Proof Generation for Saturating First-Order Theorem Provers

Stephan Schulz[1] and Geoff Sutcliffe[2]

[1] DHBW Stuttgart
schulz@eprover.org
[2] University of Miami
geoff@cs.miami.edu

1 Introduction

First-order Automated Theorem Proving (ATP) is one of the oldest and most developed areas of automated reasoning. Today, the most widely used first-order provers are fully automatic and process first-order logic with equality. Many state-of-the-art ATP systems consist of a clausifier, translating a full first-order problem specification into clause normal form, and a saturation procedure that tries to derive the empty clause to complete a proof by contradiction. Saturation procedures are typically based on variants of the superposition calculus, often combining restricted forms of paramodulation with resolution and strong redundancy elimination techniques, in particular simplification via rewriting and subsumption. The first widely used ATP system in this mold was Bill McCune's Otter [36], now succeeded by Prover9 [34]. Other major examples include Vampire [62, 47], SPASS [63, 64], and E [53, 55].

Proof production was not a primary concern early on, and provers offered different levels of support for explicit proof objects. Information was output in a variety of formats. Nowadays, proof object output is supported by most major provers, and many systems support the TPTP-3/TSTP syntax [60] for proof output. In this syntax, proofs are represented as directed acyclic graphs (DAGs), where each node is annotated with a clause or formula used in the proof. The original axioms, assumptions and goals are nodes with in-degree 0, i.e. they correspond to leaves if the DAG is unfolded into a proof tree. Inner nodes represent derived clauses and formulas, linked to the premises used in their derivation via incoming edges. The final node of the proof graph (i.e. the root in a proof tree) is the empty clause, concluding the proof by contradiction. Nodes are annotated with the inference(s) used to produce them.

The main difficulty in obtaining proof objects is the very high rate of inferences and simplifications during proof search. Most of these inferences do not contribute to the final proof, but the actual proof steps can only be identified a-posteriori. Hence careful book-keeping is necessary. If done naively, the amount of data quickly becomes unmanageable. Different provers have taken different approaches to handling this problem, either dumping all derivation steps to an external medium, keeping a full record of all inferences in main memory, or uti-

lizing invariants of the proof search algorithm that enable proof reconstruction from less extensive records.

Both SPASS and E are mainstream proof-producing provers available under open-source/free software licenses. E in particular implements the TPTP standard for proof output and includes a derivation not only for the saturation, but also for the clausification steps.

2 Calculi and Proof Systems

It was understood early on that showing the validity of a sentence in first-order logic can be reduced to demonstrating the unsatisfiability of a set of clauses. Indeed, for a long time "first order theorem proving" was nearly synonymous with "showing unsatisfiability of a formula in clause normal form". This, again, can be reduced to finding an unsatisfiable set of ground instances, or to deduce the empty clause (an explicit witness of unsatisfiability) from a set of clauses. Major early milestones were the original Davis-Putnam algorithm [9], which combined the generation of ground instances with a separate propositional satisfiability test, and Robinson's resolution [49], which uses unification to integrate instantiation and the search for an explicit contradiction in one simple inference process.

Resolution was the first major example of a saturating calculus. The search state is represented by a set of clauses. New clauses are systematically deduced using a set of *inference rules* and added to the search state. The aim is to eventually derive the empty clause. Resolution has proved to be an extremely productive line of research, and spawned a number of refinements, including ordered resolution [46] and hyper-resolution [50]. The general saturation principle and many of the inferences and techniques survive into current ATP systems.

Paramodulation [48] was introduced as a way to handle the important equality relation with an explicit inference rule. However, pure paramodulation was no significant improvement over resolution with an axiomatic description of equality. In 1970, Knuth and Bendix introduced *completion* [23] as a way to efficiently handle some pure unit equality problems, using a term ordering to transform a set of equations into a confluent rewrite rule system. This was later extended to *completion without failure* [19, 2], which provides a complete proof method for unit-equational theories. In contrast to pure resolution calculi, completion based methods make extensive use of *simplification*, in particular through rewriting. Simplification replaces a clause in the search state by a different, in some sense simpler, clause.

Resolution and completion-based techniques have merged in the current generation of superposition calculi [3, 4, 38]. These calculi combine paramodulation and (possibly) resolution inferences restricted by literal selection and orderings on terms and literals with powerful redundancy elimination techniques, in particular rewriting and subsumption. Most practical implementations combine superposition with variants of resolution to handle non-equational literals, others (in particular E) encode non-equational literals and handle them in a uniform way via superposition and simplification.

Figure 1 shows examples of the most prolific generating inference rules (*superposition*) and the most important simplification rules (unconditional rewriting) as an example of the type of rules used in saturating calculi for first-order deduction. There are additional generating inference rules (in particular *equality factoring* and *equality resolution*) that are necessary for completeness of the calculus. However, in practical applications, typically more than 95% of generating inferences are superposition or resolution inferences. Simplification has been shown to be critical for the success of the proof search, and simplification effort dominates the overall effort of most saturating first-order provers.

(SN) $$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)}$$ if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

(RN) $$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R}$$ if $u|_p = \sigma(s)$ and $\sigma(s) > \sigma(t)$.

Note that superposition is a *generating inference* (symbolized by the single line separating premises and conclusion), while rewriting is a simplifying inference (the conclusions replace the premises in the search space). The rules are instantiated with a *simplification ordering* $<$ on terms, lifted to literals and clauses, and optionally a literal selection function. See e.g. [53] for details of the notation. These rules are complemented by dual rules for positive literals with slightly different conditions for the inference.

Fig. 1. Exemplary inference rules of the superposition calculus

Over time, there have been various other approaches to the CNF refutation problem. These include model elimination [30, 29] implemented, e.g., in SETHEO [28, 37] and leanCOP [42], model evolution [6] implemented in Darwin [5], and modern instantiation-based methods as implemented in iProver [25]. These do not naturally generate a derivation-based proof. However, such a proof can generally be extracted from information gathered during the proof search.

CNF translation has been performed using straightforward algorithms as, e.g., described in [31], more often than not by tools external to the main refutation prover. As a result, the clausification process was often not considered part of the proof search, and was not represented in any proof object. FLOTTER [63] first demonstrated that advanced clausification methods as described in [41] can significantly increase the class of first-order problems that can be solved by automated theorem provers. However, FLOTTER (and its accompanying prover SPASS) do not provide the clausification steps in a form useful for a proof object. E and Vampire implement clausifiers using similar techniques as

FLOTTER, and are able to provide complete proof objects, including clausification and saturation.

3 Proof Search

All mainstream saturating provers are based on some version of the *given-clause algorithm*. This algorithm represents the proof state by two distinct sets of clauses, the set P of *processed clauses* (initially empty) and the set U of *unprocessed clauses*. In its simplest version, it moves clauses, one at a time, from U to P, at each step adding all new clauses that can be derived from the *given clause* and other premises in P using a single inference to U. Thus it maintains the invariant that all direct inferences between clauses in P are represented in $P \cup U$. Provers differ in how they add simplification to this algorithm. The DISCOUNT variant, first realized in the eponymous system [10] uses only clauses from P as side premises for simplification, and simplifies P, the given clause, and newly generated clauses. It is implemented in E, and, as an alternate method, in SPASS and VAMPIRE. The other main variant is named after *Otter*. It uses all clauses in U and P as side premises for simplification. It is implemented e.g. in Otter, Prover9, and, as an alternate method, in SPASS and Vampire.

For both of these variants, one critical parameter is the order in which given clauses are selected from U. Completeness of the proof procedure requires a rather weak fairness criterion (usually implemented by making sure that no clause is allowed to remain in U forever). However, this leaves a large amount of freedom, and heuristics for clause selection have a large effect on the practical power of an ATP system.

A major challenge for reconstructing proof objects is simplification. Simplification modifies clauses in the search state or even removes them completely. Thus, derivations that reference clauses later affected by simplification are left with dangling references. On the other hand, simplification is crucial for the success of theorem provers. Section 5 discusses possible solutions.

4 Proof Formats

Historically, there has been a large number of of languages for writing proof problems, and a different and only partially overlapping set of languages for writing proof objects.

Some languages, e.g., the LOP format [51], were designed for writing problems, and do not support writing solutions. Some languages for writing solutions are limited in scope, e.g., the PCL language [13] is limited to solutions to equational problems, and the OpenTheory language [20] is designed only to be a computer-processable form for systems that implement the HOL logic [16]. There are some general purpose languages that have features for writing derivations, e.g., Otter's `proof_object` format [35, 32] and the DFG syntax [17], but none of these (that we know of) also provide support for writing finite interpretations. Mark-up languages such as OmDoc [24], OpenMath [8], and MathML

[8] are quite expressive (especially for mathematical content), but their XML based format is not suitable for human processing. Most of these languages have not seen much use outside the groups that originally developed them.

The current standard for writing first-order problems and solutions is the TPTP language, version 3 [60]. The language was originally developed to realize the *Thousands of Problems for Theorem Provers* library [58]. Version 1 supported only clause normal form (CNF), version 2 added support for full first-order logic (FOF), and version 3 unified the syntax for CNF and FOF, and added the ability to represent proof objects, derivations, and models. Version 3 has also been conservatively extendet to cover other logics, in particular simply typed first-order logic and typed higher-order logic.

The language was designed to be suitable for writing both ATP problems and ATP solutions, to be flexible and extensible, and easily processed by both humans and computers. The syntax shares many features with Prolog, a language that is widely known in the ATP community. Indeed, with a few operator definitions, units of TPTP data can be read in Prolog using a single read/1 call, and written with a single writeq/1 call. The features were designed for writing derivations, but their flexibility makes it possible to write a range of DAG structures. Additionally, there are features of the language that make it possible to conveniently specify finite interpretations. The ability of the TPTP language to express solutions as well as problems, in conjunction with the simplicity of the syntax, sets it apart from other languages used in ATP. Overall, the TPTP language is more expressive and usable than other languages. It's use has been bolstered both by its use in the CADE ATP System Competition (CASC), and also by its support in many different provers and tools.

The TPTP language definition[3] uses a modified BNF meta-language that separates semantic, syntactic, lexical, and character-macro rules. Syntactic rules use the standard ::= separator, e.g.,

<source>   ::= <general_term>

When only a subset of the syntactically acceptable values for a non-terminal make semantic sense, a second rule for the non-terminal is provided using a :== separator, e.g.,

<source>   :== <dag_source> | <internal_source> | , etc.

Any further semantic rules that may be reached only from the right hand side of a semantic rule are also written using the :== separator, e.g.,

<dag_source> :== <name> | <inference_record>

This separation of syntax from semantics eases the task of building a syntactic analyzer, as only the ::= rules need be considered. At the same time, the semantic rules provide the detail necessary for semantic checking. The rules that produce tokens from the lexical level use a ::- separator, e.g.,

<lower_word> ::- <lower_alpha><alpha_numeric>*

with the bottom level character-macros defined by regular expressions in rules using a ::: separator, e.g.,

<lower_alpha> ::: [a-z]

[3] http://www.tptp.org/TPTP/SyntaxBNF.html

The top level building blocks of TPTP files are *annotated formulae*, *include directives*, and *comments*. An annotated formula has the form:

language`(`*name*`,` *role*`,` *formula*`[,` *source*`[,` *useful_info*`]]).`

The *language*s currently supported are `thf` - typed higher-order form, `tff` - typed first-order form, `fof` - first order form, and `cnf` - clause normal form. The *role* gives the user semantics of the *formula*, e.g., `axiom`, `lemma`, `conjecture`, and hence defines its use in an ATP system - see the BNF for the list of recognized roles and their meaning. The logical *formula* uses a consistent and easily understood notation [59] that can be seen in the BNF. The *source* describes where the formula came from, e.g., an input file or an inference. The *useful_info* is a list of arbitrary useful information, as required for user applications. The *useful_info* field is optional, and if it is not used then the *source* field becomes optional. An example of a FOF formula, supplied from a file, is:

```
fof(formula_27,axiom,
    ! [X,Y] :
      ( subclass(X,Y) <=>
        ! [U] :
          ( member(U,X) => member(U,Y) )),
    file('SET005+0.ax',subclass_defn),
    [description('Definition of subclass'), relevance(0.9)]).
```

An example of an inferred CNF formula is:

```
cnf(175,lemma,
    ( rsymProp(ib,sk_c3)
    | sk_c4 = sk_c3 ),
    inference(factor_simp,[status(thm)],[
        inference(para_into,[status(thm)],[96,78,theory(equality)])]),
    [iquote('para_into,96.2.1,78.1.1,factor_simp')]).
```

The source field of an annotated formula is most commonly a `file` record or an `inference` record. A `file` record stores the name of the file from which the annotated formula was read, and optionally the name of the annotated formula as it occurs in the file (this may be different from the name of the annotated formula itself, e.g., if the ATP system renames the annotated formulae that it reads in). An `inference` record stores three items of information about an inferred formula: the name of the inference rule; a list of "useful information items", and a list of the parents.

There currently is no fixed standard of supported inference rules, i.e. the inference rule is simply a name provided by the ATP system. However, the "useful information" field allows the system to specify the logical relation between a derived formula and its parents in the SZS ontology [59] – commonly, inferred formulae are theorems of their parents, but in some cases the semantic relationship is weaker, as in Skolemization steps. The parents are either names of existing clauses and formulas, nested inference records, or `theory` records. A theory record is used when the axioms of some theory are built into the inference rule.

A derivation is a directed acyclic graph (DAG) whose leaf nodes are formulae from the input, whose interior nodes are formulae inferred from parent formulae, and whose root nodes are the final derived formulae. For example, a proof of a FOF theorem from some axioms, by refutation of the CNF of the axioms and negated conjecture, is a derivation whose leaf nodes are the FOF axioms and conjecture, whose internal nodes are formed from the process of clausification and then from inferences performed on the clauses, and whose root node is the *false* formula.

The information required to record a derivation is, minimally, the leaf formulae, and each inferred formula with references to its parent formulae. More detailed information that may be recorded and useful includes: the name of the inference rule used in each inference step; sufficient details of each inference step to deterministically reproduce the inference; and the semantic relationships of inferred formulae with respect to their parents. The TPTP language is sufficient for recording all this, and more. A comprehensively recorded derivation provides the information required for various forms of processing, such as proof verification [57], proof visualization [56], and lemma extraction [14].

A derivation written in the TPTP language is a list of annotated formulae. Each annotated formula has a name, a role, and the logical formula. Each inferred formula has an `inference` record with the inference rule name, the semantic relationship of the formula to its parents as an SZS ontology value in a `status` record, and a list of references to its parent formulae.

Example. Consider the following toy FOF problem, to prove the `conjecture` from the `axioms`.

```
%------------------------------------------------------------------------
%----All (hu)men are created equal. John is a human. John got an F grade.
%----There is someone (a human) who got an A grade. An A grade is not
%----equal to an F grade. Grades are not human. Therefore there is a
%----human other than John.
fof(all_created_equal,axiom,(
    ! [H1,H2] : ( ( human(H1) & human(H2) ) => created_equal(H1,H2) ) )).
fof(john,axiom,(
    human(john) )).
fof(john_failed,axiom,(
    grade(john) = f )).
fof(someone_got_an_a,axiom,(
    ? [H] : ( human(H) & grade(H) = a ) )).
fof(distinct_grades,axiom,(
    a != f )).
fof(grades_not_human,axiom,(
    ! [G] : ~ human(grade(G)) )).
fof(someone_not_john,conjecture,(
    ? [H] : ( human(H) & H != john ) )).
%--------------------------------------------------------------------
```

Here is a derivation recording a proof by refutation of the CNF, adapted (removing inferences that simply copy the parent formula) from the one produced by the ATP system E 1.8 [55].

```
%------------------------------------------------------------------------
fof(c_0_0, conjecture,
    (?[X3]:(human(X3)&X3!=john)),
    file('CreatedEqual.p', someone_not_john)).
fof(c_0_1, axiom,
    (?[X3]:(human(X3)&grade(X3)=a)),
    file('CreatedEqual.p', someone_got_an_a)).
fof(c_0_2, axiom,
    (grade(john)=f),
    file('CreatedEqual.p', john_failed)).
fof(c_0_3, axiom,
    (a!=f),
    file('CreatedEqual.p', distinct_grades)).
fof(c_0_4, negated_conjecture,
    (~(?[X3]:(human(X3)&X3!=john))),
    inference(assume_negation,[status(cth)],[c_0_0])).
fof(c_0_5, negated_conjecture,
    (![X4]:(~human(X4)|X4=john)),
    inference(variable_rename,[status(thm)],
    [inference(fof_nnf,[status(thm)],[c_0_4])])).
fof(c_0_6, plain,
    ((human(esk1_0)&grade(esk1_0)=a)),
    inference(skolemize,[status(esa)],
    [inference(variable_rename,[status(thm)],[c_0_1])])).
cnf(c_0_7,negated_conjecture,
    (X1=john|~human(X1)),
    inference(split_conjunct,[status(thm)],[c_0_5])).
cnf(c_0_8,plain,
    (human(esk1_0)),
    inference(split_conjunct,[status(thm)],[c_0_6])).
cnf(c_0_9,plain,
    (grade(john)=f),
    inference(split_conjunct,[status(thm)],[c_0_2])).
cnf(c_0_10,negated_conjecture,
    (john=esk1_0),
    inference(spm,[status(thm)],[c_0_7, c_0_8])).
cnf(c_0_11,plain,
    (grade(esk1_0)=f),
    inference(rw,[status(thm)],[c_0_9, c_0_10])).
cnf(c_0_12,plain,
    (grade(esk1_0)=a),
    inference(split_conjunct,[status(thm)],[c_0_6])).
cnf(c_0_13,plain,(a!=f),
    inference(split_conjunct,[status(thm)],[c_0_3])).
cnf(c_0_14,plain,($false),
    inference(sr,[status(thm)],
    [inference(spm,[status(thm)],[c_0_11, c_0_12]), c_0_13]),
    ['proof']).
%------------------------------------------------------------------------
```

5 Proof Production

As described above, current mainstream theorem provers combine a clausifier that converts a formula in full first-order logic into clause normal form, with a saturating refutation core that tries to derive the empty clause from the clause set. Proof objects are derivation graphs, showing at least how the empty clause was derived from the initial clause set, and should also show how the initial clauses were generated from the first-order axioms.

While this is fairly straightforward without simplification, it becomes much harder if simplification is present. For reasons of both time and space efficiency, most provers use destructive simplification (i.e., the old clause is modified in memory or discarded and replaced by the modified copy). In particular, some clauses that have been used in the proof may not be present in the final clause set. There are several approaches to dealing with this problems.

Older versions of E wrote all intermediate steps into a protocol file and extracted the needed inferences in a post-processing step. This results in about 100% to 200% overhead in proof time, and fails for proof searches beyond a few minutes because the amount of data becomes unmanagable even for modern computers. Recent versions of E ensure that versions of clauses that have participated in generating inferences or as side premises in simplifications (a proportionally very small number of clauses) are archived in memory. This resulted in barely measurable overhead [55]. SPASS retains the full history and all versions of each clause in memory, and pays an overhead of about 100% [4]. Prover9 has a concept of "kept clauses", i.e. clauses the system has decided it will use or consider for future inferences and simplifications. If a "kept" clause would be affected by simplification, it is not completely deleted, but deactivated and, if necessary, replaced in the proof state by a simplified copy.

6 Proof Applications

The first and original use of proof objects is the analysis of proofs by human users. This helps to understand the proof and the application domain, to verify the correctness of the proof, but also to understand the behaviour of the ATP search process.

First-order ATP systems have directly been used for significant mathematical work, most famously for McCune's (and EQP's) proof of the Robbins problem [33]. In such case, both manual and automatic *proof checking* increases the trust placed into the proof and hence the validity if the theorem. Proof checking, analysis, and visualization is supported by tools as described in the next section.

First order ATP systems are increasingly integrated into higher-order interactive proof assistants. A prominent example is the Sledgehammer tool [43, 7] in Isabelle [40]. Via the interactive system, ATP systems contribute to large scale projects like the formal proof of Kepler's conjecture in Flyspeck [18] or the verification of the L4 micro-kernel [22, 21].

[4] Christoph Weidenbach, personal communication

In these applications, first-order proofs are used to guide the reconstruction of a proof in the native calculus of the embedding ITP system, which uses only the small set of trusted inferences of the very kernel of the system.

Another application is the validation and debugging not only of proofs, but also of specifications. Often large, manually assembled ontologies such as SUMO [39] or CYC [27, 45] contain unintended contradictions. Since most first-order systems are based on refutational calculi, they can be employed to find such contradictions, either directly on the full corpus, or a-posteriori, by checking if proofs use the negated conjecture to find the contradiction. Because of the powerful goal-directed heuristics, the second approach often is more successful.

Finally, proofs reveal a large amount of information about the domain and reasoning strategies. As such, they have been mined for useful information to help further proof attempts. One approach is the learning of heuristic evaluation functions, e.g. via annotating *patterns* [12, 15, 52] or other abstractions [54] of clauses.

Heuristic evaluation functions guide the selection of the given clauses in the refutation procedure. However, for domains with large background theories, even the original specification may overwhelm the theorem prover. One recent approach is to use machine learning techniques for premise selection, i.e. to select a subset of likely useful axioms and assumptions form a large corpus [1] [26].

7 Proof Consumption

Proof presentation is supported by two different kinds of tools. First, one can try to structure and present the proof as a sequential text, analogous to a classical mathematical textbook proof. This has been particularly successful in the case of purely equational proofs, where the reasoning can be represented as an equational chain. [11, 13].

The other approach is to visualize the proof as a DAG. IDV [61] is a tool for graphical rendering and analysis of TPTP format derivations. IDV provides an interactive interface that allows the user to quickly view features of the derivation, and access analysis facilities. The left hand side of Figure 2 shows the rendering of the derivation output by E for the TPTP problem `PUZ001+1`. The IDV window is divided into three panes: the top pane contains control buttons and sliders, the middle pane shows the rendered DAG, and the bottom pane gives the text of the annotated formula for the node pointed to by the mouse. The rendering of the derivation DAG uses shapes, colors, and tags to provide information about the derivation. The user can interact with the rendering in various ways using mouse-over and mouse clicks. The buttons and sliders in the control pane provide a range of manipulations on the rendering – zooming, hiding and displaying parts of the DAG, and access to GDV (see below) for verification. A particularly novel feature of IDV is its ability to provide a synopsis of a derivation by using the AGInTRater [44] to identify interesting lemmas, and hiding less interesting intermediate formulae. A synopsis is shown on the right hand side of Figure 2.

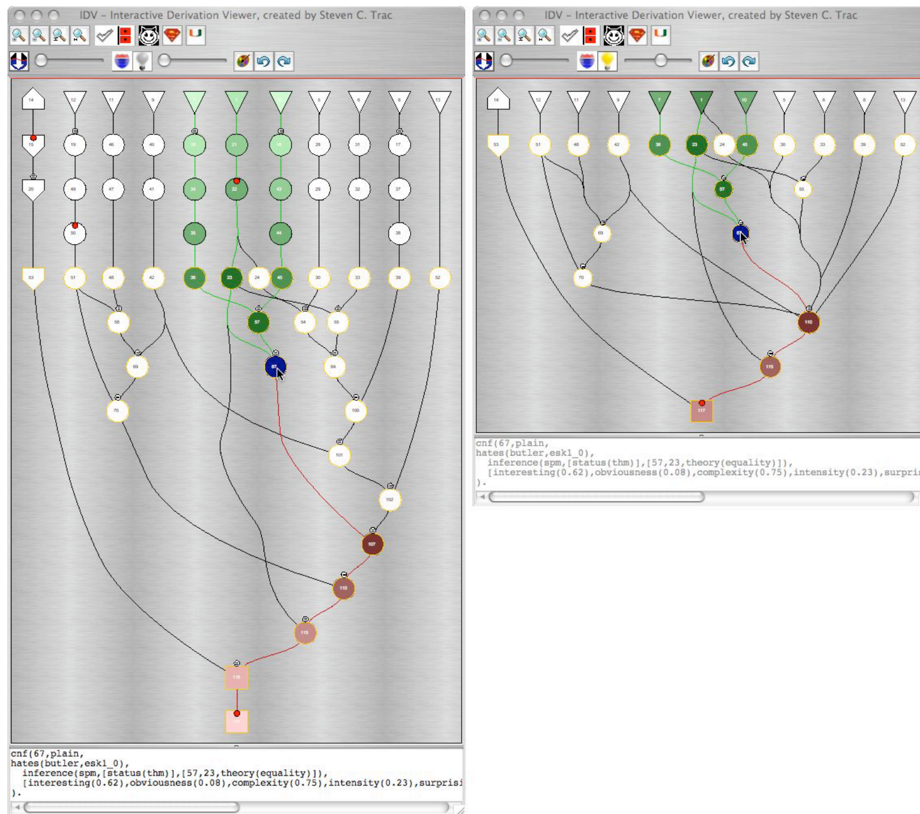

Fig. 2. E's proof by refutation of `PUZ001+1`

Proof verification can be done on a syntactic level, or on a semantic level. On a syntactic level, a trusted checker reproduces each individual inference. Most current ATP systems do no export enough information to make that directly feasible, however, as stated in the previous section, many ITP proofs first reproduce the proof in their native format, and then validate this via their own trusted kernel.

The alternative is semantic verification, i.e. showing that each derived clause and formula is in the stated semantic relationship with its parents. GDV [57] is a tool that uses structural and then semantic techniques to verify TPTP format derivations. Structural verification checks that inferences have been done correctly in the context of the derivation, e.g., checking that the derivation is acyclic, checking that assumptions have been discharged, and checking that introduced symbols (e.g., in Skolemization) are distinct. Semantic verification checks the expected semantic relationship between the parents and inferred formula of each inference step. This is done by encoding the expectation as a logical obligation

in an ATP problem, and then discharging the obligation by solving the problem with trusted ATP systems. The expected semantic relationship between the parents and inferred formula of an inference step depends on the intent of the inference rule used. For example, deduction steps expect the inferred formula to be a theorem of its parent formulae. The expected relationship is recorded as an SZS value in each inferred formula of a derivation.

8 Conclusions

The generation of explicit proof objects has not originally been a primary focus for first-order ATP systems. However, by now it is an expected feature for systems that are widely used. Earlier ad-hoc formats are now strongly converging to the TPTP-3/TSTP syntax, driven in part by the CASC competition, and in part by the increasing availability of tools that can process this information.

Proofs are used for several different applications:

- Human consumption
- Proof checking
- Embedding into interactive proofs
- Heuristics learning

The TSTP-3/TPTP format is sufficiently detailed to support these applications. However, because of its high level of abstraction, TPTP proof objects do not always allow direct step-by-step reconstruction of the proof. The future will show if this feature is important enough to emerge despite the greater effort for both producers and consumers of proofs.

References

1. J. Alama, T. Heskes, D. Kuühlwein, E. Tsivtsivadze, and J. Urban. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning*, 52(2):191–213, 2014.
2. L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion Without Failure. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press, 1989.
3. L. Bachmair and H. Ganzinger. On Restrictions of Ordered Paramodulation with Simplification. In M.E. Stickel, editor, *Proc. of the 10th CADE, Kaiserslautern*, volume 449 of *LNAI*, pages 427—441. Springer, 1990.
4. L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.
5. Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the Model Evolution Calculus. *International Journal of Artificial Intelligence Tools*, 15(1):21–52, 2006.
6. Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. In Franz Baader, editor, *Proc. of the 19th CADE, Miami*, volume 2741 of *LNCS*, pages 350–364. Springer, 2003.

7. Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement Day. In Jürgen Giesel and Reiner Hähnle, editors, *Proc. of the 5th IJCAR, Edinburgh*, volume 6173 of *LNAI*, pages 107–121. Springer, 2012.
8. O. Caprotti and D. Carlisle. OpenMath and MathML: Semantic Mark Up for Mathematics. *ACM Crossroads*, 6(2), 1999.
9. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(1):215–215, 1960.
10. J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997. Special Issue on the CADE 13 ATP System Competition.
11. J. Denzinger and S. Schulz. Analysis and Representation of Equational Proofs Generated by a Distributed Completion Based Proof System. Seki-Report SR-94-05, Universität Kaiserslautern, 1994.
12. J. Denzinger and S. Schulz. Learning Domain Knowledge to Improve Theorem Proving. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, volume 1104 of *LNAI*, pages 62–76. Springer, 1996.
13. J. Denzinger and S. Schulz. Recording and Analysing Knowledge-Based Distributed Deduction Processes. *Journal of Symbolic Computation*, 21(4/5):523–541, 1996.
14. J. Denzinger and S. Schulz. Recording and Analysing Knowledge-Based Distributed Deduction Processes. *Journal of Symbolic Computation*, 21:523–541, 1996.
15. J. Denzinger and S. Schulz. Automatic Acquisition of Search Control Knowledge from Multiple Proof Attempts. *Journal of Information and Computation*, 162:59–79, 2000.
16. M. Gordon and T. Melham. *Introduction to HOL, a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
17. R. Hähnle, M. Kerber, and C. Weidenbach. Common Syntax of the DFG-Schwerpunktprogramm Deduction. Technical Report TR 10/96, Fakultät für Informatik, Universät Karlsruhe, Karlsruhe, Germany, 1996.
18. T.C. Hales. Introduction to the Flyspeck project. In T. Coquand, H. Lombardi, and M.-F. Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings, pages 1–11. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 2006.
19. J. Hsiang and M. Rusinowitch. On Word Problems in Equational Theories. In *Proc. of the 14th ICALP, Karlsruhe*, volume 267 of *LNCS*, pages 54–71. Springer, 1987.
20. J. Hurd and R. Arthan. OpenTheory. `http://www.cl.cam.ac.uk/~jeh1004/research/opentheory`.
21. Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, 2010.
22. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles, Big Sky*, pages 207–220. ACM, 2009.
23. D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Algebra*, pages 263–297. Pergamon Press, 1970.

24. M. Kohlhase. OMDOC: Towards an Internet Standard for the Administration, Distribution, and Teaching of Mathematical Knowledge. In J.A. Campbell and E. Roanes-Lozano, editors, *Proceedings of the Artificial Intelligence and Symbolic Computation Conference, 2000*, number 1930 in Lecture Notes in Computer Science, pages 32–52. Springer-Verlag, 2000.
25. Konstantin Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proc. of the 4th IJCAR, Sydney*, volume 5195 of *LNAI*, pages 292–298. Springer, 2008.
26. D. Kuühlwein, J.C. Blanchette, C. Kaliszyk, and J. Urban. MaSh: Machine learning for Sledgehammer. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Proc. of the 4th International Conference on Interactive Theorem Proving (ITP13)*, volume 7998 of *LNCS*, pages 35–50. Springer, 2013.
27. D.B. Lenat. CYC: A Large-Scale Investment in Knowledge Infrastructure. *Communications of the ACM*, 38(11):35–38, 1995.
28. R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 1(8):183–212, 1992.
29. Reinhold Letz and Gernot Stenz. Model Elimination and Connection Tableau Procedures. In A. Robinson and A. Voronkov, editors, *Handbook of automated reasoning*, volume II, chapter 28, pages 2015–2112. Elsevier Science and MIT Press, 2001.
30. D.W. Loveland. Mechanical Theorem Proving by Model Elimination. *Journal of the ACM*, 15(2), 1968.
31. D.W. Loveland. *Automated Theorem Proving: A Logical Basis*. North Holland, Amsterdam, 1978.
32. W. McCune and O. Shumsky-Matlin. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In M. Kaufmann, P. Manolios, and J. Strother Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, number 4 in Advances in Formal Methods, pages 265–282. Kluwer Academic Publishers, 2000.
33. William McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.
34. William W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2008. (acccessed 2009-10-04).
35. W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.
36. W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.
37. M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – The CADE-13 Systems. *Journal of Automated Reasoning*, 18(2):237–246, 1997. Special Issue on the CADE 13 ATP System Competition.
38. R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science and MIT Press, 2001.
39. Ian Niles and Adam Pease. Toward a Standard Upper Ontology. In Chris Welty and Barry Smith, editors, *Proc. 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, 2001.
40. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

41. A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 5, pages 335–367. Elsevier Science and MIT Press, 2001.
42. Jens Otten and Wolfgang Bibel. leanCoP: Lean Connection-Based Theorem Proving,. *Journal of Symbolic Computation*, 36:139–161, 2003.
43. Lawrence C. Paulsson and Jasmin C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliff, Eugenia Ternovska, and Stephan Schulz, editors, *Procóf the 8th International Workshop on the Implementation of Logics (IWIL-2010), Yogyakarta, Indonesia*, volume 2 of *EPiC*, 2012.
44. Y. Puzis, Y. Gao, and G. Sutcliffe. Automated Generation of Interesting Theorems. In G. Sutcliffe and R. Goebel, editors, *Proceedings of the 19th International FLAIRS Conference*, pages 49–54. AAAI Press, 2006.
45. Deepak Ramachandran, Pace Reagan, and Keith Goolsbey. First-orderized ResearchCyc: Expressiveness and Efficiency in a Common Sense Knowledge Base. In Pavel Shvaiko, editor, *Proc. of the AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications (C&O-2005)*, 2005.
46. Raymod Reiter. Two Results on Ordering for Resolution with Merging and Linear Format. *Journal of the ACM*, 18(4):630–646, 1971.
47. A. Riazanov and A. Voronkov. Vampire 1.1 (System Description). In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 376–380. Springer, 2001.
48. G. Robinson and L. Wos. Paramodulation and Theorem Proving in First-Order Theories with Equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.
49. J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.
50. J.A. Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1(3):227–234, 1965.
51. S. Schulz. LOP-Syntax for Theorem Proving Applications. http://www4.informatik.tu-muenchen.de/ schulz/WORK/lop.syntax.
52. S. Schulz. Learning Search Control Knowledge for Equational Theorem Proving. In F. Baader, G. Brewka, and T. Eiter, editors, *Proc. of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001)*, volume 2174 of *LNAI*, pages 320–334. Springer, 2001.
53. S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
54. S. Schulz and F. Brandt. Using Term Space Maps to Capture Search Control Knowledge in Equational Theorem Proving. In A. N. Kumar and I. Russell, editors, *Proc. of the 12th FLAIRS, Orlando*, pages 244–248. AAAI Press, 1999.
55. Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.
56. G. Steel. Visualising First-Order Proof Search. In C. Aspinall, D. Lüth, editor, *Proceedings of User Interfaces for Theorem Provers 2005*, pages 179–189, 2005.
57. G. Sutcliffe. Semantic Derivation Verification. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.
58. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

59. G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In Volker Sorge and Weixiong Zhang, editors, *Distributed Constraint Problem Solving And Reasoning In Multi-Agent Systems*, Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.
60. Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Allen Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations . In Ulrich Fuhrbach and Natarajan Shankar, editors, *Proc. of the 3rd IJCAR, Seattle*, volume 4130 of *LNAI*, pages 67–81, 4130, 2006. Springer.
61. S. Trac, Y. Puzis, and G. Sutcliffe. An Interactive Derivation Viewer. In S. Autexier and C. Benzmüller, editors, *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers, 3rd International Joint Conference on Automated Reasoning*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 109–123, 2006.
62. A. Voronkov. The Anatomy of Vampire: Implementing Bottom-Up Procedures with Code Trees. *Journal of Automated Reasoning*, 15(2):238–265, 1995.
63. C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER Version 0.42. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, volume 1104 of *LNAI*, pages 141–145. Springer, 1996.
64. Christoph Weidenbach, Renate Schmidt, Thomas Hillenbrand, Dalibor Topić, and Rostislav Rusev. SPASS Version 3.0. In Frank Pfenning, editor, *Proc. of the 21st CADE, Bremen*, volume 4603 of *LNAI*, pages 514–520. Springer, 2007.