New Heights: E 3.0

Stephan Schulz[1], Petar Vukmirović[2], and Jasmin Blanchette[3]

[1] DHBW Stuttgart, Germany, schulz@eprover.org
[2] Vrije Universiteit Amsterdam, The Netherlands, petar.vukmirovic2@gmail.com
[3] Ludwig-Maximilians-Universität München, Germany,
jasmin.blanchette@ifi.lmu.de

Abstract. E 3.0 is a fully automatic theorem prover for classical first-order logic with equality, many-sorted first-order logic (with first-class Booleans), and monomorphic higher-order logic. It also supports efficient propositional reasoning, (near-)zero-overhead proof objects, and the integration of machine learning for proof search control. This paper describes the overall architecture of the system, with a focus on new features (first-class Booleans, full higher-order logic, multicore scheduling, data-driven schedule selection and representation). We discuss major aspects of the implementation and the performance of the system on the TPTP library.

1 Introduction

E is a fully automatic theorem prover. It has been under development for over 20 years, gaining 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 E 2.3. The current release, E 3.0 *Shangri-La*, introduces support for full higher-order logic as well as for most FOOL/TFX features [26].

The basic architecture of the clausal inference core has previously been described in a 2002 article [13] covering E 0.62, and the last updated description of E 2.3 was published in 2019 [19]. Details on the extension to full higher-order logic are described in a 2023 paper [31]. In this paper, we give an overview of 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 (Fig. 1): *parsing*, *preliminary analysis*, *axiom selection*, *higher-order preprocessing*, *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 or the automatic mode.

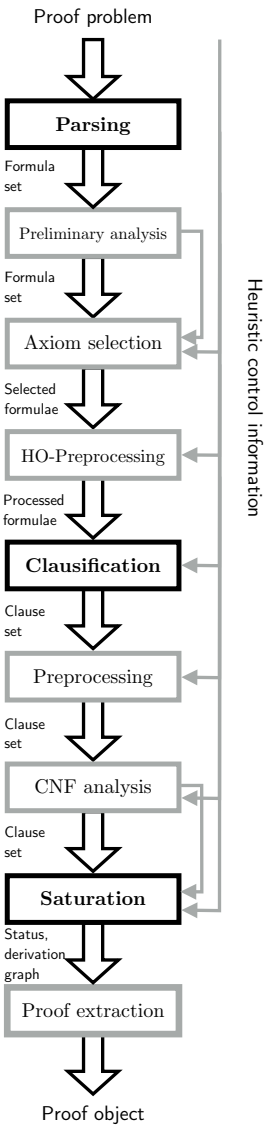

Fig. 1. Logical pipeline

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 whether 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 [7]. 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.

Next, the prover performs some preprocessing steps. It normalises Boolean equations into equivalences, it eliminates the FOOL constructs *let* and *ite*, and it may perform λ-lifting (which replaces λ-abstractions by concrete function symbols).

At the core of the prover is a refutational proof procedure for 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- or higher-order problem into a set of clauses. It is based on the ideas presented by Nonnengart and Weidenbach [11]. 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. Higher-order problems are also eagerly clausified, even though Boolean structure may re-emerge during proving and then needs to be eliminated by dedicated inference rules [3, 31]. The prover optionally performs further preprocessing of the clause set. Preprocessing removes redundant literals and tautologies, optionally unfolds some or all equational definitions, and sorts literals and clauses in a canonical order 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 procedure*. The main additional search parameter for this procedure is the scheme for the selection of the *given clause* for each iteration of the main loop. 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* [4] 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 First- and Higher-Order Superposition

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.

For version 2.0, the prover was 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 only take terms of the correct sorts as arguments. An exception is the equality predicate, which is ad hoc polymorphic but requires both arguments to be of the same sort.

The superposition calculus can straightforwardly be generalized to this logic. E implements the core inference rules from Bachmair and Ganzinger [2]: superposition, equality resolution, and equality factoring. In addition, it implements a large set of simplification rules, including unconditional rewriting, subsumption, equational literal cutting, and contextual literal cutting [17].

Version 3.0 introduces support for full higher-order logic [31]. The underlying calculus is a pragmatic, incomplete variant of the λ-superposition calculus with Booleans [3]. Compared with E 2.3, which already supported application of variables to terms, this version also supports λ-abstraction; hence, the E 3.0 term representation compatible with full higher-order logic. Terms are subject to implicit $\alpha\beta\eta$-normalization; thus, $\lambda x.\, \mathsf{f}\, x$ and $\lambda y.\, \mathsf{f}\, y$ are considered syntactically identical by the calculus, and they are both syntactically identical to f.

The core inference rules (superposition, equality resolution, and equality factoring) are analogous to those of first-order superposition, with three main differences: First, they operate on higher-order terms. Second, instead of a single

most general unifier, they compute a finite (possibly incomplete) sequence of unifiers [31]. Third, they use term orderings designed for higher-order terms [3].

Superposition inferences are performed only at positions that correspond to a first-order-style context. Consider this example, where g is considered larger than f according to the term ordering. Given the clauses $\mathsf{g} \simeq \mathsf{f}$ and $\mathsf{g}\,\mathsf{a} \not\simeq \mathsf{f}\,\mathsf{a}$, no superposition inference is possible to rewrite g in $\mathsf{g}\,\mathsf{a}$, because $[\,]\,\mathsf{a}$ is not a first-order-style context — the hole corresponds to a function, which is higher-order. Instead, a new inference rule, called argument congruence, comes into play and generates the clause $\mathsf{g}\,X \simeq \mathsf{f}\,X$ from the premise $\mathsf{g} \simeq \mathsf{f}$. This new clause can then be used to rewrite $\mathsf{g}\,\mathsf{a}$, yielding the conclusion $\mathsf{f}\,\mathsf{a} \not\simeq \mathsf{f}\,\mathsf{a}$. From there, equality resolution takes over and generates the empty clause.

Even though the outer Boolean structure is removed during the initial clausification of the problem, formulas can reappear at the top level of clauses during saturation. For example, after instantiating X with $\lambda x.\, \lambda y.\, x \wedge y$, the clause $X\,\mathsf{p}\,\mathsf{q} \vee \mathsf{a} \simeq \mathsf{b}$ becomes $(\mathsf{p} \wedge \mathsf{q}) \vee \mathsf{a} \simeq \mathsf{b}$. E dynamically clausifies every clause of the form $\varphi \vee C$ where φ is a headed by a logical symbol.

The λ-superposition calculus includes many rules that act on Boolean subterms and that are necessary for completeness. Other than Boolean simplification rules, which use simple equivalences such as $\mathsf{p} \wedge \top \leftrightarrow \mathsf{p}$ to simplify terms, we have implemented none of these Boolean rules in E. Instead, E uses an incomplete, but more easily controllable and intuitive rule, called *primitive instantiation*. This rule instantiates free predicate variables with approximations of formulas that are ground instances of the variable [33].

E also has special handling for the Hilbert choice operator, inspired by Leo-III [22], and for Leibniz equality [33]. Moreover, E treats induction axioms specially. Immediately after clausification, it abstracts literals from the goal clauses and instantiates induction axioms with these abstractions. After E has collected all the abstractions, it traverses the clauses and instantiates those that have applied variable of the same type as the abstraction.

3.2 Implementation

E is being developed in C, which provides good performance and high 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 UNIX-style operating systems, in particular 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 it has been largely moved to Python.

While C is an excellent language for performance and portability, it offers only a 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 its hearth 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 [9]. 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, for example, decide whether 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, 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 have proven to be efficient and convenient. In particular, the mark-and-sweep garbage collector reduces programmer effort and errors. Term banks are also used to represent formulas and in some roles even clauses.

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 properties, 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.

Sorts were originally represented as indices into a sort table. The LFHOL extension of E 2.3 [32] generalized this representation to support higher-order simple types and partially applied terms. Types now use 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 equality comparisons. These types were reused as is for the extension to full higher-order logic [31].

Proof Procedure The main saturation procedure is a modified version of the DISCOUNT loop [6], one of the variants of the given-clause procedure. The proof state is represented by two disjoint subsets of clauses, the *unprocessed* clauses U and the *processed* clauses P. 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 generated. Forward simplification is performed on the given clause (using clauses in P as side premises) before the clause 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 back to U. This maintains the additional invariant that the clauses in P are interreduced. We refer to the system description of E 2.3 for the pseudocode of the saturation procedure [19, Fig. 2].

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

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 premises is the given clause, the other one is a clause in P. E uses a *fingerprint index* [14] 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 between 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* [10] 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* [15], a clause indexing technique that supports finding both generalizations and instances.

Higher-Order Logic Support E 3.0 adds support for monomorphic higher-order logic. 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 to Vukmirović et al. [31, 32] for details.

We have generalized E's term representation to allow applied variables (e.g., $Y\ \mathsf{a}$), as well as the type system to support partially applied terms and λ-abstractions. The most laborious change was the extension of all three indexing data structures to support more complex terms.

E uses De Bruijn indices to represent the bound variables of λ-terms [5]. This enforces α-equivalence. For example, the term $\lambda x.\, \lambda y.\, \mathsf{f}\, x\, x$ is represented as the first-order-style term $\mathsf{LAM}(\mathsf{0}, \mathsf{LAM}(\mathsf{0}, \mathbf{f}(\mathsf{1}, \mathsf{1})))$. The first argument of LAM is redundant, since it can be deduced from the type of the λ-abstraction. However, it is convenient to have it stored there when performing basic term manipulation.

As for $\beta\eta$-equivalence, it is enforced by representing terms in $\beta\eta$-reduced forms. Since β- and η-reduction are performed very often, they need to be efficient. There are many optimizations. Notably, terms are equipped with a Boolean property that indicates whether they contain a β-redex somewhere as a subterm. If this property is not set — as will always be the case for terms belonging to the logic's first-order fragment — the term does not need to be visited by the β-reduction procedure. Similarly, for η-reduction, only terms containing λ-abstractions are visited.

When we added support for higher-order logic, another necessary change was to the unification and matching procedures, which need to cope with applied variables and λ-abstractions. In general, higher-order unification may lead to an infinite set of incomparable unifiers. For example, unifying $Y\ (\mathsf{f}\ \mathsf{a})$ with $\mathsf{f}\ (Y\ \mathsf{a})$ yields the infinite set of unifiers $\{\{Y \mapsto \lambda x.\, x\}, \{\{Y \mapsto \lambda x.\, \mathsf{f}\ x\}, \{\{Y \mapsto \lambda x.\, \mathsf{f}\ (\mathsf{f}\ x))\}, \ldots\}$. E 3.0 implements a terminating, incomplete variant of the procedure described by Vukmirović et al. [30].

The support for higher-order logic incurs very little overhead, as we would expect from a graceful generalization. Nevertheless, it is disabled by default and must be explicitly enabled at compile time. Experimental results [31, 32]

show that E extended to support higher-order logic natively outperforms the traditional encoding-based approaches.

SAT Solver Integration SAT solvers have greatly improved in 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 [8, 12], we want to use this power to improve the performance of the prover both for problems with a significant propositional component and 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 integrated the CDCL-based SAT solver PicoSAT [4] with E. The saturation loop is periodically paused, 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 passed to PicoSAT via its 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.

There are various options to control the SAT solver integration in detail [16]. With current configurations, E finds about 1% more proofs on the TPTP when PicoSAT is enabled. While this number seems low, it is significant among problems that are hard for E without SAT support. (Overall, 90% of solutions are found before the first run of the SAT solver, and thus are easy for E.)

Strong Rewriting Unfailing completion [1] allows rewriting (with some restrictions) by all orientable instances of equations. This carries over to the superposition calculus. However, to our knowledge, most provers only approximate this rewrite relation. They simplify with orientable equations (or rules). For equations, they compute a match from one side onto the target term and check if the resulting instance is orientable. This has the effect that equations with free variables in the potentially smaller side can never be used for rewriting. With E 2.5, we have introduced *strong rewriting*, where such free variables (which are implicitly universal) are instantiated with the smallest constant of the proper sort. This results in small but significant improvements [18].

Internal Proof Objects With E 1.8, we finally found a way to use the invariants of the given-clause procedure to very compactly represent the relevant parts of the derivation graph internally [21]. The core insight is that nearly all inferences involve at most one non-processed clause, while the other premises are processed clauses. Processed clauses are only rarely back-simplified, so we can keep a complete record by archiving only those few clauses, and otherwise store all side premises (and inferences) with the main premise of an inference. With this, the overhead in time and memory turned out to be negligible, so E now always builds an internal derivation graph. In addition to efficiently providing a checkable proof object in TPTP syntax [28], the presence of the derivation information enables the detection of vacuous proofs (based on an inconsistent

axiomatization). It also enables an elegant lazy implementation of *orphan clause deletion*, i.e. deletion of clauses which have lost one of their parents to simplification and hence are also redundant.

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`.

3.3 System Configuration and Search Control

Provers for first-order logic search for proofs in an infinite search space. Practical performance depends critically on making the right choices. E supports a large number of options for controlling preprocessing and actual search control.

The main parameters for the saturation are the calculus's term ordering and literal selection strategy as well as the *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 procedure processes clauses. In the simplest case, this is a single value, representing the number of symbols in the clause. 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 [20].

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. While in the past these auto-modes were represented via generated C code, we have now implemented them as a combination of a classification module and a strategy- or schedule-selection module that simply assigns a named strategy or schedule to each class.

All search control options are now collated into a single data structure that is printable and parsable. All internally used strategies are represented in symbolic form in the source code. The user can request that the parameters selected by the auto-mode are printed, and can also now modify these settings, leaving most decision to the prover, but overriding individual settings.

Multi-Core Strategy Scheduling E has supported (sequential) strategy scheduling since 2013. However, the current release has switched to a much more flexible system that also the use of multiple cores processors. On the technical level, the main instance of the prover forks of clones and connects to them via UNIX interprocess communication (pipes). It monitors their progress, and if a subordinate instance reports success, the main instance terminates all other instances and prints the result (and optional proof object) delivered by the successful instance. As usual with a UNIX fork, the children receive a (virtual) copy of the state of the system, i.e. they can all share in the work done up to the fork. No forward communication is necessary during proof search. Backward communication (child to main instance) is done via a pipe and only contains plain text analysed as such. The approach has shown itself as simple and robust.

E's original strategy scheduling shared parsing, axiom selection, clausification, and some preprocessing before starting starting the saturation with different search parameters. However, for higher-order logic we believe that there is a stronger dependence on different clausification and preprocessing steps. Hence, the current scheme allows diversification at two levels — after the preliminary analysis step, the prover can use various *preprocessing strategies*, and after the CNF analysis, each of these can create multiple *saturation strategies*. Each preprocessing strategy is assigned a fraction of available CPU time that it can further subdivide for its individual saturation schedules.

4 Experimental Evaluation

We have performed an evaluation of E 3.0 on the 22 237 non-arithmetic problems of the TPTP problem library [25], release 8.1.1. Experiments were run on the Miami instance of the StarExec cluster [23], i.e., on machines with an Intel Xeon CPU E5-2620 v4 @ 2.10 GHz processors and 256 GB of main memory. Each machine had two processors with 8 cores per processor. In normal usage, only one job is scheduled to each machine at a time.

We compare E compiled without higher order-support and λE, the version with support for higher-order logic. For both variants we present results for the simple automatic mode, the strategy-scheduling mode with strategies scheduled sequentially on a single core, and a strategy-parallel version scheduling different

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

Configuration	**All**	**HO**	**FO**	**CNF**	**FOF**	**TFF**
Class size	(22237)	(3832)	(18338)	(8344)	(9091)	(903)
E (auto)	11415	0	11415	5441	5739	235
E (schedule)	11992	0	11992	5692	6052	248
E (8-core)	12552	0	12552	5871	6421	260
λE (auto)	13807	2397	11410	5438	5738	234
λE (schedule)	14470	2502	11968	5682	6038	248
λE (8-core)	15192	2659	12533	5858	6416	259

strategies onto (up to) 8 cores (i.e. all the cores of one processor). For the sequential versions, we imposed a CPU time limit of 300 seconds, for the multi-core version a wall-clock limit of 300 seconds and a CPU time limit of 300 seconds per core (for a theoretical limit of 2400 seconds). The prover was configured to optimize memory usage to at most 2 GB per process..

Table 1 shows the number of successes – proofs and countersaturations – for the six different prover configurations. We show results for all problems, and separately for higher-order and first-order problems. For first-order we show a further breakdown into clause normal form (CNF), (unsorted) first-order (FOF) and many-sorted first-order (TFF) problems. We can see that the first-order version solves between 62% and 68% of all first-order problems, while λE solves between 62% and 69% of the higher-order version. Since the higher-order version is incomplete, all of these successes are proofs. On first-order problems, the first-order version of E is marginally stronger than λE, demonstrating that integration of the higher-order features incurs very little overhead indeed. Both versions fare much worse on TFF problems than on other problems.

The full data, including the exact command-line options, are available at `http://www.eprover.eu/E-eu/E_3.0.html`.

5 Conclusion and Future Work

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]. In the 2022 CASC-J11, it won first place in the SLH division and second places in THF, FOF, and UEQ.

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 through the third decade of its lifetime.

While E is quite mature and widely used, there are several projects for further improvement — concerning data structures, logical language, and search control. In particular, the priority queues can be more efficiently realized with lazy 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. In addition, there are various minor data structures that can be improved. On the language side, we plan to add at least basic support for interpreted arithmetic sorts.

With respect to search control, we plan a further simplification and unification of the multicore scheduling mode and sequential operation. We also want to expose full schedules to the user via suitable command-line options and configuration files. We also plan to improve problem classification (using the presence of significant patterns and structural properties of terms and formulas).

Finally, 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.

References

1. Bachmair, L., Dershowitz, N., Plaisted, D.: Completion Without Failure. In: Ait-Kaci, H., Nivat, M. (eds.) Resolution of Equations in Algebraic Structures. vol. 2, pp. 1–30. Academic Press (1989)
2. Bachmair, L., Ganzinger, H.: Rewrite-Based Equational Theorem Proving with Selection and Simplification. J. Log. Comput. **3**(4), 217–247 (1994)
3. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirovic, P.: Superposition for Higher-Order Logic. J. Autom. Reason. **67**(1), 10 (2023)
4. Biere, A.: PicoSAT Essentials. J. Satisf. Boolean Model. Comput. **4**, 75–97 (2008)
5. Charguéraud, A.: The Locally Nameless Representation. J. Autom. Reason. **49**(3), 363–408 (2012)
6. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT: A Distributed and Learning Equational Prover. J. Autom. Reason. **18**(2), 189–198 (1997)
7. Hoder, K., Voronkov, A.: Sine Qua Non for Large Theory Reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Proc. of the 23rd CADE, Wroclav, Poland. LNAI, vol. 6803, pp. 299–314. Springer (2011)
8. Korovin, K.: Inst-Gen — A Modular Approach to Instantiation-Based Automated Reasoning. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics — Essays in Memory of Harald Ganzinger, LNCS, vol. 7797, pp. 239–270. Springer (2013)
9. 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, Havana, Cuba. pp. 33–48. MPI Preprint, Max-Planck-Institut für Informatik, Saarbrücken (2001)
10. McCune, W.: Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. J. Autom. Reason. **9**(2), 147–167 (1992)
11. 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 and MIT Press (2001)
12. 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)
13. Schulz, S.: E – A Brainiac Theorem Prover. AI Commun. **15**(2–3), 111–126 (2002)
14. Schulz, S.: Fingerprint Indexing for Paramodulation and Rewriting. In: Gramlich, B., Sattler, U., Miller, D. (eds.) Proc. of the 6th IJCAR, Manchester, UK. LNAI, vol. 7364, pp. 477–483. Springer (2012)
15. 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)
16. Schulz, S.: Light-Weight Integration of SAT Solving into First-Order Reasoners — First Experiments. In: Kovács, L., Voronkov, A. (eds.) Proc. of the 4th Vampire Workshop, Gothenburg, Sweden. EPiC Series in Computing, vol. 53, pp. 9–19. EasyChair (2018)
17. Schulz, S.: E 2.4 User Manual. EasyChair preprint no. 2272 (2019), `https://easychair.org/publications/preprint/RjDx`
18. Schulz, S.: Empirical properties of term orderings for superposition. In: Konev, B., Schon, C., Steen, A. (eds.) Proc. of the 8th PAAR, Haifa, Israel. CEUR Workshop Proceedings (2022), `http://ceur-ws.org/Vol-3201/`

19. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, Higher, Stronger: E 2.3. In: Fontaine, P. (ed.) Proc. of the 27th CADE, Natal, Brasil. LNAI, vol. 11716, pp. 495–507. Springer (2019)
20. 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)
21. 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)
22. Steen, A., Benzmüller, C.: Extensional Higher-Order Paramodulation in Leo-III. J. Autom. Reason. **65**(6), 775–807 (2021)
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 Commun. **29**(5), 607–619 (2016)
25. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure — from CNF to TH0, TPTP v6.4.0. J. Autom. Reason. **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.) Proc. of the 6th Workshop on Practical Aspects of Automated Reasoning, Oxford, UK. pp. 72–87. No. 2162 in CEUR Workshop Proceedings (2018)
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, Mérida, Venezuela. 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, USA. 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`, accessed 8 July 2013
30. Vukmirovic, P., Bentkamp, A., Nummelin, V.: Efficient Full Higher-Order Unification. Log. Methods Comput. Sci. **17**(4) (2021)
31. Vukmirović, P., Blanchette, J., Schulz, S.: Extending a High-Performance Prover to Higher-Order Logic. In: Sharygina, N., Sankaranarayanan, S. (eds.) Proc. of the 29th TACAS, Paris, France. LNCS, Springer (2023), accepted for publication
32. 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. of the 25th TACAS, Prague, Czech Republic. LNCS, vol. 11427, pp. 192–210. Springer (2019)
33. Vukmirović, P., Nummelin, V.: Boolean reasoning in a higher-order superposition prover. In: Fontaine, P., Korovin, K., Kotsireas, I.S., Rümmer, P., Tourret, S. (eds.) Proc. of the 7th Workshop on Practical Aspects of Automated Reasoning and the 5th Workshop on Symbolic Computation and Satisfiability Checking. CEUR Workshop Proceedings, vol. 2752, pp. 148–166. CEUR-WS.org (2020)