Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Symbolic Computation 41 (2006) 435–470
Journal of Symbolic Computation
www.elsevier.com/locate/jsc

An automated prover for Zermelo–Fraenkel set theory in *Theorema*[☆]

Wolfgang Windsteiger*

RISC Institute, A-4232 Hagenberg, Austria

Received 19 February 2003; accepted 5 April 2005
Available online 28 October 2005

Abstract

This paper presents some fundamental aspects of the design and the implementation of an automated prover for Zermelo–Fraenkel set theory within the *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.

Keywords: Automated theorem proving; Set theory; Theorema

1. Introduction

The set theory prover in *Theorema* adapts the "**P**rove–**C**ompute–**S**olve" (for short: PCS) proving strategy for proofs containing language constructs from set theory. The PCS paradigm was introduced originally in Buchberger (2001) and it has already been applied successfully for proofs in elementary analysis in Vasaru-Dupré (2000). The main strategy in a PCS-oriented prover is to structure the proof generation into phases of

- proving (P), i.e. using inference rules for propositional connectives, the standard quantifiers from predicate logic, and for *theory-specific language constructs*,
- computing (C), i.e. rewriting w.r.t. formulae in the knowledge base,

☆ This work has been supported by "SFB Numerical and Symbolic Scientific Computing" (F013) at the University of Linz and the European Union "CALCULEMUS Project" (HPRN–CT–2000–00102).
* Tel.: +43 732 2468 9960; fax: +43 732 2468 9930.
E-mail address: Wolfgang.Windsteiger@RISC.Uni-Linz.ac.at.

0747-7171/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jsc.2005.04.013

- solving (S), i.e. finding appropriate instances of existential variables occurring in the proof goal.

In general, of course, unification serves as the key method for solving formulae in arbitrary theories. For special theories, however, specialized solving techniques may be required and special techniques for solving from computer algebra may be applied. In particular, for solving formulae involving polynomial expressions there are powerful computer algebra methods such as the Gröbner bases method for systems of algebraic equations, see Buchberger (1985), or Collins' cylindrical algebraic decomposition method for systems of polynomial inequalities over the real closed fields, see Collins (1975). Having the computer algebra system Mathematica in the background of *Theorema*, we aim at applying these methods during the S-phase of our provers. In the context of set theory, solving can also lead to "solving for sets", i.e. finding sets that fulfill certain properties, but the set theory prover in its current state does not yet fully cover this aspect.

Many mathematicians are used to building up their theories in the frame of set theory, hence, computer support for doing proofs involving language constructs from set theory is a basic ingredient for computer-supported mathematics. The *Theorema* set theory prover aims to be primarily an educational tool that can support proving at an (under-)graduate university level in arbitrary theories that are built up in the frame of set theory. For this type of application it is important to incorporate also other proving techniques apart from sole set theory, notably arithmetic simplification in basic number domains or computational simplification involving finite sets. Moreover, since the *Theorema* syntax offers commonly used language constructs from set theory and the computational aspect of (finite) sets has always been supported in the *Theorema* computation environment, the set theory prover significantly enlarges the domain of applications for the *Theorema* system, because with this prover the *Theorema* system now offers *integrated support for both proving and computing* using set theory. It is important to mention that this prover is *not* intended as an automated prover for *proving set theory itself* based on only the axioms. Rather, it should be a tool that supports automated generation of "elegant proofs" in arbitrary mathematical theories that involve set theory.

Following the philosophy of most of the *Theorema* provers, the set theory prover aims at generating automated proofs in human-like natural style. In our experience, for mathematicians the acceptance of machine-generated proofs depends heavily on the readability of the proof for a human. In the automated theorem proving community, however, this aspect has not played a major role for a long time. Of course, as long as one does not display the proof, one can expand set-theoretic language constructs into first-order predicate logic and then apply powerful first-order theorem provers, like Otter, Vampire, or SPASS. The *Theorema* set theory prover, on the other hand, implements proof strategies applied by humans in an attempt to generate machine-proofs in a style acceptable by a human. Among other things, this will have considerable impact on computer-aided math education, which we currently see as one of the application areas for the *Theorema* system. It is attractive for the teacher to compose material in a mathematical language that is at the same time suitable for rigorous formal proofs and for execution of mathematical algorithms. It is attractive for the students to be able to perform computations immediately without translating mathematics into a programming language in order to execute the algorithms and to have certain proofs generated automatically in their lecture notes being able to do their own experiments. Of course, both the didactical potential and the dangers of such systems at certain levels of maths education need to be studied separately.

The current design of provers in the *Theorema* system requires a so-called "user prover" to be composed from "special provers", see Tomuta (1998). 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. The P–C–S structure of the set theory prover is reflected in the composition of the set theory user prover from several special provers implementing the 'P', 'C', and 'S' phase, respectively. It consists of a *set theory proving unit* handling set-theory-related connectives and quantifiers in the goal or in the knowledge base, a *set theory computing unit* responsible for rewriting and simplification, and a *set theory solving unit* capable of instantiating existential goals resulting from unfolding definitions for set operations. In addition to these set theory specific components, the set theory prover re-uses several special provers already available in the *Theorema* system.

The description is structured as follows: Section 2 describes the theoretical basis upon which the set theory prover is built, Section 3 explains the interplay between user prover and special provers and gives an overview of the theorem proving procedure used in the *Theorema* system, Section 5 introduces the set theory proving units `STP` and `STKBR`, Section 6 describes the set theory computing unit `STC`, Section 7 presents the set theory solving unit `STS`, and finally we conclude with some examples of proofs generated by the set theory prover in Section 8.

2. The theoretical basis of the set theory prover

2.1. Set theory in the Theorema system

The use of "set theory" in the *Theorema* system is not tied to one particular axiomatization of set theory. Instead, a syntax for "sets" is introduced on the level of the "*Theorema* expression language", we refer to Windsteiger (2001a) for a detailed description of the language layers in the *Theorema* system. The language supports sets by providing *the braces* '{' and '}' as a flexible arity matchfix function symbol used for constructing finite sets, *the set quantifier* as a means for describing sets by a characteristic property, and several other language constructs commonly used in mathematics, such as e.g. '$\subseteq$', '$\cup$', '$\cap$', or '$\setminus$', see Kriftner (1998) for an overview on supported set syntax. By this, expressions such as $\{a, b, c\}$, $\{x \underset{x}{\mid} P_x\}$, $\{T_x \underset{x}{\mid} P_x\}$, $A \subseteq B$, $A \cup B$, $A \cap B$, or $A \setminus B$ are *syntactically valid expressions* in the *Theorema* language. In other words, the syntax of *Theorema* allows so-called "naive set theory", see Halmos (1960). However, the *Theorema* language does not fix a semantics for all set expressions supported in its syntax.

Semantics is attached to expressions in *Theorema* on the "inference rule level". The meaning of expressions is defined by inference rules that describe *how* certain operations on expressions can be performed. As an example, an inference rule for set expressions tells that in order to prove $x \in A \cap B$ we need to prove both $x \in A$ and $x \in B$. These inference rules are the elementary building blocks for provers within the *Theorema* system, see Section 3 for the details. The *intended semantics* of *Theorema* expressions is again that of naive set theory, i.e. $\{1, 4, 7\}$ is the proposed syntax for "the set containing exactly the elements 1, 4, and 7", $\{x \underset{x}{\mid} x < 10\}$ is meant to denote "the set of *all* x satisfying $x < 10$", $\{x^2 \underset{x}{\mid} x < 10\}$ is intended to mean "the set of *all* x^2, when x satisfies $x < 10$", $A \cap B$ should stand for "the intersection of A and B" and the like.

The *Theorema* language construct that deserves closer inspection in this context is the so-called *set quantifier*, which can appear in two variants $\{x \underset{x}{\mid} P_x\}$ and more generally $\{T_x \underset{x}{\mid} P_x\}$. In its first form, the set quantifier allows us to define a set from a "characteristic property" P_x. In the literature, this is often addressed as *set comprehension* or as *the abstraction* of a set

from a property and it goes back to G. Cantor, the founder of modern set theory. Naive set theory allows *unrestricted abstraction*, i.e. *for every* formula P_x the expression $\{x \mid_x P_x\}$ denotes the set of all x satisfying P_x, in combination with an *intuitive notion of membership*, namely $x \in \{x \mid_x P_x\} \Longleftrightarrow P_x$. Although intuitively "reasonable", this is the main drawback of naive set theory, since this is the main source for contradictions derivable in naive set theory such as the well known Russell paradox: We define $R := \{x \mid_x x \notin x\}$ and by straightforward reasoning on "membership" in the above mentioned intuitive sense we can quickly derive the contradiction $R \in R \Leftrightarrow R \notin R$.

Since naive set theory is inconsistent, it is not suitable as a theoretical basis for the inference rules used in our set theory prover. Hence, some *axiomatic set theory* must serve as the underlying theory for our prover. In axiomatic set theory the existence of certain sets and appropriate membership rules are introduced via axioms. There are different axiomatizations of set theory that provide fundamentally different solutions how to avoid Russell's paradox (and others):

- Zermelo–Fraenkel set theory (ZF) restricts abstraction to what is called *separation*. Roughly, it requires the structure $x \in S \wedge Q$ for P_x in an abstraction $\{x \mid_x P_x\}$, which disallows constructions like R. We refer to Ebbinghaus (1979) and Shoenfield (1967) for detailed treatments of ZF.
- Von-Neumann–Gödel–Bernays' axiomatization (NGB) of set theory, see e.g. Bernays and Fraenkel (1968) and Quine (1963), distinguishes between sets and classes and allows the membership predicate only for sets. Russell's paradox is avoided by showing that R is not a set an therefore $R \in R$ is not a well-formed assertion.
- Russell himself introduced type theory, where membership is only allowed for sets of different type, see Russell and Whitehead (1910). $R \in R$ is not allowed on the grounds that R and R are not of different type.

2.2. Possible approaches for set theory proving in the frame of Theorema

Due to the inconsistency of naive set theory there *cannot be a prover* that supports the *entire language for set theory* offered in the *Theorema* syntax and, at the same time, follows the *intended semantics* of expressions in the *Theorema* language as described above. Similar to the different approaches to axiomatization, there are different choices for how to integrate set theory proving into the *Theorema* system:

- We restrict the language, for which the prover is applicable instead of trying to support the entire language available for set theory in *Theorema*.
- We adapt the semantics of membership and deviate slightly from the intuitive semantics of well-known language constructs.
- We introduce a *typing concept* into the *Theorema* system and obey the types in all set theoretic language constructs either immediately on the syntax-level or on the inference-rule level. This would, however, require a fundamental re-design of the entire system and we decided not to follow this path for the moment.

It seemed most attractive to go for the first variant, thus we decided to choose ZF as the theoretical frame for the *Theorema* set theory prover. In particular, this choice was also motivated by the fact that evidently ZF is very popular among mathematicians and the principal target users of our

prover are mathematicians who want to formulate some mathematical theory using the language of set theory. In other words, the *Theorema* set theory prover does not support all of what the *Theorema* language syntax offers for set theory; it supports just that fragment of the *Theorema* language, which can safely be used according to the axioms of ZF as described in Section 2.3.

2.3. Zermelo–Fraenkel set theory as used in the set theory prover

ZF is an axiom system that guarantees the existence of certain sets. Based on these axioms, several new functions and predicates useful for set theory can then be introduced by explicit definitions. In the following, we will list those axioms and definitions from ZF, on which the inference rules of the set theory prover rely. At the same time, this section will describe exactly the fragment of the *Theorema* language that is actually supported by the set theory prover. Furthermore, we introduce some convenient abbreviations for commonly used formulations in set theory that are compatible with our prover. The *Theorema* set theory prover should, thus, be a useful tool for mathematicians embedding their work in some variant of ZF set theory that is consistent with these axioms, definitions, and abbreviations.

As already indicated above, the main challenge in an axiomatization of set theory is the definition of membership in a set described by the set quantifier. We now give the axioms of ZF forming the basis for those inference rules in the set theory prover that are applied for membership in expressions involving the set quantifier.

Axioms 2.1 (*Separation Axioms*). For every formula[1] P_x and every S, s.t. x is not contained in S and S is not contained in P_x, we have an axiom

$$\underset{z}{\exists}\,\underset{x}{\forall}\; x \in z \Longleftrightarrow x \in S \wedge P_x.$$

In the literature, the separation axioms are sometimes referred to as "subset axioms". They allow us—for any formula P_x and any term S (fulfilling the side-conditions given in Axiom 2.1)—to define "the set containing all x of S such that P_x", see Shoenfield (1967, pp. 239). In the *Theorema* syntax this set can be denoted as $\{x \underset{x\in S}{|} P_x\}$ or $\{x \in S \mid P_x\}$. From Axiom 2.1 we get

$$\underset{x}{\forall}\,(x \in \{x \underset{x\in S}{|} P_x\} \Longleftrightarrow x \in S \wedge P_x). \tag{1}$$

Axioms 2.2 (*Replacement Axioms*). For every formula Q_x and every S, s.t. S is not contained in Q_x, we have an axiom

$$\underset{x}{\forall}\,\underset{z}{\exists}\,\underset{y}{\forall}\,(y \in z \Leftrightarrow Q_x) \Longrightarrow \underset{z}{\exists}\,\underset{y}{\forall}\,(\underset{x\in S}{\exists} Q_x \Rightarrow y \in z).$$

In common mathematical practice, some special instances of the replacement axioms play a crucial role, namely for Q_x and S s.t. S is not contained in Q_x and Q_x has the form $P_x \wedge y = T_x$ for some formula P_x and some term T_x. For these special cases the respective replacement axioms justify the definition of "the set of all T_x when $x \in S$ satisfying P_x". The *Theorema* syntax for this set is $\{T_x \underset{x\in S}{|} P_x\}$ and, as shown in detail in Shoenfield (1967, pp. 240), from Axiom 2.2 we get

$$\underset{y}{\forall}\,(y \in \{T_x \underset{x\in S}{|} P_x\} \Longleftrightarrow \underset{x\in S}{\exists} P_x \wedge y = T_x). \tag{2}$$

[1] P_x indicates that the variable x occurs *free* in P_x. The expression P_x may contain other free variables than x as well.

At first sight, the construct $\{x \underset{x\in S}{|} P_x\}$ appears to be just a special case of $\{T_x \underset{x\in S}{|} P_x\}$, just take $T_x = x$. However, both the separation axioms and the replacement axioms are needed for the existence proof of $\{T_x \underset{x\in S}{|} P_x\}$, see Shoenfield (1967), thus, the separation axioms cannot be omitted.

The formulae (1) and (2) now define membership for special variants—note the required property $x \in S$—of the *Theorema* set quantifier as it can safely be used in ZF. The inference rules for membership as used in our set theory prover are, thus, based on (1) and (2). Once having the set quantifier, elementary set theory can be built up by just explicit definitions. For "sets" using variants of the set quantifier different from those shown in (1) and (2) ZF provides additional axioms guaranteeing their existence, see e.g. Shoenfield (1967).

From now on, if not stated otherwise, we want to use P, Q, R, and C as typed variables on the meta-level to denote *formulae*, all other letters shall denote *terms*. Free variables in formulae or terms will be indicated by subscripts. As long as the existence of the sets $\{x \underset{x}{|} P_x\}$ and $\{T_x \underset{x}{|} P_x\}$ is guaranteed by some axiom, we generalize (1) and (2) as follows:

$$\underset{x}{\forall}\ (x \in \{x \underset{x}{|} P_x\} \Longleftrightarrow P_x) \tag{3}$$

$$\underset{y}{\forall}\ (y \in \{T_x \underset{x}{|} P_x\} \Longleftrightarrow \underset{x}{\exists}\ P_x \wedge y = T_x). \tag{4}$$

Note that by this generalization we are now back at the naive set theory notion of membership for sets whose existence is guaranteed by ZF. Membership as in (4) is supported even in the more general case of a multiple range that binds more than one variable simultaneously. The multiple range in the set quantifier translates literally to the respective multiple range in the existential quantifier, i.e.

$$\underset{y}{\forall}\ (y \in \{T_{x_1,\ldots,x_n} \underset{x_1,\ldots,x_n}{|} P_{x_1,\ldots,x_n}\} \Longleftrightarrow \underset{x_1,\ldots,x_n}{\exists}\ P_{x_1,\ldots,x_n} \wedge y = T_{x_1,\ldots,x_n}).$$

It is convenient to allow also an additional condition in the set quantifier. We follow the convention to use for arbitrary range $\mathbf{x}$

$$\{\ldots \underset{\substack{\mathbf{x}\\ C}}{|} P\} \tag{5}$$

as an abbreviation for

$$\{\ldots \underset{\mathbf{x}}{|} C \wedge P\}. \tag{6}$$

Note, however, that we do not generalize the inference rules in the set theory prover to cover set quantifiers with conditions, we rather convert any expression of the form (5) in the goal or in the knowledge base into the corresponding form (6), before formulae are actually passed to the prover.

Definition 2.1 (*Subset, Set Equality*).

$$S^{(1)} \subseteq S^{(2)} :\Longleftrightarrow \underset{x}{\forall}\ (x \in S^{(1)} \Rightarrow x \in S^{(2)}) \tag{7}$$

$$S^{(1)} = S^{(2)} :\Longleftrightarrow \underset{x}{\forall}\ (x \in S^{(1)} \Leftrightarrow x \in S^{(2)}). \tag{8}$$

Definition 2.2 (*Empty Set, Set Difference*).

$$\emptyset := \{x \mid_x x \neq x\} \tag{9}$$

$$S^{(1)} \setminus S^{(2)} := \{x \mid_x x \in S^{(1)} \wedge x \notin S^{(2)}\}. \tag{10}$$

Definition 2.3 (*Finite Set Construction*). For any $n \geq 1$:

$$\{S^{(1)}, \ldots, S^{(n)}\} := \{x \mid_x x = S^{(1)} \vee \ldots \vee x = S^{(n)}\}. \tag{11}$$

Definition 2.4 (*Union, Intersection, Product*). For any $n \geq 2$:

$$S^{(1)} \cup \ldots \cup S^{(n)} := \{x \mid_x x \in S^{(1)} \vee \ldots \vee x \in S^{(n)}\} \tag{12}$$

$$S^{(1)} \cap \ldots \cap S^{(n)} := \{x \mid_x x \in S^{(1)} \wedge \ldots \wedge x \in S^{(n)}\} \tag{13}$$

$$S^{(1)} \times \ldots \times S^{(n)} := \{\langle x_1, \ldots, x_n\rangle \mid_{x_1,\ldots,x_n} x_1 \in S^{(1)} \wedge \ldots \wedge x_n \in S^{(n)}\}. \tag{14}$$

The notion $\langle \ldots \rangle$ is used for finite tuples provided as basic data type in *Theorema*. We do not model tuples *within* set theory but we use built-in knowledge about tuples provided by the semantics of the *Theorema* language.

Definition 2.5 (*Union, Intersection, Power Set*).

$$\bigcup S := \{x \mid_x \exists_{s\in S} x \in s\} \tag{15}$$

$$\bigcap S := \{x \mid_x \forall_{s\in S} x \in s\} \tag{16}$$

$$\mathcal{P}[S] := \{x \mid_x x \subseteq S\}. \tag{17}$$

Frequently used combinations of $\bigcup$ and $\bigcap$ with the set quantifier can conveniently be abbreviated when introducing $\bigcup$ and $\bigcap$ as quantifiers.

$$\bigcup_{\substack{x\in I\\ C_x}} S_x \quad \text{abbreviates} \quad \bigcup\{S_x \mid_{x\in I} C_x\} \tag{18}$$

$$\bigcap_{\substack{x\in I\\ C_x}} S_x \quad \text{abbreviates} \quad \bigcap\{S_x \mid_{x\in I} C_x\}. \tag{19}$$

When using the *Theorema* set theory prover one accepts the above definitions and assumes an underlying axiomatic system such as ZF that guarantees the existence of all these sets. We do not invent a new set theory that promises to be better suited for automated theorem proving, an approach that is taken elsewhere, e.g. in Formisano (2000).

3. How provers are organized in *Theorema*

3.1. Preliminaries on terminology

We will use the following terminology: a *proof situation* $K \vdash G$ is made up from a knowledge base of assumptions K and a goal G, and it should be understood as an abbreviation for the

phrase: "We have to prove G from K". Typically, the goal will be a single formula of the *Theorema* language, whereas the knowledge base consists of a collection of formulae, called the assumptions.

The task of the *special provers* is essentially the execution of individual *proof steps* that *reduce* the proof situation towards *terminal proof situations*, from which proof success or failure can easily read off. Terminal proof situations will be denoted by just their "value", e.g. 'proved' or 'failed'. The rules applied by the special provers guiding the reduction of proof situations are called *inference rules*. Thus, an inference rule turns a proof situation $K \vdash G$ into a proof situation $K' \vdash G'$ with a new goal G' and a new knowledge base K'. In the description of inference rules, we will denote an inference rule named 'I' transforming $K \vdash G$ into $K' \vdash G'$ by

$$\mathbf{I}: \frac{K' \vdash G'}{K \vdash G}$$

(read as: "The rule 'I' justifies a proof step to reduce the proof of G from K to a proof of G' from K'"). This notation is similar to notations used in logic for describing inference rules in formal proof calculi (e.g. the natural deduction calculus or the Gentzen calculus). 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.

We give an example of a well-known inference rule from the natural deduction calculus for predicate logic written in this style:

$$\textbf{ArbitraryButFixed}: \frac{K \vdash P_{x \to x_0}}{K \vdash \underset{x}{\forall} P_x} \quad \text{(where } x_0 \text{ is a new constant).}$$

The rule 'ArbitraryButFixed' tells us that, in order to prove $\underset{x}{\forall} P_x$ (from K) it suffices to prove $P_{x \to x_0}$ (from K) for a new constant x_0, where $P_{x \to x_0}$ stands for "P with each free occurrence of x substituted by x_0".

3.2. The generation of proofs in Theorema

The automated generation of proofs in the *Theorema* system is based on three main components: the user prover, the special provers, and the global proof search procedure.

3.2.1. User provers

The *Theorema* user interface provides the command

$$\texttt{Prove}[G,\ \text{using} \to K,\ \text{by} \to M],$$

which initiates an attempt to prove the goal G using the knowledge base K by the method M. In the *Theorema* terminology, we call the available prove-methods *user provers*. A user prover is a program that sets up a particular two-row configuration grid (see Fig. 1) of special provers and then passes control to *Theorema*'s *global proof search procedure*.

3.2.2. Special provers

A *special prover* is a sequential collection of inference rules. In Fig. 1 the sequential structure of a special prover is visualized by a top-to-bottom line-up of the inference rules in each of

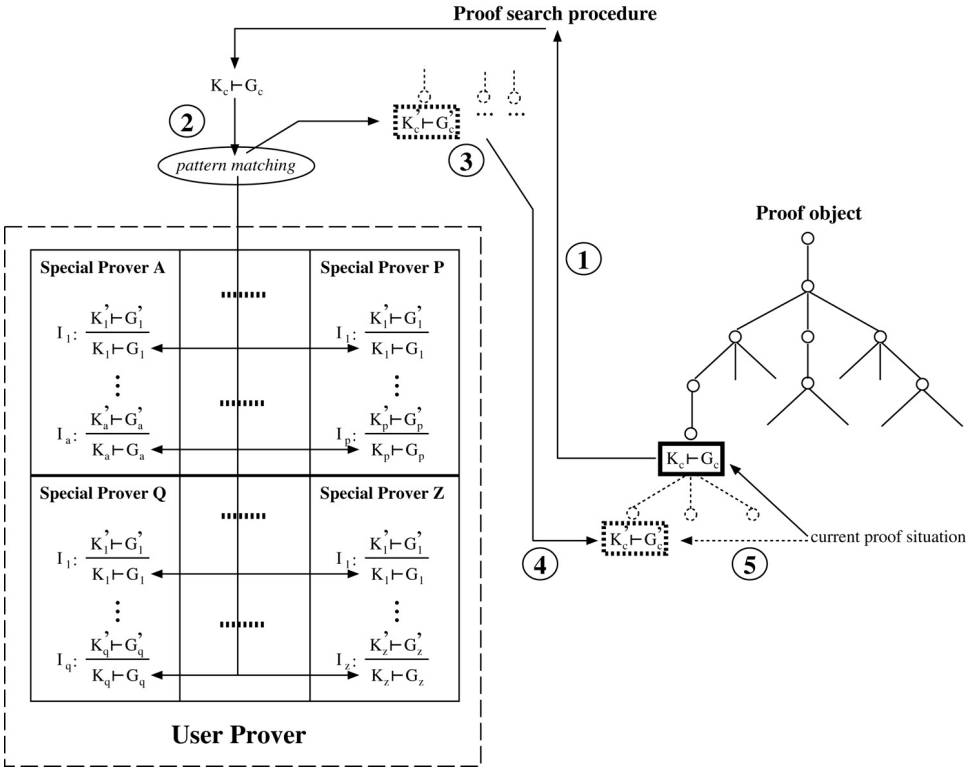

Fig. 1. Proof search and prover composition.

the special provers. The arrangement of the special provers in the user prover's configuration grid results essentially in a *hierarchical collection of inference rules* structured by the placement within the two-dimensional grid and, on a finer level, by the sequential arrangement within each cell in the grid.

3.2.3. The global proof search procedure and the proof object

The proof search procedure uses inference rules as arranged in the user prover's configuration grid in order to manipulate the *global proof object*. The proof object has a tree structure with each node containing one proof situation. The proof search procedure maintains a *current proof situation*, which specifies the node that is to be manipulated next. Initially, the proof object consists of only the root node containing the initial proof situation $K \vdash G$ given by the user in the `Prove`-command. Each proof tree manipulation is the augmentation of the proof object at the current proof situation by one or more new nodes, whose contents depend on the inference rules found by the proof search procedure in the special provers. Fig. 1 visualizes the main phases of one proof step:

1. The proof search procedure extracts the current proof situation $K_c \vdash G_c$ from the proof object.
2. Mathematica's pattern matching mechanism is used to select appropriate inference rules that allow to reduce the current proof situation. Inference rules are implemented as Mathematica programs taking goal, knowledge base, and "additional facts" of the current proof situation as input. We refer to Section 5 for more details on the role of the "additional facts" in a

proof situation. As output, an inference rule returns a new node to be inserted into the proof tree. In this phase, the configuration grid in the user prover set-up is crucial: The special provers in the first row are tried left to right, in each prover the rules are tested top to bottom. The *first rule* whose goal and assumptions match the current goal G_c and the current knowledge base K_c will be selected. If none of the special provers in the first row applies, the special provers in the second row are tried again left to right. In each special prover the rules are tried again top to bottom, and *from each applicable prover*, the first rule matching the current proof situation will also be selected. Note that the pattern language of Mathematica contains conditionals. Thus, the selection of inference rules based on Mathematica's pattern matching is not restricted to purely syntactical matches but it allows also the test of certain conditions.

3. All inference rules selected in the previous phase will be applied in this proof step to the current proof situation resulting in new nodes to be inserted into the proof object.
4. If there is more than one new node, each node is assigned a new branch in the proof object. Branches reflect *alternative proof attempts* in a proof.
5. Finally, the current proof situation is stepped to the new node on the leftmost new branch.
6. These steps are iterated until a terminal proof situation 'proved' or the search depth limit is reached. If a proof fails on one branch by reaching either a terminal proof situation 'failed' or the maximal search depth then the proof search continues on the next branch. Once all branches have failed, the entire proof has failed.

For details on the organization of the proof search within *Theorema* we refer to Tomuta (1998).

The implementation of the user prover arranges the special provers in the grid and the implementations of the special provers arrange the inference rules within the special provers. Thus, the experience of the prover programmer is reflected in a smart set-up of the user prover and the special provers. The proof search as described above is completely automated with no possibility for user-interaction. As an alternative, the *Theorema* system offers also an interactive proof search mechanism, see Piroi (2004).

4. The *Theorema* set theory prover

As discussed in Section 3.2 the implementation of a prover for set theory must consist of a user prover and several special provers. In the following, 'set theory prover' will refer to the user prover for set theory, which combines newly developed special provers for set theory with previously developed general purpose special provers available in the *Theorema* system, such as `TerminalND` for detecting terminal proof situations, `BasicND` and `PND` for basic and general predicate logic reasoning, `QR` for rewriting w.r.t. quantified equalities, equivalences, or implications in the knowledge base, or `CDP` for treatment of case distinctions, see Buchberger and Vasaru (2000), Vasaru-Dupré (2000) and Windsteiger (2001a). We will describe the four new special provers that have been developed for set theory.

`STP` collects inference rules with some set-theoretic symbol as the outermost symbol in the proof goal.

`STKBR` expands set-theoretic notions in the assumptions.

`STC` performs simplification by computation on finite sets.

`STS` applies special techniques for instantiation of existential formulae in the proof goal, which are useful in the context of set theory.

The set theory prover arranges `TerminalND`, `STKBR`, `STC`, `STP`, and `STS` in this order from left to right in the first row of the configuration grid and `BasicND`, `QR`, `CDP`, and `PND` in the second row. Roughly, this results in a heuristic to generally first expand set-theoretic language constructs in the knowledge base when they appear as outermost symbols before working on the proof goal. When reducing the proof goal, simplifications based on computational knowledge for finite sets are applied before expanding set-theoretic language constructs by their definition. If non-set-theory symbols appear as outermost symbols, proceed by the usual predicate logic reasoning and rewriting. The set theory prover can be used in interactive proving like all other provers in the *Theorema* system. The emphasis of this work is, however, to set up the prover for completely automated proof generation.

5. 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 Section 1. Inference rules for set theory specific *proving* are provided in 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. The set theory prover arranges both special provers in the first row of the configuration grid.

5.1. Set theory specific goal reduction

Set theory specific goal reduction is implemented as a special prover named `STP` (for Set Theory Proving). The inference rules within `STP` differ mainly in the syntactic patterns for the proof situation. A few inference rules are influenced in addition by global variables, by which, for instance, certain inference rules can be deactivated. Some strategies depend on the proof progress stored in `STP`'s *local proof context*, which is part of the "additional facts"-parameter in the implementation of inference rules, see Section 3.

The inference rules are grouped into rules for *membership*, rules for *inclusion*, and rules for *set equality*. The rules for membership cover proof situations, where the outermost symbol in the proof goal is '$\in$'. There is at least one inference rule for each "kind of set" introduced in Section 2, in some cases we provide specialized rules in order to offer special treatment for special cases. We show some of the membership rules as they are used in `STP`.

$$\textbf{MembershipSeparation}: \frac{K \quad \vdash \quad t \in S \wedge P_{x \to t}}{K \quad \vdash \quad t \in \{x \underset{x \in S}{|} P_x\}}$$

The inference rule 'MembershipSeparation' is just a reformulation of variant (1) of the separation Axioms 2.1. Hence, its correctness is an immediate consequence of (1) and we do not give a separate correctness proof for this rule. In fact, most of the rules in `STP` are just direct translations of one of the axioms or one of the definitions listed in Section 2. For these cases we do not give the correctness proofs of the inference rules used in our prover. Some of the inference rules, however, condense several inference steps into one compact rule to be applied. In these cases, we provide hand-proofs for the correctness of the respective rules. An example of such a rule is the elimination of the union-quantifier in the goal. Simply using abbreviation (18) would lead to an inference rule

$$\textbf{Membership-U}: \frac{K \vdash t \in \bigcup\{S_x \mid_{x\in s} C_x\}}{K \vdash t \in \bigcup\limits_{\substack{x\in s\\ C_x}} S_x}$$

The STP prover, however, implements the rule

$$\textbf{MembershipUnionOf}: \frac{K \vdash \exists\limits_{x\in s}(t \in S_x \wedge C_x)}{K \vdash t \in \bigcup\limits_{\substack{x\in s\\ C_x}} S_x}$$

'MembershipUnionOf' reduces the proof of $t \in \bigcup\limits_{\substack{x\in s\\ C_x}} S_x$ to prove $\exists\limits_{x\in s}(t \in S_x \wedge C_x)$.

Proof. Assume $\exists\limits_{x\in s}(t \in S_x \wedge C_x)$, thus $t \in S_{x_0} \wedge C_{x_0}$ for some constant $x_0 \in s$. With $z := S_{x_0}$ we can infer from this $t \in z \wedge C_{x_0} \wedge z = S_{x_0}$, hence

$$\exists_z(\exists_{x\in s} t \in z \wedge C_x \wedge z = S_x). \tag{20}$$

Separating the quantifiers in (20) gives $\exists_z(t \in z \wedge \exists\limits_{x\in s}(C_x \wedge z = S_x))$, which, by (2), is equivalent to $\exists_z(t \in z \wedge z \in \{S_x \mid_{x\in s} C_x\})$. By (15) this is equivalent to $t \in \bigcup\{S_x \mid_{x\in s} C_x\}$, thus $t \in \bigcup\limits_{\substack{x\in s\\ C_x}} S_x$ by (18). □

As special rules for membership in finite sets and finite unions, respectively, we provide for $n \geq 2$

$$\textbf{MembershipFinite}: \frac{t \neq S^{(2)}, \ldots, t \neq S^{(n)}, K \vdash t = S^{(1)}}{K \vdash t \in \{S^{(1)}, S^{(2)}, \ldots, S^{(n)}\}}$$

$$\textbf{MembershipUnion}: \frac{t \notin S^{(2)}, \ldots, t \notin S^{(n)}, K \vdash t \in S^{(1)}}{K \vdash t \in \bigcup\{S^{(1)}, S^{(2)}, \ldots, S^{(n)}\}}$$

Again, we omit the easy proofs of correctness. Note, that both membership in finite sets and finite unions would reduce by definition to a disjunction of formulae. The majority of human mathematicians would proceed by a smart choice of one of the alternatives and then just prove the chosen alternative. The inference rules given above, however, reduce the proof of a disjunction further to the proof of always the first alternative while assuming the negations of the remaining alternatives. This has the advantage that the rule keeps the proof search space small because it does not introduce additional branches into the proof object. On the other hand, the resulting proofs appear "unnatural" for human mathematicians at the point where these rules are applied. Thus, these rules might be adapted in future versions of the prover.

As we will see in Section 6, these rules will not be applied in the standard set-up of the set theory prover, because as soon as the set theory computation unit is present, membership in finite sets and finite unions will be decided based on computational semantics available in the *Theorema* language. The two rules are contained in STP only because, in the spirit of modular system design, we do not presuppose that all future *Theorema* user provers combine the available

special provers for set theory in exactly the way in which they are combined in our set theory prover now.

Furthermore, we provide one general inference rule for set inclusion and set equality, respectively, which reduce inclusion and equality to membership according to Definition 2.1. Additionally, we provide special rules for special cases in order to reduce the search depth in the proof search procedure, like e.g.

$$\textbf{ConjunctionSubset}: \frac{\text{proved}}{K \vdash \{x \mid_{x} \ldots \wedge x \in S \wedge \ldots\} \subseteq S}$$

$$\textbf{SubsetSeparation}: \frac{P_{x\to x_0}, x_0 \in X, K \vdash x_0 \in Y \wedge Q_{y\to x_0}}{K \vdash \{x \mid_{x\in X} P_x\} \subseteq \{y \mid_{y\in Y} Q_y\}}$$

(where x_0 is some new constant).

For the empty set, the expansion of Definition 2.2 would result in "unnatural" proof steps, hence, we provide special rules for the empty set, like e.g.

$$\textbf{EmptySetSubset}: \frac{\text{proved}}{K \vdash \emptyset \subseteq S}$$

$$\textbf{EqualsEmptySet}: \frac{K \vdash \neg P_{x\to x_0}}{K \vdash \{T_x \mid_{x} P_x\} = \emptyset}$$

(where x_0 is some new constant). The proofs for these rules are again straightforward. For more details and a *complete listing of all inference rules* used in STP we refer to Windsteiger (2001a).

5.2. *Set theory specific knowledge expansion*

5.2.1. *Knowledge expansion by lazy level saturation*

The special prover STKBR (for Set Theory Knowledge Base Rewriting) uses "lazy saturation" in order to infer *new knowledge* from formulae already contained in the knowledge base using knowledge about set theory specific language constructs. In contrast to classical level saturation methods, which try to obtain *all formulae* that can be inferred from the knowledge base in one saturation run, "lazy saturation" is somewhat more moderate in that it only finds formulae that can be inferred from the original knowledge base at the beginning of the saturation run. This has the advantage that usually less "potentially unimportant" formulae are generated before the prover continues with some other steps. However, if no other proof steps can be performed, the proof search procedure will continue with subsequent lazy saturation steps. This type of "iterated lazy saturation" *can* ultimately lead to a completely saturated knowledge base, but unlike classical saturation techniques it *does not necessarily*.

The STKBR prover is implemented as just *one inference rule* implementing the mechanism of lazy saturation based on knowledge from set theory combined with simplification of assumptions based on computational semantics from the *Theorema* language. STKBR is considered to be applicable to the current proof situation as soon as new formulae occur in the knowledge base compared to its previous application. This check is done with the help of an entry in the local proof context passed among the "additional facts" of the current proof situation, see Section 3. Similar to STP, most of the set theory specific knowledge expansion rules used in this phase

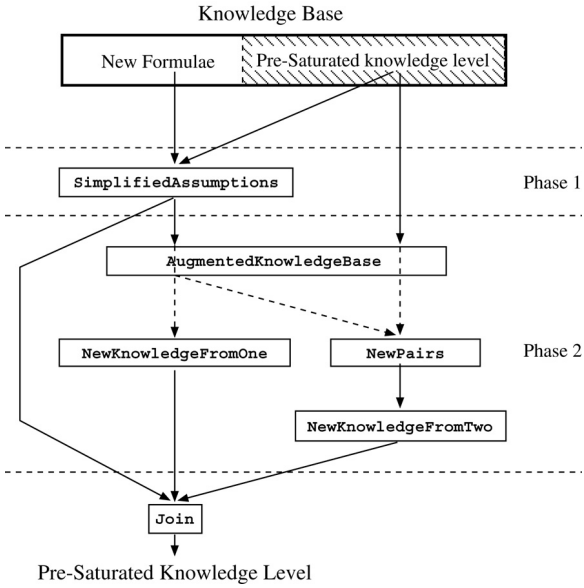

Fig. 2. Schematic flow of lazy level saturation as used in STKBR.

are just re-formulations of the axioms and the definitions in Section 2. Again they are grouped into rules for membership, inclusion, and set equality. For each "kind of set" we provide a membership rule for a proof situation, where '$\in$' appears as outermost symbol in one of the assumptions. Moreover, the prover contains rules for unfolding membership inside universally quantified formulae and, like STP, some special rules based on elementary set theory knowledge. We refer to the examples in Section 8.2.2 for more details on these special rules.

Knowledge expansion in STKBR happens in two phases:

1. New formulae are simplified using built-in semantic knowledge available in the *Theorema* language semantics.
2. New knowledge is *inferred* from the simplified knowledge base by lazy saturation as described above. The expansion rules used in lazy saturation are grouped into two groups:
   - *Group* I containing rules for inferring new knowledge from *one* known formula and
   - *Group* II containing rules for inferring new knowledge from *two* known formulae.

   Matching rules from Group I are applied to the simplified new formulae, matching rules from Group II are applied to all new pairs of formulae containing at least one new formula.

All formulae generated during these two phases are added to the knowledge base and, finally, the formula labels of formulae contained in the expanded knowledge base are stored in the local proof context in order to be accessible in the next saturation run. We call this stage a *pre-saturated knowledge level*. Complete level saturation would iterate this process until no more new formulae can be inferred. Lazy saturation, instead, passes control back to the proof search procedure after one iteration.

The schematized flow of STKBR's level saturation mechanism is shown in Fig. 2. Phase 1 is accomplished by applying the function 'SimplifiedAssumptions' to two arguments: the entire knowledge base and a list of labels 'sat' describing the pre-saturated knowledge level from a previous saturation run. Each new formula from the knowledge base is sent through a simplification function, which *computes* a simplified version of the formula w.r.t. semantic

knowledge from the *Theorema* language. In fact, the same simplification function is used that is applied also in the STC module for *goal simplification by computation* and in the top-level *Theorema*-user command Compute, see Section 6. This guarantees utmost coherence between all computations happening in different components of 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. Formulae that cannot be simplified as well as formulae from the previous saturation level leave phase 1 unchanged. Actually, STKBR contributes to both the P- and the C-phase, hence, it is not a pure proving unit! We allowed this mixture of P- and C-phase in *one* special prover in the current implementation merely for reasons of efficiency.

Phase 2 is covered in the implementation by the function 'AugmentedKnowledgeBase', which receives the simplified knowledge base resulting from phase 1 and again the list 'sat'. 'NewKnowledgeFromOne' applies Group I of expansion rules to *each (simplified) new assumption*, 'NewKnowledgeFromTwo' applies Group II of expansion rules to *all new pairs* that can be formed using at least one new assumption. The new formulae obtained in phase 2 joined with the simplified knowledge base resulting from phase 1 give the new pre-saturated knowledge level.

5.2.2. Rule locking

Rule locking is a mechanism that helps to prevent cycles in the proof search during level saturation. As an example, consider the two inference rules

$$\mathbf{I}: \frac{\ldots, x \in A, x \in B, \ldots \ \vdash \ G}{\ldots, x \in A \cap B, \ldots \ \vdash \ G} \qquad \mathbf{I'}: \frac{\ldots, x \in A \cap B, \ldots \ \vdash \ G}{\ldots, x \in A, \ldots, x \in B, \ldots \ \vdash \ G}$$

occurring in STKBR. We call two rules *inverse to each other* if one rule neutralizes the effect of the other. **I** and **I'** are an example of rules being inverse to each other. Our prover contains some pairs of inverse rules although, generally, we try to avoid to provide inverse rules wherever possible. Inverse rules need special attention because their unrestricted use immediately results in a cycle in the proof search. Rule locking allows us to dynamically disable certain inference rules for certain values of the input parameters. In the above example, the application of rule **I'** resulting in a new formula F will automatically prevent rule **I** from being applied to F in the remainder of this proof branch. Similarly, the application of rule **I** producing new formulae F_1 and F_2 will block rule **I'** on F_1 and F_2 in the remainder of this proof branch. Note, however, that affected rules are only locked for *particular values of the input parameters* whereas they stay applicable in all other situations.

In general, for each pair of inverse rules **I** and **I'** we implement both **I** and **I'** such that they lock their inverse for certain inputs. There is no general law, however, for which inputs a rule must be locked. As a special case, inference rules may even lock *themselves* in order to avoid "uninteresting" expansions in the proof search. Consider again the example from above: Applying rule **I'** once would add the new assumption $x \in A \cap B$. During the next saturation run, **I'** would add the new assumptions $x \in A \cap (A \cap B)$ and $x \in B \cap (A \cap B)$ and so forth. Thus, rule **I'** is implemented such that it locks both itself and also its inverse rule **I** on the new formulae generated by **I'**. Rule locking utilizes STKBR's local proof context to store this type of information on the proof progress.

6. 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 quantifiers introduced in Section 2, see also Buchberger (1996), i.e. set quantifiers with finite and computable range specifications, see Windsteiger (2001a). In particular, *integer ranges* and *set ranges with finite sets* are algorithmic ranges, which lead to finite sets when used in combination with the set quantifiers.

The *Theorema* language semantics allows the *explicit construction* of finite sets as an enumeration of the (finitely many) elements contained in the set, i.e. the language contains some data-structure representing a finite set. Set operations (such as union, intersection, power set, etc.) on finite sets are implemented as operations on the data-structure for finite sets in a constructive fashion, i.e. every operation on finite sets results again in a finite set. Tests for membership, inclusion, or set equality for finite sets, thus, reduces to *testing finitely many cases*, which is implemented in the frame of the *Theorema* language semantics as well.

Computation using built-in semantics knowledge is available in the *Theorema* system through the top-level user command `Compute`. A typical computation involving finite sets and numbers is

$$\texttt{Compute}[\{3x \mid_{x\in\{1,2,3,4\}} \text{is-prime}[x]\},\ \text{built-in}\rightarrow\langle\text{Built-in}[\text{“Sets”}],\ \text{Built-in}[\text{“Numbers”}]\rangle]$$

resulting in the finite set $\{6, 9\}$ and, of course,

$$\texttt{Compute}[6 \in \{3x \mid_{x\in\{1,2,3,4\}} \text{is-prime}[x]\},\ \text{built-in}\rightarrow\langle\text{Built-in}[\text{“Sets”}],\ \text{Built-in}[\text{“Numbers”}]\rangle]$$

results in *True*. Internally, `Compute` sends the expression to be computed to a simplification function, which simplifies the expression with respect to both *user-defined knowledge* given in the "using"-option and *built-in knowledge* from the *Theorema* language semantics given in the "built-in"-option to `Compute`. In the examples above, no user-defined knowledge is provided and built-in knowledge about "Sets" (for the set quantifier and the finite set in the range of the quantifier) and "Numbers" (for 'is-prime' and for the multiplication used in '$3x$' is applied. It is the intention of the `STC` (for `S`et `T`heory `C`omputing) special prover to integrate the computational power available for finite sets seamlessly into the *Theorema* proving machinery. Otherwise, all algorithmic knowledge about finite sets needs to be re-implemented inside the set theory prover, which would make it very difficult to guarantee consistent behavior in proving and computing. In order to avoid this duplication of code and knowledge and in order to achieve coherent simplifications on the top-level using `Compute` and on the proving-level in simplifications of both the proof goal and the knowledge base, the `STC` prover simplifies the goal by sending the goal formula to the same simplification 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 simplification function on the proof goal and, if the result differs from the original form, of adding a new node to the proof object, from which the effect and a complete trace of the computation can be displayed. In fact, the interface to the underlying simplification function is implemented in a more flexible fashion. Namely, it allows *arbitrary built-in knowledge* available in the *Theorema* language in addition to built-in knowledge about finite sets to be used during simplification. Similar to the `Compute`-examples above, the user may specify built-in knowledge in the call of any prover using the option "built-in". The set theory prover sets up the environment such that simplification uses set theory semantics by default and user-specified built-in semantics

in addition. Given a proof goal such as $6 \in \{3x \mid_{x\in\{1,2,3,4\}} \text{is-prime}[x]\}$ the set theory prover will produce different proof variants depending on the additional semantical knowledge provided by the user:

- Using *no semantic knowledge at all* by completely deactivating `STC`, the proof goal will be reduced by an inference rule from `STP` to prove

  $$\exists_{x\in\{1,2,3,4\}} \text{is-prime}[x] \wedge 6 = 3x,$$

  and, by predicate logic reasoning using *matching against formulae in the knowledge base*, this goal might eventually be proven since $x = 2$ is an appropriate choice for the existential variable. Note, however, that formulae such as 'is-prime[2]' and '$6 = 3 * 2$' must be provided *explicitly* in the knowledge base. Moreover, for the proof of subgoal $2 \in \{1, 2, 3, 4\}$, `STP` will apply the inference rule **MembershipFinite** described in Section 5.1 resulting in a very fine-grained proof showing all details down to the axioms of set theory.
- Using the standard setting with *only built-in sets*, the resulting proof is essentially the same, just that the subgoal $2 \in \{1, 2, 3, 4\}$ can be simplified to *True* by `STC` in one stroke. However, the set quantifier does not simplify to a finite set, since 'is-prime' is unknown and, thus, the expression stays unevaluated.
- Using *built-in semantics of "Sets" and semantics of 'is-prime'*, the proof goal will be simplified by `STC` to prove

  $$6 \in \{3 * 2, 3 * 3\},$$

  and again `STP`'s **MembershipFinite** will come into play. Still, $6 = 3 * 2$ must be available in the knowledge base in order to finish the proof.
- Using *built-in semantics of "Sets" and "Numbers"*, the proof goal will be simplified by `STC` in one step to *True* and the proof succeeds without any additional explicit knowledge.

Of course, each of these variants has its pros and cons and the *Theorema* user can decide which path to follow. We consider it of utmost importance for practical applications and acceptance of the system to offer this choice to the user. More details on combining computation with proving can be found in Windsteiger (2001a). Furthermore, we refer to Section 8.2 for more examples of interaction of proving and simplification by computing.

7. STS: The set theory solving unit

The special prover `STS` (for `Set Theory Solving`) collects inference rules for eliminating existential quantifiers in the proof goal. Its name suggests that this prover deals with set theory specific aspects of solving, but, since general predicate logic solving components in the *Theorema* system are not yet far advanced, the `STS` prover in its current state collects inference rules for existential goals as they result typically from proof goals containing language constructs from set theory. Set theory specific solving in the sense of "solving for sets" meaning "finding sets that fulfill certain properties" is not yet dealt with in this version of the prover. Rather, we concentrate on using the special structure of the existential formula in order to devise efficient instantiation methods for certain types of existential goals.

The inference rules in `STS` mainly cover proof situations of the form $K \vdash \exists_x P_x$, i.e. we have to *find some term* t such that $K \vdash P_{x\to t}$. In many cases the choice of t can be made essentially

on the basis of K. In these situations, the methods used for instantiating the existential goal are *matching* parts of P_x against ground formulae in K and *unification* of parts of P_x with formulae in K.

The more difficult cases, however, are those, where the choice of t depends strongly on the inner structure of P_x. In these situations, the instantiation of the existential goal requires, roughly speaking, some further processing of P_x before an appropriate choice of t is possible. Hence, we *introduce a solve-constant* in place of the existential variable. A solve-constant differs from a Skolem-constant in that it is a placeholder for the term t, whose exact "form" is not yet known at the time when the solve-constant is introduced, whereas a Skolem-constant is a new constant about which we do not know anything. For the proof to succeed, *all solve-constants* that have been introduced must be expressed through appropriate *ground terms* in such a manner that the resulting formula can be proven. The introduction of solve-constants, thus, allows us to delay the instantiation of existential goal formulae to a later phase of the proof in order to be able to proceed with standard reasoning techniques before actually instantiating the existential variable. What we call a solve-constant is often addressed as a *meta-variable* by other authors. The technique of meta-variables is well known and used also in other systems.

Essentially, solve-constants imitate what a human often does when proving $\exists_x P_x$, namely to "pretend to know x" and then reason on P_x in order to derive more knowledge on x until we can identify a t that fulfills all the requirements collected for x. Of course, the strategy after introducing solve-constants must always be to isolate the solve-constants, and then to apply special solving techniques depending on the nature of the remaining formulae. Hence, this strategy reduces proving to *solving over various domains*. Ideally, the formulae to be solved are algebraic equalities or inequalities such that known solution techniques available from computer algebra can be applied for finding an appropriate t. For this reason some of the inference rules in `STS` employ the Mathematica `Solve`-function in situations where an existentially quantified variable or a solve-constant appears inside an equality. For requirements formed by arbitrary set-theoretic formulae we plan to develop an appropriate *solving calculus* as future work.

We present only one typical inference rule from `STS`.

$$\textbf{IntroSolveConstant}: \frac{K \vdash Q_{y\to y^*} \wedge \exists_{x\in s} (P_x \wedge y^* = T_x) \wedge R_{y\to y^*}}{K \vdash \exists_y (Q_y \wedge y \in \{T_x \mid_{x\in s} P_x\} \wedge R_y)}$$

where Q_y and R_y are possibly empty conjunctions of formulae and y^* is a solve-constant. 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 special unions, namely goals of the form $m \in \bigcup\{T_x \mid_{x\in s} P_x\}$. It can be observed in many examples involving proof goals of this form (see in particular the example in Section 8.2.4) that this strategy leads to a well-structured proof. The rule eliminates the outermost existential quantifier by introducing a solve-constant and it introduces another existential quantifier by immediately expanding membership. `STS` contains further rules, which allow the elimination of the remaining existential quantifier in this particular case and even in other more general situations, see Windsteiger (2001a). Note, that the solve-constant already appears in an isolated position, so that it can immediately be expressed by the ground term T_x as soon as P_x has been solved for x. 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. We refer also to the discussion in Section 8.2.4,

Table 1
Comparison to systems on examples from CASC-18

Example	*Theorema*	E-SETHEO	Vampire	DCTP	Bliksem	Saturate
SET010	3.0	15.8	23.9	1.2	>300	?
SET014	3.2	>300	>300	281.0	>300	1.8
SET096	2.0	9.6	17.0	113.7	7.1	8.1
SET171	4.0	>300	>300	>300	>300	2.9
SET580	8.7	0.4	0.1	1.5	>300	1.7
SET612	2.1	>300	>300	>300	>300	9.9
SET624	43.7	0.7	0.8	1.7	>300	10.2
SET630	2.4	0.4	62.3	1.5	>300	116.8
SET716	6.8	>300	>300	>300	>300	8.8

where, in particular, the important role of *solving as a sub-problem in proving* is discussed in a concrete example.

8. Comparison and examples

8.1. Comparison to state-of-the-art theorem provers

In this section, we test the *Theorema* set theory prover on some examples from the SET section of TPTP, see TPTP: Thousands of Problems for Theorem Provers (n.d.). Timings refer to CPU seconds consumed on a 1500 MHz Intel P4 running Mathematica 4.2 and include the time needed for generating the proof, simplifying the successful proof, and displaying the formatted proof as shown in Section 8.2. Table 1 shows a comparison of the computing times to state-of-the-art theorem provers as they performed in CASC-18[2], see CADE-18 ATP System Competition (CASC-18) (n.d.), which refer to CPU time on a 993 MHz Intel P4. The timings of the "Saturate"-prover were taken from Ganzinger and Stuber (2003) and were measured on a 2000 MHz CPU (timings are only available for examples from the FOF division (first-order form) of CASC-18).

The former winner of the FOF division of previous CASCs, SPASS, did not participate in CASC-18. Table 2 compares timings of the *Theorema* set theory prover on some of the SET examples contained in TPTP to the performance of a revised version of SPASS as reported in Afshordel et al. (2001). SPASS's timings have been recorded on a 333 MHz Intel P2, *Theorema* timings refer to experiments on a 400 MHz Intel P2.

We want to emphasize, however, that the absolute computation times are not our main focus in the current stage of development, mainly because the *Theorema* system is currently implemented in the programming language of Mathematica, which does not offer possibilities for compiling programs. Hence, comparing the run-time of *interpreted Mathematica code* to computing times of *optimized compiled machine code* does not tell us much. The timings in Tables 1 and 2 are meant to demonstrate that our set theory prover generates proofs "within a few seconds" even for examples where other provers fail completely or need considerably more time, see e.g. (SET014), (SET171), (SET612), or (SET716). The Saturate prover performs very well on the previously mentioned examples, but interestingly it is slower by a factor of almost 50 in (SET630). Note, on the other hand, that the *Theorema* set theory prover is considerably slower than the CASC

[2] Software versions used: E-SETHEO csp02, Vampire 5.0, DCTP 10.1p, Bliksem 1.12a.

Table 2
Theorema versus SPASS

Example	*Theorema*	SPASS 2.0
SET010	6.1	1
SET612	7.5	1
SET624	155.4	101
SET694	5.5	1
SET698	22.7	71
SET722	6.6	18
SET751	5.04	3

provers in (SET624), which will be discussed in detail in Section 8.2.5. Table 2 shows that the *Theorema* set theory prover and SPASS show "similar" behavior.

Of course, proof generation should be fast, but we are currently much more interested in having automatically generated "nice proofs" that are easily understandable for a human reader. We therefore aim at designing provers that apply inference rules in a smart and "natural human-like way" without too many failing branches during the proof search. Once this is achieved, the absolute computation times depend only on the efficiency of executing the programs on particular hardware. We can speed up the entire system (i.e. *all provers* available in the *Theorema* system!) by improving the runtime environment, on which the *Theorema* system is based. One possibility, which we are currently investigating for a re-design of *Theorema*, is to develop an efficient execution engine (based on e.g. Java) for a certain fragment of the Mathematica programming language that would allow the compilation of our provers. From first experiments an envisaged speed-up by a factor between 50 and 100 seems realistic. One can observe in practical examples as shown in the subsequent sections that the proofs generated by our set theory prover contain only a *few failing branches*, and each branch contains only a *few useless formulae*.

8.2. Proofs generated by the Theorema set theory prover

In this section, we collect some representative proofs that were generated *completely automatically* by the *Theorema* set theory prover. In order to justify our claim from Section 8.1 that "the set theory prover generates proofs that are easily understandable for a human reader", the examples in the subsequent sections will not only describe the methods and heuristics used in our set theory prover, but they will also include the generated proofs. The proofs are displayed in *simplified form*, i.e. they do not contain anymore failing proof branches and they do not show any formulae that did not contribute to the final proof success. The *fully automated simplification* of the "raw proof object" as it is produced by the set theory prover is a standard post-processing feature available in the *Theorema* system and the timings given in Section 8.1 include also the time needed for proof simplification.

The optical appearance of the proofs in the *Theorema* system corresponds exactly to how they are typeset in this paper! Within *Theorema*, the standard presentation of proofs is generated in a Mathematica notebook document, a document format provided by Mathematica that allows typeset mathematical text to be intermixed with Mathematica input and output expressions as well as graphics. The proofs have been translated from Mathematica notebook format into LaTeX as accurately as possible without manual beautification. Some of the features of the *Theorema* standard proof presentation utilize, however, special capabilities of the Mathematica notebook format and can therefore not be rendered in this paper:

- Formulae in the knowledge base and goal formulae are displayed in different color.
- Formula labels in running text are "click-able" and show the entire referenced formula in a popup-window when clicked.
- Proof branches are organized in a hierarchy of nested cells that reflects the structure of the proof. Collapsing entire proof-branches by mouse-click allows us to quickly browse through the structure of a proof and easily "zoom into" the interesting proof parts or skip uninteresting proof parts, respectively.

8.2.1. Properties of functions built into the set theory prover

The *Theorema* mathematical language supports the notion $f :: A \to B$ denoting the predicate "f is a function from A to B (in intensional form)". In intensional form, a function from A to B is something that can be applied to some term in A resulting in a term in B. Alternatively, *Theorema* offers the concept of a function from A to B in extensional form (written $f : A \to B$) from set theory, where a function is a certain subset of $A \times B$. As an example, we take (SET722), where the set theory prover succeeds for both intensional and extensional representation for functions. The computing times do not essentially differ between the two variants. We present the proof of (SET722) based on the intensional function concept, in order to demonstrate that the use of the set theory prover does not require the user to force all of mathematics into set representation.

The *Theorema* set theory prover does not require the definition of surjectivity in its knowledge base. Rather, it recognizes surjectivity on the inference rule level, i.e. the prover contains inference rules for proving surjectivity and for expanding surjectivity in the knowledge base, respectively, regardless of whether the intensional or the extensional function concept is used.

Proof (*SET722*). $\underset{A,B,C,f,g}{\forall} f :: A \to B \wedge g \circ f :: A \overset{surj.}{\to} C \Rightarrow g :: B \overset{surj.}{\to} C$,

under the assumption:

(Definition (Composition)) $\underset{f,g,x}{\forall} (g \circ f)[x] := g[f[x]]$.

We assume

(1) $f_0 :: A_0 \to B_0 \wedge g_0 \circ f_0 :: A_0 \overset{surj.}{\to} C_0$,

and show

(2) $g_0 :: B_0 \overset{surj.}{\to} C_0$.

In order to show surjectivity of g_0 in (2) we assume

(3) $x1_0 \in C_0$,

and show

(4) $\underset{B1}{\exists} B1 \in B_0 \wedge g_0[B1] = x1_0$.

From (1.1) we can infer

(6) $\underset{A1}{\forall} A1 \in A_0 \Rightarrow f_0[A1] \in B_0$.

From (1.2) we know by definition of "surjectivity"

(7) $\underset{A2}{\forall} A2 \in A_0 \Rightarrow (g_0 \circ f_0)[A2] \in C_0$,

(8) $\underset{x2}{\forall} x2 \in C_0 \Rightarrow \underset{A2}{\exists} A2 \in A_0 \wedge (g_0 \circ f_0)[A2] = x2$.

By (8), we can take an appropriate Skolem function such that

(9) $\underset{x2}{\forall} x2 \in C_0 \Rightarrow A2_0[x2] \in A_0 \wedge (g_0 \circ f_0)[A2_0[x2]] = x2$.

Formula (3), by (9), implies:

$A2_0[x1_0] \in A_0 \wedge (g_0 \circ f_0)[A2_0[x1_0]] = x1_0$,

which, by (6), implies:

$f_0[A2_0[x1_0]] \in B_0 \wedge (g_0 \circ f_0)[A2_0[x1_0]] = x1_0$,

which, by (Definition (Composition)), implies:

(10) $f_0[A2_0[x1_0]] \in B_0 \wedge g_0[\, f_0[A2_0[x1_0]]] = x1_0$.

Formula (4) is proven because, with $B1 := f_0[A2_0[x1_0]]$, (10) is an instance. □

The use of the special inference rules for function properties like e.g. surjectivity can be suppressed by an option in the Prove-call. The knowledge base would then need to contain the definition of surjectivity explicitly. The proof of (SET722), however, succeeds even in this setting. It differs only in that the special inference rule combines several proof steps into one compact step. Special inference rules are available also for injectivity, which are used in (SET716), where the proof takes just 6.8 s, which is about the same time that the "Saturate"-prover needs. Note, however, that all the CASC provers fail in (SET716).

8.2.2. Set theory specific knowledge built into the prover

The examples in this section shall demonstrate how set theory specific knowledge is built into the prover. As already mentioned in the description of the individual special provers, most of the inference rules used in the set theory prover are in essence only re-formulations of the *axioms and definitions of ZF*. Some rules it uses, however, are based *on certain theorems* that are valid in ZF. Clearly, these theorems are themselves only consequences of the axioms, therefore *all inference rules in the set theory prover are based on the axioms of ZF*. What we want to say is that there are certain rules that correspond to direct application of an axiom and there are other rules that hide a chain of logical arguments based on the axioms. The examples in this section try to show that this is reasonable because the *Theorema* set theory prover is intended for mathematicians who want to have support in their every-day work using sets. It is not intended to be a prover that reduces every mathematical proof to the axioms of ZF.

Proof (*Proposition (Intersection Powerset)*). $\bigcap \mathcal{P}[A] = \emptyset$,

with no assumptions.
We have to prove (Proposition (intersection powerset)), hence, we have to show:

(1) $A1_0 \notin \bigcap \mathcal{P}[A]$.

We prove (1) by contradiction.
We assume

(2) $A1_0 \in \bigcap \mathcal{P}[A]$,

and show (a contradiction).
From (2) we can infer

(3) $\underset{A2}{\forall} A2 \in \mathcal{P}[A] \Rightarrow A1_0 \in A2$.

From (3) we can infer

(4) $A1_0 \in \emptyset$,

(5) $A1_0 \in A$.

Using available computation rules we can simplify the knowledge base:
Formula (4) simplifies to

(6) *False*.

Formula (a contradiction) is true because the assumption (6) is false. □

It can be proven in ZF that the power set $\mathcal{P}[A]$ always contains $\emptyset$ and A itself, and, of course, the *Theorema* set theory prover can prove this. Thus, we can instantiate a universally quantified variable running over the power set of A with $\emptyset$ and A. This theorem is coded into a special inference rule in `STKBR`, which allows the instantiation of the universally quantified assumption (3) to infer (4) and (5). The simplification of (4) to (6) is then accomplished in phase 1 of the subsequent saturation run in `STKBR` by built-in semantic knowledge about finite sets (in particular, the empty set) as described in Section 5.2.

The second example is taken from the case study on the mutilated checkerboard, see McCarthy (1964, 1995) and Windsteiger (2001b,a). The theorem says that an 8×8 checkerboard with two opposite corners missing can always be covered by dominos. A proof of this theorem can be given using a formulation of the problem in set theory. A proof of the theorem has been generated using *Theorema* by building up the theory of dominos, checkerboards, coverings, etc. and by completely exploring new notions as they are defined. "Completely exploring", in this context, means that sufficiently many properties of a new notion are proven before the next notion will be introduced, see Buchberger (1999). Although each of the proofs in this exploration is generated completely automatically, the whole proof cannot be called "fully automated", because the exploration itself is the interaction of the user with the *Theorema* system. This is a highly non-trivial, mathematically very interesting and challenging, and didactically very instructive experience for the human user. Using systems such as *Theorema*, the mathematician can then concentrate on this task of structuring mathematical knowledge in big theories, while the actual proofs are then assisted by the computer. For students this opens the possibility to experiment on building up own theories and to, in the optimal case, *understand why* certain known theories are built up in a certain way.

During one of these so-called exploration rounds, we arrived at the proposition that whenever X is a domino on the board then the domino covers two distinct fields that are adjacent to each other.

Proof (*Proposition (Dominos Adjacent)*).

$$\underset{X}{\forall} \text{domino-on-board}[X] \Rightarrow X \subseteq \text{Board} \wedge \underset{x,y}{\exists} x \in X \wedge y \in X \wedge x \neq y \wedge \text{adjacent}[x, y],$$

under the assumption:

(Definition (Domino))

$$\forall_{x} \text{ (domino-on-board}[x] :\Leftrightarrow x \subseteq \text{Board} \wedge |x| = 2 \wedge$$

$$\forall_{x1,x2} x1 \in x \wedge x2 \in x \wedge x1 \neq x2 \Rightarrow \text{adjacent}[x1, x2]).$$

We assume

(2) domino-on-board$[X_0]$,

and show

(3) $X_0 \subseteq \text{Board} \wedge \exists_{x,y} x \in X_0 \wedge y \in X_0 \wedge x \neq y \wedge \text{adjacent}[x, y]$.

Proof of (3.1) $X_0 \subseteq$ *Board*: (SKIPPED)
Proof of (3.2):
Formula (2), by (Definition (Domino)), implies:

(5) $|X_0| = 2 \wedge X_0 \subseteq \mathit{Board} \wedge$

$$\forall_{x1,x2} x1 \in X_0 \wedge x2 \in X_0 \wedge x1 \neq x2 \Rightarrow \text{adjacent}[x1, x2].$$

From (5.1) we can infer

(6) $X1_0 \in X_0$,

(7) $X1_1 \in X_0$,

(8) $X1_0 \neq X1_1$.

Now, let $y := X1_0$. Thus, for proving (3.2) it is sufficient to prove:

(10) $\exists_{x} x \in X_0 \wedge X1_0 \in X_0 \wedge x \neq X1_0 \wedge \text{adjacent}[x, X1_0]$.

Now, let $x := X1_1$. Thus, for proving (10) it is sufficient to prove:

(15) $X1_1 \in X_0 \wedge X1_0 \in X_0 \wedge X1_1 \neq X1_0 \wedge \text{adjacent}[X1_1, X1_0]$.

Using available computation rules we evaluate (15) using (8) and (5.1) as additional assumption(s) for simplification:

(16) $X1_1 \in X_0 \wedge X1_0 \in X_0 \wedge \text{adjacent}[X1_1, X1_0]$.

Proof of (16.1) $X1_1 \in X_0$:
Formula (16.1) is true because it is identical to (7).
Proof of (16.2) $X1_0 \in X_0$:
Formula (16.2) is true because it is identical to (6).
Proof of (16.3) adjacent$[X1_1, X1_0]$:
Formula (16.3), using (5.3), is implied by:

(17) $X1_0 \in X_0 \wedge X1_1 \in X_0 \wedge X1_1 \neq X1_0$.

Using available computation rules we evaluate (17) using (8) and (5.1) as additional assumption(s) for simplification:

(18) $X1_0 \in X_0 \wedge X1_1 \in X_0$.

Proof of (18.1) $X1_0 \in X_0$:
Formula (18.1) is true because it is identical to (6).
Proof of (18.2) $X1_1 \in X_0$:
Formula (18.2) is true because it is identical to (7). □

In this proof, special knowledge about cardinality goes into the proof. STKBR uses an inference rule that allows us to choose distinct elements from a finite set, i.e. if we know $|A| = n$ for some natural number n then we can choose *new constants* $x_1, \ldots, x_n$ such that $x_i \in A$ (for each $1 \leq i \leq n$) and "all x_i distinct". This rule allows us to infer (6), (7), and (8) from (5.1) in the proof above, the remaining proof is straightforward.

8.2.3. The interplay between Theorema and Mathematica

The two proofs in this section will show, how *Theorema* interacts with the underlying Mathematica system. We want to emphasize the strict separation between *Theorema* and Mathematica in the sense that no Mathematica algorithm from the rich computer algebra library available through Mathematica is applied "quietly" during a proof in the *Theorema* system unless the user explicitly allows the *Theorema* set theory prover to do so. The first example uses Mathematica's Solve function for instantiating existential variables in the proof goal.

Proof (*SET751*).

$$\underset{A,B,f,X,Y}{\forall} X \subseteq A \wedge Y \subseteq A \wedge X \subseteq Y \wedge f :: A \to B \Rightarrow \text{image}[f, X] \subseteq \text{image}[f, Y],$$

under the assumption:

(Definition (Image)) $\underset{f,X}{\forall} \text{image}[f, X] := \{ f[x] \mid_{x} x \in X \}$.

We assume

(1) $X_0 \subseteq A_0 \wedge Y_0 \subseteq A_0 \wedge X_0 \subseteq Y_0 \wedge f_0 :: A_0 \to B_0$,

and show

(2) $\text{image}[f_0, X_0] \subseteq \text{image}[f_0, Y_0]$.

For proving (2) we choose

(3) $f1_0 \in \text{image}[f_0, X_0]$,

and show:

(4) $f1_0 \in \text{image}[f_0, Y_0]$.

From (1.3) we can infer

(8) $\underset{X2}{\forall} X2 \in X_0 \Rightarrow X2 \in Y_0$.

Formula (3), by (Definition (Image)), implies:

(11) $f1_0 \in \{ f_0[x] \mid_x x \in X_0 \}$.

From (11) we know by definition of $\{T_x \mid_x P\}$ that we can choose an appropriate value such that

(12) $x1_0 \in X_0$,

(13) $f1_0 = f_0[x1_0]$.

Formula (4), using (13), is implied by:

$f_0[x1_0] \in \text{image}[f_0, Y_0]$,

which, using (Definition (Image)), is implied by:

(19) $f_0[x1_0] \in \{ f_0[x] \mid_x x \in Y_0 \}$.

In order to prove (19) we have to show

(20) $\exists_x x \in Y_0 \wedge f_0[x1_0] = f_0[x]$.

Since $x := x1_0$ solves the equational part of (20) it suffices to show

(21) $x1_0 \in Y_0$.

Formula (21), using (8), is implied by:

(22) $x1_0 \in X_0$.

Formula (22) is true because it is identical to (12). □

Since a sub-formula of the existential goal (20) is an equality containing the existential variable, we instantiate the existential variable x in the proof goal with the help of Mathematica. In this example, a candidate for x was found by solving the equation $f_0[x1_0] = f_0[x]$ for x, which is done by a call to the Mathematica function `Solve` for solving (systems of) equations. Of course, unification or even matching would have done this job as well, but, in the case of equational sub-formulae, the `STS`-prover tries to apply the *specific rule* using `Solve` *before it tries general predicate logic solving* using matching and unification.

The second example shows, how arithmetic knowledge on natural numbers provided by Mathematica is accessible for the set theory prover. As already discussed in Section 6 on the set theory computation unit `STC`, semantic knowledge about natural numbers from the *Theorema* language is not available in the set theory prover by default, but it can be provided by the user on demand in the call of the prover using the "built-in"-option.

Proof (*G*). $36 \in \bigcup_{i \in \mathbb{N}} \{ j^2 \mid_{j \in \mathbb{N}} j \geq i \wedge j \leq i + 5 \}$

under the assumption

(A) $\forall_{m,n} n > m \Rightarrow \exists_i i \leq n \wedge i \geq m \wedge i \in \mathbb{N}$.

In order to show (G) we have to show

(1) $\exists_i 36 \in \{ j^2 \mid_{j \in \mathbb{N}} j \geq i \wedge j \leq i + 5 \} \wedge i \in \mathbb{N}$.

In order to prove (1) we have to show

(2) $\exists_i \exists_j\ j \geq i \wedge j \in \mathbb{N} \wedge j \leq i + 5 \wedge i \in \mathbb{N} \wedge 36 = j^2$.

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

(4) $\exists_i\ i \in \mathbb{N} \wedge 6 \geq i \wedge 6 \in \mathbb{N} \wedge 6 \leq 5 + i$.

Using available computation rules we evaluate (4):

(6) $\exists_i\ i \leq 6 \wedge i \geq 1 \wedge i \in \mathbb{N}$.

Formula (6), using (A), is implied by:

(7) $6 > 1$.

Using available computation rules we evaluate (7):

(8) *True*.

Formula (8) is true because it is the constant *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. Similar to the previous example, since a sub-formula of the existential goal (2) is an equality containing the quantified variable, the Mathematica `Solve` function is used internally to solve the quadratic equation $36 = j^2$ for j, which finds two solutions $j = -6$ and $j = 6$. Of course, in this example matching and unification would not be an alternative, since theory-specific arithmetic knowledge is necessary for solving this formula. The first solution results in a failing proof attempt, since $-6 \in \mathbb{N}$ simplifies to *False* by built-in knowledge about $\mathbb{N}$. The failing branch is eliminated when finally simplifying the successful proof. Note that the labels of the formulae indicate a "missing branch". Formulae (3) and (5) do not appear in the proof presentation because they have been eliminated during proof simplification. Simplifications from (4) to (6) and from (7) to (8) were made using available semantic knowledge about natural numbers by STC ($6 \in \mathbb{N}$ and $6 > 1$, respectively) and, finally, reduction from (6) to (7) and the detection of proof success were made by standard predicate logic inference rules. We have no specialized solving methods for natural numbers available, therefore we needed assumption (A) in the knowledge base. An appropriate solver for $\mathbb{N}$ would be able to verify (6) without any additional knowledge. We will investigate necessary solving techniques in future work.

8.2.4. Theory exploration versus isolated theorem proving

We consider (SET770), an example from the TPTP library concerning equivalence classes, namely the theorem that two equivalence classes are equal or disjoint. Note again, that none of the provers in the CASC competition could solve this problem. In Windsteiger (2001a), an entire exploration of the theory of equivalence relations, equivalence classes, factor sets, partitions, induced relations, etc. is given. Instead of proving (SET770) from first principles, i.e. from the axioms, it is preferable to first prove some auxiliary lemmata, which later facilitate the proof of the theorem. This is just what a human mathematician would be doing. We present here the proof of (SET770) using the two auxiliary propositions (equal classes) and (not in distinct classes) in the knowledge base. The computing time for the proof is 5.8 s on a 2000 MHz Intel P4, the proofs

of the auxiliary propositions take 8.5 and 8.6 s, respectively. Of course, this example using the two additional propositions is not anymore (SET770) in the sense of TPTP! It should be clear that the timings for the examples given in Section 8.1 refer to the problem formulation as specified in the TPTP library, except that, of course, we may omit definitions in the knowledge base that refer to set theory specific constructs, which are covered by inference rules in our prover. We do not claim this example to be a "solution for (SET770) as given in TPTP" and, therefore, we also did not include it in the tables of timings in Section 8.1. We rather show this example in order to advocate for "theory exploration" being superior to "proving from first principles", in particular if we want mathematicians to appreciate our systems.

Proof (*SET770*). $\underset{R,x,y}{\forall}$ is-symmetric$[R] \wedge$ is-transitive$[R] \Rightarrow$

$$(\text{class}[x, R] = \text{class}[y, R]) \vee (\text{class}[x, R] \cap \text{class}[y, R] = \{\}).$$

under the assumptions:

(Proposition (equal classes)) $\underset{R,x,y}{\forall}$ is-transitive$[R] \wedge$ is-symmetric$[R] \wedge$

$$\langle x, y\rangle \in R \Rightarrow \text{class}[x, R] = \text{class}[y, R],$$

(Proposition (not in distinct classes)) $\underset{R,x,y,z}{\forall}$ is-symmetric$[R] \wedge$ is-transitive$[R] \wedge$

$$x \in \text{class}[y, R] \wedge x \in \text{class}[z, R] \Rightarrow \langle y, z\rangle \in R.$$

We assume

(1) is-symmetric$[R_0] \wedge$ is-transitive$[R_0]$,

and show

(2) $(\text{class}[x_0, R_0] = \text{class}[y_0, R_0]) \vee (\text{class}[x_0, R_0]) \cap \text{class}[y_0, R_0] = \{\})$.

We prove (2) by proving the first alternative negating the other(s).
We assume

(4) $\text{class}[x_0, R_0] \cap \text{class}[y_0, R_0] \neq \{\}$.

We now show

(3) $\text{class}[x_0, R_0] = \text{class}[y_0, R_0]$.

From (4) we know that we can choose an appropriate value such that

(5) $x3_0 \in \text{class}[x_0, R_0] \cap \text{class}[y_0, R_0]$.

From (5) we can infer

(7) $x3_0 \in \text{class}[x_0, R_0]$,

(8) $x3_0 \in \text{class}[y_0, R_0]$.

Formula (3), using (Proposition (equal classes)), is implied by:

(11) is-symmetric$[R_0] \wedge$ is-transitive$[R_0] \wedge \langle x_0, y_0\rangle \in R_0$.

Proof of (11.1) is-symmetric$[R_0]$:

Formula (11.1) is true because it is identical to (1.1).
Proof of (11.2) is-transitive[R_0]:
Formula (11.2) is true because it is identical to (1.2).
Proof of (11.3) $\langle x_0, y_0\rangle \in R_0$:
Formula (11.3), using (Proposition (not in distinct classes)), is implied by:

(12) $\underset{x}{\exists}$ is-symmetric[R_0] $\wedge$ is-transitive[R_0] $\wedge\ x \in$ class[x_0, R_0] $\wedge\ x \in$ class[y_0, R_0].

Now, let $x := x3_0$. Thus, for proving (12) it is sufficient to prove:

(13) is-symmetric[R_0] $\wedge$ is-transitive[R_0] $\wedge\ x3_0 \in$ class[x_0, R_0] $\wedge\ x3_0 \in$ class[y_0, R_0].

Proof of (13.1) is-symmetric[R_0]:
Formula (13.1) is true because it is identical to (1.1).
Proof of (13.2) is-transitive[R_0]:
Formula (13.2) is true because it is identical to (1.2).
Proof of (13.3) $x3_0 \in$ class[x_0, R_0]:
Formula (13.3) is true because it is identical to (7).
Proof of (13.4) $x3_0 \in$ class[y_0, R_0]:
Formula (13.4) is true because it is identical to (8). □

The same case study has been carried out for an intensional concept of relations. Similar to the intensional concept of a function described in Section 8.2.1, an intensional relation is something that can be applied to terms yielding true or false. An intensional relation is nothing else than a predicate in the sense of logic. We show one of the proofs and explain its key steps, since this proof shows the natural interplay between P-, C-, and S-phases as implemented in the *Theorema* set theory prover very nicely. This example also demonstrates that the PCS-strategy, which has already been used in a prover for elementary analysis within the *Theorema* system, see Buchberger (2001); Vasaru-Dupré (2000), yields natural proofs very similar to the style a human mathematician would give the proof.

Proof (*Lemma (Union Inverse Factor Set)*). $\underset{A}{\forall}$ is-reflexive$_A$[$\sim$] $\Rightarrow \bigcup$ factor-set$_\sim$[A] $= A$,

under the assumptions:

(Definition (relation sets): class) $\underset{A,x}{\forall}$ class$_{A,\sim}$[x] $:= \{a \mid_a a \in A \wedge a \sim x\}$,

(Definition (relat. sets): factor-set) $\underset{A}{\forall}$ factor-set$_\sim$[A] $:= \{$class$_{A,\sim}$[x] $\mid_x x \in A\}$,

(Definition (reflexivity)) $\underset{A}{\forall}$ is-reflexive$_A$[$\sim$] $:\Leftrightarrow \underset{x}{\forall}\ (x \in A \Rightarrow x \sim x)$.

We assume

(1) is-reflexive$_{A_0}$[$\sim$],

and show

(2) $\bigcup$ factor-set$_\sim$[A_0] $= A_0$.

Formula (2), using (Definition (relation sets): factor-set), is implied by:

$$\bigcup\{\mathrm{class}_{A_0,\sim}[x] \mid_x x \in A_0\} = A_0,$$

which, using (Definition (relation sets): class), is implied by:

(3) $\bigcup\{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\} = A_0.$

Formula (1), by (Definition (reflexivity)), implies:

(4) $\forall_x (x \in A_0 \Rightarrow x \sim x).$

We show (3) by mutual inclusion:
$\subseteq$: We assume

(5) $x1_0 \in \bigcup\{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\}$

and show:

(6) $x1_0 \in A_0.$

From (5) we know by definition of the big $\bigcup$-operator that we can choose an appropriate value such that

(7) $x2_0 \in \{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\},$

(8) $x1_0 \in x2_0.$

From (7) we know by definition of $\{T_x \mid_x P\}$ that we can choose an appropriate value such that

(9) $a1_0 \in A_0,$

(10) $x2_0 = \{a \mid_a a \in A_0 \wedge a \sim a1_0\}.$

Formula (8), by (10), implies:

(23) $x1_0 \in \{a \mid_a a \in A_0 \wedge a \sim a1_0\}.$

From (23) we can infer

(24) $x1_0 \in A_0 \wedge x1_0 \sim a1_0.$

Formula (6) is true because it is identical to (24.1).
$\supseteq$: Now we assume

(6) $x1_0 \in A_0.$

and show:

(5) $x1_0 \in \bigcup\{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\}.$

In order to show (5) we have to show

(29) $\exists_{x4} x1_0 \in x4 \wedge x4 \in \{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\}.$

In order to solve (29) we have to find $x4^*$ such that

(30) $x1_0 \in x4^* \wedge \exists_x (x \in A_0 \wedge x4^* = \{a \mid_a a \in A_0 \wedge a \sim x\}).$

Since (6) matches a part of (30) we try to instantiate, i.e. let now $x := x1_0$.
Thus, by (30), we choose $x4^* := \{a \mid_a a \in A_0 \wedge a \sim x1_0\}$.

Now, it suffices to show

(32) $x1_0 \in A_0 \wedge x1_0 \in \{a \mid_a a \in A_0 \wedge a \sim x1_0\}$.

Proof of (32.1) $x1_0 \in A_0$:
Formula (32.1) is true because it is identical to (6).
Proof of (32.2) $x1_0 \in \{a \mid_a a \in A_0 \wedge a \sim x1_0\}$:

In order to prove (32.2) we have to show:

(33) $x1_0 \in A_0 \wedge x1_0 \sim x1_0$.

Formula (33), using (4), is implied by:

(34) $x1_0 \in A_0$.

Formula (34) is true because it is identical to (6). □

We briefly comment on the essential steps in the proof:

- The proof starts with a P-phase, in which the universally quantified implication in the proof goal is reduced by natural deduction inference rules for predicate logic from the special prover `BasicND`, see (1) and (2).
- In a C-phase, the special prover `QR` rewrites the goal and the knowledge base using the definitions in the knowledge base, see (3) and (4).
- The prover switches back again to a P-phase, but now the `STP` prover reduces set equality $X = Y$ to the two subgoals $X \subseteq Y$ and $X \supseteq Y$. In fact, the inference rule for set equality reduces the subgoals by Definition of '$\subset$' immediately, see (5) and (6).
- For proving the first subgoal (6), staying in a P-phase, `STKBR` expands membership in a union and a set quantifier in the knowledge base in two subsequent level saturation runs, see (7), (8), (9) and (10).
- In a C-phase, `QR` uses the equality (10) for rewriting (8) into (23).
- In the final P-phase, expanding membership proves the subgoal (6).
- For proving the second subgoal (5), first `STP` reduces membership in a union during a P-phase into the existential goal (29).
- The set theory prover enters an S-phase. The goal (29) has the special structure $\exists_{x4} x1_0 \in x4 \wedge x4 \in \{T_x \mid_x P_x\}$, which can be handled by rule 'IntroSolveConstant' from Section 7. Thus, the existential quantifier is eliminated by introducing the solve constant $x4^*$, and the expansion of the inner membership $x4^* \in \{T_x \mid_x P_x\}$ introduces another existential quantifier (now for x), see (30).
- The existential sub-formula in (30) is solved for x by unification with formulae in the knowledge base. In fact, in this example *matching* is sufficient, but we provide unification in this step for the general case. Having solved for x, the solve constant $x4^*$ can be instantiated from the equational sub-formula $x4^* = \ldots$ in (30), reducing the solve problem (30) again to a proof problem, see (32).
- In the P-phase, the goal (32) is split using general predicate logic, subgoal (32.1) is trivially true, and subgoal (32.2) is handled first by a set theory specific proof rule from `STP`, see (33).

- Finally, the goal (33) is proved by simple rewriting using implications from the knowledge base in a C-phase, see (34).

8.2.5. An example of "Weak Performance" of the set theory prover

Proof (*SET624*). $\underset{B,C,D}{\forall} B \cap (C \cup D) \neq \{\} \Leftrightarrow B \cap C \neq \{\} \vee B \cap D \neq \{\}$.

For proving (SET624) we take all variables arbitrary but fixed and prove:

(1) $B_0 \cap (C_0 \cup D_0) \neq \{\} \Leftrightarrow B_0 \cap C_0 \neq \{\} \vee B_0 \cap D_0 \neq \{\}$.

Direction from left to right:
We assume

(3) $B_0 \cap (C_0 \cup D_0) \neq \{\}$

and show

(2) $B_0 \cap C_0 \neq \{\} \vee B_0 \cap D_0 \neq \{\}$.

From (3) we know that we can choose an appropriate value such that

(6) $B1_0 \in B_0 \cap (C_0 \cup D_0)$.

From (6) we can infer

(8) $B1_0 \in B_0$,

(9) $B1_0 \in C_0 \cup D_0$.

From (9) we can infer

(11) $B1_0 \in C_0 \vee B1_0 \in D_0$.

We prove (2) by proving the first alternative negating the other(s).
We assume

(13) $\neg(B_0 \cap D_0 \neq \{\})$.

We now show

(12) $B_0 \cap C_0 \neq \{\}$.

Formula (12) means that we have to show that

(14) $\underset{B2}{\exists} B2 \in B_0 \cap C_0$.

We prove (14) by splitting up the intersection into its individual components:
We have to prove:

(15) $\underset{B2}{\exists} B2 \in B_0 \wedge B2 \in C_0$.

Formula (13) is simplified to

(16) $B_0 \cap D_0 = \{\}$.

From (16) we can infer

(17) $\underset{B3}{\forall} B3 \notin B_0 \cap D_0$.

From (17) we can infer

(18) $\underset{B3}{\forall} B3 \notin B_0 \vee B3 \notin D_0$.

We prove (15) by case distinction using (11).
Case (11.1) $B1_0 \in C_0$:
Now, let $B2 := B1_0$. Thus, for proving (15) it is sufficient to prove:

(20) $B1_0 \in B_0 \wedge B1_0 \in C_0$.

Proof of (20.1) $B1_0 \in B_0$:
Formula (20.1) is true because it is identical to (8).
Proof of (20.2) $B1_0 \in C_0$:
Formula (20.2) is true because it is identical to (11.1).
Case (11.2) $B1_0 \in D_0$:
From (8), by (18), we obtain:

(29) $B1_0 \notin D_0$.

Formula (15) is proved because (29) and (11.2) are contradictory.
Direction from right to left:
We assume

(5) $B_0 \cap C_0 \neq \{\} \vee B_0 \cap D_0 \neq \{\}$

and show

(4) $B_0 \cap (C_0 \cup D_0) \neq \{\}$.

Formula (4) means that we have to show that

(30) $\underset{B4}{\exists} B4 \in B_0 \cap (C_0 \cup D_0)$.

We prove (30) by splitting up the intersection into its individual components:
We have to prove:

(31) $\underset{B4}{\exists} B4 \in B_0 \wedge B4 \in C_0 \cup D_0$.

We prove (31) by case distinction using (5).
Case (5.1) $B_0 \cap C_0 \neq \{\}$:
From (5.1) we know that we can choose an appropriate value such that

(32) $B5_0 \in B_0 \cap C_0$.

From (32) we can infer

(34) $B5_0 \in B_0$,

(35) $B5_0 \in C_0$.

Now, let $B4 := B5_0$. Thus, for proving (31) it is sufficient to prove:

(38) $B5_0 \in B_0 \wedge B5_0 \in C_0 \cup D_0$.

We prove the individual conjunctive parts of (38):
Proof of (38.1) $B5_0 \in B_0$:
Formula (38.1) is true because it is identical to (34).
Proof of (38.2) $B5_0 \in C_0 \cup D_0$:
In order to prove (38.2) we may assume

(40) $B5_0 \notin D_0$

and show:

(39) $B5_0 \in C_0$.

(Note, that in all other cases the formula (38.2) trivially holds!)
Formula (39) is true because it is identical to (35).
Case (5.2) $B_0 \cap D_0 \neq \{\}$:
From (5.2) we know that we can choose an appropriate value such that

(41) $B6_0 \in B_0 \cap D_0$.

From (41) we can infer

(43) $B6_0 \in B_0$,

(44) $B6_0 \in D_0$.

Now, let $B4 := B6_0$. Thus, for proving (31) it is sufficient to prove:

(47) $B6_0 \in B_0 \wedge B6_0 \in C_0 \cup D_0$.

We prove the individual conjunctive parts of (47):
Proof of (47.1) $B6_0 \in B_0$:
Formula (47.1) is true because it is identical to (43).
Proof of (47.2) $B6_0 \in C_0 \cup D_0$:
In order to prove (47.2) we may assume

(49) $B6_0 \notin D_0$

and show:

(48) $B6_0 \in C_0$.

(Note, that in all other cases the formula (47.2) trivially holds!)
Formula (48) is proved because (49) and (44) are contradictory. □

Although the prover does not generate any failing branches in this example it is substantially slower than the CASC provers. SPASS shows similar behavior like the *Theorema* prover in that (SET624) is the example in which SPASS performs by far worst. Most probably, the reason for the weak performance of the *Theorema* set theory prover is an inefficient implementation of matching the existentially quantified variable against constants available in the knowledge base, which is needed several times in this example. Note, however, that the proof is straightforward and easy to comprehend for a human reader.

9. Conclusion

This paper describes the design and the implementation of an automated prover for Zermelo–Fraenkel set theory (ZF) in the frame of the *Theorema* system. In particular, we describe how the PCS paradigm for structuring automated theorem provers, which has already been used in other provers provided in *Theorema*, has been accommodated to set theory. The prover is intended to support mathematicians working in arbitrary areas of mathematics that are formulated *using ZF* rather than for proving theorems of ZF from the axioms. This means, we aim at proving theorems in the flavor of the examples shown in Sections 8.2.2 and 8.2.4 much more than most of the examples from the TPTP library. The proofs shown in Section 8 demonstrate that the *Theorema* set theory prover is able to produce proofs of non-trivial theorems in a human-comprehensible style. On average, the computing times for automatically generating the formatted proofs are comparably low. The set theory prover as described in this paper is contained in the public version of the *Theorema* system, which is freely available at http://www.theorema.org.

From the point of view of prover design, the set theory prover is the first prover in the *Theorema* system that interfaces *proving* with *computing* based on available language semantics. The special provers `STKBR` and `STC` will be used as models for future special provers requiring access to the *Theorema* computation engine. Further investigations will be necessary in order to handle conditional rewriting more efficiently and to improve the S-phase by developing more powerful special solvers and by interfacing solvers available in the computer algebra and the constraint solving community.

References

Afshordel, B., Hillenbrand, T., Weidenbach, C., 2001. First order atom definitions extended. In: Nieuwenhuis, R., Voronkov, A. (Eds.), LPAR 2001. In: LNAI, vol. 2250. Springer Verlag, Berlin, Heidelberg, pp. 309–319.

Bernays, P., Fraenkel, A., 1968. Axiomatic set theory. In: Studies in Logic and the Foundations of Mathematics, 2nd ed. North-Holland Publishing Company.

Buchberger, B., 1985. Gröbner bases: an algorithmic method in polynomial ideal theory. In: Bose, N. (Ed.), Multidimensional Systems Theory. D. Reidel Publishing Company, Dordrecht, Boston, Lancaster, pp. 184–232.

Buchberger, B., 1996. Mathematics: an introduction to mathematics integrating the pure and algorithmic aspect. In: Volume I: A Logical Basis for Mathematics. In: Lecture Notes for the Mathematics Course in the First and Second Semester at the Fachhochschule for Software Engineering in Hagenberg, Austria.

Buchberger, B., 1999. Theory Exploration Versus Theorem Proving. In: Armando, A., Jabelean, T. (Eds.), Electronic Notes in Theoretical Computer Science, vol. 23-3. Elsevier, pp. 67–69. CALCULEMUS Workshop, University of Trento, Trento, Italy.

Buchberger, B., 2001. The PCS prover in Theorema. In: Moreno-Diaz, R., Buchberger, B., Freire, J. (Eds.), Proceedings of EUROCAST 2001 (8th International Conference on Computer Aided Systems Theory — Formal Methods and Tools for Computer Science). In: Lecture Notes in Computer Science, vol. 2178, 2201. Springer, Berlin, Heidelberg, New York, pp. 469–478.

Buchberger, B., Vasaru, D., 2000. The Theorema PCS Prover. Jahrestagung der DMV, Dresden, September 18–22.

CADE-18 ATP System Competition (CASC-18), n.d. http://www.cs.miami.edu/~tptp/CASC/18/.

Collins, G. E., 1975. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Second GI Conference on Automata Theory and Formal Languages. In: LNCS, vol. 33. Springer Verlag, Berlin, pp. 134–183.

Ebbinghaus, H., 1979. Einführung in die Mengenlehre, 2nd ed., Wissenschaftliche Buchgesellschaft Darmstadt. ISBN 3-534-06709-6.

Formisano, A., 2000. Theory-based resolution and automated set reasoning. Ph.D. Thesis. Universita degli Studi di Roma La Sapienza.

Ganzinger, H., Stuber, J., 2003. Superposition with equivalence reasoning and delayed clause normal form transformation. In: Proc. 19th Int. Conf. on Automated Deduction (CADE-19). In: LNCS, vol. 2741. Miami, USA, pp. 335–349.

Halmos, P., 1960. Naive Set Theory. D. Van Nostrand Company, Princeton, NJ (Reprinted by Springer-Verlag, New York, 1974, ISBN: 0-387-90092-6 (Springer-Verlag edition)).

Kriftner, F., 1998. Theorema: The language. In: Buchberger, B., Jebelean, T. (Eds.), Proceedings of the Second International Theorema Workshop, pp. 39–54. RISC report 98-10.

McCarthy, J., 1964. A tough nut for proof procedures. Stanford AI project memo.

McCarthy, J., 1995. The mutilated checkerboard in set theory. In: Matuszewski, R. (Ed.), The QED Workshop II, Warshaw University, pp. 25–26. Technical Report L/1/95.

Piroi, F., 2004. Tools for using automated provers in mathematical theory exploration. Ph.D. Thesis. RISC Institute, University of Linz.

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

Russell, B., Whitehead, A., 1910. Principia Mathematica. Cambridge University Press (Reprinted 1980).

Shoenfield, J.R., 1967. Mathematical Logic. Logic, Addison Wesley Publishing Company.

Tomuta, E., 1998. An architecture for combining provers and its applications in the Theorema system. Ph.D. Thesis. The Research Institute for Symbolic Computation, Johannes Kepler University. RISC report 98-14.

TPTP: Thousands of Problems for Theorem Provers, n.d. http://www.cs.miami.edu/~tptp/.

Vasaru-Dupré, D., 2000. Automated theorem proving by integrating proving, solving and computing. Ph.D. Thesis. RISC Institute. RISC report 00-19.

Windsteiger, W., 2001a. A set theory prover in *Theorema*: implementation and practical applications. Ph.D. Thesis. RISC Institute, http://www.risc.uni-linz.ac.at/people/wwindste/publications.html.

Windsteiger, W., 2001b. On a solution of the mutilated checkerboard problem using the *Theorema* set theory prover. In: Linton, S., Sebastiani, R. (Eds.). Proceedings of the Calculemus 2001 Symposium, http://www.risc.uni-linz.ac.at/people/wwindste/publications.html.