Article Submitted to Journal of Symbolic Computation

An Automated Prover for Zermelo-Fraenkel Set Theory in *Theorema*[*]

Wolfgang Windsteiger

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

Abstract

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

1. Introduction

The set theory prover in *Theorema* adapts the "**P**rove-**C**ompute-**S**olve" (short: PCS) proving strategy for proofs containing language constructs from set theory. The PCS paradigm was introduced originally in (Buchberger 2000) 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. application of 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,
- solving (S), i.e. instantiation of existential variables.

Having the computer algebra system Mathematica in the background of *Theorema*, we aim towards applying known solution methods from computer algebra during the S-phase, such as the Gröbner bases method for systems of algebraic

[*]This work has been supported by the "SFB Numerical and Symbolic Scientific Computing" (F013) at the University of Linz and the European Union "CALCULEMUS Project" (HPRN-CT-2000-00102).

equations, see (Buchberger 1985), or Collins' CAD method for systems of inequalities over the reals, see (Collins 1975).

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.

Following the philosophy of most of the *Theorema* provers, the set theory prover aims at generating automated proofs in a human-like natural style. Since many mathematicians are used to building up their theories in the frame of set theory, computer support for doing proofs in this area of mathematics is a basic ingredient for computerized mathematics. In our experience, 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 central 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. Apart from others, this will have considerable impact on computer-aided math education.

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 in the *Theorema* system, Section 4 introduces the set theory proving units `STP` and `STKBR`, Section 5 describes the set theory computing unit `STC`, Section 6 presents the set theory solving unit `STS`, and finally we conclude with some examples of proofs generated by the set theory prover in Section 7.

2. The Theoretical Basis of the Set Theory Prover

The use of set theory in *Theorema* is not tied to one particular axiomatization of set theory. Instead, we introduce "sets" on the level of the language by providing the braces '{' and '}' as a flexible arity matchfix function symbol used for constructing finite sets, the set quantifier as a means for describing sets by a characteristic property, and several other language constructs commonly used in mathematics, such as '$\subseteq$' (subset), '$\cup$' (union), '$\cap$' (intersection), or '$\setminus$' (set difference), see (Kriftner 1998) for an overview on supported set syntax. Providing these language constructs, we implicitly assume that sets such as $\{a\}$, $\{1, b\}$, $\{\underset{x}{x \mid} P_x\}$ (the set of *all* x satisfying P_x), or $\{T_x \underset{x}{\mid} P_x\}$ (the set of *all* T_x, when x satisfies P_x) actually exist, which is typically guaranteed by some axioms of the underlying set theory. There are different approaches, in the Zermelo-Fraenkel axiomatization (ZF) as described e.g. in (Ebbinghaus 1979) the existence of the singleton $\{a\}$ follows from an axiom on power sets and the existence of $\{1, b\}$ follows from the existence of singletons together with an axiom on unions, whereas in an axiomatization given in (Takeuti & Zaring 1971), which also follows the spirit of ZF, the existence of $\{1, b\}$ is guaranteed by an axiom of pairing and the singleton $\{a\}$ is then just defined to denote the pair $\{a, a\}$. (Shoenfield 1967), also following ZF, shows the existence of the pair $\{1, b\}$ from an axiom on power sets and defines the singleton $\{a\}$ to stand for $\{a, a\}$.

A *Theorema* language construct that deserves closer inspection in this context is the so-called *set quantifier*, i.e. the expression $\{x \underset{x}{\mid} P_x\}$, which allows one to define a set from a property P_x. In the literature, this is often addressed as *the abstraction* of a set from a property and it goes back to G. Cantor, the founder of modern set theory. As explained in (nearly) every introductory course in mathematics, the unrestricted use of abstraction soon leads to contradictions such as the well-known Russell paradox. With R denoting the "Russell-set" $\{x \underset{x}{\mid} x \notin x\}$ it is straight-forward to derive the contradiction $R \in R \Leftrightarrow R \notin R$. Different axiomatizations of set theory provide fundamentally different solutions how to avoid Russell's paradox (and others):

- ZF set theory restricts abstraction to what is called *separation*. Roughly, it requires the structure $x \in S \wedge Q$ for P_x in an abstraction $\{x \underset{x}{\mid} P_x\}$, which disallows constructions like R.
- Von-Neumann-Gödel-Bernays' axiomatization (NGB) of set theory, see e.g. (Bernays & Fraenkel 1968) or (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 & Whitehead 1910). $R \in R$ is not allowed on the grounds that R and R are not of different type.

The *Theorema* system as such does not force the user into one of the above mentioned axiomatizations. The *Theorema language* allows unrestricted use of both the set quantifier and the membership predicate, therefore allowing both the definition of R and formulae such as $R \in R \Leftrightarrow R \notin R$. The *set theory prover*, however, relies on ZF and therefore refuses to apply inference rules on formulae involving constructs such as R. In other words, the *Theorema* set theory prover does not support all of what the *Theorema* language offers for set theory. If a user desires to work e.g. in NGB set theory the *Theorema* language would allow this but *this* set theory prover would not support it.

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 sequel, we will list those axioms and definitions from ZF, on which the inference rules of the set theory prover rely. Furthermore, we introduce some convenient abbreviations for commonly used formulations in set theory supported by our prover. The *Theorema* set theory prover should, thus, be a useful tool for mathematicians embedding their work in some set theory that is consistent with these axioms, definitions, and abbreviations.

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*

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

The separation axioms[2] 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). The *Theorema* syntax for this set is $\{x \underset{x\in S}{|} P_x\}$ (alternatively $\{x \in S \mid P_x\}$) and 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*

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

As a special case, let Q_x be defined as $P_x \wedge y = T_x$ for some formula P_x and some term T_x and let S be some term not contained in Q_x. Then the respective replacement axiom justifies the definition of the set of all T_x for all $x \in S$ satisfying P_x, see (Shoenfield 1967). In *Theorema* we may write $\{T_x \underset{x\in S}{|} P_x\}$ and 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.

[2] In the literature, the separation axioms are sometimes referred to as "subset axioms".

At first sight, the construct $\{T_x \underset{x\in S}{|} P_x\}$ appears to cover also sets of the form $\{x \underset{x\in S}{|} P_x\}$, 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.

(1) and (2) define membership for special variants of the *Theorema* set quantifier that can safely be used in ZF. Based on the set quantifier, we can now give the definitions used in the set theory prover. The existence of the sets defined in the sequel is guaranteed by axioms of ZF, see e.g. (Shoenfield 1967) for the justifications.

From now on, if not stated otherwise, we want to use P_x, Q_x, R_x, and C_x as typed variables on the meta-level to denote *formulae* (with x among the free variables), all other letters shall denote *terms* (with free variables as indicated in the subscript position). 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:

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

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

The latter 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.

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

It is convenient to allow also an additional condition in the set quantifier. We follow the convention to use

$$\{\ldots \underset{\substack{x \\ C_x}}{|} P_x\} \tag{5}$$

as an abbreviation for

$$\{\ldots \underset{x}{|} C_x \wedge P_x\} \ . \tag{6}$$

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), when formulae are passed to the prover[3].

[3]Conditions in set quantifiers with multiple ranges are handled analogously.

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):

$$\emptyset := \{x \underset{x}{|} x \neq x\} \tag{9}$$

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

Definition 2.3 (finite set construction): For any $n \geq 1$:

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

Definition 2.4 (union, intersection, product): For any $n \geq 2$:

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

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

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

(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. The logical operators '$\wedge$' and '$\vee$' are assumed to be associative and commutative, thus, the set operators '$\cup$', '$\cap$', '$\times$' and the finite set construction are associative and commutative "by definition".)

Definition 2.5 (union, intersection, power set):

$$\bigcup S := \{x \underset{x}{|} \underset{s \in S}{\exists} x \in s\} \tag{15}$$

$$\bigcap S := \{x \underset{x}{|} \underset{s \in S}{\forall} x \in s\} \tag{16}$$

$$\mathcal{P}[S] := \{x \underset{x}{|} x \subseteq S\} \tag{17}$$

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

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

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

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).

2.1. Preliminaries on Terminology

We will use the following terminology in the description of the proof modules: 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.

Now, 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. 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[4]:

[4] $P_{x\to x_0}$ stands for P with each free occurrence of x substituted by x_0.

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

The rule 'ArbitraryButFixed' tells 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.

3. How Provers are Setup in *Theorema*

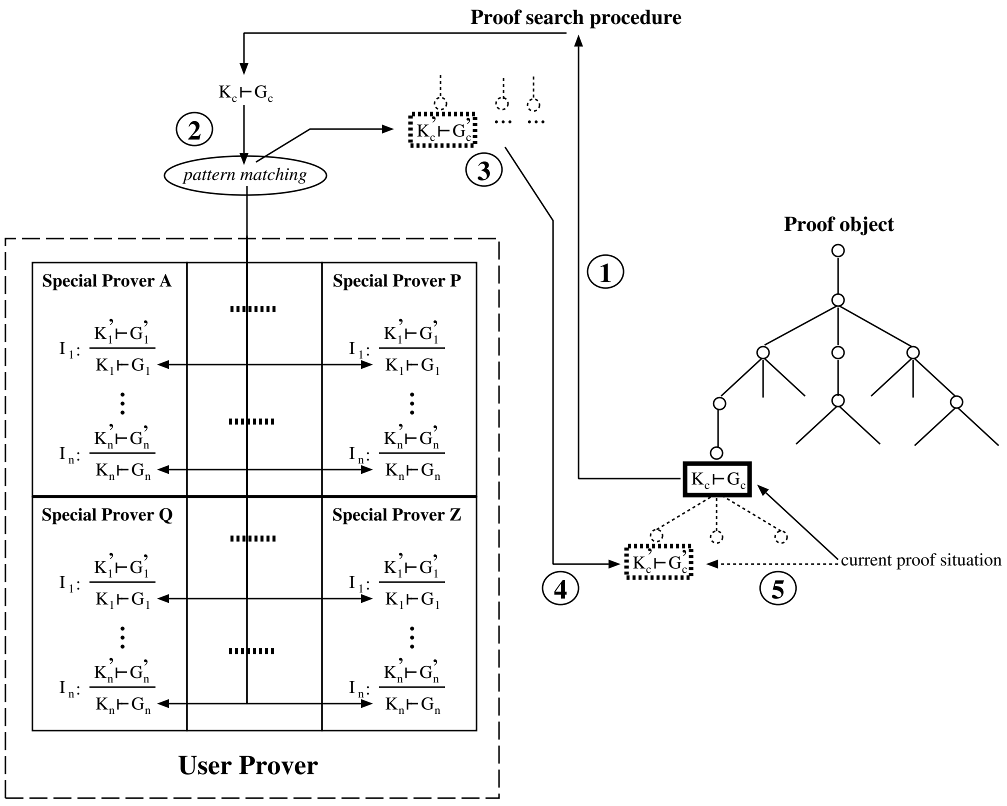

Figure 1: Proof search and prover composition

The *Theorema* user interface provides the command

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

which lets *Theorema* 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 configuration grid (see Figure 1) of *special provers* and then passes control to *Theorema*'s *global proof search procedure*. The proof search procedure manipulates the *global proof object* with the help of the user prover chosen in the Prove-call. 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 containing the current proof situation given by the user. 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 inference rules found by the proof search procedure in the special provers. Figure 1 explains 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 in order to reduce the current proof situation. Inference rules are implemented as Mathematica functions with goal, knowledge base, and "additional facts" (see Section 4) as input and a new node to be inserted into the proof tree as output. In this phase, the setup of the special provers 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 matching $K_c \vdash G_c$ 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 / top to bottom. From each applicable prover, the first rule matching $K_c \vdash G_c$ will also be selected. The implementation of the user prover arranges the special provers in the grid, 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 choice of inference rules.
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 left-most new branch.

For details on the organization of the proof search within *Theorema* we refer to (Tomuta 1998). A concrete example demonstrating the process shown in Figure 1 will be given in Section 4.1. In its current status, the proof search is completely automated with no possibility for user-interaction. We are investigating possibilities how to incorporate user-interaction into the existing proof-search environment for future versions of the *Theorema* system, see (Piroi & Jebelean 2002). In the sequel, 'set theory prover' will refer to the user prover for set theory. In the subsequent sections, we will describe only new special provers that have been developed for set theory. The set theory prover utilizes, of course, also other special provers available in *Theorema*, e.g. `BasicND` for basic 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 & Vasaru 2000), (Vasaru-Dupré 2000), and (Windsteiger 2001*a*).

4. 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 language constructs 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.

4.1. Inference Rules used in STP

Set theory specific goal reduction is implemented as a special prover named `STP`. The inference rules 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 '∈'. 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`.

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

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

$$\mathrm{STP}[\bullet\mathrm{lf}[\texttt{"G"}, a \in \{x \underset{x \in s}{\mid} x < 10\}, \bullet\mathrm{finfo}[]], \bullet\mathrm{asml}[\mathrm{K}], \mathrm{af}] \qquad (20)$$

where •lf[...] represents the proof-goal (labelled "G"), •asml[K] is the current knowledge base, and 'af' are the additional facts containing among others `STP`'s local proof context. Note, that this is *not* how the *user* needs to call a prover! The actual application of the special prover is based on internal data structures such as •lf[...] or •finfo[...], which are built-up automatically during the proof search. The interface between the user and the internal data structures as they show up above is provided through the user prover and the Prove-call as shown in Section 3.

For readers familiar with the Mathematica programming language we show part of the actual implementation of the inference rule 'MembershipSeparation' as a Mathematica function[5]

$$\text{STP}[\ \bullet\text{lf}[\ \text{l}\_,\ t\_ \in \{\underset{y\_\in S\_}{y\_ \mid} P\_\},\ \text{i}\_],\ \text{a}\_\bullet\text{asml},\ \text{af}\_]:=\ \langle \textit{program} \rangle$$

The pattern in this function definition obviously matches the function call (20), thus, the rule 'MembershipSeparation' applies, and the Mathematica program ⟨*program*⟩ will return the new node

{"AndNode",
  {"MembershipSeparation", •usedFormulae["G"],
    •generatedFormulae[•lf["G'", $a \in s \wedge a < 10$, •finfo[]]]},
  {{"ProofSituation", •lf["G'", $a \in s \wedge a < 10$, •finfo[]], •asml[K], af}},
  {}, {}, "pending"}

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

In order to prove (G) we have to show:
(G') $a \in s \wedge a < 10$.

The correctness of the inference rule 'MembershipSeparation' follows immediately from consequence (1) of the separation axiom 2.1. 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[6]. An example of such a rule is the elimination of the union-quantifier in

[5]The *Theorema* language parser is applied to formula expressions in a prover program before the program is processed further. This allows to use nicely formatted expressions such as '$t\_ \in \{\underset{y\_\in S\_}{y\_ \mid} P\_\}$' in a program, which hides internal data structures to some extent from the prover programmer.

[6]Ideally, the *Theorema* Predicate Logic Prover should be capable of producing these proofs when having the definitions from Section 2 in its knowledge base. The correctness assertion for an inference rule is, however, always higher-order. Moreover, some of the inference rules, e.g. 'MembershipSeparation', require formula manipulations such as variable substitution available in the object language. In its current status, we neither have a higher-order predicate prover nor does the *Theorema* language provide the necessary language constructs on the object level!

the goal. Simply using abbreviation (18) would lead to an inference rule

$$\textbf{Membership-U}: \frac{K \;\vdash\; t \in \bigcup\{S_x \underset{x\in s}{\mid} C_x\}}{K \;\vdash\; t \in \bigcup_{\substack{x\in s\\ C_x}} S_x}$$

The STP prover, however, implements the rule

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

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

Proof: Assume $\underset{x\in s}{\exists} (t \in S_x \wedge C_x)$, 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

$$\underset{z}{\exists} (\underset{x\in s}{\exists}\; t \in z \wedge C_x \wedge z = S_x) \; . \tag{21}$$

Separating the quantifiers in (21) gives $\underset{z}{\exists} (t \in z \wedge \underset{x\in s}{\exists} (C_x \wedge z = S_x))$, which, by (2), is equivalent to $\underset{z}{\exists} (t \in z \wedge z \in \{S_x \underset{x\in s}{\mid} C_x\})$. By (15) this is equivalent to $t \in \bigcup\{S_x \underset{x\in s}{\mid} C_x\}$, thus $t \in \bigcup_{\substack{x\in s\\ C_x}} S_x$ by (18). □

As special rules for membership, we provide e.g. for $n \geq 2$

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

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

Both set inclusion and set equality reduce, by Definition 2.1, to membership. Additionally, we provide special rules for special cases in order to reduce the search depth in the proof search procedure, e.g.

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

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

(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 like e.g.

$$\textbf{EmptySetSubset}: \frac{\text{proved}}{K \vdash \emptyset \subseteq \{\underset{x}{T_x \mid P_x}\}}$$

$$\textbf{EqualsEmptySet}: \frac{K \;\vdash\; \neg P_{x \to x_0}}{K \;\vdash\; \{\underset{x}{T_x \mid P_x}\} = \emptyset} \quad \text{where } x_0 \text{ is some new constant.}$$

For more details and a complete listing of inference rules used in STP we refer to (Windsteiger 2001*a*).

4.2. The Structure of STKBR

The special prover STKBR (for Set Theory Knowledge Base Rewriting) uses a level saturation technique, see also (Konev & Jebelean 2000), to infer *new knowledge* from the knowledge base using inference rules for set theory specific language constructs. It differs from most of the other special provers in the *Theorema* system in that it does not implement inference rules as *separate* Mathematica functions differing in the argument patterns. This "classic" implementation scheme for *Theorema* special provers as described in Section 3 is not suitable for an efficient implementation of a level saturation mechanism. Since it would allow each inference rule to infer only one new formula, it would result in massive growth of the required search depth and a considerable administrative overhead in the proof search.

The STKBR prover, instead, is just *one Mathematica function* implementing a mechanism that uses inference rules *in parallel* in order to generate *all possible new formulae during only one application.* It is considered to be applicable to the current proof situation as soon as new formulae occur in the knowledge base compared to the previous run. This check is done with the help of an entry in the local proof context passed among the "additional facts", see Section 3. The saturation of the current knowledge level happens in two phases:

1. New formulae are simplified using built-in semantic knowledge available in the *Theorema* language semantics[7], see also STC in Section 5. Application of built-in semantics can be suppressed through a user option in the Prove-call.
2. New knowledge is *inferred* from the simplified knowledge base. Inference rules as used in STKBR are implemented as Mathematica functions taking formulae as parameters returning some internal data structure containing

[7]Here we see that STKBR contributes to both the P- and the C-phase, hence, we should not call it a pure proving unit! For reasons of efficiency we allowed this mixture of P- and C-phase in *one* special prover in the current implementation.

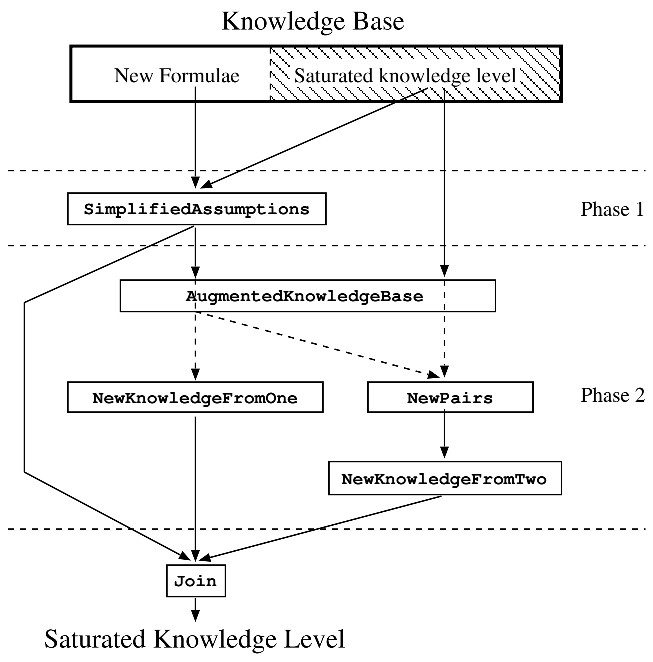

Figure 2: Schematic flow of the `STKBR` level saturation

the new formulae inferred from the given ones together with additional information for later proof text generation. These inference rules are grouped into two groups,

- *Group One* containing rules for inferring new knowledge from *one* known formula and
- *Group Two* containing rules for inferring new knowledge from *two* known formulae.

Matching rules from Group One are applied to the simplified new formulae, matching rules from Group Two are applied to all new pairs of formulae containing at least one new formula[8].

All formulae generated during these two phases make up the new knowledge base and a new proof situation is inserted into the proof object. Since the new proof situation contains all knowledge, that can be made available at that point, we call it a *saturated knowledge level.* Finally, the formula labels contained in the saturated level are stored in the local proof context being accessible in the next saturation run.

The schematized flow of `STKBR` level saturation mechanism is shown in Figure 2. Phase 1 is accomplished by calling the function 'SimplifiedAssumptions'

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

with two arguments: the entire knowledge base and a list of labels 'sat' specifying the saturated knowledge level. Each new formula from the knowledge base is sent through the function 'EvaluateFromProve', which *computes* a simplified version of the formula w.r.t. semantic knowledge from the *Theorema* language. 'EvaluateFromProve' is the function used also in the `STC` module for *goal simplification by computation*, see Section 5. It is based on the function 'EvaluateStandard', which is the basic evaluation function for computations using *Theorema* semantics, which is used also by `Compute`, the top-level user function to initiate computations. This guarantees utmost coherence between all computations happening in the *Theorema* system, be it on the user level by calling `Compute`, be it on the prover level by doing simplifications on the goal or on the knowledge base. Formulae that cannot be simplified as well as formulae from the previous saturation level leave phase 1 unchanged.

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 One of inference rules componentwise to *all new assumptions*, 'NewKnowledgeFromTwo' applies Group Two of inference rules to *all new pairs* that can be formed using the new assumptions. The results joined with the simplified knowledge base resulting from phase 1 give the saturated knowledge level.

The inference rules applied by `STKBR` can more or less be read off the Definitions in Section 2. Again they are grouped into rules for membership, inclusion, and set equality. For each "kind of set" introduced in Section 2 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.

4.2.1. Rule Locking

Rule locking helps to prevent cycles 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. Unrestricted use of inverse rules immediately results in a cycle in the proof search. Rule locking allows 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 would automatically prevent rule **I** from being applied to F and vice versa.

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, for which inputs an inverse rule must be locked. As a special case, inference rules may 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.

Rule locking utilizes `STKBR`'s local proof context to store this type of information on the proof progress. More precisely, the local proof context contains a lookup table of so-called *rewrite exceptions*. The rewrite exception for an inference rule **I** from Group One is a list of formula labels. The inference rule **I** will only apply to a formula if **I**'s rewrite exception does not contain the formula label. The rewrite exception for an inference rule **I** from Group Two is a list of pairs of formula labels. The inference rule **I** will only apply to two formulae if **I**'s rewrite exception does not contain the pair of formula labels.

5. 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 2001*a*). In particular, *integer ranges* and *set ranges* for finite sets are algorithmic ranges, which lead to finite sets when used in combination with the set quantifiers.

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

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 is

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

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

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

Basically, when the STC prover applies to a proof situation, one proof step consists of calling the evaluation function 'EvaluateFromProve' (see also Section 4.2) and, in case the result differs from the original form, of adding a node to the proof object, from which the effect and a complete trace of the computation can be displayed. Again, the use of 'EvaluateFromProve' preserves coherence with STKBR and Compute. Many details on combining computation with proving can be found in (Windsteiger 2001*a*).

6. STS: The Set Theory Solving Unit

The special prover STS collects inference rules for eliminating existential quantifiers in the proof goal[9]. Methods used for instantiating existential goals range from *matching* against formulae in the knowledge base, over *unification* and *introduction of solve constants* until the use of Mathematica's 'Solve' function. We present only one typical inference rule from STS.

$$\textbf{IntroSolveConstant}: \frac{K \;\vdash\; Q_{y\to y^*} \wedge \underset{x\in s}{\exists} (P_x \wedge y^* = T_x) \wedge R_{y\to y^*}}{K \;\vdash\; \underset{y}{\exists} (Q_y \wedge y \in \{T_x \underset{x\in s}{\mid} P_x\} \wedge R_y)}$$

where Q_y and R_y are possibly empty conjunctions of formulae and y^* is a *solve constant.*

A solve constant[10] is some constant, whose value is not yet known at the time when it is introduced. Solve constants allow to eliminate existential quantifiers and delay their instantiation to a later phase of the proof. For the proof to succeed, *the values for 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. A solve constant differs from a Skolem constant in that it is a placeholder for a concrete term that needs to be determined during the proof whereas a Skolem constant is a new constant about which we do not know anything. 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 formula in order to determine the concrete value of the solve constants. Applying this strategy reduces proving to solving over various domains, and it offers the possibility to benefit from the great advances that have been accomplished in developing powerful solving methods in computer algebra.

The inference rule described above might appear random. It is part of STS

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

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

since it applies exactly to proof situations left after expanding membership in special unions, namely goals of the form $t \in \bigcup\{T_x \underset{x\in s}{|} P_x\}$. It can be observed in many examples involving proof goals of this form (see in particular the example in Section 7.4) that this strategy leads to a well-strucured 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 2001*a*). Note, that the solve constant already appears in isolated position, so that it can immediately be expressed by the ground term T_x as soon as P_x is 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. Some examples of different instantiation techniques will be shown in Section 7.

7. Comparison and Examples

7.1. 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 the 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 7.2. Table 1 shows a comparison of the computing times to state-of-the-art theorem provers as they performed in CASC-18[11], 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 & 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).

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

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

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.

Table 2: *Theorema* vs. 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

The comparison of computing times to other systems is, however, difficult, since the *Theorema* system is implemented in the programming language of Mathematica. There is no possibility of compiling Mathematica programs, hence, comparing the run-time of interpreted code to computing times of optimized compiled machine code does not tell much. One can observe though in practice that the proofs generated by the set theory prover contain only few failing branches, and each branch contains only few useless formulae.

We want to emphasize, that the absolute computation times are not our main focus. We are much more interested in having automatically generated "nice proofs" that are easily understandable for a human reader. As a consequence, we aim at designing provers that apply inference rules in a smart way without too many failing branches during the proof search.

7.2. 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. The proofs shown in this section are taken from the system comparison in Section 7.1, the label of the goal formula refers to the example name as given in Tables 1 and 2. The proofs below 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 simplification of the "raw proof object" produced by the set theory prover is a standard post-processing feature available in the *Theorema* system and the timings given in Section 7.1 include also the time needed for proof simplification. As already mentioned above, most of the proofs generated by the *Theorema* set theory prover succeed in the first branch that is tried and the prover generates only few unnecessary formulae.

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 being intermixed with Mathematica input and output expressions as well as graphics. Some of the features of the *Theorema* standard proof presentation utilize 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 "clickable" and show the entire referenced formula in a popup-window.
- Proof branches are organized in a hierarchy that reflects the structure of the proof. Collapsing entire proof-branches by mouse-click allows to quickly browse through the structure of a proof and easily "zoom into" the interesting proof parts or skip uninteresting proof parts, respectively.

Proof: (SET010) $B \setminus (C \cap D) = (B \setminus C) \cup (B \setminus D)$.

$\subseteq$: We assume

(1) $B1 \in B \setminus (C \cap D)$

and show

(2) $B1 \in (B \setminus C) \cup (B \setminus D)$

From (1) we can infer

(3) $B1 \in B$

(4) $B1 \notin C \cap D$.

From (4) we can infer

(5) $B1 \notin C \vee B1 \notin D$.

In order to prove (2) we may assume

(6) $B1 \notin B \setminus D$

and show

(7) $B1 \in B \setminus C$

From (6) we can infer

(8) $B1 \notin B \vee B1 \in D$.

We have to prove (7), thus, we first show:

(9) $B1 \in B$:

Formula (9) is true because it is identical to (3).
For proving (7) it still remains to show

(10) $B1 \notin C$:

From (3) and (8) we obtain

(11) $B1 \in D$.

From (5) and (11) we obtain

(12) $B1 \notin C$.

Formula (10) is true because it is identical to (12).
$\supseteq$: Now we assume (2) and show (1).
From (2) we can infer

(13) $B1 \in B \setminus C \vee B1 \in B \setminus D$.

We have to prove (1), thus, we first have to show

(14) $B1 \in B$.

(We skip the proof of (14). It succeeds by case distinction based on (13).)
For proving (1) it still remains to show:

(15) $B1 \notin C \cap D$.

Assume

(16) $B1 \in C \cap D$

From (16) we can infer

(17) $B1 \in C$

(18) $B1 \in D$.

Case (13.1) $B1 \in B \setminus C$:
From (13.1) we can infer

(19) $B1 \in B$

(20) $B1 \notin C$.

(17) and (20) are contradictory.
Case (13.2) $B1 \in B \setminus D$:
From (13.2) we can infer

(21) $B1 \in B$

(22) $B1 \notin D$.

(18) and (22) are contradictory. □

Set theory specific reasoning requires only rules from STP and STKBR in this example. The case distinction based on formula (13) is done by the special prover `CDP`, the rest is basic predicate logic from `BasicND`.

Proof: (SET171) $\underset{A,B,C}{\forall} (A \cup B) \cap (A \cup C) = A \cup (B \cap C)$

For proving (SET171) we take all variables arbitrary but fixed and prove:

(1) $(A_0 \cup B_0) \cap (A_0 \cup C_0) = A_0 \cup (B_0 \cap C_0)$.

$\subseteq$: We assume

(2) $A1_0 \in (A_0 \cup B_0) \cap (A_0 \cup C_0)$

and show:

(3) $A1_0 \in A_0 \cup (B_0 \cap C_0)$.

From (2) we can infer

(5) $A1_0 \in A_0 \cup B_0$,

(6) $A1_0 \in A_0 \cup C_0$.

From (5) we can infer

(7) $A1_0 \in A_0 \vee A1_0 \in B_0$.

From (6) we can infer

(8) $A1_0 \in A_0 \vee A1_0 \in C_0$.

In order to prove (3) we may assume

(10) $A1_0 \notin B_0 \cap C_0$

and show:

(9) $A1_0 \in A_0$.

(Note, that in all other cases the formula (3) trivially holds!)
From (10) we can infer

(11) $A1_0 \notin B_0 \vee A1_0 \notin C_0$.

We prove (9) by case distinction using (11).
Case (11.1) $A1_0 \notin B_0$:
From (11.1) and (7) we obtain by resolution

(12) $A1_0 \in A_0$.

Formula (9) is true because it is identical to (12).
Case (11.2) $A1_0 \notin C_0$:
From (11.2) and (8) we obtain by resolution

(13) $A1_0 \in A_0$.

Formula (9) is true because it is identical to (13).
$\supseteq$: Now we assume

(3) $A1_0 \in A_0 \cup (B_0 \cap C_0)$

and show:

(2) $A1_0 \in (A_0 \cup B_0) \cap (A_0 \cup C_0)$.

From (3) we can infer

(14) $A1_0 \in A_0 \vee A1_0 \in B_0 \cap C_0$.

We prove (2) by splitting up the intersection into its individual components:
We have to prove:

(15) $A1_0 \in A_0 \cup B_0$.

In order to prove (15) we may assume

(18) $A1_0 \notin B_0$

and show:

(17) $A1_0 \in A_0$.

(Note, that in all other cases the formula (15) trivially holds!)
We prove (17) by case distinction using (14).
Case (14.1) $A1_0 \in A_0$:
Formula (17) is true because it is identical to (14.1).
Case (14.2) $A1_0 \in B_0 \cap C_0$:
From (14.2) we can infer

(20) $A1_0 \in B_0$,

(21) $A1_0 \in C_0$.

Formula (17) is proved because (20) and (18) are contradictory.
We have to prove:

(16) $A1_0 \in A_0 \cup C_0$.

In order to prove (16) we may assume

(23) $A1_0 \notin C_0$

and show:

(22) $A1_0 \in A_0$.

(Note, that in all other cases the formula (16) trivially holds!)
We prove (22) by case distinction using (14).
Case (14.1) $A1_0 \in A_0$:
Formula (22) is true because it is identical to (14.1).
Case (14.2) $A1_0 \in B_0 \cap C_0$:
From (14.2) we can infer

(25) $A1_0 \in B_0$,

(26) $A1_0 \in C_0$.

Formula (22) is proved because (26) and (23) are contradictory. □

Note that the *Theorema* set theory prover comes up with this straight-forward proof in only 4 seconds whereas all the systems from the CASC competition failed in this example after 300 seconds.

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) $\underset{B2}{\exists}\ \mathit{B2} \in B_0 \land \mathit{B2} \in C_0$.

Formula (13) is simplified to

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

From (16) we can infer

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

From (17) we can infer

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

We prove (15) by case distinction using (11).
Case (11.1) $\mathit{B1}_0 \in C_0$:
Now, let $\mathit{B2} := \mathit{B1}_0$. Thus, for proving (15) it is sufficient to prove:

(20) $\mathit{B1}_0 \in B_0 \land \mathit{B1}_0 \in C_0$.

Proof of (20.1) $\mathit{B1}_0 \in B_0$:
Formula (20.1) is true because it is identical to (8).
Proof of (20.2) $\mathit{B1}_0 \in C_0$:
Formula (20.2) is true because it is identical to (11.1).
Case (11.2) $\mathit{B1}_0 \in D_0$:
From (8), by (18), we obtain:

(29) $\mathit{B1}_0 \notin D_0$.

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

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

and show

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

Formula (4) means that we have to show that

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

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

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

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 for example (SET624) 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.

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 notion $f :: A \to B$ denotes 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. *Theorema* offers also the concept of a function from A to B in extensional form (written $f : A \to B$), where a function is a certain subset of $A \times B$. The *Theorema* set theory prover can handle both and it 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. The use of these inference rules can be suppressed by an option in the Prove-call. In this case, the knowledge base needs to contain the definition of surjectivity. The proof, 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 seconds, which is about the same time that the "Saturate"-prover needs. Note, however, that all the CASC provers fail in (SET716).

We now want to present two variants of a proof for (SET751). The first one uses intensional function notation as already shown in the previous example, whereas the second uses the extensional function concept typically used in set theory[12]. The computing times do not essentially differ between the two variants.

Proof: (SET751)

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

under the assumption:

(Definition (Image)) $\underset{f,X}{\forall} \mathrm{image}[f, X] := \left\{ f[x] \underset{x}{|} x \in X \right\}$.

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) $\mathrm{image}[f_0, X_0] \subseteq \mathrm{image}[f_0, Y_0]$.

For proving (2) we choose

[12]We chose to present both variants of the proof in order to demonstrate the flexibility of the prover. The set theory prover does not require the user to force all of mathematics into set representation.

(3) $\mathit{f1}_0 \in \mathrm{image}[f_0, X_0]$,

and show:

(4) $\mathit{f1}_0 \in \mathrm{image}[f_0, Y_0]$.

From (1.3) we can infer

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

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

(11) $\mathit{f1}_0 \in \left\{ f_0[x] \underset{x}{|}\ x \in X_0 \right\}$.

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

(12) $\mathit{x1}_0 \in X_0$,

(13) $\mathit{f1}_0 = f_0[\mathit{x1}_0]$.

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

$f_0[\mathit{x1}_0] \in \mathrm{image}[f_0, Y_0]$,

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

(19) $f_0[\mathit{x1}_0] \in \left\{ f_0[x] \underset{x}{|}\ x \in Y_0 \right\}$.

In order to prove (19) we have to show

(20) $\underset{x}{\exists}\ x \in Y_0 \wedge f_0[\mathit{x1}_0] = f_0[x]$.

Since $x := \mathit{x1}_0$ solves the equational part of (20) it suffices to show

(21) $\mathit{x1}_0 \in Y_0$.

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

(22) $\mathit{x1}_0 \in X_0$.

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

The instantiation of the existential variable x in the proof goal (20) was done by solving the equation $f_0[\mathit{x1}_0] = f_0[x]$ for x, which is done by a call to the Mathematica function `Solve` for solving (systems of) equations.

Proof: (SET751)

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

under the assumption:

(Definition (Image)) $\underset{f,X}{\forall}\ \mathrm{image}[f,X] := \left\{ \underset{y}{y} \middle| \underset{x}{\exists}\ x \in X \wedge \langle x,y \rangle \in f \right\}$.

We assume

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

and show

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

For proving (2) we choose

(3) $\mathit{f1}_0 \in \mathrm{image}[f_0, X_0]$,

and show:

(4) $\mathit{f1}_0 \in \mathrm{image}[f_0, Y_0]$.

From (1.3) we can infer

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

Formula (4), using (Definition (Image)), is implied by:

(11) $\mathit{f1}_0 \in \left\{ \underset{y}{y} \middle| \underset{x}{\exists}\ x \in Y_0 \wedge \langle x,y \rangle \in f_0 \right\}$.

In order to prove (11) we have to show:

(12) $\underset{x}{\exists}\ x \in Y_0 \wedge \langle x, \mathit{f1}_0 \rangle \in f_0$.

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

(14) $\mathit{f1}_0 \in \left\{ \underset{y}{y} \middle| \underset{x}{\exists}\ x \in X_0 \wedge \langle x,y \rangle \in f_0 \right\}$.

From (14) we can infer

(15) $\underset{x}{\exists}\ x \in X_0 \wedge \langle x, \mathit{f1}_0 \rangle \in f_0$.

By (15) we can take appropriate values such that:

(16) $x_0 \in X_0 \wedge \langle x_0, \mathit{f1}_0 \rangle \in f_0$.

Now, let $x := x_0$. Thus, for proving (12) it is sufficient to prove:

(19) $x_0 \in Y_0 \wedge \langle x_0, \mathit{f1}_0 \rangle \in f_0$.

Proof of (19.1) $x_0 \in Y_0$:
Formula (19.1), using (8), is implied by:

(20) $x_0 \in X_0$.

Formula (20) is true because it is identical to (16.1).

Proof of (19.2) $\langle x_0, f1_0 \rangle \in f_0$:
Formula (19.2) is true because it is identical to (16.2). □

7.3. Theory-specific Knowledge Built into the Prover

In this section, we want to show two examples that demonstrate, how set theory specific knowledge is built into the prover. The set theory prover does not only apply axioms of ZF, but it uses in addition some theorems that are valid in ZF. 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 can prove the entire build-up of set theory based on 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 is a special inference rule in `STKBR` that 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 by built-in semantic knowledge about finite sets, in particular, the empty set.

The following example is taken from the case study on the mutilated checkerboard, see (McCarthy 1964), (McCarthy 1995), (Windsteiger 2001*b*), (Windsteiger 2001*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). 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))

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

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

We assume

(2) $\text{domino-on-board}[X_0]$,

and show

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

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

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

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

From (5.1) we can infer

(6) $\mathit{X1}_0 \in X_0$,

(7) $\mathit{X1}_1 \in X_0$,

(8) $\mathit{X1}_0 \neq \mathit{X1}_1$.

Now, let $y := \mathit{X1}_0$. Thus, for proving (3.2) it is sufficient to prove:

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

Now, let $x := \mathit{X1}_1$. 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) $\mathit{X1}_1 \in X_0$:
Formula (16.1) is true because it is identical to (7).
Proof of (16.2) $\mathit{X1}_0 \in X_0$:
Formula (16.2) is true because it is identical to (6).
Proof of (16.3) adjacent$[\mathit{X1}_1, \mathit{X1}_0]$:
Formula (16.3), using (5.3), is implied by:

(17) $\mathit{X1}_0 \in X_0 \wedge \mathit{X1}_1 \in X_0 \wedge \mathit{X1}_1 \neq \mathit{X1}_0$.

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

(18) $\mathit{X1}_0 \in X_0 \wedge \mathit{X1}_1 \in X_0$.

Proof of (18.1) $\mathit{X1}_0 \in X_0$:
Formula (18.1) is true because it is identical to (6).
Proof of (18.2) $\mathit{X1}_1 \in X_0$:
Formula (18.2) is true because it is identical to (7). □

The next example shows, how arithmetic knowledge on natural numbers provided by Mathematica is accessible for the prover.

Proof: (G) $36 \in \bigcup\limits_{i \in \mathbb{N}} \{j^2 \underset{j \in \mathbb{N}}{|} j \geq i \wedge j \leq i + 5\}$

under the assumption

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

In order to show (G) we have to show

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

In order to prove (1) we have to show

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

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

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

Using available computation rules we evaluate (3):

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

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

(5) $6 > 1$.

Using available computation rules we evaluate (5):

(6) *True* .

Formula (6) 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. The Mathematica 'Solve' function is used internally to solve the quadratic equation $36 = j^2$, which finds two solutions $j = -6$ and $j = 6$. 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. Simplifications from (3) to (4) and from (5) to (6) were made using available semantic knowledge by STC ($6 \in \mathbb{N}$ and $6 > 1$, respectively) and, finally, reduction from (4) to (5) and the detection of proof success were made by standard predicate logic inference rules. 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 (4) without any additional knowledge. We will investigate necessary solving techniques in future work.

7.4. Theory Exploration vs. 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 2001*a*), 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 very often does. We present here the proof of (SET770) using the two auxiliary propositions (equal classes) and (not in distinct class) in the knowledge base. The computing time for the proof is 5.8 seconds on a 2000 MHz Intel P4, the proofs of the auxiliary propositions take 8.5 and 8.6 seconds, respectively.

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

Now, let $x := x\mathit{3}_0$. Thus, for proving (12) it is sufficient to prove:

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

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) $x\mathit{3}_0 \in \text{class}[x_0, R_0]$:
Formula (13.3) is true because it is identical to (7).
Proof of (13.4) $x\mathit{3}_0 \in \text{class}[y_0, R_0]$:
Formula (13.4) is true because it is identical to (8).

□

The same case study has been carried ot for an intensional concept of relations. Similar to the intensional concept of a function described in Section 7.2, 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.

Proof:

(Lemma (union inverse factor set)) $\underset{A}{\forall}\ \text{is-reflexive}_A[\sim] \Rightarrow \bigcup \text{factor-set}_\sim[A] = A$,

under the assumptions:

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

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

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

We assume

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

and show

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

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

$$\bigcup \{\text{class}_{A_0,\sim}[x] \underset{x}{|} x \in A_0\} = A_0 ,$$

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

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

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

(4) $\underset{x}{\forall} (x \in A_0 \Rightarrow x \sim x)$.

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

(5) $x1_0 \in \bigcup\{\{a \underset{a}{\mid} a \in A_0 \wedge a \sim x\} \underset{x}{\mid} x \in A_0\}$

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) $x2_0 \in \{\{a \underset{a}{\mid} a \in A_0 \wedge a \sim x\} \underset{x}{\mid} x \in A_0\}$,

(8) $x1_0 \in x2_0$.

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

(9) $a1_0 \in A_0$,

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

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

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

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) $x1_0 \in \bigcup\{\{a \underset{a}{\mid} a \in A_0 \wedge a \sim x\} \underset{x}{\mid} x \in A_0\}$

In order to show (5) we have to show

(29) $\underset{x4}{\exists}\, x1_0 \in x4 \wedge x4 \in \{\{a \underset{a}{\mid} a \in A_0 \wedge a \sim x\} \underset{x}{\mid} x \in A_0\}$.

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

(30) $\mathit{x1}_0 \in \mathit{x4}^* \wedge \underset{x}{\exists} (x \in A_0 \wedge \mathit{x4}^* = \{a \underset{a}{|} a \in A_0 \wedge a \sim x\})$.

Since (6) matches a part of (30) we try to instantiate, i.e. let now $x := \mathit{x1}_0$. Thus, by (30), we choose $\mathit{x4}^* := \{a \underset{a}{|} a \in A_0 \wedge a \sim \mathit{x1}_0\}$.

Now, it suffices to show

(32) $\mathit{x1}_0 \in A_0 \wedge \mathit{x1}_0 \in \{a \underset{a}{|} a \in A_0 \wedge a \sim \mathit{x1}_0\}$.

Proof of (32.1) $\mathit{x1}_0 \in A_0$:
Formula (32.1) is true because it is identical to (6).
Proof of (32.2) $\mathit{x1}_0 \in \{a \underset{a}{|} a \in A_0 \wedge a \sim \mathit{x1}_0\}$:

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

(33) $\mathit{x1}_0 \in A_0 \wedge \mathit{x1}_0 \sim \mathit{x1}_0$.

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

(34) $\mathit{x1}_0 \in A_0$.

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 `BasicND`, see (1) and (2).
- In a C-phase, `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 $\underset{x4}{\exists}\ x1_0 \in x4 \wedge x4 \in \{T_x \underset{x}{|} P_x\}$, which can be handled by rule 'IntroSolveConstant' from Section 6. Thus, the existential quantifier is eliminated by introducing the solve constant $x4^*$, and the expansion of the inner

membership $x4^* \in \{T_x \underset{x}{|} P_x\}$ 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 subformula $x4^* = \ldots$ 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. 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, the prover follows the PCS paradigm for structuring automated provers used already earlier in other provers provided in *Theorema*. 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. This means, we aim at proving theorems in the flavor of the examples shown in Sections 7.3 and 7.4 much more than most of the examples from the TPTP library. The proofs shown in Section 7 demonstrate that the *Theorema* set theory prover is able to produce proofs of non-trivial theorems in a human-comprehensible style. In average, the computing times for automatically generating the formatted proofs are comparably low. Further improvements of the prover will have to go into the rewriting part of the prover in order to handle conditional rewriting more efficiently.

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 done in order 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* R. Nieuwenhuis & A. Voronkov, eds, 'LPAR 2001', number 2250 *in* 'LNAI', Springer Verlag Berlin Heidelberg, pp. 309–319.

Bernays, P. & Fraenkel, A. (1968), *Axiomatic Set Theory*, Studies in Logic and the Foundations of Mathematics, 2 edn, North-Holland Publishing Company.

Buchberger, B. (1985), Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory, *in* N. Bose, 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. Volume I: A Logical Basis for Mathematics. Lecture notes for the mathematics course in the first and second semester at the Fachhochschule for Software Engineering in Hagenberg, Austria.

Buchberger, B. (1999), Theory Exploration Versus Theorem Proving, *in* A. Armando & T. Jebelean, eds, 'Electronic Notes in Theoretical Computer Science', Vol. 23-3, Elsevier, pp. 67–69. CALCULEMUS Workshop, University of Trento, Trento, Italy.

Buchberger, B. (2000), Computer-assisted proving by the pcs-method, *in* M. Hazewinkel, ed., 'Proceedings of the Workshop on Constructive Algebra', LNCS, Springer. To appear.

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', number 33 *in* 'LNCS', Springer Verlag, Berlin, pp. 134–183.

Ebbinghaus, H. (1979), *Einführung in die Mengenlehre*, 2 edn, Wissenschaftliche Buchgesellschaft Darmstadt. ISBN 3-534-06709-6.

Formisano, A. (2000), Theory-based resolution and automated set reasoning, PhD 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)', LNCS ?, Miami, USA, pp. ?–?

Konev, B. & Jebelean, T. (2000), Combining Level-Saturation Strategies and Meta-Variables for Predicate Logic Proving in Theorema, *in* 'Proceedings of IMACS ACA 2000, St.Petersburg, Russia'.

Kriftner, F. (1998), Theorema: The Language, *in* B. Buchberger & T. Jebelean, 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* R. Matuszewski, ed., 'The QED Workshop II', Warshaw University, pp. 25–26. Technical Report L/1/95.

Piroi, F. & Jebelean, T. (2002), Interactive Proving in Theorema, *in* 'Collected Abstracts. Ninth Workshop on Automated Reasoning, AISB'02', Imperial College of Science, Technology and Medicine, University of London. England.

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.

Takeuti, G. & Zaring, W. (1971), *Introduction to Axiomatic Set Theory*, Graduate Texts in Mathematics 1, Springer Verlag. ISBN 0-387-05302-6.

Tomuta, E. (1998), An Architecture for Combining Provers and its Applications in the Theorema System, PhD 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, PhD thesis, RISC Institute. RISC report 00-19.

Windsteiger, W. (2001*a*), A Set Theory Prover in Theorema: Implementation and Practical Applications[13], PhD thesis, RISC Institute.

Windsteiger, W. (2001*b*), On a Solution of the Mutilated Checkerboard Problem using the Theorema Set Theory Prover[14], *in* S. Linton & R. Sebastiani, eds, 'Proceedings of the Calculemus 2001 Symposium'.

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

[14]http://www.risc.uni-linz.ac.at/people/wwindste/Public/Reports/Calculemus01