An Automated Prover for Set Theory in *Theorema*

Wolfgang Windsteiger*

RISC Institute
A-4232 Hagenberg, Austria
Wolfgang.Windsteiger@RISC.Uni-Linz.ac.at

Abstract. This paper presents some fundamental aspects of the design and the implementation of an automated prover for set theory within the well-known *Theorema* system. The method applies the "Prove-Compute-Solve"-paradigm as its major strategy for generating proofs in a natural style for statements involving constructs from set theory.

1 Introduction

The Set Theory prover in *Theorema* adapts the "**P**rove-**C**ompute-**S**olve" (short: PCS) prove strategy for proofs containing language constructs from set theory. The PCS paradigm was invented in [3] and it has already been applied successfully for proofs in elementary analysis in [13]. The main strategy in a PCS-oriented prover is to structure the proof generation into phases of

- proving (P), i.e. application of logical inference rules for propositional connectives and for quantifiers,
- computing (C), i.e. rewriting w.r.t. formulae in the knowledge base,
- solving (S), i.e. instantiation of existential variables.

Having the computer algebra system Mathematica in the background of *Theorema*, we aim towards applying known solution methods from computer algebra during the S-phase, such as the Gröbner bases method for systems of algebraic equations or Collins' CAD method for systems of inequalities over the reals.

The current design of provers in the *Theorema* system requires a so-called "user prover" to be composed from "special provers" (see [12]). A special prover consists of a collection of inference rules, whereas the user prover guides *the strategy*, through which the proof search procedure applies the inference rules. Consequently, the set theory user prover consists of a *set theory proving unit* handling set-related quantifiers in the goal or in the knowledge base, a *set theory computing unit*, and a *set theory solving unit*. In addition to these set theory specific components, the Set Theory prover utilizes several special provers available in the *Theorema* system, such as `BasicND` for handling basic quantifiers

* This work has been supported by the "SFB Numerical and Symbolic Scientific Computing" (F013) at the University of Linz and the european union "CALCULEMUS Project" (HPRN–CT–2000–00102).

from predicate logic using natural deduction, `QR` for rewriting using quantified formulae in the knowledge base, and `CDP` for applying case distinction (invented in [3]; see [13] for detailed description).

The description is structured as follows: Section 2 describes the theoretical basis upon which the set theory user prover `SetTheoryPCSProver` is built, Section 3 introduces the set theory proving units `STP` and `STKBR`, Section 4 describes the set theory computing unit `STC`, Section 5 presents the set theory solving unit `STS`, and finally we conclude with an example of a proof generated by the `SetTheoryPCSProver` in Section 6.

More details on implementation and a number of case studies using the Set Theory prover can be found in [14].

2 The Theoretical Basis for the Set Theory Prover

The use of set theory in *Theorema* is not tied to one particular axiomatization of set theory. Instead, we introduce "sets" on the level of the language by providing the braces '{' and '}' as a flexible arity matchfix function symbol used for constructing finite sets and the set quantifier. Providing these language constructs, we implicitly assume that sets such as $\{a\}$, $\{1, b\}$, $\{x \underset{x}{|} \mathfrak{P}_x\}$, or $\{\mathfrak{T}_x \underset{x}{|} \mathfrak{P}_x\}$ actually exist, which is typically guaranteed by some axioms of the underlying set theory. There are different approaches, in the Zermelo-Fraenkel axiomatization (ZF) as described e.g. in [4] the existence of the singleton $\{a\}$ follows from an axiom on power sets and the existence of $\{1, b\}$ follows from the existence of singletons together with an axiom on unions, whereas in an axiomatization given in [11], which also follows the spirit of ZF, the existence of $\{1, b\}$ is guaranteed by an axiom of pairing and the singleton $\{a\}$ is then just defined to denote the pair $\{a, a\}$.

Among mathematicians *using set theory*, however, there is a common understanding of the intuition behind these constructs, which is more or less independent of the concrete axiomatization of set theory. The *Theorema* Set Theory prover tries to support the formalization of mathematics *using some parts of set theory*, it is not priorly targeted towards generating proofs in the *formal construction of set theory itself*! Following this spirit, we provide *definitions of the basic constructs of set theory* supported by the *Theorema* Set Theory prover, which follow the ZF-style of axiomatizing set theory. Thus, the *Theorema* Set Theory prover should be a useful tool for mathematicians embedding their work in ZF set theory.

How does *Theorema* then handle the set quantifier, i.e. the expression $\{x \underset{x}{|} \mathfrak{P}_x\}$, which allows one to define a set from a property $\mathfrak{P}_x$? In the literature, this is often addressed as *the abstraction* of a set from a property and it goes back to G. Cantor, the founder of modern set theory. As explained in (nearly) every introductory course in mathematics, the unrestricted use of abstraction soon leads to contradictions such as the well-known Russel paradox. With R denoting the "Russel-set" $\{x \underset{x}{|} x \notin x\}$ it is straight-forward to derive the contradiction

$R \in R \Leftrightarrow R \notin R$. ZF set theory resolves this paradox by imposing a certain structure on the formula $\mathfrak{P}_x$ in an abstraction $\{\underset{x}{x \mid \mathfrak{P}_x}\}$, which disallows constructions like R. Von-Neumann-Gödel-Bernays' axiomatization (NGB) of set theory (see e.g. [1] or [9]) distinguishes between sets and classes and allows the membership predicate only for sets. Russel's paradox is avoided by showing that R is not a set an therefore $R \in R$ is not a well-formed assertion. Russel himself introduced types as a way out by allowing membership only for sets of different type (see [10]). $R \in R$ is not allowed on the grounds that R and R are not of different type.

Theorema as such does not force the user into one of the above mentioned axiomatizations. The *language* allows unrestricted use of both the set quantifier and the membership predicate, therefore allowing both the definition of R and formulae such as $R \in R \Leftrightarrow R \notin R$. The Set Theory *prover*, however, relies on ZF and therefore refuses to apply inference rules on formulae involving constructs such as R. It is based on the following definitions[1].

Definition 1.

$$a \in \{x \underset{x \in \mathfrak{s}}{\mid} \mathfrak{P}_x\} :\Longleftrightarrow a \in \mathfrak{s} \wedge \mathfrak{P}_{x \to a} \tag{1}$$

$$a \in \{\mathfrak{T}_x \underset{x \in \mathfrak{s}}{\mid} \mathfrak{P}_x\} :\Longleftrightarrow \underset{x \in \mathfrak{s}}{\exists} (\mathfrak{P}_x \wedge a = \mathfrak{T}_x) \tag{2}$$

$$\emptyset := \{x \underset{x \in \mathfrak{s}}{\mid} x \neq x\} \quad \text{(for some set } \mathfrak{s}) \tag{3}$$

$$a \in \{a_1, \ldots, a_n\} :\Longleftrightarrow a = a_1 \vee \ldots \vee a = a_n \quad (\text{for } n \geq 1) \tag{4}$$

$$a \in \mathfrak{S}_1 \cup \ldots \cup \mathfrak{S}_n :\Longleftrightarrow a \in \mathfrak{S}_1 \vee \ldots \vee a \in \mathfrak{S}_n \quad (\text{for } n \geq 2) \tag{5}$$

$$a \in \bigcup \mathfrak{S} :\Longleftrightarrow \underset{s \in \mathfrak{S}}{\exists} a \in s \tag{6}$$

$$\bigcup_{\substack{x \in \mathfrak{s} \\ \mathfrak{C}_x}} \mathfrak{S}_x := \bigcup \{\mathfrak{S}_x \underset{x \in \mathfrak{s}}{\mid} \mathfrak{C}_x\} \tag{7}$$

$$a \in \mathfrak{S}_1 \cap \ldots \cap \mathfrak{S}_n :\Longleftrightarrow a \in \mathfrak{S}_1 \wedge \ldots \wedge a \in \mathfrak{S}_n \quad (\text{for } n \geq 2) \tag{8}$$

$$a \in \bigcap \mathfrak{S} :\Longleftrightarrow \underset{s \in \mathfrak{S}}{\forall} a \in s \tag{9}$$

$$\bigcap_{\substack{x \in \mathfrak{s} \\ \mathfrak{C}_x}} \mathfrak{S}_x := \bigcap \{\mathfrak{S}_x \underset{x \in \mathfrak{s}}{\mid} \mathfrak{C}_x\} \tag{10}$$

$$\mathfrak{S}_1 \subseteq \mathfrak{S}_2 :\Longleftrightarrow \underset{a}{\forall} a \in \mathfrak{S}_1 \Rightarrow a \in \mathfrak{S}_2 \tag{11}$$

$$\mathfrak{S}_1 = \mathfrak{S}_2 :\Longleftrightarrow \underset{a}{\forall} a \in \mathfrak{S}_1 \Leftrightarrow a \in \mathfrak{S}_2 \tag{12}$$

We list only the most important definitions. In the concrete implementation, the prover can handle some more like e.g. cross product or set complement, see [14]. When using the *Theorema* Set Theory prover one accepts these definitions and assumes an underlying axiomatic system such as ZF that guarantees the existence of all sets defined above.

[1] $\mathfrak{P}_{x \to a}$ stands for $\mathfrak{P}$ with each free occurrence of x substituted by a.

2.1 Preliminaries on Terminology

We will use the following terminology in the description of the prove modules: a proof situation $\kappa \vdash \mathfrak{G}$ is made up from a knowledge base of assumptions κ and a goal $\mathfrak{G}$, and it should be understood as an abbreviation for the phrase: "We have to prove $\mathfrak{G}$ from κ". Typically, the goal will be a single formula of the *Theorema* language, hence we use gothic letters such as $\mathfrak{G}$ or $\mathfrak{S}$ like in Def. 1, whereas the knowledge base consists of *a collection of formulae*, called the assumptions.

Now, the task of the special provers is essentially the execution of individual *proof steps* that *reduce* the proof situation, where the rules applied by the special provers guiding the transformations of proof situations are called *inference rules*. Thus, an inference rule turns a proof situation $\kappa \vdash \mathfrak{G}$ into a proof situation $\kappa' \vdash \mathfrak{G}'$ with a new goal $\mathfrak{G}'$ and a new knowledge base κ', where we call $\kappa \vdash \mathfrak{G}$ and $\kappa' \vdash \mathfrak{G}'$ the *in-situation* and the *then-situation* of the rule, respectively. In the description of inference rules, we will denote an inference rule named $\mathcal{I}$ transforming $\kappa \vdash \mathfrak{G}$ into $\kappa' \vdash \mathfrak{G}'$ by

$$\mathcal{I} : \frac{\kappa' \vdash \mathfrak{G}'}{\kappa \vdash \mathfrak{G}}$$

(read as: "The rule $\mathcal{I}$ justifies a proof step to reduce the proof of $\mathfrak{G}$ from κ to a proof of $\mathfrak{G}'$ from κ'"). This notation is similar to notations used in logic for describing inference rules in formal prove calculi, like for instance the sequent calculus (e.g. [5]), the Gentzen calculus (e.g. [6]), or the calculus of natural deduction (e.g. [8]). Certain similarities to these formalisms are desired, but we use it purely as a symbolic description for proof steps, and we do not refer to any meaning of the symbols in any known logic system.

An example of a well-known inference rule written in this style is

$$\textbf{ArbitraryButFixed} : \frac{\kappa \vdash \mathfrak{P}_{x \to x_0}}{\kappa \vdash \underset{x}{\forall} \mathfrak{P}_x} \quad \text{(where } x_0 \text{ is a new constant)}$$

meaning that, in order to prove $\underset{x}{\forall} \mathfrak{P}_x$ (from κ) it suffices to prove $\mathfrak{P}_{x \to x_0}$ (from κ) for a new constant x_0.

3 STP and STKBR: The Set Theory Proving Units

The PCS proof strategy imposes a structure on proofs as alternating phases of proving, computing, and solving, as already described in Sect. 1. *Proving* can in this context be interpreted as *eliminating theory-specific language constructs*, in particular eliminating *quantifiers*. The Set Theory prover is a prover that can handle language constructs from set theory *in addition* to standard predicate logic. Therefore, it re-uses the special provers available in the *Theorema* system for handling propositional connectives and the $\forall$-quantifier (see [3], [13], and [14]). Set theory specific *proving* is covered by the two new special provers `STP` and STKBR. During the Prove-phase, we alternate steps of *reducing the goal* with steps of *expanding the knowledge base*. While STP reduces set theory specific language constructs in the proof-goal, STKBR expands them in the knowledge base.

3.1 Inference Rules used in STP

Set theory specific goal reduction is implemented as a special prover named STP. As most of the special provers in *Theorema*, STP implements individual inference rules as individual function definitions for one Mathematica function STP, which differ in the patterns specifying the proof situation. The choice, which inference rule to apply next, depends mainly on the pattern of the proof situation given in the inference rule, a few inference rules are influenced in addition by global variables used by STP, some strategies depend on the proof progress stored in STP's *local proof context*, which is the third parameter in a call to STP in addition to the proof-goal and the knowledge base.

Mainly, the inference rules are grouped into rules for *membership*, rules for *inclusion*, and rules for *set equality*. The rules for membership contain at least one inference rule for each "kind of set" introduced in Def. 1, in some cases we provide tailored rules in order to offer special treatment for special cases. We show some of the membership rules as they are used in STP.

$$\textbf{MembershipAbstraction}: \frac{\mathfrak{K} \vdash \mathfrak{t} \in \mathfrak{s} \wedge \mathfrak{P}_{x \to \mathfrak{t}}}{\mathfrak{K} \vdash \mathfrak{t} \in \{x \underset{x \in \mathfrak{s}}{|} \mathfrak{P}_x\}}$$

We give an impression of what the result of this inference rule is in a concrete example. If, during a concrete proof, the proof search procedure arrives at a proof situation, where we need to prove $a \in \{x \underset{x \in s}{|} x < 10\}$ w.r.t. some knowledge base KB, then the special prover STP would be called in the following format:

$$\mathrm{STP}[\bullet \mathrm{lf}[\texttt{"1"}, a \in \{x \underset{x \in s}{|} x < 10\}, \bullet \mathrm{finfo}[]], \bullet \mathrm{asml}[\mathrm{KB}], \mathrm{af}]$$

where •lf[...] represents the proof-goal labelled "1", $\bullet\mathrm{asml}[KB]$ is the current knowledge base, and "af" are the additional facts containing among others STP's local proof context. Note, that this is *not* how the *user* needs to call the prover, the actual call of the special prover is based on internal data structures, which are built-up automatically during the proof search. It is the task of the *Theorema* User Language (see [14]) to serve as an interface between the user and the internal data structures as they show up above. The result of this call is the *new proof situation*

```
{"AndNode",
 {"MembershipAbstraction", .usedFormulae["1"],
   .generatedFormulae[.lf["1'",And[Element[a,s],a<10],.finfo[]]]},
 {{"ProofSituation", .lf["1'",And[Element[a,s],a<10],.finfo[]],
   .asml[kb], af}}, {}, {}, "pending"}
```

The proof search procedure will insert this node into the *Theorema* proof object. The node contains enough information in order to later *simplify* a successful proof (object) and to generate the natural language text from it. Note, however, that it *does not contain* the natural language text representation itself! When later generating the proof presentation from a proof object, this step of the proof would read as follows:
In order to prove (1) we have to show:
(1') $a \in s \wedge a < 10$.

The correctness of the inference rule "MembershipAbstraction" follows immediately from the definition of set abstraction. Some of the inference rules, however, condense several inference steps into one compact rule to be applied. In these cases, we provide correctness proofs for the respective rules[2]. An example of such a rule is the elimination of the union-quantifier in the goal.

$$\textbf{MembershipUnionOf}: \frac{\mathsf{K} \vdash \underset{x \in \mathfrak{s}}{\exists} (\mathfrak{t} \in \mathfrak{S}_x \wedge \mathfrak{C}_x)}{\mathsf{K} \vdash \mathfrak{t} \in \bigcup\limits_{\substack{x \in \mathfrak{s} \\ \mathfrak{C}_x}} \mathfrak{S}_x}$$

MembershipUnionOf reduces the proof of $\mathfrak{t} \in \bigcup\limits_{\substack{x \in \mathfrak{s} \\ \mathfrak{C}_x}} \mathfrak{S}_x$ to prove $\underset{x \in \mathfrak{s}}{\exists} (\mathfrak{t} \in \mathfrak{S}_x \wedge \mathfrak{C}_x)$.

Proof. Assume $\underset{x \in \mathfrak{s}}{\exists} (\mathfrak{t} \in \mathfrak{S}_x \wedge \mathfrak{C}_x)$, thus $\mathfrak{t} \in \mathfrak{S}_{x_0} \wedge \mathfrak{C}_{x_0}$ for some constant $x_0 \in \mathfrak{s}$. With $z := \mathfrak{S}_{x_0}$ we can infer from this $\mathfrak{t} \in z \wedge \mathfrak{C}_{x_0} \wedge z = \mathfrak{S}_{x_0}$, hence

$$\underset{z}{\exists} (\underset{x \in \mathfrak{s}}{\exists} \mathfrak{t} \in z \wedge \mathfrak{C}_x \wedge z = \mathfrak{S}_x) \ . \tag{13}$$

Separating the quantifiers in (13) gives $\underset{z}{\exists} (\mathfrak{t} \in z \wedge \underset{x \in \mathfrak{s}}{\exists} (\mathfrak{C}_x \wedge z = \mathfrak{S}_x))$, which, by (2), is equivalent to $\underset{z}{\exists} (\mathfrak{t} \in z \wedge z \in \{\mathfrak{S}_x \underset{x \in \mathfrak{s}}{|} \mathfrak{C}_x\})$. By (6) this is equivalent to $\mathfrak{t} \in \bigcup \{\mathfrak{S}_x \underset{x \in \mathfrak{s}}{|} \mathfrak{C}_x\}$, thus $\mathfrak{t} \in \bigcup\limits_{\substack{x \in \mathfrak{s} \\ \mathfrak{C}_x}} \mathfrak{S}_x$ by (7). □

Set inclusion reduces, by definition, to membership and set equality reduces to membership. In addition to these reductions, we implemented several inference rules for special cases that reduce the search depth for the proof search, e.g.

$$\textbf{ConjunctionSubset}: \frac{\text{proved}}{\mathsf{K} \vdash \{x \underset{x}{|} \ldots \wedge x \in \mathfrak{S} \wedge \ldots\} \subseteq \mathfrak{S}}$$

$$\textbf{EqualsEmptySet}: \frac{\mathsf{K} \vdash \neg \mathfrak{P}_{x \to x_0}}{\mathsf{K} \vdash \{\mathfrak{T}_x \underset{x}{|} \mathfrak{P}_x\} = \emptyset} \quad \text{where } x_0 \text{ is some new constant,}$$

[2] Ideally, the *Theorema* Predicate Logic Prover should be capable of producing these proofs when having Def. 1 in its knowledge base. Unfortunately, however, some of the definitions, notably (1), and some inference rules "live" on the language expression level and they refer to variable substitution, free variables and the like. In its current status, the *Theorema* language cannot express these things on the object level!

and some extensions so that the prover can also deal with cardinality and function properties such as bijectivity. For details see [14].

3.2 The Structure of STKBR

The special prover `STKBR` (for Set Theory Knowledge Base Rewriting) uses a level saturation technique (see also [7]), to infer *new knowledge* from the knowledge base by unfolding definitions of set theoretic language constructs. It differs drastically from most of the other special provers in the *Theorema* system in that it does not implement inference rules as *separate definitions* for *one* Mathematica function. This "classic" implementation scheme for *Theorema* special provers, which introduces one definition per proof situation, is not suitable for an efficient implementation of a level saturation mechanism, because inferring new formulae one at the time would result in a massive growth of the required search depth for the proof search. The `STKBR` function, instead, is implemented as just *one definition*, which produces *all possible new formulae during only one application*. This has the advantage, that several inference rules can be applied in parallel during one `STKBR`-step instead of adding only one new formula at the time to the knowledge base.

As a consequence, the `STKBR` function does not specify the syntactic pattern of the proof situation in its parameters but it is considered to be applicable to the current proof situation as soon as new formulae occur in the knowledge base compared to `STKBR`'s previous run. This check is done with the help of an entry in the local proof context that stores the labels of all assumptions that have already been treated in the preceding saturation level. In case new formulae have been added to the assumptions, the saturation of the current knowledge level happens in two phases:

- In a first phase, new formulae are, if desired, simplified by computation using built-in semantic knowledge available in the *Theorema* language semantics, see also `STC` in Sect. 4. In case this type of simplification is not desired, this phase can be skipped through a user option in the call of the prover.
- In a second phase, new knowledge is *inferred* from the simplified new formulae using inference rules for set theory. These inference rules are again grouped into two groups,
  - *Group One* containing rules for inferring new knowledge from *one* known formula and
  - *Group Two* containing rules for inferring new knowledge from *two* known formulae.

  Matching rules from Group One are applied to the simplified new formulae, matching rules from Group Two are applied to *all new pairs* of formulae that can be formed using additionally the simplified new formulae[3].

[3] Up to now, no inference rules have been implemented that depend on three formulae. As soon as such inference rules are needed, we will provide a Group Three of inference rules, which will be applied to all possible triples of formulae.

All formulae generated during these two phases are adjoined to the knowledge base for the new proof situation. The augmented knowledge base is considered to contain *all knowledge*, that can be made available at that point, thus, we call it a *saturated knowledge level*. The schematized flow of STKBR level saturation mechanism is shown in Fig. 1, where the boxed names are the names of the respective functions in the actual implementation.

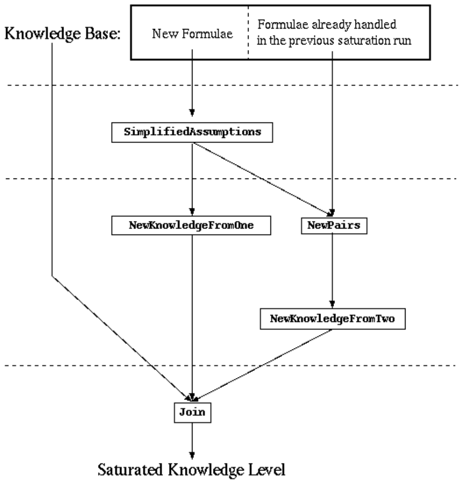

Fig. 1. Schematic flow of the STKBR level saturation

Phase one is accomplished by calling the function 'SimplifiedAssumptions' with two arguments 'kb' and 'handled', where 'kb' is the entire knowledge base and 'handled' is the list of labels of assumptions that have already been used in the previous saturation phase. Each formula from the knowledge base, whose label is *not among the handled labels*, is sent through the function 'Evaluate-FromProve', which *computes* a simplified version of the formula w.r.t. semantic knowledge from the *Theorema* language. 'EvaluateFromProve' is the function used also in the STC module for *goal simplification by computation*, see Sect. 4. Note, that 'EvaluateFromProve' is based on the function 'EvaluateStandard', which is the basic evaluation function for computations using *Theorema* semantics, which is used also by `Compute`, the top-level user function to initiate computations. This guarantees utmost coherence between all computations happening in the *Theorema* system, be it on the user level by calling `Compute`, be it on the prover level by doing simplifications on the goal or on the knowledge base.

Phase two is covered in the implementation by the function 'Augmented-KnowledgeBase', which receives two arguments: the simplified knowledge base 'kb' resulting from phase one and 'handled' as above. Group One of inference rules is applied componentwise to *all new assumptions*, Group Two of inference rules is applied to *all new pairs* that can be formed using the new assumptions and the results are added to the knowledge base.

The inference rules applied by `STKBR` can more or less be read off Def. 1, hence we do not list them here. The interested reader can look up [14].

Rule Locking Now consider the two inference rules

- from $x \in A \cap B$ we can infer $x \in A$ and $x \in B$ and
- from $x \in A$ and $x \in B$ we can infer $x \in A \cap B$

occurring in `STKBR`. We call these rules *inverse* to each other. Unrestricted use of inverse rules immediately results in a cycle in the proof search, which we avoid by a mechanism of *rule locking*. As soon as an inference rule $\mathcal{I}$ infers a new formula $\mathfrak{F}$, we lock the inverse of $\mathcal{I}$ from being applied to $\mathfrak{F}$. Rule locking utilizes `STKBR`'s local proof context to store this type of information on the prove progress.

4 STC: The Set Theory Computing Unit

The *Theorema* language contains semantics essentially for *finite sets*, namely

- sets that are constructed using the set braces '{' and '}' as set constructor applied to finitely many arguments, and
- sets that are constructed using *algorithmic versions* of the set quantifier (see also [2]), i.e. set quantifiers with finite and computable range specifications (see [14]). In particular, *integer ranges* and *set ranges* for finite sets are algorithmic ranges, which lead to finite sets when used in combination with the set quantifier.

The *Theorema* semantics enables to *construct* finite sets as an enumeration of the (finitely many) elements contained in the set. Set operations (such as union, intersection, power set, etc.) on finite sets are implemented in a constructive fashion. *Proving properties* (such as membership, inclusion, or set equality) of finite sets therefore reduces to *testing finitely many cases*, which is implemented in the frame of the *Theorema* language as well.

From the user's point of view, computation using built-in semantics knowledge is available in the *Theorema* system through the top-level user function `Compute`. A typical computation involving finite sets is

$$\mathrm{Compute}[\{3x \underset{x\in\{1,2,3,4\}}{|} \text{is-prime}[x]\}]$$

resulting in the finite set $\{6, 9\}$.

It is the intention of the `STC` special prover to integrate the knowledge available for computations seamlessly into the *Theorema* proving machinery. Otherwise, all algorithmic knowledge about finite sets needed to be re-implemented inside the Set Theory prover, which would make it next to impossible to guarantee identical behavior in proving and computing. In order to avoid this duplication of code and knowledge, the `STC` prover simplifies the goal by sending the formula to the same evaluation function that is also used in `Compute` and in `STKBR`.

Basically, when the `STC` prover applies to a proof situation, one proof step consists of calling the evaluation function 'EvaluateFromProve' (see also Sect. 3.2) and, in case the result differs from the original form, of adding a node to the proof object, from which the effect and a complete trace of the computation can be displayed. Again, the use of 'EvaluateFromProve' preserves coherence with `STKBR` and `Compute`. Many details on combining computation with proving can be found in [14].

5 STS: The Set Theory Solving Unit

The special prover `STS` collects inference rules for eliminating existential quantifiers[4]. Methods used for instantiating existential goals range from *matching* against formulae in the knowledge base, over *unification* and *introduction of solve constants* until employing the Mathematica 'Solve' function to obtain solutions of equational goals. We present only one typical inference rules from `STS`.

$$\textbf{IntroSolveConstant}: \frac{\mathsf{K} \vdash \mathfrak{Q}_{y \to y^*} \wedge \underset{x \in \mathfrak{s}}{\exists} (\mathfrak{P}_x \wedge y^* = \mathfrak{T}_x) \wedge \mathfrak{R}_{y \to y^*}}{\mathsf{K} \vdash \underset{y}{\exists} (\mathfrak{Q}_y \wedge y \in \{\mathfrak{T}_x \underset{x \in \mathfrak{s}}{|} \mathfrak{P}_x\} \wedge \mathfrak{R}_y)}$$

where $\mathfrak{Q}_y$ and $\mathfrak{R}_y$ are possibly empty conjunctions of formulae and y^* is a *solve constant.*

A solve constant[5] is some constant, which we still have to assign a concrete value. Solve constants are introduced in order to eliminate existential quantifiers by substituting a constant for the quantified variable, where at the moment of introducing the constant, its concrete value can not yet be determined. For the proof to succeed, *all solve constants* that have been introduced must be eliminated by substituting appropriate *ground terms* in such a manner that the resulting formula can be proven. Of course, the strategy after introducing solve constants must always be to isolate the solve constants, which is typically done by *solving*, using methods depending on the nature of the remaining formula. Applying this strategy reduces proving to solving over various domains, and it offers the possibility to benefit from the great advances that have been accomplished in developing powerful solution methods in computer algebra.

[4] In fact, it should contain only the set theory specific part of solving. Since the solving components in the *Theorema* system are not yet far-advanced, we started with `STS` collecting inference rules for proof situations as they appear in typical proofs in set theory.

[5] What we call solve constant is often addressed as *meta variable* by other authors. The technique of meta variables is well known and used also in other systems. Essentially, it imitates what a human does when instantiating existential quantifiers, in particular, in the well-known proof on limits, continuity, etc.

The inference rule described above might appear random. It is part of STS since it applies exactly to proof situations left after expanding membership in a union, i.e. goals of the form $\mathfrak{t} \in \bigcup\{\mathfrak{T}_x \underset{x \in \mathfrak{s}}{|} \mathfrak{P}_x\}$. The rule eliminates the outermost existential quantifier, but it introduces another existential quantifier. STS contains further rules, which allow the elimination of the existential quantifier in this particular and even in other more general situations (see [14]). In addition to rules introducing solve constants, the STS prover, of course, also contains several rules for instantiating solve constants as soon as they appear in an isolated position.

6 An Example of an Automatically Generated Proof

Prove: $(G) \quad 36 \in \bigcup_{i \in \mathbb{N}} \{j^2 \underset{j \in \mathbb{N}}{|} j \geq i \wedge j \leq i+5\}$ under the assumption

$(A) \quad \underset{m,n}{\forall}\, n > m \Rightarrow \underset{i}{\exists}\, i \leq n \wedge i \geq m \wedge i \in \mathbb{N}$.

In order to show (G) we have to show

$$(1) \quad \underset{i}{\exists}\, 36 \in \{j^2 \underset{j \in \mathbb{N}}{|} j \geq i \wedge j \leq i+5\} \wedge i \in \mathbb{N} \ .$$

In order to prove (1) we have to show

$$(2) \quad \underset{i}{\exists}\, \underset{j}{\exists}\, j \geq i \wedge j \in \mathbb{N} \wedge j \leq i+5 \wedge i \in \mathbb{N} \wedge 36 = j^2 \ .$$

Since $j := 6$ solves the equational part of (2) it suffices to show

$$(3) \quad \underset{i}{\exists}\, i \in \mathbb{N} \wedge 6 \geq i \wedge 6 \in \mathbb{N} \wedge 6 \leq 5+i \ .$$

Using available computation rules we evaluate (3):

$$(4) \quad \underset{i}{\exists}\, i \leq 6 \wedge i \geq 1 \wedge i \in \mathbb{N} \ .$$

Formula (4), using (A), is implied by:

$$(5) \quad 6 > 1 \ .$$

Using available computation rules we evaluate (5):

$$(6) \quad \mathit{True} \ .$$

The derivations of formulae (1) and (2) result from applying STP inference rules for membership in a union and membership in a set abstraction, respectively. Reduction of (2) to (3) is accomplished by instantiating j by a solution of a quadratic equation done in STS. Simplifications from (3) to (4) and from (5) to (6) were made using available semantic knowledge by STC ($6 \in \mathbb{N}$ and $6 > 1$, respectively) and, finally, reduction from (4) to (5) and the detection of proof success were made by standard predicate logic inference rules.

References

[1] P. Bernays and A. Fraenkel. *Axiomatic Set Theory*. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 2 edition, 1968.

[2] B. Buchberger. Mathematics: An introduction to mathematics integrating the pure and algorithmic aspect. volume i: A logical basis for mathematics. Lecture notes for the mathematics course in the first and second semester at the Fachhochschule for Software Engineering in Hagenberg, Austria, 1996.

[3] B. Buchberger and D. Vasaru. The Theorema PCS Prover. Jahrestagung der DMV, Dresden, September 18-22, 2000.

[4] H. Ebbinghaus. *Einführung in die Mengenlehre*. Wissenschaftliche Buchgesellschaft Darmstadt, 2 edition, 1979. ISBN 3-534-06709-6.

[5] H. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. BI Wissenschaftsverlag Mannheim/Leipzig/Wien/Zürich, 3 edition, 1992. ISBN 3-411-15603-1.

[6] J. Gallier. *Logic for Computer Science*. Foundations of Automated Theorem Proving. Harper & Row, 1986.

[7] B. Konev and T. Jebelean. Combining Level-Saturation Strategies and Meta-Variables for Predicate Logic Proving in Theorema. In *Proceedings of IMACS ACA 2000, St.Petersburg, Russia*, June 2000.

[8] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Book Company, 1974. ISBN 0-07-039910-7.

[9] W. Quine. *Set Theory and its Logic*. Belknap Press of Harvard University Press, Cambridge, Massachusetts, 1963.

[10] B. Russell and A. Whitehead. *Principia Mathematica*. Cambridge University Press, 1910. Reprinted 1980.

[11] G. Takeuti and W. Zaring. *Introduction to Axiomatic Set Theory*. Graduate Texts in Mathematics 1. Springer Verlag, 1971. ISBN 0-387-05302-6.

[12] E. Tomuta. *An Architecture for Combining Provers and its Applications in the Theorema System*. PhD thesis, The Research Institute for Symbolic Computation, Johannes Kepler University, 1998. RISC report 98-14.

[13] D. Vasaru-Dupré. *Automated Theorem Proving by Integrating Proving, Solving and Computing*. PhD thesis, RISC Institute, May 2000. RISC report 00-19.

[14] W. Windsteiger. *A Set Theory Prover in Theorema: Implementation and Practical Applications*[6]. PhD thesis, RISC Institute, May 2001.

[6] http://www.risc.uni-linz.ac.at/people/wwindste/Public/Reports/PhdThesis