Faster, Higher, Stronger: E 2.3

Stephan Schulz[1], Simon Cruanes[2], and Petar Vukmirović[3]

[1] DHBW Stuttgart, Germany, schulz@eprover.org
[2] Aesthetic Integration, Austin, Texas, USA, simon.cruanes.2007@m4x.org
[3] Vrije Universiteit Amsterdam, The Netherlands, p.vukmirovic@vu.nl

Abstract. E 2.3 is a theorem prover for many-sorted first-order logic with equality. We describe the basic logical and software architecture of the system, as well as core features of the implementation. We particularly discuss recently added features and extensions, including the extension to many-sorted logic, optional limited support for higher-order logic, and the integration of SAT techniques via PicoSAT. Minor additions include improved support for TPTP standard features, *always-on* internal proof objects, and lazy orphan removal. The paper also gives an overview of the performance of the system, and describes ongoing and future work.

1 Introduction

E is a fully automated theorem prover for first-order logic with equality. It has been under development for about 20 years, adding support for full first-order logic with E 0.82 in 2004, many-sorted first-order logic with E 2.0 in 2017, and both optional support for λ-free higher-order logic (LFHOL) and improved handling of propositional logic with the current release, E 2.3.

The basic architecture of the clausal inference core has previously been described in [15] (covering E 0.62), and the last updated description of E 1.8 was published in 2013 [18]. The recent support for λ-free higher-order logic is covered in detail in [30,31]. In this paper, we describe the current state of the prover, with a particular focus on recent developments.

E is available as free software under the GNU General Public License. Official point releases are available as source distributions from `https://www.eprover.org`. Development versions and the full history of changes can be found at `https://github.com/eprover`.

2 System Design and Architecture

The system is designed around a pipeline of largely distinct processing steps (compare Fig. 1): *Parsing*, *preliminary analysis*, *axiom selection*, *clausification*, *clausal preprocessing*, *auto-mode CNF analysis*, *saturation* and *proof object extraction*. Parsing, clausification and saturation are necessary for actual theorem proving, the other steps are optional and enabled by command line options.

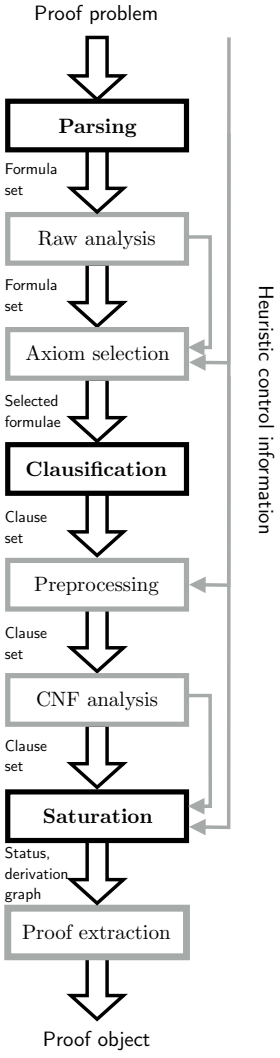

Fig. 1. Logical pipeline

In the first step, the input problem is parsed as a set of annotated formulas, where each logical formula is represented as a shared term over a signature including the usual logical operators, and wrapped in a data structure that allows annotations to capture additional extra-logical properties such as the formula role (in particular *axiom* and *conjecture*), the name of the formula, and its provenience.

The next step is an optional analysis of the parsed problem, primarily to automatically determine if and how an axiom selection scheme should be applied in the third step to reduce the number of axioms. Axiom selection is based on a variant of the SInE algorithm [8]. This step is optional. Axiom selection can be manually triggered or blocked by the user or triggered automatically based on the results of the preceding analysis step.

At the core of the prover is a refutational proof procedure for first-order clausal logic with equality. It is based on the superposition calculus (with some extensions and modifications), and works on a clausal representation of the problem. The *clausification* step converts the full first-order problem into a set of clauses. It is based on the ideas presented by Nonnengart and Weidenbach [13]. As usual with refutational theorem provers, if an explicit conjecture is given, it is negated before clausification, so that the resulting clause set is unsatisfiable if the conjecture logically follows from the axioms.

The prover optionally performs preprocessing of the clause set. Preprocessing removes redundant literals and tautologies, optionally unfolds some or all equational definitions, and orders literals and clauses in a canonical ordering so that the prover behaves in a more deterministic way.

The resulting clause set can be extracted after this stage. Indeed, several other clausal provers use E as an external clausifier. If E continues, the clause set is then analyzed to determine the search control strategy to be used by the inference core. The superposition calculus is parameterized by a term ordering and (optionally) a literal selection function. The implementation uses a variant of the *given-clause algorithm* (Fig. 2). The main additional search parameter for this algorithm is the scheme for the selection of the *given clause* for each iteration of the main loop. However, there are a large

number of additional flags controlling e.g. different options for simplification. All aspects of the search strategy can again be explicitly set by the user, or automatically determined by the automatic mode of the prover.

The inference core performs a classical saturation of the clause set, optionally interspersed with calls to the CDCL-based SAT-solver *PicoSAT* [3] to detect conflicts hidden in (so far) unprocessed clauses. The procedure terminates successfully when either the empty clause has been derived directly, when the SAT-solver detects unsatisfiability of the proof state, or when the clause set is saturated. It terminates unsuccessfully, if it cannot reach success within user-defined limits (e.g. CPU time, memory, iterations, elementary term operations).

In the case of success, an optional final step can extract a proof object from the search state and present the proof to the user.

3 Calculus and Implementation

3.1 Superposition for many-sorted logic

E was originally built around untyped first-order logic, distinguishing only predicate symbols (returning a Boolean value) and function symbols (returning an individual, represented as a term). Variables would implicitly range over all terms, and hence could be bound to any term. As of version 2.0, the prover has been extended to support many-sorted first order logic in the style described by Sutcliffe, Schulz, Claessen and Baumgartner [27].

In this logic, every plain function symbol has an associated function type, accepting a fixed number of arguments of defined sorts, and constructing a term of a defined return sort. Predicate symbols also accept terms of the correct sorts only. An exception is the equality-predicate, which is ad-hoc polymorphic, but requires terms of the same sort for both arguments.

Supporting many-sorted logic unlocks access to many useful features:

- expressing some size constraints over models, using axioms such as $\exists a, b : \tau, \forall x : \tau, x \simeq a \vee x \simeq b$;
- enabling more efficient encodings from systems with richer logics, such as proof assistants [4] or program verification tools [5]. While types can be encoded within plain first-order logic, these encodings tend to bloat formulas, adding sort predicates to every axiom, and to bloat terms, which reduces the effectiveness of many simplification rules;
- supporting some built-in theories, as SMT solvers typically do; indeed, the SMT-LIB language [2] is typed;
- supporting the FOOL extension [10] of first-order logic and its realization in the TFX format [26], allowing Boolean sub-terms as well as `let` and `if-then-else` constructs.

Basic support for interpreted numbers is planned for the near future, full support for TFX is already in progress and will most likely be included in the next release.

The superposition calculus readily generalizes to this logic. E implements the standard inference rules from [1]: Superposition, equality resolution, and equality

factoring. In addition, it implements a large array of simplification rules, the most important of which are unconditional rewriting, subsumption, equational literal cutting and contextual literal cutting. A more detailed description of the calculus (and its realization in the proof procedure) is provided in the manual [19].

In practice, supporting simple types requires every variable and term to be annotated with its sort. Unification and retrieval of terms from indices (e.g. for demodulation) check that types are compatible before binding a variable to any given term. This is what prevents an axiom such as $\forall x : \text{side}, x \simeq \text{left} \vee x \simeq \text{right}$ to rewrite another clause's subterm of an incompatible type. This change had negligible impact on performance, but significantly improves the expressiveness of the logic.

E can currently parse the TFF0 sub-grammar of the TPTP TFF format [27], and prints typed terms and formulas using the same syntax.

Sorts were originally represented as indices into a sort table. The LFHOL extension of E 2.3 [31] further generalizes this representation to support higher-order simple types and partially applied terms. The implementation of types now uses a lightweight term-like structure, in which complex types are build from basic sorts and the arrow operator. Like terms, types are perfectly shared for efficient type equality comparisons.

3.2 Implementation

The system is being developed in C, providing good performance and maximal portability. The code of the prover proper largely restricts itself to features from C99, with some POSIX extensions. It has been successfully built on a large range of different UNIX-style operating systems, in particular OS-X/macOS (with both LLVM and GCC as compilers) and Linux, the two main development and testing platforms. It has also been compiled and run under versions of Windows, using the CygWin libraries for POSIX/UNIX compatibility.

In the past, supporting software for testing and optimizing the system has been built in a number of scripting languages, but more recently has been largely moved to Python.

While C is an excellent language for performance and portability, it offers a relatively small number of built-in data structures and programming constructs. As a consequence, E has been built on a layer of libraries providing generic data types such as unlimited size stacks, splay trees, dynamic arrays, as well as convenient abstractions for a number of operating system services.

On top of these generic libraries, the prover implements logical data types and operations. At the heart of the system is the term bank data type, an efficient and garbage-collected data structure originally for aggressively shared first-order terms. All persistent terms are inserted in a bottom-up manner into this term bank. Thus, identical terms are represented by identical pointers. This results in a saving in the number of cells needed to represent the proof state of several orders of magnitude [11]. It also enables us to precompute a number of properties and store them in the term cells. Examples include the number of function symbols in the term and the number of variable occurrences. Thus, we

can e.g. decide if a shared term is ground in constant time. More importantly, we can cache the result of rewrite attempts at the term level — in the case of success with a link to the result (and, for proof reconstruction, with the clause used as a side premise), in the case of failure with the age of the youngest clause tried, so that future attempts can be restricted to newer clauses.

The term bank data structure and its API has proven to be efficient and convenient. In particular, the mark-and-sweep garbage collector makes the creation and destruction of terms very convenient and reduces programmer effort and errors. As a result, shared terms are now also used to represent formulas and in some roles even clauses (which, as of E 2.2, are parsed as a special case of formulas).

Literals and clauses for the inference core are implemented as dedicated data structures. Internally, all literals are equational. In addition to the two terms making up the equation, literals include polarity (positive or negative), a number of Boolean flags, and a pointer for creating linked lists. Clauses consist of such a linked list of literals, wrapped in a container for meta-data, heuristic evaluations, and information about the derivation of the clause.

Proof procedure Fig. 2 depicts the main saturation procedure. It is a modified version of the DISCOUNT loop [6], one of the variants of the *given-clause algorithm*. The proof state is represented by two disjoint subsets of clauses, the *processed* clauses P and the *unprocessed* clauses U. Initially, all clauses are unprocessed. At each iteration of the main loop, the prover heuristically selects a *given clause* from U, adds it to P, and performs all generating inferences between this clause and all clauses in P. The resulting new clauses are added to U. This maintains the invariant that all direct consequences between clauses in P have been performed. Forward simplification is performed on the given clause (using clauses in P as side premises) before it is used for generation, and on new clauses before they are added to U. In addition, clauses in P are back-simplified with the given clause, and simplified clauses are moved to U. This maintains the additional invariant that the clauses in P are interreduced, or maximally simplified with respect to other clauses in P.

In addition to saturation, the current version may trigger a propositional check for unsatisfiability of a grounded version of the proof state, as described below.

Internal proof objects Originally, E only offered the option to log all inferences to an external medium and then generate a proof object in a post-mortem analysis. Since these logs often reached extreme sizes, this was costly and not even practically possible for long runs.

With E 1.8, we finally found a way to use the invariants of the given-clause algorithm to very compactly represent the derivation graph internally [22]. Since the overhead in time and memory turned out to be negligible, we have simplified the code and now always build an internal proof object. In addition to efficiently providing a checkable proof object in TPTP syntax [28], the presence of the

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

```
while U ≠ {}
    if prop_trigger(U,P)
        if prop_unsat_check(U,P)
            SUCCESS, Proof found
    g = extract_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)
        T' = {}
        foreach c ∈ T
            c = cheap_simplify(c, P)
            if c is not trivial
                T' = T' ∪ {c}
        U = U ∪ T'
SUCCESS, original U is satisfiable
```

Fig. 2. The modified *given-clause* algorithm as implemented in E

derivation information enables the detection of vacuous proofs (based on an inconsistent axiomatization). It also enables an elegant lazy implementation of *orphan killing*. An orphan is a generated clause which has lost at least one of its parents to interreduction before being selected for processing, and which hence can be deleted as well. Older versions of E maintained an explicit list of direct descendants for processed clauses, and actively removed such descendents if the parent clause became redundant. As of E 2.2, we instead check the status of the parent clauses (which are either active or archived) only when the clause is selected for processing. This removes the bookkeeping overhead and simplifies both code and data structures.

In addition to the generation of proof objects, the system supports the proposed TPTP standard for answers [29]. An *answer* is an instantiation for an existential conjecture (or *query*) that makes the conjecture true. E can supply bindings for the outermost existentially quantified variables in a TPTP formula with type `question`.

Indexing Most of the generating and simplifying inference rules require two premises - the main premise and a side premise. For generating inferences, one

of the inference partners is the given clause, the other one is a clause in P. E uses a *fingerprint index* [16] to efficiently find clauses with (sub-)terms that are unifiable with the maximal terms of inference literals of the given clause.

For simplification, the DISCOUNT loop distinguishes two situations. In *forward simplification*, all clauses in P are used as side premises to simplify a given clause - either *the* given clause, or a newly generated clause. E uses *perfect discrimination trees* [12] with size- and age-constraints for forward rewriting. Backward simplification uses a single clause to simplify all clauses from P. E uses fingerprint indexing for backwards rewriting. Subsumption and contextual literal cutting use *feature vector indexing* [17], a clause indexing technique that supports the finding of both generalizations and instances.

SAT integration SAT solvers have greatly improved performance in the last decades, and can handle propositional problems that are far beyond the practical scope of classical first-order provers with ease. Following other attempts [9,14], we want to utilize this power to improve the performance of the prover both for problems with a significant propositional component as well as for first-order problems where contradictory instances are generated early, but are not detected until all involved clauses have been selected for processing.

We have thus integrated the CDCL solver PicoSAT [3] with E. The saturation loop is periodically interrupted, and all clauses in the current proof state are grounded, i.e. all variables are bound to a constant of the proper sort. The instantiated clauses are efficiently translated into propositional clauses and handed to PicoSAT. Our original implementation used PicoSAT as an external tool via files and UNIX pipes [20], but as of E 2.2 we link PicoSAT as a library and use its documented C API. If PicoSAT refutes the given propositional problem, E extracts the unsatisfiable core and relates it back to the original first-order clauses to construct a proof object. If PicoSAT fails to find unsatisfiability, the saturation is resumed.

E users can control this process by choosing the following options:

- the point when PicoSAT is called – currently it is after N generated or processed clauses or after N new subterms created, where N is a user-chosen constant
- the way variables are instantiated with constants – some of the options are to use the most or the least frequent constant, a fresh constant (for each type), or a frequent or infrequent constant appearing in the conjecture.

We have only started to explore the parameter space. With current configurations, E finds about 1% more proofs on TPTP when PicoSAT is enabled. While this number seems low, it is significant among hard problems - 90% of solutions are found before the first run of the SAT solver.

3.3 Higher-order logic support

One of the most recent updates for E adds support for λ-free higher-order logic (LFHOL). Supporting richer logics in a highly optimized theorem prover without

compromising performance required some changes to fundamental data structures and algorithms. Here we will only briefly describe the changes. We refer the reader to [31] for details.

LFHOL is a fragment of simply-typed higher-order logic, with no λ-abstraction, but supporting functional variables and partial application of terms. It is expressive enough to axiomatize, for example, frequently used functional programming combinators such as map. We have generalized E's term representation to allow applied variables, as well as the type system to support partially applied terms. The most laborious change was the extension of all three indexing data structures to support more complex terms. Our experimental results show that E extended to support LFHOL natively outperforms the traditional encoding-based approaches. E 2.3 users can specify LFHOL problems in TPTP THF syntax. Support for LFHOL can be specified as an option at compile time.

3.4 Search Control

All provers for first-order logic search for proofs in an infinite search space. While *fairness* is a minimal requirement for completeness, practical performance depends critically on making the right choices. The prover supports a large number of options for controlling preprocessing and actual search control.

The most important parameters for the saturation are the *term ordering*, the *literal selection strategy*, and *clause evaluation heuristics*. Term orderings primarily determine in which direction equations can be applied (and as a consequence, which terms are overlapped for superposition inferences), and which literals are maximal and hence available for inferences. Literal selection can be used to overwrite the default inference literals and restrict inferences to particular (negative) literals. Finally, clause evaluations determine the order in which the given-clause algorithm processes clauses. In the simplest case, this is a single value, representing the number of symbols in the clause (known as a *symbol-counting* heuristic — 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. Each queue is ordered by a particular evaluation function. 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 likely connected to the conjecture. We have so far only evaluated a small part of the possibility space opened by this design [21].

More complex clause evaluation functions allow the system to evaluate clauses based on a user-provided *watch list*. Clauses that match clauses on the watch list are preferred over other clauses. Watch lists can either be created based on human intuition, by manual analysis of similar proofs, or by automated mining of related proofs. The watch list mechanism in E has been improved several times, with the current incarnation [7] being successfully used for challenging mathematical problems.

Automatic prover configuration 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.

3.5 Usage and formats

Recent versions of E have made minor changes to the usage and options, as well as to I/O formats. These are mostly conservative, i.e. they should not significantly impact integration of the prover into larger systems.

The first such change is automatic detection of the input format. E supports three different formats: The original LOP format inherited from SETHEO, the old TPTP format (TPTP format version 1/2) and the current TPTP format version 3. The prover has originally used command line options to select the desired format. However, with E 1.9.1, we introduced automatic detection of the input format (and automatic setting of the corresponding output format). This feature is implemented by checking the first proper input token, and selecting TPTP-3 format if it is one of the TPTP-3 language identifiers (`cnf`, `fof`, ...), or `include`, TPTP-2 format if it is one of `input_clause` or `input_formula`, and LOP otherwise. It is not completely foolproof (it can e.g. misidentify LOP input that uses TPTP-3 language identifiers as normal function symbols), but it works very well in practice. If it misidentifies the format, it fails towards the more modern TPTP-3 format. We have not yet encountered that situation. If TPTP-3 syntax is identified, the output syntax is also set to TPTP-3, otherwise it is set to PCL2 (E's original, more limited format). These choices can be independently overwritten via explicit use of the existing command line options.

The second change is more strict checking of TPTP language constraints. In particular, E now requires FOF, TFF and TCF style formulas to be fully quantified. CNF formulas are implicitly considered universally quantified. This change was prompted by frequent user errors dur to misjudging quantifier scopes and hence inadvertently creating free variables. Such cases will now be flagged as errors.

Similarly, E will now automatically type numerical constants as `$int`/`$rat` or `$real` (which will result in errors if they are used in untyped formulas) unless they are explicitly marked as free constants by a command line switch.

Finally, the prover now can detect proofs resulting from an inconsistent axiom set, and explicitly report the problem status as `ContradictoryAxioms`.

4 Experimental Evaluation

We have performed an evaluation of E 2.3 on the 16094 CNF and FOF problems of the TPTP problem library [25], release 7.2.0. Experiments were run on the StarExec cluster [23], i.e. on machines with an Intel Xeon E5/2.40 GHz processor

Strategy	**UEQ**	**CNE**	**CEQ**	**FNE**	**FEQ**	**All**
Class size	(1193)	(2383)	(4442)	(1771)	(6305)	(16094)
Auto (proofs)	814	1603	2360	1156	3704	9637
Auto (sat)	16	280	220	304	203	1023
Auto (all)	830	1883	2580	1460	3907	10660
E 1.8 Auto (all)	*812*	*1851*	*2561*	*1456*	*3909*	*10589*
Schedule (proofs)	829	1625	2470	1165	3961	10050
Schedule (sat)	16	286	219	307	206	1034
Schedule (all)	845	1911	2689	1472	4167	11084
E 1.8 Schedule (all)	*828*	*1889*	*2655*	*1463*	*4113*	*10948*

Table 1. Proofs and (counter-)saturations found within 300 seconds

and at least 128 GB of main memory. We used a CPU time limit of 300 seconds per problem and the prover was configured to optimize memory usage to at most 2 GB.

Table 1 summarizes the results of the experiment. We list the performance for unit-equational problems, clausal and non-clausal problems with and without equality. The two tested strategies are the automatic mode and the automatic strategy scheduler. For each strategy, we list the number of proofs found, the number of counter-saturations (i.e. saturations not including the empty clause), and the total number of successes. For comparison, we have also included data for E 1.8, the last version with a formally published description. The full data, including the exact command line options, is available at `http://www.eprover.eu/E-eu/E_2.3.html`.

5 Future Work

While E is quite mature and widely used, there is a number of projects for further improvement - in data structures, search control, and supported language. In particular, terms can be more compactly represented with variable length arrays (a feature not yet available in standard C when the data type was first designed), and priority queues can be more efficiently realized with heaps. Feature vector indexing works very well for classical theorem proving problems, but is less than optimal for problems with very large and sparsely used signatures. We plan to develop it into a more adaptive and efficient variant.

On the language side, we plan to support the full TFX language [26] and hence the FOOL logic. We also plan to add at least basic support for interpreted arithmetic sorts.

A lot of recent improvements have only been evaluated in isolation, not in concert. A major project is such a large-scale evaluation and a regeneration of the automatic modes to make better use of the new features.

Finally, E has grown over more than 20 years now. While we have tried to integrate new techniques in as modular and elegant a way as possible, some of the higher level-code can profit from significant refactoring and streamlining.

6 Conclusion

E is a mature and yet still developing fully automated theorem prover for first-order logics and some extensions. It has good performance, as demonstrated in the yearly CASC competitions [24].

The prover is available as free and open source software, and has been used and extended by a large number of parties. We hope and expect that this success will continue throughout the third decade of its lifetime.

References

1. Bachmair, L., Ganzinger, H.: Rewrite-Based Equational Theorem Proving with Selection and Simplification. Journal of Logic and Computation **3**(4), 217–247 (1994)
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-lib standard: Version 2.0. In: Proc. of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010), `http://homepage.cs.uiowa.edu/~tinelli/papers/BarST-SMT-10.pdf`
3. Biere, A.: PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation **4**, 75–97 (2008)
4. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formal. Reasoning **9**(1), 101–148 (2016). https://doi.org/10.6092/issn.1972-5787/4593
5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011), `http://proval.lri.fr/publications/boogie11final.pdf`
6. Denzinger, J., Kronenburg, M., Schulz, S.: 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
7. Goertzel, Z., Jakubův, J., Schulz, S., Urban, J.: ProofWatch: Watchlist guidance for large theories in E. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving: 9th International Conference, Oxford, UK. pp. 270–288. Springer (2018)
8. Hoder, K., Voronkov, A.: Sine Qua Non for Large Theory Reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Proc. of the 23rd CADE, Wroclav. LNAI, vol. 6803, pp. 299–314. Springer (2011)
9. Korovin, K.: Inst-Gen - A Modular Approach to Instantiation-Based Automated Reasoning. In: Programming Logics - Essays in Memory of Harald Ganzinger, LNCS, vol. 7797, pp. 239–270. Springer (2013)
10. Kotelnikov, E., Kovács, L., Reger, G., Voronkov, A.: The Vampire and the FOOL. In: Avigad, J., Chlipala, A. (eds.) Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, USA. pp. 37–48. ACM (2016)
11. Löchner, B., Schulz, S.: An Evaluation of Shared Rewriting. In: de Nivelle, H., Schulz, S. (eds.) Proc. of the 2nd International Workshop on the Implementation of Logics. pp. 33–48. MPI Preprint, Max-Planck-Institut für Informatik, Saarbrücken (2001)
12. McCune, W.: Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. Journal of Automated Reasoning **9**(2), 147–167 (1992)

13. 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)
14. Reger, G., Suda, M., Voronkov, A.: Playing with AVATAR. In: Felty, A.P., Middeldorp, A. (eds.) Proc. of the 25th CADE, Berlin, Germany. LNAI, vol. 9195, pp. 399–415. Springer (2015)
15. Schulz, S.: E – A Brainiac Theorem Prover. Journal of AI Communications **15**(2/3), 111–126 (2002)
16. 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)
17. 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)
18. Schulz, S.: System Description: E 1.8. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) Proc. of the 19th LPAR, Stellenbosch. LNCS, vol. 8312, pp. 735–743. Springer (2013)
19. Schulz, S.: E 2.0 User Manual. EasyChair preprint no. 8 (2018). https://doi.org/10.29007/m4jw
20. Schulz, S.: Light-weight integration of SAT solving into first-order reasoners – first experiments. In: Kovács, L., Voronkov, A. (eds.) Vampire 2017. Proceedings of the 4th Vampire Workshop. EPiC Series in Computing, vol. 53, pp. 9–19. EasyChair (2018). https://doi.org/10.29007/89kc, `https://easychair.org/publications/paper/94vW`
21. Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: Olivetti, N., Tiwari, A. (eds.) Proc. of the 8th IJCAR, Coimbra. LNAI, vol. 9706, pp. 330–345. Springer (2016)
22. Schulz, S., Sutcliffe, G.: Proof generation for saturating first-order theorem provers. In: Delahaye, D., Woltzenlogel Paleo, B. (eds.) All about Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55, pp. 45–61. College Publications, London, UK (January 2015)
23. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A Cross-Community Infrastructure for Logic Solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Proc. of the 7th IJCAR, Vienna. LNCS, vol. 8562, pp. 367–373. Springer (2014)
24. Sutcliffe, G.: The 8th IJCAR automated theorem proving system competition–CASC-J8. AI Communications **29**(5), 607–619 (2016)
25. Sutcliffe, G.: The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017)
26. Sutcliffe, G., Kotelnikov, E.: TFX: The TPTP extended typed first-order form. In: Konev, B., Urban, J., Rümmer, P. (eds.) Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning (PAAR), Oxford, UK. pp. 72–87. No. 2162 in CEUR Workshop Proceedings (2018), `http://ceur-ws.org/Vol-2162/#paper-07`
27. 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)
28. 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 (2006)

29. 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)
30. Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic - report version. Tech. rep., Matryoshka Project (2018), http://matryoshka.gforge.inria.fr/pubs/ehoh_report.pdf
31. Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. In: Vojnar, T., Zhang, L. (eds.) Proc. 25th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19). pp. 192–210. No. 11427 in LNCS, Springer (2019)