System Description: E 1.8

Stephan Schulz

Institut für Informatik, Technische Universität München,
D-80290 München, Germany
schulz@eprover.org

Abstract. E is a theorem prover for full first-order logic with equality. It reduces first-order problems to clause normal form and employs a saturation algorithm based on the equational superposition calculus. E is built on shared terms with cached rewriting, and employs several innovations for efficient clause indexing. Major strengths of the system are automatic problem analysis and highly flexible search heuristics. The prover can provide verifiable proof objects and answer substitutions with very little overhead. E performs well, solving more than 69% of TPTP-5.4.0 FOF and CNF problems in automatic mode.

1 Introduction

E is a theorem prover for full first-order logic with equality, built around a fully equational implementation of the superposition calculus. For the last 12 years the prover has been one of the major participants at the CADE ATP System Competition in the MIX, CNF, FOF, UEQ and LTB categories, usually finishing among the top systems in all these categories. E is available as Free Software under the GNU GPL. It is implemented in C, widely portable, and has been used, in whole or part, as a component in many other systems.

Fig. 1 shows the high-level functional decomposition of the theorem prover, and the data flow between the components. A proof problem is read into main memory, and is passed through several different processing stages:

- The problem is parsed and converted into a set of clauses and formulas by a simple but efficient recursive descent parser. The parser supports E-LOP, and the TPTP CNF/FOF syntax [13].
- In the next stage, *Relevancy Pruning*, the problem is optionally simplified by discarding clauses and formulas deemed unlikely to contribute to a proof. E implements both strict relevancy pruning and a configurable variant of the SInE algorithm [4].
- The third stage, *Clausification*, converts the problem from full first-order logic to clausal form. Clausification uses a slightly simplified version of the algorithm presented by Nonnengart and Weidenbach [8]. The implementation takes advantage of E's shared term/shared formula representation
- The resulting clause set can be pre-processed. Preprocessing removes redundant literals and tautologies, and optionally expands equational definitions. If requested, preprocessing can also perform complete interreduction of the problem specification.

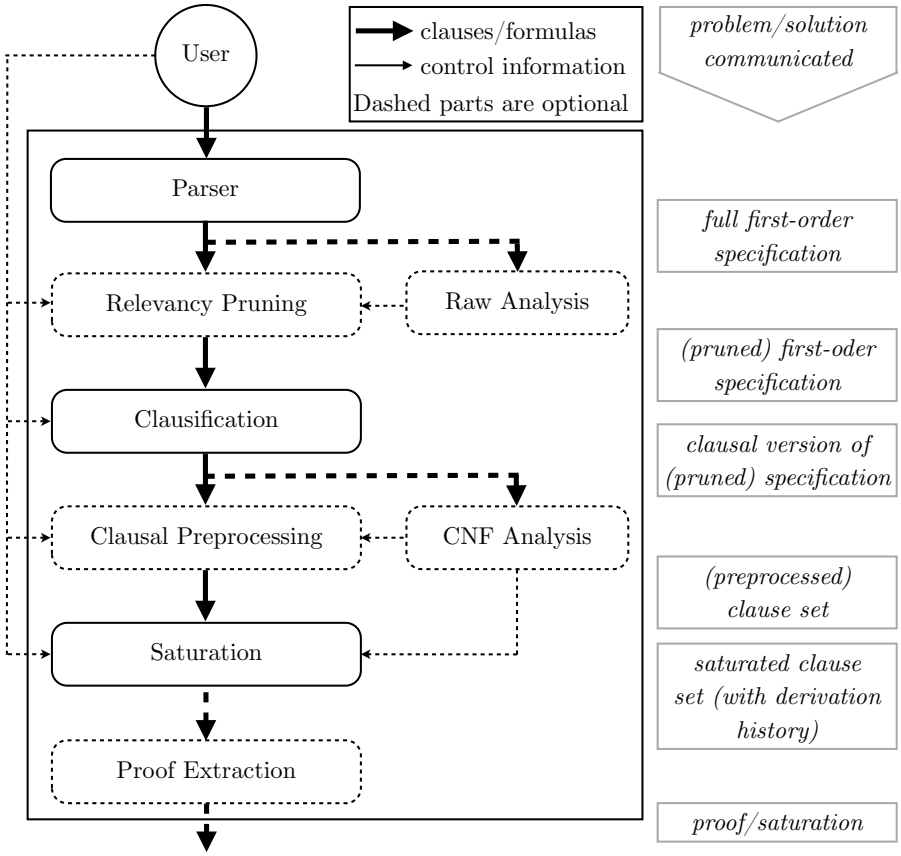

Fig. 1. Decomposition of E and major data flows

- After preprocessing, the clause set is passed to the main saturation algorithm. This is realized as an instance of the DISCOUNT variant of the given-clause algorithm and implements a variant of the superposition calculus with a number of contraction techniques. The saturation ends when the empty clause has been derived, the set is saturated, or a user-defined resource limit is reached.
- The prover can store enough information to generate a checkable proof object. In the final (optional) step, this information is collected into a proof tree (or saturation derivation), which can be printed in E's original PCL-2 or TPTP-3/TSTP syntax.

Various aspects of the process are controlled by parameters that are either provided by the user or heuristically determined by the *automatic mode* of the prover.

2 Saturation

The core of the prover is a saturation procedure that tries to show the inconsistency of a set of clauses (the *search state*). New clauses are deduced using generating inference rules, and existing clauses are simplified or discarded using contraction rules. The algorithm terminates either when it has derived the empty clause as an explicit witness of inconsistency, or if all non-redundant inferences have been computed. In this case, the resulting saturated set describes a model of the clause set.

2.1 Calculus

E implements an instance of the superposition calculus with negative literal selection, as originally described by Bachmair and Ganzinger [2]. It uses the rules *equality resolution* (ER), *equality factoring* (EF), and *superposition into positive and negative literals*, (SP) and (SN). Alternatively, the latter two rules can be replaced by *simultaneous superposition* (SSP and SSN), which often results in slightly better search behaviour and hence is the default. Simultaneous superposition is inspired by simultaneous paramodulation [3], and maintains completeness[1].

Contraction is critical for practical performance. E implements deletion of duplicate and resolved literals (DD, DR), syntactic and semantic tautology deletion (TD1, TD2, SD), *destructive equality resolution* (DR), unconditional rewriting (RP, RN), equational literal cutting (PS, NS), subsumption (CS, ES), *contextual literal cutting* (CLC), condensing (CD), AC-tautology deletion (ACD) and AC-simplification (ACS). The last two rules handle associative and commutative function symbols as suggested in [1].

2.2 Implementation

Fig. 2 sketches the proof procedure. The algorithm maintains the invariant that the set P of unprocessed clauses is interreduced, and that all generating inferences between clauses from P have been performed. Derivations are *fair* if no clause remains unprocessed forever.

The implementation is built around perfectly shared terms. Each distinct term is represented exactly once in a *term bank*. Unconditional rewriting is cached. Whenever a possible simplification is detected, it is recorded in the term bank. Future simplifications simply follow these *rewrite links* before trying new equations.

Indexing enables the prover to quickly find inference partners for a given premise. E indexes the set P of processed clauses. It uses *Perfect Discrimination Trees* [7] with size- and age-constraints for forward rewriting (finding positive

[1] On the ground level, a simultaneous superposition inference can be simulated by a single conventional superposition step, followed by a series of (simplifying) conditional rewrite steps.

Search state: $U \cup P$
U contains *unprocessed* clauses, P contains *processed* clauses.
Initially, all clauses are in U, P is empty.
The *given clause* is denoted by g.

```
while U ≠ {}
   g = delete_best(U)
   g = simplify(g, P)
   if g == □
      SUCCESS, Proof found
   if g is not subsumed by any clause in P (or otherwise redundant w.r.t. P)
      P = P\{c ∈ P | c subsumed by (or otherwise redundant w.r.t.) g}
      T = {c ∈ P | c can be simplified with g}
      P = (P\T) ∪ {g}
      T = T ∪ generate(g, P)
      foreach c ∈ T
         c = cheap_simplify(c, P)
         if c is not trivial
            U = U ∪ {c}
SUCCESS, original U is satisfiable
```

Remarks: delete_best(U) finds and extracts the clause with the best heuristic evaluation (see 3.3) from U. generate(g, P) performs all generating inferences using g as one premise, and clauses from P as additional premises. It uses inference rules (SP) or (SSP), (SN) or (SSN), (ER) and (EF).
simplify(c, S) applies all simplification inferences in which the main (simplified) premise is c and all the other premises are clauses from S. This typically includes full rewriting, (CD) and (CLC). cheap_simplify(c, S) works similarly, but only applies inference rules with a particularly low cost implementation, usually including rewriting with orientable units, but not (CLC). The exact set of contraction rules used is configurable in either case.

Fig. 2. Saturation procedure of E

unit clauses that can rewrite new clauses), *Fingerprint Indexing* [10] for backward rewriting (finding clauses in P that can be rewritten with the given clause) and superposition, and *Feature Vector Indexing* [11] for subsumption and contextual literal cutting.

Term orderings (LPO and KBO) are implemented using the elegant and efficient reformulations presented by Löchner [5, 6].

3 Search Control

Proof search depends on a number of different parameters. The three major choice points are the selection of a term ordering, the (optional) selection of inference literals, and the order in which clauses from U are picked for processing.

3.1 Term Orderings

E supports KBO and LPO. Both orderings are parameterized. KBO uses a weight function assigning weights to individual function symbols (and a fixed weight to all variables), and both orderings use a precedence on the function symbols. E currently supports about a dozen precedence generation schemes, and more than two dozen weight generation schemes. Orderings showing the best performance use the frequency of symbols in the specification, making terms with rarer symbols larger in the ordering.

3.2 Literal Selection

Literal selection is a major strength of E. Even quite naive approaches (always select the largest negative literal, if any) lead to a significant improvement over the plain superposition calculus. Good literal selection strategies seem to prefer ground literals, literals that are large in the term ordering, and to avoid literals that contain little structure, e.g. literals of the form $p(X, Y, Z)$.

3.3 Clause Evaluation

The given-clause algorithm selects clauses according to a heuristic evaluation. In the simplest case, this is a single value, representing the number of symbols in the clause (smaller is better). E generalizes this concept and allows the user to specify an arbitrary number of priority queues and a weighted round-robin scheme that determines how many clauses are picked from each queue. A major feature is the use of goal-directed evaluation functions. These give a lower weight to symbols that occur in the goal, and a higher weight to other symbols, thus preferring clauses which a likely connection to the conjecture. As an alternative, E can also *learn* good clause evaluations from previous proof experience [9].

3.4 Automatic Prover Configuration

Performance of first-order theorem provers critically depends on the search strategy and heuristics. Finding good heuristics for a given problem is challenging even for an experienced user. E supports a number of *automatic modes* that analyze the problem and apply either a single strategy or a schedule of several strategies. The selection of strategies and generation of schedules for each class of problems is determined automatically by analyzing previous performance of the prover on similar problems.

4 Proofs and Answers

4.1 Proofs

E 1.8 can internally record all necessary information for proof output. It makes use of the DISCOUNT loop property that only processed clauses (usually a

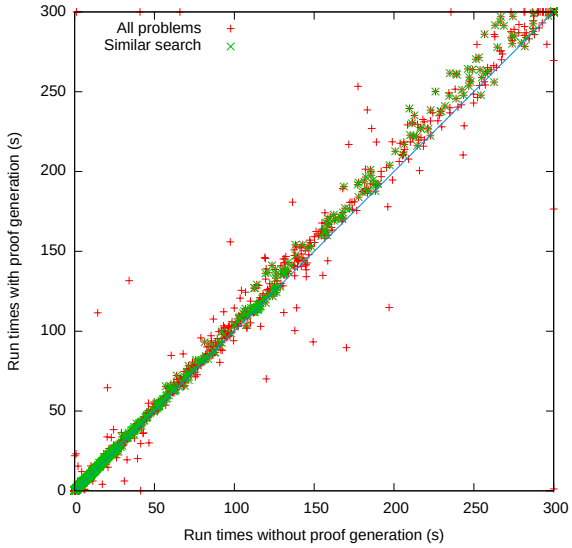

Fig. 3. Comparison of run times

small subset of all clauses in the search state) can ever participate in generating inferences or be used to simplify other clauses. For each clause, the system stores its origin (usually a generating inference and the parents), and a history of simplifications (inference rules and side premises). A processed clause is archived and replaced by a simplified copy (pointing to the original as the parent) only if it itself is back-simplified.

When the empty clause has been derived and hence a proof concluded, the proof tree is extracted by tracing the recorded dependencies. Proof steps are topologically sorted, ensuring that all dependencies of a step are listed before the step itself. The linearized proof can then be printed.

Recording of the derivation history does not systematically change the search behaviour. However, changes in memory usage and layout can cause some operations (e.g. iteration over a set) to be performed differently, potentially disturbing the proof search. Fig. 3 shows the run times of the prover in automatic mode with and without proof generation over TPTP 5.4.0, for both the majority of problems where both versions performed the same search and the small number with differing search behaviour. Performing a simple linear regression over the problems with the same search suggests an overhead of only 0.24% for proof generation.

4.2 Answers

The system supports the proposed TPTP standard for answers [14]. An *answer* is an instantiation for an existential conjecture (or *query*) that makes the conjecture

Strategy	**UEQ**	**CNE**	**CEQ**	**FNE**	**FEQ**	**All**
Class size	(1179)	(2352)	(5867)	(1713)	(5867)	(15560)
Best	764	1642	3211	1251	3211	9305
... with proof object	764	1648	3210	1251	3210	9301
SatAuto	800	1833	3671	1418	3671	10334
... with proof object	799	1833	3664	1421	3664	10326
Auto	801	1834	3758	1424	3758	10432
... with proof object	799	1834	3749	1424	3749	10415
Auto-Scheduling	824	1867	3939	1430	3939	10783
... with proof object	823	1864	3936	1430	3936	10776

UEQ: Unit equational problems, CNE: (non-unit) CNF problems without equality, CEQ: CNF problems with equality, FNE: Full first-order problems without equality, FEQ: Full first-order problems with equality

Table 1. Number of proofs/models found within 300 seconds CPU limit

true. In practice, E supplies bindings for the outermost existentially quantified variables in a TPTP formula with type `question`.

The implementation is straightforward. The query is extended by adding the atomic formula `~$answer(new_fun(<varlist>))`, where `new_fun` is a previously unused function symbol, and `<varlist>` is the list of outermost existentially quantified variables. This atom is carried through clausification and ends up as a positive literal in the CNF. The literal ordering is automatically chosen so that the answer literal never participates in inferences. Semantically, the `$answer` predicate always evaluates to false. It is evaluated only in clauses where all remaining literals are answer literals. Answers are extracted and printed in tuple form at the time of the evaluation. Consider the following example:

Specification

```
fof(greeks, axiom, (philosopher(socrates)|philosopher(plato))).
fof(scot, axiom, (philosopher(hume))).
fof(phils_wise, axiom, (![X]:(philosopher(X) => wise(X)))).
fof(is_there_wisdom, question, (?[X]:wise(X))).
```

Answers (`eprover --tptp3-format -s --answers`)

```
# SZS status Theorem
# SZS answers Tuple [[hume]|_]
# SZS answers Tuple [([socrates]|[plato])|_]
# Proof found!
```

The system correctly handles disjunctive answers (at least one of `socrates` or `plato` is a philosopher and hence `wise`, but the theory does not allow us to decide who is). While the example has been kept intentionally simple, the system also supports complex terms and variables as parts of answers, in that case representing the set of all instances.

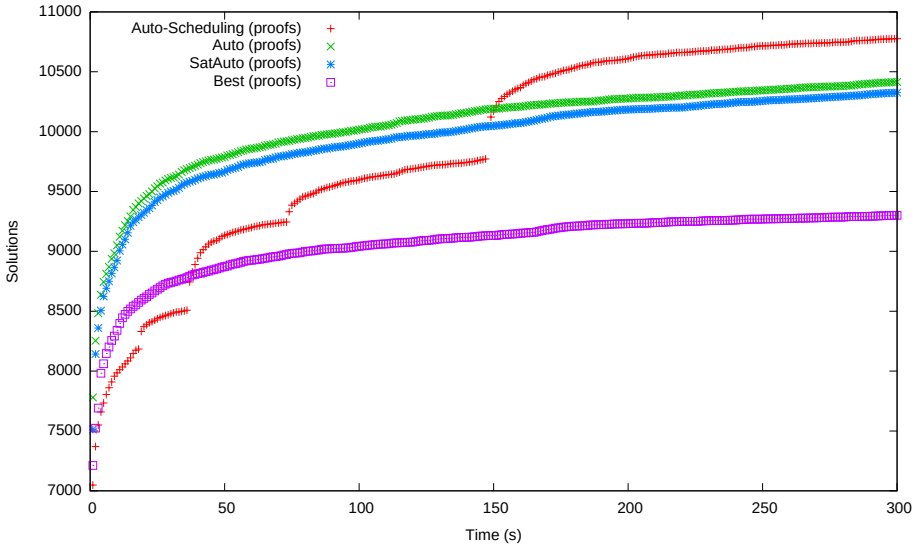

Fig. 4. Performance (number of solutions over time) of E with different strategies and meta-strategies

5 Performance

Table 1 lists the performance of E for 4 different search regimens and different classes of problems. Tests were run on the University of Miami *Pegasus cluster*. Each node of the cluster is equipped with 8 Intel Xeon cores, running at 2.5 GHz, and 16 GB of RAM. Test runs were done with a CPU time limit of 300 seconds per job, a memory limit of 1024 MB per job, and with 8 jobs scheduled per node. All 15560 untyped first-order problems (including CNF, FOF and UEQ) from TPTP 5.4.0 were used as test examples.

Best is the currently strongest single strategy known. *SatAuto* analyses the input problem and picks an appropriate strategy based on the performance on similar problems. *Auto* additionally performs problem pruning, potentially losing completeness, but improving behaviour on very large problems. Finally, *Auto-Scheduling* runs up to 5 complementary strategies for each problem class.

Search performance over time is visualized in Fig. 4 for all 15560 problems. In all cases, the first 7000 solutions are found within less than 1 second. Of the 10783 solutions found by AutoScheduling, 1000 are saturations, 9783 are proofs.

6 Conclusion

E has reached good maturity for untyped first-order logic. It is stable, reliable, and has improved usability with strong automatic search, proof object generation and answer substitutions.

Future planned changes include support for simply typed first-order logic with arithmetic as defined in [12], improved support for repetitive queries against large axiom sets, and the use of new data-driven methods for search control.

Acknowledgements: I thank the University of Miami's *Center for Computational Science* HPC team for making their cluster available for the evaluation.

References

1. Avenhaus, J., Hillenbrand, T., Löchner, B.: On Using Ground Joinable Equations in Equational Theorem Proving. Journal of Symbolic Computation 36(1-2), 217–233 (2003)
2. Bachmair, L., Ganzinger, H.: Rewrite-Based Equational Theorem Proving with Selection and Simplification. Journal of Logic and Computation 3(4), 217–247 (1994)
3. Benanav, D.: Simultaneous paramodulation. In: Proc. of the 10th CADE, Kaiserslautern. LNCS, vol. 449, pp. 442–455. Springer (1990)
4. Hoder, K., Voronkov, A.: Sine Qua Non for Large Theory Reasoning. In: Bjørner, N., Stokkermans, S.V. (eds.) Proc. of the 23rd CADE, Wroclav. LNAI, vol. 6803, pp. 299–314. Springer (2011)
5. Löchner, B.: Things to Know when Implementing KBO. Journal of Automated Reasoning 36(4), 289–310 (2006)
6. Löchner, B.: Things to Know When Implementing LPO. International Journal on Artificial Intelligence Tools 15(1), 53–80 (2006)
7. McCune, W.: Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. Journal of Automated Reasoning 9(2), 147–167 (1992)
8. Nonnengart, A., Weidenbach, C.: Computing Small Clause Normal Forms. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, chap. 5, pp. 335–367. Elsevier Science and MIT Press (2001)
9. Schulz, S.: Learning Search Control Knowledge for Equational Theorem Proving. In: Baader, F., Brewka, G., Eiter, T. (eds.) Proc. of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001). LNAI, vol. 2174, pp. 320–334. Springer (2001)
10. Schulz, S.: Fingerprint Indexing for Paramodulation and Rewriting. In: Gramlich, B., Sattler, U., Miller, D. (eds.) Proc. of the 6th IJCAR, Manchester. LNAI, vol. 7364, pp. 477–483. Springer (2012)
11. Schulz, S.: Simple and Efficient Clause Subsumption with Feature Vector Indexing. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics: Essays in Memory of William W. McCune, LNAI, vol. 7788, pp. 45–67. Springer (2013)
12. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP Typed First-order Form with Arithmetic. In: Bjørner, N., Voronkov, A. (eds.) Proc. of the 18th LPAR, Merida. LNAI, vol. 7180, pp. 406–419. Springer (2012)
13. Sutcliffe, G., Schulz, S., Claessen, K., Gelder, A.V.: Using the TPTP Language for Writing Derivations and Finite Interpretations . In: Fuhrbach, U., Shankar, N. (eds.) Proc. of the 3rd IJCAR, Seattle. LNAI, vol. 4130, pp. 67–81. Springer, 4130 (2006)
14. Sutcliffe, G., Stickel, M., Schulz, S., Urban, J.: Answer Extraction for TPTP. `http://www.cs.miami.edu/~tptp/TPTP/Proposals/AnswerExtraction.html`, (acccessed 2013-07-08)