SAD as a mathematical assistant — how should we go from here to there?

Alexander Lyaletski [a], Andrey Paskevich [a,b], Konstantin Verchinine [b]

[a] *Faculty of Cybernetics, Kyiv National Taras Shevchenko University, Ukraine*

[b] *Math-Info Department, Paris 12 "Val de Marne" University, France*

Abstract

The System for Automated Deduction (SAD) is developed in the framework of the Evidence Algorithm research project and is intended for automated processing of mathematical texts. The SAD system works on three levels of reasoning: (a) the level of text presentation where proofs are written in a formal natural-like language for subsequent verification; (b) the level of foreground reasoning where a particular theorem proving problem is simplified and decomposed; (c) the level of background deduction where exhaustive combinatorial inference search in classical first-order logic is applied to prove end subgoals.

We present an overview of SAD describing the ideas behind the project, the system's design, and the process of problem formalization in the fashion of SAD. We show that the choice of classical first-order logic as the background logic of SAD is not too restrictive. For example, we can handle binders like Σ or *lim* without resort to second order or to a full-powered set theory. We illustrate our approach with a series of examples, in particular, with the classical problem $\sqrt{2} \notin \mathbf{Q}$.

1 Introduction

In the early 1960s the Academician V. Glushkov initiated a series of researches aimed at the development of a computer system that would be able to aid a working mathematician to verify formally long but in some sense "evident" proofs (e.g. routine algebraic transformations superposed with applications of a number of definitions).

According to the initial idea three main components of the system should be: an inference engine that implements the basic level of evidence, an extensible collection of tools that reinforce the basic engine and a powerful input language that must be close to the natural mathematical language and easy to use.

Preprint submitted to Elsevier Science *11 April 2006*

Thus, a statement is "evident" if it can be proved by the system without any intervention of a user. So the more powerful is the system, the larger the "evidence domain" is. One can hope to augment the system's power by adding new "reinforcement methods" (e.g. proof schemes, reasoning styles, etc) as well as by accumulating useful and frequently used mathematical knowledge.

The work began and an original theoretical basis for the inference engine was proposed soon, which was called the Evidence Algorithm [1]. The background logic was classical first-order logic. Note that the choice of logic predefined somehow a set-theoretical formalization style (as opposed to type-theoretical). Later, a version of the input language, called Theory Language [2], was developed. Some steps towards more intelligent system were made as well [3].

The project was really interesting and ambitious and required significant human and computer resources (two of the three authors took part in the first research team that developed the project), and the life of the project turned to be hard. It was almost abandoned in the beginning of 90-th but fortunately restored several years later (thanks to permanent efforts of one of the authors) and took the name "Evidence Algorithm" (EA) for the whole [4].

The architecture of an ideal mathematical assistant from the viewpoint of EA could be outlined as follows. A user communicates with the system with the help of texts written in the above-mentioned high-level formal input language. She or he submits a problem like "verify whether the given text is correct" or "how to prove the following statement", or "what is the given text about" and so on. The text, provided being syntactically correct, is treated by the part of the system that we call "reasoner". The reasoner analyzes the problem and formulates a series of tasks that it submits to the inference engine, a prover. If the prover succeeds, the resulting conclusion (e.g. proof trace) is given to the user and the game is over. If it fails then a kind of "morbid anatomist" makes a diagnosis and supplies it to the reasoner who tries to repair the situation. In particular, the reasoner can decide that an auxiliary statement (lemma) might be useful and start the search for those in the mathematical archives. To do that it submits a request to the archive service, we call it "librarian". After getting an answer the reasoner begins a new proof search cycle with the modified problem and the process goes on.

The user can interact with the system by playing for the reasoner, librarian, for the morbid anatomist (provided that she or he understands the internal prover's life) or for the prover itself, deciding whether a given conjecture should be considered as valid.

Below we present the current incarnation of the Evidence Algorithm, the System for Automated Deduction (SAD) [5,6], in its current state of development. We start by describing the design of the SAD system and the services it provides (Section 2). Then we give a brief introduction to ForTheL, the input language of SAD (Section 3), followed by a kind of tutorial on how to formalize "live" texts (Section 4). In Section 5, we explain the verification procedure and define the notion of a correct ForTheL text. Section 6 demonstrates how to go beyond the first-order setting imposed by the background logic of SAD: we study formalization issues for operations that bind variables. The native inference engine of SAD is described in Section 7. We conclude with a brief review of related approaches and an outline of the future work.

2 Architecture of SAD

In the domain of mathematical assistance we distinguish two main groups of problems, which we can describe as "knowledge management" and "knowledge application". The first group includes organization of libraries of mathematical facts, search for relevant information, communication with external mathematical services, and, probably, user interface. Problems from the second group may vary, depending on a mathematical assistant in question: proof planning, equation solving, or text verification.

The SAD system in its current state is focused on knowledge application problems. A mathematical archive of SAD and the tools to interact with external mathematical services are under development. Three main services provided by SAD are:

- inference search in classical first-order logic;
- theorem proving in the context of a self-contained mathematical text;
- verification of a self-contained mathematical text.

In what follows, we will talk about the inference search mode, the theorem proving mode, and the text verification mode of SAD. Each mode relies on the previous ones: theorem proving in SAD resorts to inference search if needed, and verification of a text consists in proving a number of separate conjectures in the frame of that text.

In order to explain how a mathematical problem can be formalized and processed in SAD, let us consider in more detail its verification mode. We relate the stages of this process to the parts of the SAD design scheme (Figure 1).

A user, which we call an *author*, starts by writing a ForTheL text, containing the problem itself (e.g. a theorem or a number of theorems), the relevant information (problem specific lemmas, definitions, axioms), and the common preliminary facts used throughout the text and called *preliminaries* below (e.g. general properties of sets or numbers). Any claim in the text may be

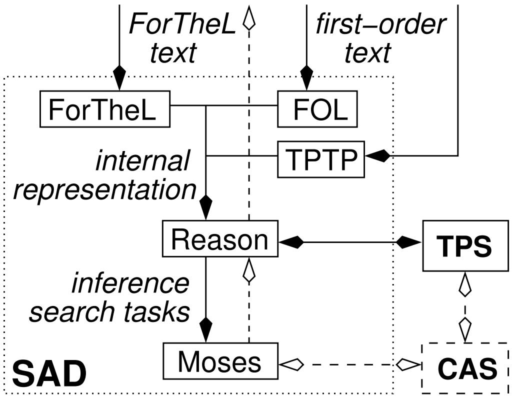

Fig. 1. The architecture of SAD

supplied with a proof. Preparation of a ForTheL text is entirely manual, no assistance is provided for this stage. In Sections 4, 5, 6 we give several examples that demonstrate how a problem written in natural language can be converted to a ForTheL text.

Now, the SAD system enters the scene. The parser module [ForTheL] analyzes the structure of the input text, defined with the help of the ForTheL markup, and the syntax of sentences. Then the parser translates the text into its internal representation. In this representation, the initial structure is made more straightforward (for example, in the proof by case analysis, the disjunction of the case hypotheses is explicitly added as a goal). Also, ForTheL statements are rendered to annotated formulas. Though the result of translation is a collection of internal data structures, it corresponds to a ForTheL text, too — not the original input text, but some straightened, *normalized* version of it.

In order to verify a submitted ForTheL text (see Section 5), the system transforms its normalized version into a series of goal statements, which must be deduced from their predecessors in the text. Thus text verification is reduced to theorem proving.

The goal statements are processed one by one by the module [Reason], the *foreground reasoner* of SAD. Given a *proof task* — a statement to prove along with the list of its logical predecessors in the text under consideration — the reasoner tries to simplify it in various ways: to decompose it to a number of smaller subtasks (e.g. to split a conjunctive goal or to perform a case analysis), to filter the set of premises, to generate suitable instantiations of relevant lemmas, to expand definitions, and so on.

Thus, the reasoner is a kind of automated heuristic based prover, supplied with a collection of proof task transformation rules. This collection is not intended to be a complete logic calculus. The purpose of the reasoner is not to find the entire proof on its own, but to prepare the inference search tasks for the *background prover* of SAD. The latter is a combinatorial automated prover in classical first-order logic, whose duty is to complete the proofs started by the reasoner. If the background prover fails to find the inference at some instant, the reasoner may continue the proof task transformation or try an alternative way, or just reject the text.

The richer the reasoner's collection of transformation rules is, the more complex proof tasks it can fulfill, the more coarse-grained and terse mathematical texts can be verified by the system. The simplest reasoner would just send the received proof task to the background prover — that was the point where the development of SAD has started. At present, the capabilities of [Reason] are still quite poor: it performs propositional goal splitting and implements a simple variant of definition expansion strategy.

The native background prover of SAD, called [Moses], is based on a special goal-driven sequent calculus, described in Section 7. The prover explores the search space using bounded depth-first search with iterative deepening and backtracking, it uses constraints and folding-up [7] to increase the efficiency of search. In order to provide SAD with equality handling, [Moses] implements a variation of Brand's modification method [8]. The original notion of admissible substitution used in the calculus allows to preserve the initial signature of the task so that accumulated unification problems (sets of equations) can be sent to a specialized solver, e.g. an external computer algebra system (**CAS**).

We call [Moses] native, because the background prover is supposed to be independent of SAD by design, so that an external theorem proving system (**TPS**), such as Otter [9], SPASS [10], or Vampire [11] could be used. We made a number of experiments with SPASS and some of them are presented in the paper (Subsection 4.3).

Being started in the inference search mode, SAD accepts the input texts written directly in first-order language. The modules [FOL] and [TPTP] are parsers that process the native first-order syntax of SAD and the TPTP-syntax, respectively. The second format is used in the well-known TPTP library that contains several thousands of logical problems for automated proof systems in classical first-order logic [12]. The SAD system can download problems and axioms directly from the web-site of the TPTP library. Of course, one can submit her or his own problem in the syntax of TPTP.

Dashed lines in Figure 1 reflect some links to be implemented in future: cooperation with external computer algebra systems and automated translation of the inferences found by the reasoner and/or the prover to ForTheL.

The SAD system is implemented in Haskell [13] and C. It can be used online on the site of the project (http://ea.unicyb.kiev.ua) or downloaded from there.

3 Introduction to the ForTheL language

ForTheL (Formal Theory Language) [14] is intended for representation of mathematical texts including axioms, definitions, theorems and proofs. In this section, we provide a brief informal introduction to the language. The details of ForTheL can be found in the reference manual [15].

ForTheL is designed to be close to the natural language of real-life mathematical publications issued by mathematicians. It was frequently discussed (see, for example, the archives of the QED Project [16]) that it would be nice to get a formal language that is both formal and resembles a natural one, and we do not want to repeat here all pro and contra. We will only mention two reasons that were important for us while deciding to pursue a verbose "natural" style instead of using the unified notation of some traditional logical language.

The first reason is to provide our framework with a user-friendly interface. A text composed with correct English sentences will hopefully be more readable and easier to write than a collection of formulas built with quantifiers, parentheses, lambdas, junctors, and so on.

Second, we share the viewpoint that a text written in human language and in a traditional mathematical style usually contains a useful information which lies beyond the scope of classical logic. For example, we distinguish definitions from axioms, theorems from intermediate statements inside a proof, and we tend to use them in different ways when learning and understanding a text. Sentences in natural language contain nouns, which denote classes of entities; adjectives and verbs, which act as attributes and restrict classes; adjectives and verbs, which act as predicates and may relate different entities. However, where human language makes distinctions, the language of mathematical logic unifies. Theorems, definitions, ordinary statements become formulas. Class nouns, attributes, predicates disappear and first-order atoms appear instead.

Our intention is to preserve these distinctions in the formalization. We investigate how a human mathematician treats definitions as compared with axioms or lemmas, class nouns as compared with adjectives, proofs by case analysis as compared with proofs that follow other schemes, and so on. Taking these observations into account, we hope to improve the reasoning capabilities of a machine, to implement heuristic routines directing proof search or reducing a search space with the help of "non-logical" knowledge extracted from a formal, yet natural-like input text.

The syntax of a ForTheL sentence follows the rules of English grammar. Sentences are built of units: statements, predicates, notions (that denote classes of objects) and terms (that denote individual entities). Units are composed of syntactical primitives: nouns which form notions (e.g. "`subset of`") or terms ("`closure of`"), verbs and adjectives which form predicates ("`belongs to`", "`compact`"), symbolic primitives that use a concise symbolic notation for predicates and functions and allow to consider usual quantifier-free first-order formulas as ForTheL statements. Some syntactical primitives (for example, "`equal to`") are predefined in ForTheL and the rest must be introduced with the help of special instructions to the parser included in the text. Of course, just a little fragment of English is formalized in the syntax of ForTheL.

In order to define the semantics of a sentence, it is sufficient to define the semantics of a statement unit. The latter is determined by a series of transformations that convert a ForTheL statement to a first-order formula — the *formula image*. For example, the formula image of a simple statement "`all closed subsets of any compact set are compact`" is:

$$\forall\, \texttt{A}\, ((\texttt{A is\_a\_set} \wedge \texttt{A is compact}) \supset \\ \forall\, \texttt{B}\, ((\texttt{B is\_a\_subset of A} \wedge \texttt{B is closed}) \supset \texttt{B is compact}))$$

There are three kinds of sentences in ForTheL: assumptions, selections, and affirmations. Assumptions are statements preceded with the words "`let`" or "`assume that`". They serve to declare variables or to provide some hypotheses for the following text. For example, the following sentences are typical assumptions: "`Let S be a nonempty set.`", "`Suppose that m is greater than n.`". Selections are formed by notion units preceded with the words "`take`" or "`choose`". They state the existence of representatives of the mentioned notions and can be used to declare variables, too. Here follows an example of a selection: "`Take an even prime number X.`". Finally, affirmations are just statements: "`If p divides n - p then p divides n.`".

A ForTheL text is a sequence of top-level sections. These sections — axioms, definitions, and theorems — play in ForTheL the same role they do in usual mathematical texts. Any top-level section consists of zero or more assumptions followed by an affirmation.

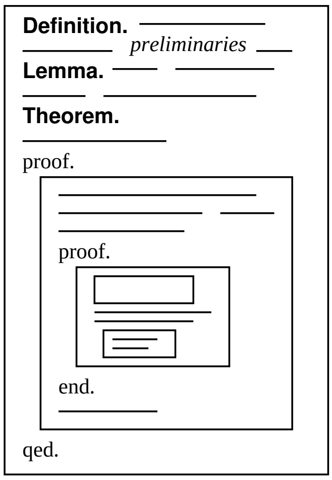

Fig. 2. ForTheL text's structure

An author can facilitate the verification process by supplying a proof for affirmations in the text. A ForTheL proof is a sequence of assumptions, selections, affirmations (which may have their own proofs), and compound low-level sections: cases and raw blocks. The case sections occur in the proofs by case analysis. The raw blocks are used to structure a proof: to limit the scope of assumptions and variable declarations. ForTheL supports special proof schemes like proof by contradiction, by case analysis, and by general induction. The ForTheL sentences are considered as low-level sections, too. In particular, an affirmation with proof is a compound section that contains the proof as a subsection.

Semantics of a ForTheL text, in what verification is concerned, is considered in Section 5. We explicitly mention verification, since a ForTheL text can be processed in different ways, depending on a problem being solved. For example, if the system wants to mine some relevant information in the text in question, the text's semantics would be based on the hierarchy of notions and definitions rather than on logical precedence or formula images.

The ForTheL language can be seen as a variant of controlled English. It has much in common with Attempto Controlled English (ACE) [17] (though the semantics of a sentence is different in ACE) and especially with Common Logic Controlled English (CLCE) [18]. In fact, many ForTheL statements are well-formed CLCE statements with the same meaning, and vice versa. However, ForTheL has a more elaborated concept of text which captures reasoning in a natural deduction style. On the level of text, ForTheL is quite similar to the language of Mizar [19] and Isar [20] (see Section 8).

Another approach to formalization of the commonly used mathematical language is presented in Mathematical Vernacular of de Bruijn (see [21]) and its refinement, Weak Type Theory (WTT) [22]. It is noteworthy that notions ("nouns") and attributes ("adjectives") in WTT are particular weak types and participate in type derivations.

In what follows, we will see examples of ForTheL texts and find some additional information on the language.

4 Formalization in SAD as seen by a user

The formal proof of the fact that $\sqrt{2}$ is not a rational number is a classical touchstone problem formalized in many mathematical assistants [23]. We have chosen its generalized version to demonstrate step by step the whole cycle of formalization and verification in SAD. We show how the initial problem is transformed to a self-contained ForTheL text, what kinds of reasoning are involved during verification, and how large (or short) are gaps in a proof that the system can fill autonomously.

4.1 Comprehending the problem

The existence of numbers which cannot be represented as a fraction of two integers was proved by mathematicians of the Pythagorean School who demonstrated that the diagonal of a square with side 1 is not such a fraction. Here is the theorem in its traditional form:

Theorem. $\sqrt{2}$ is not rational.

Proof. Assume that $\sqrt{2}$ is rational. Then there exist relatively prime natural numbers n,m such that $\sqrt{2} = n/m$, and therefore, $2m^2 = n^2$. Since n^2 is even, n also is; say $n = 2k$. Then $2m^2 = 4k^2$, that is $m^2 = 2k^2$. Thus, m^2 is even too, whence m is even. So, n and m have a common nontrivial divisor in contradiction to our hypothesis. □

The claim can be easily generalized as follows:

Theorem. The square root of any prime number is not rational.

It is not surprising that the proof of this more general fact is in a manner more logical though less straightforward (an auxiliary lemma is required):

Proof. Let p be a prime such that $\sqrt{p}$ is rational. Then there exist relatively prime natural numbers n,m such that $\sqrt{p} = n/m$, and therefore, $pm^2 = n^2$. Since p is prime, p divides n; say $n = kp$. Then $pm^2 = p^2k^2$, that is $m^2 = pk^2$. Thus, p divides m^2 and m. So, n and m have a common nontrivial divisor and we have a contradiction. □

The fact that $2|n^2$ implies $2|n$ can be proved directly: $(2k{+}1)^2 = 2(2k^2{+}2k){+}1$. The generalized statement $(p|n^2 \supset p|n)$ is not that obvious. So, in order to make our proof verifiable, we involve the following lemma (PDP stands for "Prime that Divides a Product"):

Lemma PDP. Let m,n be natural numbers and p be a prime. If p divides mn then p divides m or n.

Proof. We proceed by induction on $(m + n + p)$. If $m \geq p$ then $p|mn$ implies $p|(m - p)n$ and we apply the induction hypothesis ($p|m - p$ obviously implies $p|m$). In the same manner we treat the case $n \geq p$.

Say $m, n < p$ and $pk = mn$. If k is zero or one, the lemma's statement is immediately proved. Otherwise, let r be a prime divisor of k. Then r divides mn and, therefore $r|m$ or $r|n$, using the induction hypothesis and the fact that $r \leq k < p$.

If r divides m then $p(k/r) = (m/r)n$ and the induction hypothesis is applicable. Obviously, $p|(m/r)$ implies $p|m$. We treat the case $r|n$ in the same way. □

What we have, is not yet a self-contained text. We must complete it with an ensemble of preliminary facts, including, in the order of introduction: basic properties of natural numbers (addition, subtraction, and ordering), some Euclidean arithmetic (divisibility, quotient, primes and relative primes), definitions of a rational number and square root. These *preliminaries* are well-known properties of well-known mathematical objects and we can take them for granted.

4.2 ForTheL'ization

The mathematical notation used in our example is quite simple. Thus it suffices just to reduce the syntax to ASCII character set and to get rid of some syntactic sugar in order to write down this text in ForTheL. For the sake of brevity, we do not cite the preliminaries. In the rest of the paper, we will refer to the following ForTheL fragment as to the Square Root Text.

```
Lemma PDP.  For all natural numbers n,m,p
  if p is prime and p | n * m then p | n or p | m.
Proof by induction on ((n + m) + p).
  Let n,m,p be natural numbers.
  Assume that p is prime and p divides n * m.
  Case p <= n.
    p divides (n - p) * m and n - p < n.
    Then p divides n - p or p divides m.
    If p divides n - p then p divides n.
  end.
  Case p <= m.
    p divides n * (m - p) and m - p < m.
    Then p divides n or p divides m - p.
    If p divides m - p then p divides m.
  end.
  Case n < p and m < p.
    Take a natural number k such that n * m = p * k.
    Case k = 0 \/ k = 1. Obvious.
    Case k != 0 /\ k != 1.
      Take a prime divisor r of k.
      r divides n * m and r <= k and k < p.
      Then r divides n or r divides m.
      Case r divides n.
        p divides (n / r) * m and (n / r) < n.
        Then p divides (n / r) or p divides m.
        If p divides (n / r) then p divides n.
      end.
      Case r divides m.
        p divides n * (m / r) and (m / r) < m.
        Then p divides n or p divides (m / r).
        If p divides (m / r) then p divides m.
      end.
    end.
  end.
qed.

Theorem Main.  Let p be a prime natural number.
  For no rational number q the square of q is p.
Proof by contradiction.
```

```
    Let q be a rational number such that q * q = p.
    Take relatively prime natural numbers n,m such that q * m = n.
    Then p * (m * m) = (n * n).
    Hence p divides n * n and p divides n.
    Choose a natural number k such that n = p * k.
    Then we have p * (m * m) = p * (k * n).
    The square of m is equal to (p * k) * k.
    Hence p divides m * m and p divides m.
    We have a contradiction.
  qed.
```

The text above is a syntactically valid ForTheL text. Moreover, being supplied with suitable preliminary facts, this text is probably correct, meaning that every statement in it can be deduced from its logical predecessors (so far, we do not precise what are the statements to be proved and which predecessors they have). We say "probably", because this ForTheL fragment just reflects the proofs written above in natural language and we believe those proofs to be correct; however, we cannot be completely sure in any one of these two claims, so the formal verification is required.

Let us explain how a text in natural language is converted to ForTheL. First of all, we add explicit structural markup. Proofs, subproofs, cases should have explicit headers and endings. Note that SAD does not support proof by analogy, so we have to write twice almost the same reasoning in the proof of Lemma PDP (compare the cases "`p <= n`" and "`p <= m`"; "`r divides n`" and "`r divides m`").

Since our motivation is not to show the existence of irrational numbers, but just to prove a property of primes, we can slightly simplify the reasoning by eliminating some operations which we use sporadically, such as square root or division on rational numbers.

We remove some reminders for a human reader — such as "Since p is prime" — since they do not help a reasoner to find a proof (indeed, the system will not forget that p is prime), and therefore are not included in the syntax of ForTheL (they may be easily added on the user's request). On the other hand, we may have to add some supplementary statements which would seem superfluous to a reader but make the automated verification possible (see Subsection 4.3).

Finally, since ForTheL formalizes just a small fragment of English (even "scientific English"), there are some natural expressions that should be reformulated. For example, we unfold English statements where a verb applies to a list of objects: in ForTheL, we have to say "p divides m^2 and p divides m" instead of "p divides m^2 and m".

In conclusion, let us give some explanations on using induction in SAD. Proofs by induction do not have any particular structure in ForTheL; for example, there is no special markup for the base case and the step case. A proof section is considered as a proof by induction whenever it satisfies the following conditions. The section begins with the header "`proof by induction on` *term*". The goal must be a universally quantified formula of the form $\forall \vec{v} F$ and the free variables of the induction term (say, t) must be in $\vec{v}$. These variables must be declared in the beginning of the proof section in the same way as they are described in the goal. If these requirements are satisfied then the parser formulates and inserts in the proof the appropriate induction hypothesis:

$$\forall \vec{u}\, (t[\vec{u}/\vec{v}] \prec t \supset F[\vec{u}/\vec{v}])$$

Here, the binary relation $\prec$ stands for an abstract well-founded ordering, as explained below. In other words, the hypothesis can be formulated as follows: *the property being proved holds for all the values of the induction term that are less than the current one with respect to a fixed well-founded ordering.*

If we succeed to prove the property for the current value of the induction term in presence of the induction hypothesis, we can conclude that the property holds for all values of that term and the initial goal is thus proved. In other words, ForTheL applies the *general induction principle*.

Since there is no finite axiomatization of well-foundness in first-order logic, the SAD system can by no means check whether a given user-defined ordering is well-founded. Instead, ForTheL provides a special binary predicate symbol `-<-` which is used as an abstract a priori well-founded relation in induction arguments. It is left to the author to write down the axioms defining particular properties of the relation `-<-`, and the system will take for granted that the supplied axioms are compatible with well-foundness.

In the current example, the corresponding axiom is:

```
Axiom IH.   For all natural numbers n,m (n < m => n -<- m).
```

and the induction hypothesis for the proof of PDP is as follows:

```
For all natural numbers N,M,P if ((N + M) + P) -<- ((n + m) + p)
  then if P is prime and P | N * M then P | N or P | M.
```

4.3 Filling the gaps

Despite our confidence in the correctness of the Square Root Text, SAD fails to verify it. The reasoning capabilities of the system are too weak to establish automatically the truth of the very first affirmation in the proof of Lemma PDP:

"`p divides (n - p) * m`", the fourth line of the proof. In fact, almost all the affirmations in the Square Root Text will not be proved as they are.

Generally, there are several ways to make an affirmation more "evident" to the system. The simplest one is to provide it with *references*, a list of top-level premises to use in verification. The rest of top-level facts (axioms, definitions, lemmas) will be excluded from the proof search [1]. When your problem has a large set of preliminaries, such an explicit filtering of premises can be very helpful, reducing the search time from half an hour to half a second. Note that a top-level fact mentioned in references is not required to actually contribute to the found proof.

However, the exact list of needed premises can be sometimes hard to find (in fact, the author has to foresee the proof) and can turn out to be quite long, making the text unnatural. Finally, it may happen that even if the set of premises is filtered, the prover still fails to find the proof. The needed proof step may be just too large.

Another way is to insert an intermediate statement which is easier to verify than the affirmation in question (say, A) and which may shorten the proof of A. The author may provide a proof section for A and put there not just a single statement but the whole intermediate reasoning. The rule of thumb is as follows: if your intermediate argument $\mathcal{P}$ is going to be used merely to support verification of A then place $\mathcal{P}$ in a subproof; if it can help to verify other claims in the current proof section, too, then put $\mathcal{P}$ above A, so that $\mathcal{P}$ will be a premise for the rest of the section; if $\mathcal{P}$ reappears several times in your proof under different instantiations, then make it a top-level lemma.

The needed level of detail is usually found by trial and error, until the problematic affirmation becomes checkable. Of course, this level strongly depends on the reasoner and the prover being used. Therefore, some knowledge about the deductive machinery of SAD can help the author in preparation of a verifiable text. In hard cases, the author may have to revise the whole proof and the set of preliminaries.

Let us consider the first case section in the proof of Lemma PDP in its final, verifiable state:

```
Case p <= n.
  Let us show that p divides (n - p) * m and n - p < n.
    n = p + (n - p) and n * m = p * ((n * m) / p).
    n * m = (p * m) + ((n - p) * m) (by AMDistr).
    p divides (n - p) * m (by DefDiv, DivMin).
```

[1] however, the author can explicitly mark some frequently used top-level facts as persistent, making them participate in each proof search.

```
    end.
    Then p divides n - p or p divides m (by IH).
    Indeed ((n - p) + m) + p < (n + m) + p (by MonAdd). end.
    If p divides n - p then p divides n (by DefDiv,DivSum).
  end.
```

Let us compare the case section above with the corresponding one in the Square Root Text. It consists of the same three affirmations which are now provided with subproofs and references. Note that purely algebraic tasks turn to be difficult for a generic first-order prover (especially in presence of the axioms of associativity, commutativity, and distributivity), so that we give three intermediate statements in order to deduce $p|(n-p)m$ from $p|nm$.

Another subgoal, $n - p < n$, does not require any support from us. On the other hand, SAD fails to deduce $((n-p)+m)+p < (n+m)+p$ from $n-p < n$ on its own, and we have to mention it explicitly in a subproof in order to apply the induction hypothesis.

Other "hard" affirmations in the text are processed in a similar way. At the end, the proof of Lemma PDP becomes twice longer than the initial one. At the same time, the proof of the main theorem does not grow, since we can make it verifiable just by referencing the necessary preliminary facts:

```
Theorem Main.  Let p be a prime natural number.
  For no rational number q the square of q is p.
Proof by contradiction.
  Let q be a rational number such that q * q = p.
  Take relatively prime natural numbers n,m such that q * m = n.
  Then p * (m * m) = (n * n) (by MulAsso,MulComm).
  Hence p divides n * n and p divides n (by DefDiv,PDP).
  Choose a natural number k such that n = p * k (by DefDiv).
  Then we have p * (m * m) = p * (k * n) (by MulAsso).
  The square of m is equal to (p * k) * k (by MulComm,MulCanc).
  Hence p divides m * m and p divides m (by MulAsso,DefDiv,PDP).
  We have a contradiction (by DefRelPr).
qed.
```

In its final state, the ForTheL text about square roots of prime numbers consists of 205 non-blank lines: 126 lines of preliminaries and 79 lines for Lemma PDP and the main theorem. The text contains 95 goals (lemmas in preliminaries taken into account, too) and is fully verified by SAD using SPASS [10] as the background prover in 45 seconds (0.5 sec/goal on average, 4.7 seconds for the longest session).

Some other interesting problems formalized and verified in SAD are Finite and Infinite Ramsey Theorems, a series of theorems about a particular notion of refinement, Cauchy-Bunyakowski-Schwarz (CBS) inequality for real vectors, Chinese Remainder Theorem, a criterion of series convergence (see Section 6).

5 Text verification

Let us demonstrate how does SAD work on a fragment taken from the proof of Lemma PDP. We consider the innermost case analysis where the "gaps" are not filled yet:

```
Case r divides n.
  p divides (n / r) * m and (n / r) < n.
  Then p divides (n / r) or p divides m.
  If p divides (n / r) then p divides n.
end.
Case r divides m.
  p divides n * (m / r) and (m / r) < m.
  Then p divides n or p divides (m / r).
  If p divides (m / r) then p divides m.
end.
```

The process of translation reveals the semantics of different proof methods and encodes it in the structure of the resulting text. If we leave individual sentences as they are and translate only the compound sections, the text above will be translated (or *normalized*, in terms of [15]) into the following one:

```
p | n or p | m.                               # 1
proof.
  r divides n or r divides m.                 # 2
  if r divides n then p | n or p | m.         # 3
  proof.
    let r divide n.
    p divides (n / r) * m and (n / r) < n.    # 4
    Then p divides (n / r) or p divides m.
    If p divides (n / r) then p divides n.
    p | n or p | m.
  end.
  if r divides m then p | n or p | m.         # 5
  proof.
    let r divide m.                           # 6
    p divides n * (m / r) and (m / r) < m.    # 7
    Then p divides n or p divides (m / r).
    If p divides (m / r) then p divides m.
    p | n or p | m.
  end.
end.
```

As we see, the sequence of case sections is transformed to a proof of the statement "`p | n or p | m`". Why this one? The reason is that it was the *current goal* when the case analysis started.

The current goal is a statement associated to each position inside a proof section. Roughly, it is the statement being proved in that position. Let us consider how the current goal evolves in the proof of Lemma PDP. At the beginning of the proof, the current goal is "`for all natural numbers n,m,p if p is prime and p | n * m then p | n or p | m`". After the first assumption, which declares the variables `n`, `m`, and `p` as natural numbers, the current goal is reduced to "`if p is prime and p | n * m then p | n or p | m`". After the second assumption, the current goal is reduced further to "`p | n or p | m`". Assumptions that modify the current goal are called *motivated*. Unmotivated assumptions are not shown in the example above and we will return to them later.

So at the moment when the case analysis under consideration starts, the current goal is still "`p | n or p | m`". Therefore the translated text has the form of the affirmation "`p | n or p | m`" supplied with a proof section.

Before considering particular cases, the reasoner must ensure that the case analysis is exhaustive, that is, the disjunction of case hypotheses is valid. This disjunction becomes the first affirmation in the proof section.

Each case section is translated to a proof of the affirmation of the form "`if` H `then` G" where H is the case hypothesis and G is the current goal. At the beginning of the proof, the current goal is set to this implication but the very first assumption "`let` H" reduces it back to G. Then the proof follows the lines of the original case section.

At the end of any proof section that came from the initial text or was obtained from a case section, the translator puts the affirmation of the current goal (unless the proof section already ends with an equivalent statement). Thus, both case sections end with the affirmation "`p | n or p | m`". Note that this affirmation is not put at the end of the outer proof section, since the conducted case analysis gives itself the infallible proof of the current goal.

The definitions below determine the semantics of ForTheL w.r.t. verification.

Definition 1 A section A is a *logical predecessor* of a section B in a normalized text $\mathbf{T}$ whenever A textually precedes B and no section containing A textually precedes B.

Definition 2 Let Γ be a set of sections in a normalized text $\mathbf{T}$ and S be a ForTheL statement. We say that S is a logical consequence of Γ whenever the formula image of S is a logical consequence of the formula image of Γ.

The formula image of Γ is the conjunction of images of sections from Γ. The definition of the formula image of a ForTheL section is quite long and we will not give it here in full. The idea can be seen from the following examples. The image of an affirmation is the image of the affirmed statement independently of the presence of a proof section. The image of an assumption is the assumed statement. A two-sentence raw block of the form "`block. Let X be a` N`. Then` S`. end.`" where `X` is a fresh variable, N, a notion, and S, a statement, will have the image $\forall X(\mathrm{FI}(X\ \texttt{is}\ N) \supset \mathrm{FI}(S))$, where $\mathrm{FI}(X\ \texttt{is}\ N)$ and $\mathrm{FI}(S)$ are the formula images of the corresponding statements.

Definition 3 An affirmation A in a normalized text $\mathbf{T}$ is *correct* whenever one of the following conditions holds: (a) A occurs in an axiom or a definition; (b) A is supplied with a proof section; (c) the statement of A is a logical consequence of the set of logical predecessors of A in $\mathbf{T}$.

Definition 4 A selection A in a normalized text $\mathbf{T}$ is *correct* whenever the existence of representatives of the mentioned notions is a logical consequence of the set of logical predecessors of A in $\mathbf{T}$.

Definition 5 A ForTheL text $\mathbf{T}$ is correct whenever all the selections and affirmations in its normalization are correct.

In the normalized text on page 15, the affirmation `#7` is logically preceded by sections `#2` (affirmation without proof), `#3` (affirmation with a proof), and `#6` (assumption). Affirmations `#1` and `#5` are not predecessors of `#7` since we consider an affirmation with a proof as a single section and, therefore, both `#1` and `#5` *contain* `#7` and do not textually precede it. The affirmation `#4` and other sentences in that proof section are not predecessors of `#7` since the section `#3` that contains `#4` and its neighbors precedes `#7`.

Thus, in order to confirm correctness of the Square Root Text, the reasoner must, in particular, deduce "`p divides n*(m/r) and (m/r) < m`" from the set of premises that contains "`r divides m`" (`#6`), "`if r divides n then p|n or p|m`" (`#3`), "`r divides n or r divides m`" (`#2`), and other logical predecessors of `#7` outside the fragment under consideration.

Let us now return to motivated and unmotivated assumptions. Consider the following ForTheL fragment:

```
Let us prove that any natural number n greater than 2 is odd.
  Let n be a natural number greater than 2.
  Assume that n is prime. Hence 2 does not divide n.
end.
```

The first assumption in the proof is certainly motivated and changes the current goal to "`n is odd`". The second one does not correspond to any part of the (new) current goal and leaves it unchanged. Now, if we try to translate this fragment just as described above:

```
Any natural number n greater than 2 is odd.
Proof.
  Let n be a natural number greater than 2.
  Assume that n is prime. Hence 2 does not divide n.
  n is odd.
end.
```

and apply the Definition 5, we will conclude that the fragment is correct while it is certainly not. The reason is that correctness of the current goal in the end of a proof implies correctness of the statement being proved only if all the assumptions we made in the proof were motivated. Therefore, we enclose unmotivated assumptions in raw blocks, and the fragment in question is translated as follows:

```
Any natural number n greater than 2 is odd.
Proof.
  Let n be a natural number greater than 2.
  block.
    Assume that n is prime. Hence 2 does not divide n.
  end.
  n is odd.
end.
```

The newly introduced raw block section is equivalent to an affirmation "`if n is prime then 2 does not divide n`", which is correct but does not prove the final "`n is odd`".

Assumptions of the form "`assume the contrary`" or, in general, "`assume that` H" where H is equivalent to the negation of the current goal, are considered as motivated and change the current goal to *falsum*, so that the subsequent reasoning must lead to a contradiction.

To conclude: the ForTheL language determines what are the statements to verify and what are the statements to use in any particular verification process. The reasoner and the prover will accomplish the job.

6 Extending the scope: binding constructions

The SAD system relies a lot on the deductive power of its background prover. The foreground reasoner of SAD can help the prover by simplifying the prover's tasks but its special methods and heuristics cannot replace extensive inference search. SAD uses classical first-order logic for inference search. Therefore any construction we use in our input texts should be either translatable into first order language (possibly, in some tricky way) or must be excluded from the scope of the prover and handled by the reasoner or even by the parser.

Let us now consider a more complex domain than that of natural numbers.

Theorem. Let (a_i) be a nonnegative, monotone decreasing sequence of reals that converges towards 0. Then the following series converges, too:

$$\sum_{i=0}^{\infty}(-1)^i a_i$$

Proof. First, we prove that for all N, d

$$0 \leq (-1)^N \sum_{i=N}^{N+d}(-1)^i a_i \leq a_N$$

We proceed by induction on d. The base case of the induction ($d = 0$) is obvious. Let us assume that the statement holds for $d = k$ (and for arbitrary N) and show that it also holds for $d = k + 1$:

$$(-1)^N \sum_{i=N}^{N+(k+1)} (-1)^i a_i = a_N - \left((-1)^{N+1} \sum_{i=N+1}^{(N+1)+k} (-1)^i a_i \right)$$

By induction hypothesis, the subtracted value lies between 0 and a_{N+1} which is not greater than a_N, hence the induction step is proved.

Now, we must demonstrate that for arbitrary $\varepsilon > 0$ there exists some N so that for any $n > N$

$$\left| \sum_{i=0}^{n}(-1)^i a_i - \sum_{i=0}^{N}(-1)^i a_i \right| = \left| \sum_{i=N+1}^{n} (-1)^i a_i \right| < \varepsilon$$

Let us fix ε and take such N that for any $n > N$, $a_n < \varepsilon$ (recall that (a_n) converges to zero). Take some arbitrary $n > N$. Then we obtain

$$0 \leq \left| \sum_{i=N+1}^{n} (-1)^i a_i \right| = (-1)^{N+1} \sum_{i=N+1}^{n} (-1)^i a_i \leq a_{N+1} < \varepsilon$$

and the theorem is proved. □

In this example, we deal with sequences and series which, in general, are second-order objects. They occur throughout the proof, sometimes inside algebraic calculations, and we cannot hope to eliminate all of them from the scope of the prover. So we must supply them with a first-order semantics.

There are several first-order formalisms where functions and predicates are treated as individual objects. First, we can build a subtheory of sequences and series in the frame of some set theory. This approach would provide a common basis for many different domains. The well-known system Mizar [19] uses it to construct a whole library of interrelated mathematical texts.

However, when working with concepts defined in a set-theoretic style, we often have to descend in definition expansion down to the membership relation. As a consequence, we obtain proof tasks that are quite long (because of all the needed preliminaries) and where numerous membership literals unify with each other. Combinatorial inference search is unpractical for such problems. To cope just with sequences and series, a theory of sets is too general.

We could also use the equational theory of combinators **CL** by Curry [24]. The binary application operation and combinators **S** and **K** are sufficient to describe any computational behavior that we can meet on the second-order level of traditional mathematics and beyond that. Unfortunately, unification (i.e. term construction) in such a theory is very expensive; two axioms of **CL** produce a lot of inferences and tend to explode the search space. In this regard, they can be compared with the axioms of equality or with AC-axioms.

At the same time, the theory of combinators is also too general for our purposes: we do not work with self-application, fixed points, uncomputable objects, or arbitrary high orders. We would like to find a more specific set of combinators whose application during inference search could be restricted.

Such a set of combinators can be found in the theory of recursive functions [25]; besides, we need neither primitive recursion nor minimization. The problem is that we cannot describe higher-order functions in such a theory without suitable encoding, e.g. gödelization. However, if we weaken the recursion theory leaving just superpositions and add the appropriate combinators for functions like $\sum$ or *lim*, we obtain what we need.

Before proceeding to definitions, consider a ForTheL version of the theorem:

```
Theorem.
  Let A[i] be a real number for every natural number i.
  Assume that for every natural number i
              (0 <= A[i] and A[i+1] <= A[i]).
  Assume that A[i] converges wrt i to zero.
  Then Sum(mn(i)*A[i],i,0,N) converges wrt N
    to some real number.
Proof.
  For all natural numbers N,d
    0 <= mn(N) * Sum(mn(i)*A[i],i,N,N+d) and
    mn(N) * Sum(mn(i)*A[i],i,N,N+d) <= A[N].
```

```
  proof by induction on d.
    Let N,d be natural numbers.
    Case d = 0. Obvious.
    Case d = k+1 for some natural number k.
      Sum(mn(i)*A[i],i,N,N+d) =
        (mn(N)*A[N]) + Sum(mn(i)*A[i],i,N+1,N+d).
      Then mn(N) * Sum(mn(i)*A[i],i,N,N+d) =
        A[N] - (mn(N+1) * Sum(mn(i)*A[i],i,N+1,N+d)).
      We have 0 <= mn(N+1) * Sum(mn(i)*A[i],i,N+1,N+d) and
        mn(N+1) * Sum(mn(i)*A[i],i,N+1,N+d) <= A[N+1] (by IH).
      A[N+1] <= A[N] and we have the thesis.
    end.
  end.

  Let e be a positive real number.
  Take a natural number N such that for every natural number n
    if n > N then |A[n]| < e.
  Let n be a natural number such that n > N.
  Sum(mn(i)*A[i],i,0,n) =
    Sum(mn(i)*A[i],i,0,N) + Sum(mn(i)*A[i],i,N+1,n).
  We have 0 <= mn(N+1) * Sum(mn(i)*A[i],i,N+1,n).
  Then |Sum(mn(i)*A[i],i,N+1,n)| =
    mn(N+1) * Sum(mn(i)*A[i],i,N+1,n).
  Hence |Sum(mn(i)*A[i],i,N+1,n)| <= A[N+1]
    and |Sum(mn(i)*A[i],i,N+1,n)| < e.
  |Sum(mn(i)*A[i],i,0,n) - Sum(mn(i)*A[i],i,0,N)| < e.
  We have the thesis.
qed.
```

In the text above, `mn(i)` stands for $(-1)^i$, `Sum(mn(i)*A[i],i,0,n)` denotes $\sum_0^n (-1)^i A_i$, and `A[i] converges wrt i to zero` means $\lim A_i = 0$. The variable `A` is a "second-order" variable, which are allowed only in terms of the form `A[`s_1`,...,`s_n`]`. The "arity" of such a variable should stay fixed throughout the text. Both in premises and in goals, second-order variables are implicitly universally quantified.

In binding functions (`Sum`) or predicates (`converges to`) we mention the bound variable as an ordinary argument. By definition of the corresponding syntactical primitive, the parser knows which argument place holds a variable to bind, and so the term `Sum(mn(i)*A[i],i,0,n)` will be translated to `Sum(`λ`i.mn(i)*A[i],0,n)`. Let us denote this step by Iλ (introduction of λ).

Lambda terms are not allowed in the first-order language, so translation must proceed and replace lambdas with combinator terms. In order to guarantee the consistency of our formalization, we need to distinguish ordinary individual objects from individuals representing functions. Therefore we take, as the destination language, a many-sorted first-order language where sorts are natural numbers: ordinary individuals have the sort 0, and each lambda-abstraction on top of a term increments its sort by 1. In other words, a term of the sort n is understood as a function of n arguments. Thus, all "second-order" variables obtain the sort equal to their arity and the rest of variables ("first-order" ones) get the sort 0. Each non-binding function symbol (such as `mn(.)`) obtains the sort $0 \times \cdots \times 0 \to 0$; each non-binding predicate symbol obtains the sort $0 \times \cdots \times 0$. Only the variables of the sort 0 are allowed to be bound.

Then we extend the signature of our destination language with the following constants, functions and predicates:

- for each non-binding function symbol f (such as `mn(.)`) with the arity $n > 0$, we add a *twin* constant f^n of the sort n;
- for all m, n such that $0 < m \leq n$, we add a *projection* constant I^n_m of the sort n;
- for all m, n, we add a *superposition* function S^n_m of the arity $(m+1)$ and the sort $m \times n \times \cdots \times n \to n$; initial occurrences of application `t[x1,...,xm]` are replaced with S^0_m(`t`,`x1`,...,`xm`);
- for all n and each binding function symbol Σ (such as `Sum(`λ`i.,.,.)`) with the arity $1 + m$, we add a *special superposition* function Σ^n_m of the arity $(m+1)$ and the sort $(n+1) \times n \times \cdots \times n \to n$; initial occurrences of Σ are replaced with Σ^0_m;
- for each binding predicate symbol Lim (such as λ`i. converges to .`) with the arity $1 + m$ we add a new predicate Lim_m of the arity $(m + 1)$ and the sort $1 \times 0 \times \cdots \times 0$; initial occurrences of Lim are replaced with Lim_m.

$$
\begin{array}{rll}
\lambda u.u & \to I^1_1 & (\mathrm{E}\lambda I) \\
\lambda u.s & \to \mathrm{S}^{n+1}_n(s, I^{n+1}_1, \ldots, I^{n+1}_n) & (\mathrm{E}\lambda K) \\
\lambda u.f(s_1, \ldots, s_m) & \to \mathrm{S}^1_m(f^m, \lambda u.s_1, \ldots, \lambda u.s_m) & (\mathrm{E}\lambda f) \\
\lambda u.\mathrm{S}^n_m(t, s_1, \ldots, s_m) & \to \mathrm{S}^{n+1}_{m+1}(\lambda u.t, \lambda u.s_1, \ldots, \lambda u.s_m, I^{n+1}_{n+1}) & (\mathrm{E}\lambda S) \\
\lambda u.\Sigma^n_m(t, s_1, \ldots, s_m) & \to \Sigma^{n+1}_m(\lambda u.t, \lambda u.s_1, \ldots, \lambda u.s_m) & (\mathrm{E}\lambda\Sigma)
\end{array}
$$

(rule $\mathrm{E}\lambda K$: s has the sort n and u does not occur in s)

Fig. 3. Rules of lambda-elimination

Figure 3 shows the rules we apply to eliminate lambda. For example, the main statement of the theorem above will be transformed as follows. Note that bound variables `N` and `i` disappear in the final statement.

`Sum(mn(i)*A[i],i,0,N) converges wrt N to R`

$\Downarrow$ (by Iλ)

λ`N.Sum(` λ`i.mn(i)*A[i], 0,N) converges to R`

$\Downarrow$ (introduction of sort-indexed symbols)

λ`N.Sum`$^0_2$`(` λ`i.mn(i)*`S^0_1`(A,i), 0,N) (converges to)`$_1$ `R`

$$\Downarrow \text{ (by E}\lambda f)$$

$$\lambda\texttt{N.Sum}^0_2(\mathrm{S}^1_2(\texttt{*}^2,\ \lambda\texttt{i.mn(i)},\ \lambda\texttt{i.}\mathrm{S}^0_1(\texttt{A,i})),\texttt{0,N})\ (\texttt{converges to})_1\ \texttt{R}$$

$$\Downarrow \text{ (by E}\lambda f, \text{E}\lambda I)$$

$$\lambda\texttt{N.Sum}^0_2(\mathrm{S}^1_2(\texttt{*}^2, \mathrm{S}^1_1(\texttt{mn}^1, I^1_1),\ \lambda\texttt{i.}\mathrm{S}^0_1(\texttt{A,i})),\ \texttt{0,N})\ (\texttt{converges to})_1\ \texttt{R}$$

$$\Downarrow \text{ (by E}\lambda S, \text{E}\lambda K, \text{E}\lambda I)$$

$$\lambda\texttt{N.Sum}^0_2(\mathrm{S}^1_2(\texttt{*}^2, \mathrm{S}^1_1(\texttt{mn}^1, I^1_1), \mathrm{S}^1_2(\mathrm{S}^2_1(\texttt{A}, I^2_1), I^1_1, I^1_1)),\texttt{0,N})\ (\texttt{converges to})_1\ \texttt{R}$$

$$\Downarrow \text{ (by E}\lambda \Sigma, \text{E}\lambda K, \text{E}\lambda I)$$

$$\texttt{Sum}^1_2(\mathrm{S}^2_1(\ \mathrm{S}^1_2(\texttt{*}^2,\ \mathrm{S}^1_1(\texttt{mn}^1, I^1_1),\ \mathrm{S}^1_2(\mathrm{S}^2_1(\texttt{A}, I^2_1), I^1_1, I^1_1)),\ I^2_1),\ \mathrm{S}^1_0(\texttt{0}),\ I^1_1)\ (\texttt{converges to})_1\ \texttt{R}$$

The final expression looks quite cumbersome. Such a natural and simple construction like the initial statement surely should not turn into a hard-readable several-line bush of special function symbols S^i_j and I^i_j. Can we simplify the translation using the properties of superpositions and projections?

$$\mathrm{S}^n_n(x, I^n_1, \ldots, I^n_n) = x \tag{1}$$
$$\mathrm{S}^n_m(I^m_k, x_1, \ldots, x_m) = x_k \tag{2}$$
$$\mathrm{S}^0_m(f^m, x_1, \ldots, x_m) = f(x_1, \ldots, x_m) \tag{3}$$
$$\mathrm{S}^n_m(\mathrm{S}^m_k(x, y_1, \ldots, y_k), z_1, \ldots, z_m) = \mathrm{S}^n_k(x, \mathrm{S}^n_m(y_1, z_1, \ldots, z_m), \ldots, \mathrm{S}^n_m(y_k, z_1, \ldots, z_m)) \tag{4}$$
$$\begin{aligned}\mathrm{S}^n_m(\Sigma^m_k(x, y_1, \ldots, y_k), z_1, \ldots, z_m) = \\ \Sigma^n_k(\mathrm{S}^{n+1}_{m+1}(x, I^{n+1}_1, \mathrm{S}^{n+1}_n(z_1, I^{n+1}_2, \ldots, I^{n+1}_{n+1}), \ldots, \mathrm{S}^{n+1}_n(z_m, I^{n+1}_2, \ldots, I^{n+1}_{n+1})), \\ \mathrm{S}^n_m(y_1, z_1, \ldots, z_m), \ldots, \mathrm{S}^n_m(y_k, z_1, \ldots, z_m))\end{aligned} \tag{5}$$

Fig. 4. First-order theory of limited second-order superposition

In Figure 4, we introduce the axiom schemes varying over natural m, n, k, twin constants f^m, and special superpositions Σ^n_m. These axioms constitute what we call the *theory of limited second-order superposition* **SP**. The axioms of **SP** permit us to continue the transformation above:

$$\texttt{Sum}^1_2(\mathrm{S}^2_1(\ \mathrm{S}^1_2(\texttt{*}^2,\ \mathrm{S}^1_1(\texttt{mn}^1, I^1_1),\ \mathrm{S}^1_2(\mathrm{S}^2_1(\texttt{A}, I^2_1), I^1_1, I^1_1)),\ I^2_1),\ \mathrm{S}^1_0(\texttt{0}),\ I^1_1)\ (\texttt{converges to})_1\ \texttt{R}$$

$$\Downarrow (1,4)$$

$$\texttt{Sum}^1_2(\mathrm{S}^2_1(\ \mathrm{S}^1_2(\texttt{*}^2, \texttt{mn}^1,\ \mathrm{S}^1_1(\texttt{A}, \mathrm{S}^1_2(I^2_1, I^1_1, I^1_1))),\ I^2_1), \mathrm{S}^1_0(\texttt{0}), I^1_1)\ (\texttt{converges to})_1\ \texttt{R}$$

$$\Downarrow (2)$$

$$\texttt{Sum}^1_2(\mathrm{S}^2_1(\ \mathrm{S}^1_2(\texttt{*}^2, \texttt{mn}^1,\ \mathrm{S}^1_1(\texttt{A}, I^1_1)),\ I^2_1),\ \mathrm{S}^1_0(\texttt{0}),\ I^1_1)\ (\texttt{converges to})_1\ \texttt{R}$$

$$\Downarrow (1)$$

$$\texttt{Sum}^1_2(\mathrm{S}^2_1(\ \mathrm{S}^1_2(\texttt{*}^2, \texttt{mn}^1,\ \texttt{A}),\ I^2_1),\ \mathrm{S}^1_0(\texttt{0}),\ I^1_1)\ (\texttt{converges to})_1\ \texttt{R}$$

$$\Downarrow (4)$$

$$\texttt{Sum}^1_2(\mathrm{S}^2_2(\texttt{*}^2,\ \mathrm{S}^2_1(\texttt{mn}^1, I^2_1),\ \mathrm{S}^2_1(\texttt{A}, I^2_1)),\ \mathrm{S}^1_0(\texttt{0}),\ I^1_1)\ (\texttt{converges to})_1\ \texttt{R}$$

Let us analyze the first argument of the relation (`converges to`). It has the sort one and so is considered as an unary function which takes the index (say, N) as the argument and returns the N-th sequence element. This element is evaluated by a binding function `Sum(λ.,.,.)`, but first we must evaluate its arguments. The term $\mathrm{S}^1_0(\mathtt{0})$ denotes the constant-zero unary function. That means that the second argument for `Sum` does not depend on N. The term I^1_1 is the identity function. That means that the third argument is N itself.

The first argument of `Sum` is a term of the sort 2: it takes the index (say, i) for the inner sequence being summed and the index N. To evaluate it, we pass i and N to $\mathrm{S}^2_1(\mathtt{mn}^1, I^2_1)$ (resulting in $\mathtt{mn}(i)$) and to $\mathrm{S}^2_1(\mathtt{A}, I^2_1)$ (resulting in $\mathrm{S}^0_1(\mathtt{A}, i)$). And the first argument of `Sum` — the i-th element of the inner sequence — will be evaluated to $(\mathtt{mn}(i)\ \ast\ \mathrm{S}^0_1(\mathtt{A}, i))$ as it should be.

Main properties of the theory **SP** are as follows.

Lemma 6 Let t be a **SP**-term of the sort n and $s_1, \ldots, s_n$ be **SP**-terms of the sort k, and u be a variable of the sort 0. Then the following holds in **SP**:

$$\mathrm{S}^k_{n+1}(\lambda u.t, s_1, \ldots, s_n, \mathrm{S}^k_0(u)) =_{\mathbf{SP}} \mathrm{S}^k_n(t, s_1, \ldots, s_n)$$

Corollary 7 (consistency of abstraction) Let $t, s_1, \ldots, s_n$ be **SP**-terms of the sort 0 and $u_1, \ldots, u_n$ be variables of the sort 0 that do not occur in $s_1, \ldots, s_n$. Then the following equality is true in **SP**:

$$\mathrm{S}^0_n(\lambda u_n \cdots \lambda u_1.t, s_1, \ldots, s_n) =_{\mathbf{SP}} t[s_1/u_1, \ldots, s_n/u_n]$$

Proposition 8 (scope of SP) For any **SP**-term t of the sort n there exists a term t' in the source language that does not contain free variables, except maybe those of t and first-order variables $u_1, \ldots, u_n$, such that

$$\lambda u_1 \cdots \lambda u_n.t' =_{\mathbf{SP}} t$$

The last proposition shows that the theory **SP** fits well our needs: it describes exactly those expressions with binding constructions and second-order variables which we can write in ForTheL. This is the difference between **SP** and the general theory of combinators **CL**.

Unfortunately, direct application of the **SP** axioms in inference search is not very fruitful. Similar to the case of **CL**, unification is very expensive in **SP** and an author has to include in his text almost all the intermediate steps of **SP** transformations. Moreover, to cope with **SP** we need a prover supporting sorts which is rarely the case. At present, we are working on a mechanism to generate the appropriate instantiations for second-order variables before launching the prover, which could permit to reduce or to eliminate completely the use of **SP** axioms during inference search.

7 Proof search in SAD

It was mentioned above that a special sequent formalism was developed and implemented in [Moses], the prover of SAD. In order to make this paper self-contained, we give a brief description of a certain modification GD' of the calculus GD introduced in [26] and used in [Moses].

7.1 Admissible substitutions

We work in classical first-order logic with negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$), universal and existential quantifiers ($\forall$ and $\exists$).

Let Γ be a set of closed formulas such that no two quantifiers in Γ bind the same variable (a *collision-free* set). Let $\mathcal{V}_\Gamma$ be the set of *indexed variables* $\{\, {}^k v \mid v \text{ occurs in } \Gamma,\ k \in \mathbf{N} \,\}$. The expression ${}^k F$ will denote the formula F with each variable v replaced with an indexed variable ${}^k v$.

We write $F\lfloor G^+ \rfloor$ ($F\lfloor G^- \rfloor$) to denote a formula F with a positive (respectively, negative) occurrence of a subformula G. A variable ${}^k v \in \mathcal{V}_\Gamma$ is said to be *unknown* in Γ (${}^k v \in \mathcal{V}^+$) if for some formula $P \in \Gamma$, $P\lfloor (\forall v F)^+ \rfloor$ or $P\lfloor (\exists v F)^- \rfloor$. Correspondingly, x is said to be *fixed* in Γ (${}^k v \in \mathcal{V}^-$) if for some $P \in \Gamma$, $P\lfloor (\forall v F)^- \rfloor$ or $P\lfloor (\exists v F)^+ \rfloor$. Obviously, each variable in $\mathcal{V}_\Gamma$ is either unknown or fixed in Γ. Below, we typeset fixed variables in bold: ${}^k\mathbf{v}$.

The set Γ induces a relation $\prec_\Gamma$: $\mathcal{V}_\Gamma^+ \times \mathcal{V}_\Gamma^-$ as follows: ${}^k u \prec_\Gamma {}^m\mathbf{w}$ if and only if $k = m$ and the quantifier on w occurs in the scope of the quantifier on u in Γ, that is, some formula $F \in \Gamma$ is of the form $(\ldots \mathcal{Q}_1\, u\, (\ldots \mathcal{Q}_2\, w\, (\ldots) \ldots) \ldots)$.

Given a substitution σ, we define another relation $\lll_\Gamma^\sigma$: $\mathcal{V}_\Gamma^- \times \mathcal{V}_\Gamma^+$ as follows: ${}^m\mathbf{w} \lll_\Gamma^\sigma {}^k u$ if and only if ${}^m\mathbf{w}$ occurs in ${}^k u\sigma$.

A substitution σ is *admissible* in Γ whenever the following conditions hold:

(1) for any fixed ${}^k\mathbf{w} \in \mathcal{V}_\Gamma^-$, ${}^k\mathbf{w}\sigma = {}^m\mathbf{w}$, for some index m;
(2) for any ${}^k\mathbf{w}, {}^m\mathbf{w} \in \mathcal{V}_\Gamma^-$, ${}^k u \in \mathcal{V}_\Gamma^+$, if ${}^k\mathbf{w}\sigma = {}^m\mathbf{w}\sigma$ and ${}^k u \prec_\Gamma {}^k\mathbf{w}$, then ${}^k u\sigma = {}^m u\sigma$;
(3) the transitive closure of the composition $(\lll_\Gamma^\sigma \circ \prec_\Gamma)$ is irreflexive.

A kind of the Herbrand theorem can be proved for admissible substitutions:

Proposition 9 Let Γ be a collision-free set of closed formulas. The set Γ' is obtained from Γ by erasing all the quantifiers in formulas (so that all the variables in Γ' are free). The set Γ is consistent if and only if for any $n \in \mathbf{N}$ and any finite admissible substitution σ in the signature of Γ, the set $\{{}^1\Gamma', \ldots, {}^n\Gamma'\}\sigma$ is consistent.

7.2 Inference rules

The calculus GD' deals with *a-sequents*, which are expressions of the form $[\Lambda]\ \Gamma \to G$, where Λ is a list of literals, called *framed literals*, Γ is a set of formulas, called *premises*, and G is a *goal formula*. In our setting, Λ and G contain indexed variables, and Γ contains non-indexed variables.

When a proof of an initial a-sequent is searched in GD', an *inference tree* is constructed. At the beginning of a search process the tree consists of the initial a-sequent. The tree grows from top to bottom, with the subsequent nodes generated in accordance with the inference rules of GD'.

At some moments of inference, branches in the tree may be *terminated*: a leaf node containing an equation of the form $\langle L_1 \approx L_2 \rangle$ (where L_1, L_2 are literals) is added to such a branch and no more expansions are possible on it.

By $F^\neg$ we denote the negation of F with the negation sign moved inside F:

$$
\begin{array}{lllll}
(\forall\, x\ P)^\neg = \exists\, x\, \neg P & (P \vee Q)^\neg = \neg P \wedge \neg Q & (P \supset Q)^\neg = P \wedge \neg Q & & \\
(\exists\, x\ P)^\neg = \forall\, x\, \neg P & (P \wedge Q)^\neg = \neg P \vee \neg Q & (\neg P)^\neg = P & A^\neg = \neg A &
\end{array}
$$

Figure 5 summarizes the inference rules of GD'. In the figure, A, B denote unifiable atoms and L, M denote unifiable literals. The expression ${}^*F$ stands for ${}^kF$ where k is a fresh index with respect to the whole inference tree. Recall that premises are considered as unordered and frame literals, as ordered, so that L denotes the last literal in $[\Lambda, L]$.

The following *goal-drivenness constraint* is applied to the inferences in GD'. Whenever an AG-rule is applied and the atom A (or its negation) goes to the end of the list of framed literals, the corresponding occurrence of B in ${}^*F^\neg$ must be fixed and remembered. We require that the subsequent applications of goal-splitting rules never remove that occurrence of B from the proof. In other words, that occurrence of B must form a literal goal after a number of splittings. Then, we require the branch containing this goal to be terminated by an application of TR_Λ-rule, with the equation $\langle A \approx B \rangle$ in the leaf.

7.3 Proof trees

Let us consider an inference tree T where every branch is terminated. Let Γ denote the set of premises in the initial a-sequent. Let E denote the overall

Goal-Splitting Rules (GS)

$$(\rightarrow \forall): \quad \frac{[\Lambda]\ \ \Gamma \rightarrow \forall\, x\, F}{[\Lambda]\ \ \Gamma \rightarrow F} \qquad (\rightarrow \exists): \quad \frac{[\Lambda]\ \ \Gamma \rightarrow \exists\, x\, F}{[\Lambda]\ \ \Gamma \rightarrow F}$$

$$(\rightarrow\supset)_1: \quad \frac{[\Lambda]\ \ \Gamma \rightarrow F \supset G}{[\Lambda]\ \ \Gamma \rightarrow G} \qquad (\rightarrow\supset)_2: \quad \frac{[\Lambda]\ \ \Gamma \rightarrow F \supset G}{[\Lambda]\ \ \Gamma \rightarrow F^{\neg}}$$

$$(\rightarrow \vee)_1: \quad \frac{[\Lambda]\ \ \Gamma \rightarrow F \vee G}{[\Lambda]\ \ \Gamma \rightarrow G} \qquad (\rightarrow \vee)_2: \quad \frac{[\Lambda]\ \ \Gamma \rightarrow F \vee G}{[\Lambda]\ \ \Gamma \rightarrow F}$$

$$(\rightarrow \wedge): \quad \frac{[\Lambda]\ \ \Gamma \rightarrow F \wedge G}{[\Lambda]\ \ \Gamma \rightarrow F \quad [\Lambda]\ \ \Gamma \rightarrow G} \qquad (\rightarrow \neg): \quad \frac{[\Lambda]\ \ \Gamma \rightarrow \neg F}{[\Lambda]\ \ \Gamma \rightarrow F^{\neg}}$$

Auxiliary-Goal Rules (AG)

$$(AG^+): \quad \frac{[\Lambda]\ \ \Gamma, F\lfloor B^+ \rfloor \rightarrow A}{[\Lambda, \neg A]\ \ \Gamma, F \rightarrow {}^*F^{\neg}} \qquad (AG^-): \quad \frac{[\Lambda]\ \ \Gamma, F\lfloor B^- \rfloor \rightarrow \neg A}{[\Lambda, A]\ \ \Gamma, F \rightarrow {}^*F^{\neg}}$$

Termination Rules (TR)

$$(TR_\Lambda): \quad \frac{[\Lambda_1, M, \Lambda_2]\ \ \Gamma \rightarrow L}{\langle M \approx L \rangle} \qquad (TR_\Gamma): \quad \frac{[\Lambda]\ \ \Gamma, M \rightarrow L}{\langle {}^*M \approx L \rangle}$$

Fig. 5. Inference rules of GD'

system of equations in the leaves of the tree.

The inference tree T is considered to be a *proof tree* whenever there exists a substitution σ, admissible in Γ, which is a solution for E (that is, $E\sigma$ is a set of identities). The initial a-sequent of a proof tree is called *deducible* in GD'.

Proposition 10 (soundness and completeness of GD') Let Γ be a consistent set of formulas and G be a formula. The sequent $\Gamma \rightarrow G$ is deducible in Gentzen's calculus **LK** [27] if and only if the a-sequent $[\ ]\ \Gamma, \neg G \rightarrow {}^0G$ is deducible in the calculus GD'.

Note that equation solving is separated from application of inference rules. Equations may be accumulated and sent to a solver in an arbitrary moment. This property of GD' allows to organize a flexible proof search procedure. Due to the absence of preliminary skolemization, the initial signature is preserved and, therefore, we can use a relevant domain-oriented solver. The submodule of [Moses] responsible for equation handling acts as a mediator between the prover and external solvers. It checks a substitution found by a solver for admissibility and builds additional equations if necessary. The procedure computing the most general unifier is used as a solver by default.

7.4 GD' as a clausal tableau calculus

Let us demonstrate the deductive technique of SAD on a simple propositional problem. Assume that we want to prove the sequent $A \vee (B \vee A) \to A \vee B$. Let $\Gamma = \{A \vee (B \vee A), \neg(A \vee B)\}$. A GD'-inference used to prove the a-sequent $[\,]\ \Gamma \to A \vee B$ is shown in Figure 6.

$$
\dfrac{[\ ]\ \Gamma \to A \vee B}{\dfrac{[\ ]\ \Gamma \to A}{\dfrac{[\neg A]\ \Gamma \to \neg A \wedge \neg(B \vee A)}{\dfrac{[\neg A]\ \Gamma \to \neg A}{\langle \neg A \approx \neg A\rangle}\,(TF_\Lambda) \qquad \dfrac{[\neg A]\ \Gamma \to \neg(B \vee A)}{\dfrac{[\neg A]\ \Gamma \to \neg B \wedge \neg A}{\dfrac{[\neg A]\ \Gamma \to \neg B}{\dfrac{[\neg A, B]\ \Gamma \to A \vee B}{\dfrac{[\neg A, B]\ \Gamma \to B}{\langle B \approx B\rangle}\,(TF_\Lambda)}\,(\to\vee)}\,(AG^-) \qquad \dfrac{[\neg A]\ \Gamma \to \neg A}{\langle \neg A \approx \neg A\rangle}\,(TF_\Lambda)}\,(\to\wedge)}\,(\to\neg)}\,(\to\wedge)}\,(AG^+)}\,(\to\vee)_1
$$

Fig. 6. Proof tree in GD'

Note that in any GD'-inference the premises are the same in each a-sequent. Moreover, the list of framed literals in a given node is exactly the list of the complements of literal goals above that node. Then we can present the proof tree given above in a quite abbreviated form, as the tree of literal goals. In the inference shown in Figure 7, applications of the auxiliary-goal rule are joined with the subsequent applications of goal-splitting rules in a single inference step, and termination by a framed literal becomes termination by a contradiction in a branch.

$$
\dfrac{[\ ]\ A \vee (B \vee A), \neg(A \vee B) \to A \vee B}{\dfrac{A}{\dfrac{\neg A}{\langle \neg A \approx \neg A\rangle}\,(TF_\Lambda) \quad \dfrac{\neg B}{\dfrac{B}{\langle B \approx B\rangle}\,(TF_\Lambda)}\,(AG^- + GS) \quad \dfrac{\neg A}{\langle \neg A \approx \neg A\rangle}\,(TF_\Lambda)}\,(AG^+ + GS)}\,(GS)
$$

Fig. 7. Abbreviated proof tree in GD'

Consider now the clauses resulting from a "naive" clausification of Γ:

$$\mathbf{Cls}(\Gamma) = \{\, A \vee B \vee A,\ \neg A,\ \neg B \,\}$$

We see that literals expanding non-leaf nodes in the abbreviated proof tree come from the complemented clauses above. In fact, the goal-splitting rules of GD' are exactly the rules of "naive" clausification (with the only difference that quantifiers are ignored and no Skolem symbols arise). Then we can reformulate our calculus as shown in Figure 8.

<ins>*Start rule (SR)*</ins>

$$\frac{[\ ]\ \ \Gamma \to G}{[\ ]\ \ \Gamma \to L_1^{\neg} \quad \dots \quad [\ ]\ \ \Gamma \to L_n^{\neg}}$$

<ins>*Auxiliary-Goal Rules (AG)*</ins>

$$(AG^{+}): \quad \frac{[\Lambda]\ \ \Gamma \to A}{\langle \neg A \approx {}^{k}L^{\neg} \rangle \quad [\Lambda, \neg A]\ \ \Gamma \to {}^{k}L_1^{\neg} \quad \dots \quad [\Lambda, \neg A]\ \ \Gamma \to {}^{k}L_n^{\neg}}$$

$$(AG^{-}): \quad \frac{[\Lambda]\ \ \Gamma \to \neg A}{\langle A \approx {}^{k}L^{\neg} \rangle \quad [\Lambda, A]\ \ \Gamma \to {}^{k}L_1^{\neg} \quad \dots \quad [\Lambda, A]\ \ \Gamma \to {}^{k}L_n^{\neg}}$$

<ins>*Termination Rules (TR)*</ins>

$$(TR_{\Lambda}): \quad \frac{[\Lambda_1, M, \Lambda_2]\ \ \Gamma \to L}{\langle M \approx L \rangle} \qquad (TR_{\Gamma}): \quad \frac{[\Lambda]\ \ \Gamma, M \to L}{\langle {}^{k}M \approx L \rangle}$$

Fig. 8. Inference rules of GD' (reformulation)

In the start rule, $(L_1 \vee \cdots \vee L_n)$ is one of the clauses from $\mathbf{Cls}(\neg G)$. In the auxiliary-goal rules, $(L \vee L_1 \vee \cdots \vee L_n)$ belongs to $\mathbf{Cls}(\Gamma)$. In the auxiliary-goal and termination rules, k is a fresh index with respect to the whole inference. Note that the goal-drivenness constraint is now incorporated in the AG-rules.

Now, since premises and framed literals can be omitted and we can work directly with trees of literals our reformulation of GD' becomes just a sequent-based counterpart of the *(strong) connection tableau calculus* also known as *model elimination* [7]. Of course, there is still a difference in quantifier handling: GD' uses the notion of admissible substitution instead of skolemizing initial formulas.

8 Related approaches and systems

We find convenient to consider the variety of existing approaches with respect to three characteristics: the style of formalization they suggest, the style of proof they require, and the granularity of proof they accept. We will try to show the place of the SAD system in this three-dimensional space.

Formalization style. By this, we mean the choice of preliminaries, the form of definitions (whether they are recursive), the way of reasoning (whether it is constructive, or strictly typed, or calculation-based), and so on. It is obvious that this style is influenced, above all, by the choice of the base logic and of fundamental theories used in formalizations.

At present, two main trends consist in adopting higher-order logic (type theory) and first-order logic (set theory). Types are used in the majority of the well-known mathematical assistants such as Isabelle/HOL [28], Coq [29], ΩMEGA [30], PVS [31], HOL [32], Automath [33], λCLAM [34], and others. The type-theoretic approach favors inductively defined domains, recursive definitions and is completely natural to formalize the concepts of programming or engineering. It is, probably, not so natural for traditional mathematics [35] — not denying that most of the mentioned systems accumulate a vast collection of purely mathematical theories.

On the other side we see the system Mizar [19] that uses classical first-order logic and Tarski-Grothendieck set theory. This approach corresponds well to the traditional style of mathematical presentation and the Mizar Mathematical Library constitutes, to our knowledge, the largest collection of formalized mathematical texts for today.

Classical first-order logic was the choice of SAD, too. The SAD system does not adopt any kind of set theory (or any other foundational theory) as the common basis for all formalizations. We prefer to define a particular set of preliminaries for a problem under consideration, choosing the basic concepts on the appropriate level of the development of the theory. (In other words, SAD does not grow a tall tree but works in the garden of green bushes.)

A special kind of logic was adopted in the Nqthm prover [36] and its successor, ACL2 [37]: quantifier-free first-order logic with the theory of inductively defined data structures. This basis resembles the type-theoretic approach in that it is convenient to define recursive functions and formalize induction-based reasoning over finitely generated domains (integers, trees, etc).

The Theorema project [38] works in higher-order predicate logic, enriched with sequence variables and binding constructions: lambda-abstractor, sum, limit, and so on.

Proof style. Another important property of a proof assistant is what kind of input it takes. Interactive systems are most often tactic-driven, meaning that a given statement is proved by a sequence of instructions given to the system. These instructions, *tactics*, can be primitive, like applying an inference rule, or rather complex, like applying a simplification routine or a decision procedure. Systems of that type are Isabelle, PVS, Coq, HOL and others. Working with such a system is easy if it provides a terse set of powerful tactics which is enough most of the time.

The systems of another kind accept the proofs written in the same language that axioms and propositions to prove. Of course, this language must be extended with certain structuring means to organize logic formulas in a proof. Then the system must autonomously verify each proof step. Mizar and SAD are systems of that kind; and Isabelle, with introduction of Isar [20] (a structured proof language imitating a language of mathematical proofs) can be considered as a proof-driven system, too.

Though there are no "proof sections" in Nqthm and ACL2, they are also systems of the second kind, where verification of a principal statement is supported with a number of supplementary preceding lemmas.

Proof checkers like Automath are proof-driven systems by definition, though, in Automath, the proof is just a typed lambda-term inhabiting a type that represents a statement being proved (many higher-order tactic-based proof assistants use the same representation in the background).

Of course, the distinction is quite conditional. If one can prove theorems in a tactic-driven proof assistant using mostly the tactics of intermediate goal introduction and of automated subgoal closure, such a system can be as well seen as a proof-driven. If the steps in a proof you submit to a proof-driven system should be supplied with detailed hints to the verifier, such a system can be as well seen as a tactic-driven, with the tactics being your hints (the case of Isar).

The systems using proof planning, such as λClam, Ωmega, and IsaPlanner [39] stand apart. In a proof planner, we do not supply a particular problem with a particular proof, whether explicit or encoded in tactic applications. Instead, the system attempts to find the proof without user assistance by generating a proof plan from the collection of general proof methods whose usage is guided by method preconditions, various control rules, or strategies. Of course, any proof method, control rule, or strategy is supposed to be applicable in a quite large class of problems, not just for the problem in question.

Proof granularity. The reasoning power of a mathematical assistant can be weaker or stronger, requiring a user to perform more or less proof steps by himself: measured by the number of instructions, in a tactic-driven system, or by the length of written proofs, in a proof-driven one. Among the mathematical assistants, we can distinguish proof checkers and "proof finders". The former accept only those proofs where each step consists in an inference rule application, and hence supplied proofs have to be fully detailed. Mizar is a system of that kind, though the set of inference rules of Mizar is quite large. Automath is another pure proof checker, with a different kind of proofs.

Systems which we called "proof finders" apply proof search or proof planning methods and try to close the gaps by their own means. SAD, Nqthm (ACL2), Theorema are proof-driven systems of that kind. λCLAM and ΩMEGA, the systems using proof planning, should be classified as proof finders, too.

Interactive tactic-driven systems usually have enhanceable sets of tactics, so that the "reasoning power" is not specific to a system as such, and almost any tactic-driven system can be classified as a proof finder. However, our experiments with Coq and Isabelle show that one strongly needs to follow the suggested formalization style: if you try to prove a complex theorem "at once", without splitting it to a number of ad-hoc lemmas, without using special tactics and existing libraries, the "proof construction dialog" quickly becomes complex, wide-branched, and hardly traceable.

9 Future work

At present, the researches in the framework of the Evidence Algorithm advance in the following directions:

Background prover: equality handling. Efficient handling of equality in sequent-based or tableau calculi stays a challenging problem for prover developers. Recently, we developed an extension of the Model Elimination calculus with a lazy paramodulation rule [40] and proved it to be complete in classical first-order logic with equality. This calculus, called LPCT (for Lazy Paramodulation Connection Tableau), uses basic paramodulation and term ordering. The corresponding modification of [Moses] is under development now.

Background prover: simple rules. In ForTheL, any object is introduced as member of a class described by some notion. The hierarchy of notions is, in essence, a kind of weak typing system (not to confuse with the weak typing in WTT [22]), and, therefore, proving that a given individual belongs to a given class is a frequently arising subgoal in inference search. Quite often, this subgoal can be resolved with the help of what we call a *simple rule*. In its general form, a simple rule is an implication of the form $\forall v_1, \dots, v_n \, Q_1(v_{i_1}) \wedge \dots \wedge Q_m(v_{i_m}) \supset Q(s[v_1, \dots, v_n])$, where $Q, Q_1, \dots, Q_m$ are unary predicate symbols. Given a set of simple rules Δ, an unary predicate symbol Q, and a ground term t, it is possible to decide whether $\Delta \vdash Q(t)$ in polynomial time with respect to the length of t. This property of simple rules allows to extend a resolution-based or a tableau-based calculus with a non-destructive inference rule that, in case it can be applied, removes a literal from a clause (in resolution) or closes a branch without substituting in the rest of the tree (in tableau).

Background prover: many-sorted language. To be implemented consistently, the method of second-order superposition combinators (Section 6) requires a prover to support sorts.

Foreground reasoner: definition expansion. As our experience shows, definitions complicate inference search when being passed to a prover "as is". In particular, it is explained by their "double-edged", self-applicable form. Consider the definition of the subset relation: $A \subseteq B \equiv \forall x (x \in A \supset x \in B)$. After a traditional clausification, this equivalence turns into three clauses: $(A \nsubseteq B \vee x \notin A \vee x \in B)$, $(sk(A, B) \in A \vee A \subseteq B)$, $(sk(A, B) \notin B \vee A \subseteq B)$. The first clause resolves with the second one, producing the clause $(sk(A, B) \in A \vee x \notin A \vee x \in B)$. The resulting clause is non-tautological and will not be eliminated from the search space, yet it is completely useless.

At present, we investigate another approach: the reasoner does not include definitions in a task for the prover; instead, provided the inference search fails, the reasoner expands some defined symbols and sends the task again. A weaker variant is to include definitions but make some expansions in advance.

Foreground reasoner: recursive definitions. This important extension is to be made soon in SAD. A recursive definition must be a subject for verification: the existence of the described object must be proved. A similar extension should be made to support the set comprehension scheme. Of course, these modifications will concern ForTheL as well.

Language: tuples. Mathematical objects defined as tuples are frequently met: a group is defined as a triple (carrier set, multiplication, inverse), a graph as a pair (vertex set, edge set), and so on. Support for the tuples, in particular, in definitions, is one of the facilities lacking in ForTheL.

System as a whole. The SAD team is currently working to provide SAD with tools for maintaining a library of mathematical texts. The most important concern here is how to estimate the (ir)relevance of a given fact with respect to the current proof task.

Acknowledgments. We are grateful to the referee for his valuable comments and suggestions which were extremely useful. We thank everybody who took part in the development of the SAD system.

References

[1] V. M. Glushkov, Some problems of automata theory and artificial intelligence (in Russian), Kibernetika 2 (1970) 3–13.

[2] V. Glushkov, K. Vershinin, et al., On a formal language for description of mathematical texts (in Russian), in: Avtomatizatsiya poiska dokazatel'stv teorem v matematike, Institute of Cybernetics, Kiev, 1974, pp. 3–36.

[3] K. Vershinin, Using auxiliary statements in the process of proof search (in Russian), Semiotika i Informatika 12 (1979) 12–21.

[4] The Evidence Algorithm project, http://ea.unicyb.kiev.ua.

[5] K. Verchinine, A. Degtyarev, A. Lyaletski, A. Paskevich, SAD, a System for Automated Deduction: a current state, in: F. Kamareddine (Ed.), Proc. Workshop on 35 Years of Automath, Heriot-Watt University, Edinburgh, Scotland, 2002.

[6] A. Lyaletski, A. Paskevich, K. Verchinine, Theorem proving and proof verification in the system SAD, in: A. Asperti, G. Bancerek, A. Trybulec (Eds.), Mathematical Knowledge Management: Third International Conference, MKM 2004, Vol. 3119 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 236–250.

[7] R. Letz, G. Stenz, Model elimination and connection tableau procedures, in: A. Robinson, A. Voronkov (Eds.), Handbook for Automated Reasoning, Vol. II, Elsevier Science, 2001, pp. 2017–2116.

[8] D. Brand, Proving theorems with the modification method, SIAM Journal of Computing 4 (1975) 412–430.

[9] W. McCune, Otter 3.0 reference manual and guide, Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, USA (1994).

[10] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, D. Topic, SPASS Version 2.0, in: A. Voronkov (Ed.), Automated Deduction: 18th International Conference, CADE-18, Vol. 2392 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 275–279.

[11] A. Riazanov, A. Voronkov, The design and implementation of VAMPIRE, AI Communications 15 (2–3) (2002) 91–110.

[12] G. Sutcliffe, C. B. Suttner, T. Yemenis, The TPTP problem library, in: A. Bundy (Ed.), Automated Deduction: 12th International Conference, CADE-12, Vol. 814 of Lecture Notes in Computer Science, Springer-Verlag, 1994, pp. 252–266, see also http://tptp.org.

[13] S. P. Jones, Haskell 98 Language and Libraries, Cambridge University Press, 2003.

[14] K. Vershinin, A. Paskevich, ForTheL — the language of formal theories, International Journal of Information Theories and Applications 7 (3) (2000) 120–126.

[15] K. Verchinine, A. Paskevich, ForTheL Reference Manual (unpublished draft), hosted at http://ea.unicyb.kiev.ua/download/forthel.ps.gz (2004).

[16] The QED Project, http://www.mcs.anl.gov/qed/.

[17] N. E. Fuchs, U. Schwertel, R. Schwitter, Attempto Controlled English — not just another logic specification language, in: P. Flener (Ed.), Logic-Based Program Synthesis and Transformation: 8th International Workshop, LOPSTR'98, Vol. 1559 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 1–20.

[18] J. F. Sowa, Common logic controlled english (unpublished draft), hosted at http://www.jfsowa.com/clce/specs.htm (2004).

[19] A. Trybulec, H. Blair, Computer assisted reasoning with Mizar, in: Proc. 9th International Joint Conference on Artificial Intelligence, 1985, pp. 26–28.

[20] M. Wenzel, Isar — a generic interpretative approach to readable formal proof documents, in: Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99, Vol. 1690 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 167–184.

[21] R. P. Nederpelt, J. H. Geuvers, R. C. de Vrijer (Eds.), Selected Papers on Automath, North-Holland, 1994.

[22] F. Kamareddine, R. P. Nederpelt, A Refinement of de Bruijn's Formal Language of Mathematics, Journal of Logic, Language and Information 13 (3) (2004) 287–340.

[23] F. Wiedijk, The seventeen provers of the world (unpublished draft), hosted at http://www.cs.ru.nl/~freek/comparison/ (2004).

[24] H. Curry, R. Feys, Combinatory Logic, Vol. 1, North-Holland, 1958.

[25] H. Rogers, Theory of Recursive Functions and Effective Computability, McGraw-Hill, 1967.

[26] A. Lyaletski, K. Verchinine, A. Degtyarev, A. Paskevich, System for Automated Deduction (SAD): Linguistic and deductive peculiarities, in: M. A. Klopotek, S. T. Wierzchon, M. Michalewicz (Eds.), Intelligent Information Systems: IIS'2002 Symposium, Advances in Soft Computing, Physica-Verlag, 2002, pp. 413–422.

[27] G. Gentzen, Untersuchungen uber das Logische Schliessen, Math. Zeit. 39 (1934) 176–210.

[28] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Vol. 2283 of Lecture Notes in Computer Science, Springer-Verlag, 2002.

[29] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, B. Werner, The Coq proof assistant reference manual — version v6.1, Tech. Rep. 0203, INRIA (1997).

[30] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. H. Siekmann, V. Sorge, ΩMEGA: Towards a mathematical assistant, in: W. McCune (Ed.), Automated Deduction: 14th International Conference, CADE-14, Vol. 1249 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 252–255.

[31] S. Owre, J. M. Rushby, N. Shankar, PVS: a prototype verification system, in: D. Kapur (Ed.), Automated Deduction: 11th International Conference, CADE-11, Vol. 607 of Lecture Notes in Computer Science, Springer-Verlag, 1992, pp. 748–752.

[32] M. J. C. Gordon, T. F. Melham, Introduction to HOL: a theorem proving environment for higher order logic, Cambridge University Press, 1993.

[33] R. P. Nederpelt, J. H. Geuvers, R. C. de Vrijer (Eds.), Selected Papers on Automath, Vol. 133 of Studies in Logic and the Foundations of Mathematics, North-Holland, 1994.

[34] J. Richardson, A. Smaill, I. Green, System description: Proof planning in higher-order logic with lambda-clam, in: Automated Deduction: 15th International Conference, CADE-15, Vol. 1421 of Lecture Notes in Computer Science, Springer-Verlag, 1998, pp. 129–133.

[35] L. Lamport, Types considered harmful, DEC SRC internal note (Dec. 1992).

[36] R. S. Boyer, J. S. Moore, A Computational Logic Handbook (2nd edition), Academic Press, 1998.

[37] M. Kaufmann, P. Manolios, J. S. Moore, Computer-Aided Reasoning: An Approach, Kluwer Academic Publishers, 2000.

[38] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, D. Vasaru, A survey of the Theorema project, in: W. Küchlin (Ed.), ISSAC'97 — Proc. International Symposium on Symbolic and Algebraic Computation, ACM Press, Maui, Hawaii, USA, 1997, pp. 384–391.

[39] L. Dixon, J. D. Fleuriot, IsaPlanner: A prototype proof planner in Isabelle, in: F. Baader (Ed.), Automated Deduction: 19th International Conference, CADE-19, Vol. 2741 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 279–283.

[40] W. Snyder, C. Lynch, Goal directed strategies for paramodulation, in: Rewriting Techniques and Applications: 4th International Conference, RTA 1991, Vol. 448 of Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 150–161.