, Y.Usenko, D.Vasaru, and W.Windsteiger.
Theorema: An Integrated System for Computation and Deduction in Natural Style.
Technical Report 98-25, RISC Report Series,University of Linz,Austria, December 1998. Also available as SFB Report No.98-06.

[Tma99a]
B.Buchberger, C.Dupre, T.Jebelean, F.Kriftner, K.Nakagawa, D.Vasaru, and W.Windsteiger.
Theorema: A Progress Report.
Technical Report 99-42, RISC Report Series,University of Linz,Austria, December 1999.
Also available as SFB Report 99-35, Johannes Kepler University Linz, Spezialforschungsbereich F013, December, 1999.
Published in: *Symbolic Computation and Automated Reasoning* (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning), M.Kerber, M.Kohlhase (ed.), pp.98-113.
A.K.Peters, Natick, Massachusetts, St.Andrews, Scotland, August 2000.

[Tma99b]
B.Buchberger, C.Dupre, T.Jebelean, F.Kriftner, K.Nakagawa, D.Vasaru, and W.Windsteiger.
Theorema: A Short Demo.
Technical Report 99-45, RISC Report Series, University of Linz, Austria, December 1999.
Also available as SFB Report 99-37, Johannes Kepler University Linz, Spezialforschungsbereich F013, December, 1999.

[Tma00a]
B.Buchberger, D.Vasaru, and T.Jebelean.
The Theorema System: Current Status and the Proving-Solving-Computing Cycle.
Technical Report 00-37, RISC Report Series, University of Linz, Austria, May 2000.

[Tma00b]
B.Buchberger, C.Dupre, T.Jebelean, B.Konev, F.Kriftner, T.Kutsia, K.Nakagawa, F.Piroi, D.Vasaru, and W.Windsteiger.
The Theorema System: Proving, Solving, and Computing for the Working Mathematician.
Technical Report 00-38, RISC Report Series,University of Linz, Austria, August 2000.

[Tma00c]
B.Buchberger, C.Dupre, T.Jebelean, B.Konev, F.Kriftner, T.Kutsia, K.Nakagawa, F.Piroi, D.Vasaru, and W.Windsteiger.
The Natural Style Provers of Theorema: A Survey of Strategies for Different Mathematical Domains.
Technical Report 00-39, RISC Report Series, University of Linz, Austria, June 2000.

[Wolf03]
S.Wolfram.
The Mathematica Book
Wolfram Media Inc., Champaign, Illinois (USA), 2003.

[Wolf09]
Wolfram Mathemtica Documentation center
http://reference.wolfram.com/mathematica/guide/Mathematica.html, 20. August 2009