CASC-29

CASC-29
CASC-29
CASC-29

# Proceedings of CASC-29 – the CADE-29 ATP System Competition

Geoff Sutcliffe

University of Miami, USA

**Abstract**

The CADE ATP System Competition (CASC) is the annual evaluation of fully automatic, classical logic, Automated Theorem Proving (ATP) systems - the world championship for such systems. CASC-29 was the twenty-eighth competition in the CASC series. Twenty-four ATP systems competed in the various divisions. These proceedings present the competition design and information about the competing systems.

## 1 Introduction

The CADE ATP System Competition (CASC) [100] is the annual evaluation of fully automatic, classical logic, ATP systems – the world championship for such systems. One purpose of CASC is to provide a public evaluation of the relative capabilities of ATP systems. Additionally, CASC aims to stimulate ATP research, motivate development and implementation of robust ATP systems that can be easily and usefully deployed in applications, provide an inspiring environment for personal interaction between ATP researchers, and expose ATP systems within and beyond the ATP community. CASC evaluates the performance of the ATP systems in terms of the number of problems solved, the number of acceptable proofs and models output, and the average time taken for problems solved, in the context of a bounded number of eligible problems and specified time limits.

CASC is held at each CADE (the International Conference on Automated Deduction) and IJCAR (the International Joint Conference on Automated Reasoning) conference – the major forums for the presentation of new research in all aspects of automated deduction. CASC-29 was held on 3rd July 2023, as part of the 29th International Conference on Automated Deduction (CADE-29). It was the twenty-eighth competition in the CASC series; see [121, 127, 124, 69, 71, 120, 118, 119, 76, 78, 80, 82, 85, 87, 89, 91, 93, 95, 97, 126, 99, 102, 105, 108, 110, 114, 115] and the CASC web site `http://www.tptp.org/CASC`, for information about previous competitions. CASC-29 was organized by Geoff Sutcliffe, assisted by Martin Desharnais for the SLH division, and overseen by a panel consisting of Cláudia Nalon, Sophie Tourret, and Christoph Weidenbach. The competition was run on computers provided by the StarExec project [64] at the University of Miami. The CASC-29 web site provides access to all the resources used before, during, and after the event: `http://www.tptp.org/CASC/29`.

The design and organization of CASC has evolved over the years to a sophisticated state [121, 122, 117, 123, 67, 68, 70, 72, 73, 74, 75, 77, 79, 81, 84, 86, 88, 90, 92, 94, 96, 98, 101, 104, 106, 107, 109, 111]. Important changes for CASC-29 were (for readers already familiar with the general design of CASC):

- The LTB division went on hiatus.
- The Typed First-order Non-theorem (TFN) division returned from hiatus, using typed (monomorphic) first-order problems without arithmetic.

The CASC rules, specifications, and deadlines are absolute. Only the panel has the right to make exceptions. It is assumed that all entrants have read the documentation related to the

competition, and have complied with the competition rules. Non-compliance with the rules can lead to disqualification. A "catch-all" rule is used to deal with any unforeseen circumstances: *No cheating is allowed*. The panel is allowed to disqualify entrants due to unfairness, and to adjust the competition rules in case of misuse.

These proceedings are organized as follows: Section 2 describes the competition divisions and the ATP systems that entered the various divisions. Sections 3 and 4 describe the competition infrastructure and the requirements for the ATP systems. Section 5 describes how the systems are evaluated. Sections 6 and 7 describe the practical steps for registering a system and checking that is has the required properties. Section 8 provides descriptions (written by the entrants) of the systems entered into CASC-29. Section 9 concludes.

*A Tense Note:* Attentive readers will notice changes between the present and past tenses in this paper. Many parts of CASC are established and stable – they are described in the present tense (the rules are the rules). Aspects that were particular to CASC-29 are described in the past tense so that they make sense when reading this after the event.

# 2 Divisions and Systems

CASC is divided into divisions according to problem and system characteristics, in a coarse version of the TPTP problem library's Specialist Problem Classes (SPCs) [125]. Each division uses problems that have certain logical, language, and syntactic characteristics, so that the systems that compete in a division are, in principle, able to attempt all the problems in the division. Some divisions are further divided into problem categories that make it possible to analyze, at a more fine-grained level, which systems work well for what types of problems. Table 1 catalogs the divisions and problem categories of CASC-29. The example problems can be viewed online at `http://www.tptp.org/cgi-bin/SeeTPTP?Category=Problems`. Sections 3.2 and 3.3 explain what problems are eligible for use in each division and category.

Systems that do not run in the competition divisions for any reason (e.g., the system requires special hardware, or the entrant is an organizer) can be entered into the demonstration division. The demonstration division uses the same problems as the competition divisions, and the entry specifies which competition divisions' problems are to be used.

Twenty-four ATP systems competed in the various divisions. The division winners from CASC-J11 (the previous CASC), and the Prover9 1109a system, are automatically entered into the corresponding demonstration divisions to provide benchmarks against which progress can be judged. The systems, the divisions in which they were entered, and their entrants, are listed in Tables 2 and 3. A division acronym in *italics* indicates the system was in the demonstration division. System descriptions are in the competition proceedings [112] and on the CASC-29 web site.

# 3 Infrastructure

## 3.1 Computers

The competition computers had:

- An octa-core Intel(R) Xeon(R) E5-2667, 3.20GHz CPUs, without hyperthreading
- 128GB memory
- The CentOS Linux release 7.4.1708 (Core) operating system, with Linux kernel 3.10.0-693.el7.x86_64.

Table 1: Divisions and Problem categories

| Division | Problems | Problem categories |
|---|---|---|
| **THF** | **T**yped (monomorphic) **H**igher-order **F**orm theorems (axioms with a provable conjecture). | **TNE** – **T**HF with **N**o **E**quality, e.g., `NUM738^1`.<br>**TEQ** – **T**HF with **EQ**uality, e.g., `SET171^3`. |
| **TFA** | **T**yped (monomorphic) **F**irst-order form theorems with **A**rithmetic (axioms with a provable conjecture). | **TFI** – **TF**A with only **I**nteger arithmetic, e.g., `DAT016_1`.<br>**TFE** – **TF**A with only r**E**al arithmetic, e.g., `MSC022_2`. |
| **TFN** | **T**yped **F**irst-order form **N**on-theorems (axioms with a countersatisfiable conjecture, and satisfiable axiom sets) without arithmetic. | E.g., `COM002_20`. |
| **FOF** | **F**irst-**O**rder **F**orm theorems (axioms with a provable conjecture). | **FNE** – **F**OF with **N**o **E**quality, e.g., `COM003+1`.<br>**FEQ** – **F**OF with **EQ**uality, e.g., `SEU147+3`. |
| **FNT** | **F**OF **N**on-**T**heorems (axioms with a countersatisfiable conjecture, and satisfiable axioms without a conjecture). | **FNN** – **FNT** with **N**o equality, e.g., `KRS173+1`.<br>**FNQ** – **FNT** with e**Q**uality, e.g., `MGT033+2`. |
| **UEQ** | **U**nit **EQ**uality theorems in clause normal form (unsatisfiable clause sets). | E.g., `RNG026-7`. |
| **SLH** | Typed (monomorphic) higher-order theorems (axioms with a provable conjecture), generated by Isabelle's **SL**edge**H**ammer system [43]. | The problems were generated from sessions in Isabelle's Archive of Formal Proofs [19]. |

One ATP system runs on one CPU at a time. Systems can use all the cores on the CPU, which is advantageous in divisions where a wall clock time limit is used. One ATP system ran on one CPU at a time. (Each StarExec computer has two sockets, i.e., two CPUs, and 256 GiB memory. StarExec uses Linux's `sched_setaffinity` to restrict each system run to a single CPU, and `setrlimit` to limit memory use to 128 GiB.) Systems can use all the cores on the CPU, which is advantageous in divisions where a wall clock time limit is used. The StarExec computers used for CASC are the same as are publicly available to the TPTP community, which allows system developers to test and tune their systems in exactly the same environment as is used for the competition.

Demonstration division systems can run on the competition computers, or the computers can be supplied by the entrant. The CASC-29 demonstration division systems all used the competition computers.

## 3.2 Problems for the TPTP-based Divisions

The problems for the THF, TFA, TFN, FOF, FNT, and UEQ divisions were taken from the Thousands of Problems for Theorem Provers (TPTP) problem library [103], v8.2.0. The TPTP version used for CASC is released only after the competition has started, so that new problems

| ATP System | Divisions | | | | | | | Entrant (Associates) | Entrant's Affiliation |
|---|---|---|---|---|---|---|---|---|---|
| Beagle 0.9.47 | | | *TFN* | | | | | CASC | CASC-J8 winner |
| CSE 1.6 | | | | FOF | | | | Feng Cao (Yang Xu, Peiyao Liu, Jun Liu, Shuwei Chen, Guoyan Zeng, Jian Zhong, Guanfeng Wu, Xingxing He, Peng Xu) | JiangXi University of Science and Technology |
| CSE_E 1.5 | | | | FOF | | | | Peiyao Liu (Yang Xu, Feng Cao, Stephan Schulz, Jun Liu, Shuwei Chen, Guoyan Zeng, Jian Zhong, Guanfeng Wu, Xingxing He, Peng Xu) | Southwest Jiaotong University |
| cvc5 1.0 | | *TFA* | | | | | | CASC | CASC-J11 winner |
| cvc5 1.0.5 | THF | TFA | TFN | FOF | FNT | | SLH | Andrew Reynolds (Haniel Barbosa, Cesare Tinelli, Clark Barrett) | University of Iowa |
| Drodi 3.5.1 | | | | FOF | | UEQ | | Oscar Contreras | Amateur Programmer |
| Duper 1.0 | THF | | | FOF | | | SLH | Joshua Clune (Alexander Bentkamp, Yicheng Qian) | Carnegie Mellon University |
| E 3.0 | | | | | | | *SLH* | CASC | CASC-J11 winner |
| E 3.1 | THF | | TFN | FOF | FNT | UEQ | SLH | Stephan Schulz | DHBW Stuttgart |
| GKC 0.8 | | | | FOF | | UEQ | | Tanel Tammet | Tallinn University of Technology |
| iProver 3.8 | | TFA | TFN | FOF | FNT | UEQ | | Konstantin Korovin (André Duarte, Edvard Holden) | University of Manchester |
| Lash 1.13 | THF | | | | | | SLH | Cezary Kaliszyk (Chad Brown, Jan Jakubuv) | University of Innsbruck |

Table 2: The ATP systems and entrants

| ATP System | Divisions | | | | | | | Entrant (Associates) | Entrant's Affiliation |
|---|---|---|---|---|---|---|---|---|---|
| LEO-II 1.7.0 | THF | | | | | | | Alexander Steen (Christoph Benzmüller) | University of Greifswald |
| Leo-III 1.7.8 | THF | | | | | | SLH | Alexander Steen (Christoph Benzmüller) | University of Greifswald |
| Princess 230619 | | TFA | | FOF | | | | Philipp Rümmer | University of Regensburg |
| Prover9 1109a | | | | *FOF* | | | | CASC (Bob Veroff, Bill McCune) | CASC fixed point |
| Satallax 3.4 | THF | | | | | | SLH | Cezary Kaliszyk (Chad Brown, Michael Färber) | University of Innsbruck |
| Toma 0.4 | | | | | | UEQ | | Teppei Saito (Nao Hirokawa) | Japan Advanced Institute of Science and Technology |
| Twee 2.4.1 | | | | | | *UEQ* | | CASC | CASC-J11 winner |
| Twee 2.4.2 | | | | | | UEQ | | Nick Smallbone | Chalmers University of Technology |
| Vampire 4.7 | | | | *FOF* | *FNT* | | | CASC | CASC-J11 winner |
| Vampire 4.8 | THF | TFA | TFN | FOF | FNT | UEQ | SLH | Michael Rawson (Ahmed Bhayat, Márton Hajdú, Petra Hozzová, Laura Kovács, Jakob Rath, Giles Reger, Martin Reiner, Martin Suda, Johannes Schoisswohl, Andrei Voronkov) | TU Wien |
| Zipperp'n 2.1.999 | *THF* | | | | | | | CASC | CASC-J11 winner |
| Zipperp'n 2.1.9999 | THF | | | FOF | | | *SLH* | Jasmin Blanchette (Alexander Bentkamp, Simon Cruanes, Petar Vukmirović) | Vrije Universiteit Amsterdam |

Table 3: The ATP systems and entrants, continued

in the release have not been seen by the entrants. The problems have to meet certain criteria to be eligible for use:

- The TPTP tags problems that are designed specifically to be suited or ill-suited to some ATP system, calculus, or control strategy as *biased* They are excluded from the competition.
- The problems must be syntactically non-propositional.
- The TPTP uses system performance data in the Thousands of Solutions from Theorem Provers (TSTP) solution library to compute problem difficulty ratings in the range 0.00 (easy) to 1.00 (unsolved) [125]. Problems with ratings in the range 0.21 to 0.99 are eligible. Problems of lesser and greater ratings are made eligible if there are not enough problems with ratings in that range.

  Systems can be submitted before the competition so that their performance data is used in computing the problem ratings – problems that are solved get a rating less than 1.00 and are thus eligible (unless the rating drops below 0.21). The rating system also uses performance data from ATP systems that are not entered into the competition, which can produce ratings that make some problems eligible for selection but easy or unsolvable by the systems in the competition. Using problems that are solved by all or none of the competition systems does not affect the competition rankings, has the benefit of placing the systems' performances in the context of the state-of-the-art in ATP, but does reduce the differentiation between the systems in the competition.

In order to ensure that no system receives an advantage or disadvantage due to the specific presentation of the problems in the TPTP, the problems are preprocessed to

- strip out all comment lines (in particular, the problem header)
- randomly reorder the formulae (`include` directives are left before the formulae, and type declarations are left before the symbols' uses)
- randomly swap the arguments of associative connectives and randomly reverse implications
- randomly reverse equalities

The numbers of problems used in each division and problem category are constrained by the numbers of eligible problems, the number of systems entered across the divisions, the number of CPUs available, the time limits, and the time available for running the competition live in one conference day, i.e., in about 6 hours. The numbers of problems used are set within these constraints, according to the judgement of the organizers. The problems used are randomly selected from the eligible problems based on a seed supplied by the competition panel:

- The selection is constrained so that no division or category contains an excessive number of very similar problems, according to the "very similar problems" (VSP) lists distributed with the TPTP [69]: For each problem category in each division, if the category is going to use $N$ problems and there are $L$ VSP lists that have an intersection of at least $N/(L+1)$ with the eligible problems for the category, then maximally $N/(L+1)$ problems are taken from each of those VSP lists.
- In order to combat excessive tuning towards problems that are already in the preceding TPTP version, the selection is biased to select problems that are new in the TPTP version used, until 50% of the problems in each problem category have been selected or there are no more new problems to select, after which random selection from old and new problems continues. The number of new problems used depends on how many new problems are eligible and the limitation on very similar problems.

The problems are given to the ATP systems as files in TPTP format, with `include` directives, in increasing order of TPTP difficulty rating.

### 3.3 Problems for the SLH Divisions

The problems for the SLH division are generated by Isabelle's Sledgehammer system. 200 problems were generated from each of 42 new sessions in Isabelle's Archive of Formal Proofs (AFP) [19], providing 8400 problems that could be used. 1000 appropriately difficult problems were chosen based on performance data similar to that in the TSTP. The formulae are not modified by any preprocessing, thus allowing the ATP systems to take advantage of natural structure that occurs in the problems. While it was hoped that all would be theorems, it was possible that some are countersatisfiable for reasons including unlucky fact selection, unlucky monomorphization, and dependence on clever induction that only a human could come up with.

The number of problems was based on the CPU time limit, using a calculation similar to that used for the TPTP-based divisions. The problems are given in a roughly estimated increasing order of difficulty.

### 3.4 Time Limits

In the TPTP-based divisions a time limit is imposed for each problem. The minimal time limit for each problem is 120s. The time limit for each problem is constrained by the same factors that constrain the numbers of problems that are used. The time limit is set within the constraints, according to the judgement of the organizers, and is announced at the competition. In CASC-29 a wall clock time limit was imposed for each problem, and no CPU time limits were imposed (so that it could be advantageous to use all the cores on the CPU). StarExec copies the systems and problems to a compute node before starting execution, so that the system does not experience any network delay loading the problem.

In the SLH division a CPU time limit is imposed for each problem. The minimal time limit per problem is 15s, and the maximal time limit per problem is 90s, which is the range of CPU time that can be usefully allocated for a proof attempt in the Sledgehammer context.[1] The time limit is chosen as a reasonable value within the range allowed, and is announced at the competition.

## 4 System Entry, Delivery, and Execution

Systems can be entered at only the division level, and can be entered into more than one division. A system that is not entered into a division is assumed to perform worse than the entered systems, for that type of problem – wimping out is not an option. Entering many similar versions of the same system is deprecated, and entrants may be required to limit the number of system versions that they enter. Systems that rely essentially on running other ATP systems without adding value are deprecated; the competition panel may disallow or move such systems to the demonstration division.

The ATP systems entered into CASC are delivered to the competition organizer as StarExec installation packages, which the organizer installs and tests on StarExec. Source code is delivered separately, under the trusting assumption that the installation package does correspond to the source code. After the competition all competition division systems' StarExec and source code packages are made available on the CASC web site. This allows anyone to use the systems

[1] According to Jasmin Blanchette, and he should know.

on StarExec, and to examine the source code. An open source license is encouraged, to allow the systems to be freely used, modified, and shared. Many of the StarExec packages include statically linked binaries that provide further portability and longevity of the systems.

For systems running on entrant supplied computers in the demonstration division, entrants must email a `.tgz` file containing the source code and any files required for building the executable system to the competition organizers by the system delivery deadline. In the demonstration division the entrant specifies whether or not the source code is placed on the CASC web site.

The ATP systems must be fully automatic. They are executed as black boxes, on one problem at a time. Any command line parameters have to be the same for all problems/batches in each division. The ATP systems must be sound, and are tested for soundness by submitting non-theorems to the systems in the THF, TFA, FOF, UEQ, and SLH divisions, and theorems to the systems in the TFN and FNT divisions. Claiming to have found a proof of a non-theorem or a disproof of a theorem indicates unsoundness. If a system fails the soundness testing it must be repaired by the unsoundness repair deadline or be withdrawn.

# 5 System Evaluation

CASC ranks the ATP systems at only the division level. For each ATP system, for each problem, four items of data are recorded: whether or not the problem was solved, the CPU and wall clock times taken (as measured by StarExec's `runsolver` utility [53], and prepended to each line of the system's `stdout`), and whether or not a solution (proof or model) was output. The systems are ranked according to the number of problems solved with an acceptable proof/model output. Ties are broken according to the average time taken over problems solved. Trophies are awarded to the competition divisions' winners.

The competition panel decides whether or not the systems' solutions are "acceptable". The criteria include:

- Derivations must be complete, starting at formulae from the problem, and ending at the conjecture (for axiomatic proofs) or a $false$ formula (for proofs by contradiction, e.g., CNF refutations).
- For solutions that use translations from one form to another, e.g., translation of FOF problems to CNF, the translations must be adequately documented.
- Derivations must show only relevant inference steps.
- Inference steps must document the parent formulae, the inference rule used, and the inferred formula.
- Inference steps must be reasonably fine-grained.
- An unsatisfiable set of ground instances of clauses is acceptable for establishing the unsatisfiability of a set of clauses.
- Models must be complete, documenting the domain, function maps, and predicate maps. The domain, function maps, and predicate maps may be specified by explicit ground lists (of mappings), or by any clear, terminating algorithm.

In addition to the ranking data, other measures are made and presented in the results:

- The *state-of-the-art contribution* (SotAC) quantifies the unique abilities of each system. For each problem solved by a system, its SotAC for the problem is the fraction of systems that do not solve the problem. A system's overall SotAC is the average SotAC over the problems it solves but that are not solved by all the systems.

- The *efficiency* measure combines the number of problems solved with the time taken. It is the average solution rate over the problems solved (the solution rate for one problem is the reciprocal of the time taken to solve it), multiplied by the fraction of problems solved. Efficiency is computed for both CPU time and wall clock time, to measure how efficiently the systems use one core and multiple cores respectively.
- The *core usage* measures the extent to which the systems take advantage of multiple cores. It is the average of the ratios of CPU time used to wall clock time used, over the problems solved. The competition ran on octa-core computers, thus the maximal core usage was 8.0.

At some time after the competition all high ranking systems in each division are tested over the entire TPTP. This provides a final check for soundness. If a system is found to be unsound during or after the competition, but before the competition report is published, and it cannot be shown that the unsoundness did not manifest itself in the competition, then the system is retrospectively disqualified. At some time after the competition, the solutions from the winners are checked by the panel. If any of the solutions are unacceptable, i.e., they are sufficiently worse than the samples provided, then that system is retrospectively disqualified. All disqualifications are explained in the competition report.

The demonstration division results are presented along with the competition divisions' results, but might not be comparable with those results. The demonstration division is not ranked.

# 6 System Registration

ATP systems must be registered for the competition using the CASC system registration form, by the registration deadline. For each system an entrant must be nominated to handle all issues (e.g., installation and execution difficulties) arising before, during, and after the competition. The nominated entrant must formally register for CASC. It is not necessary for entrants to physically attend the competition.

## 6.1 System Descriptions

A system description has to be provided for each ATP system, using the HTML schema supplied on the CASC web site. The schema has the following sections:

- Architecture. This section introduces the ATP system, and describes the calculus and inference rules used.
- Strategies. This section describes the search strategies used, why they are effective, and how they are selected for given problems. Any strategy tuning that is based on specific problems' characteristics must be clearly described (and justified in light of the tuning restrictions described in Section 7).
- Implementation. This section describes the implementation of the ATP system, including the programming language used, important internal data structures, and any special code libraries used. The availability of the system is also given here.
- Expected competition performance. This section makes some predictions about the performance of the ATP system for each of the divisions and categories in which it is competing.
- References.

The system description has to be emailed to the competition organizers by the system description deadline. The system descriptions form part of the competition proceedings (Section 8).

### 6.2 Sample Solutions

For systems in the divisions that require solution output, representative sample solutions must be emailed to the competition organizers by the sample solutions deadline. Use of the TPTP format for proofs and finite interpretations is encouraged. The competition panel decides whether or not each system's solutions are acceptable (see Section 5).

Proof/model samples are required as follows:

- THF: `SET014^4`
- TFA: `DAT013=1`
- TFN: `HWV042_1` and `HWV042_3` (but as these problems from TPTP v8.1.0 are rather difficult, some systems were given two easier problems from TPTP v8.2.0: `COM001_10` and `DAT335_2`.
- FOF: `SEU140+2`
- FNT: `NLP042+1` and `SWV017+1`
- UEQ: `BOO001-1`

An explanation must be provided for any non-obvious features.

## 7 System Requirements

Entrants must ensure that their systems execute in the competition environment, and have the following properties. Entrants are advised to finalize their installation packages and check these properties well in advance of the system delivery deadline. This gives the competition organizers time to help resolve any difficulties encountered.

**Execution, Soundness, and Completeness**

- Systems must be fully automatic, i.e., all command line switches have to be the same for all problems in each division.
- Systems' performances must be reproducible by running the system again.
- Systems must be sound.
- Systems do not have to be complete in any sense, including calculus, search control, implementation, or resource requirements.
- All techniques used must be general purpose, and expected to extend usefully to new unseen problems. The precomputation and storage of information about individual problems that might appear in the competition, or their solutions, is not allowed. Strategies and strategy selection based on individual problems or their solutions are not allowed. If machine learning procedures are used to tune a system, the learning must ensure that sufficient generalization is obtained so that no there is no specialization to individual problems. The system description must explain any such tuning or training that has been done. The competition panel may disqualify any system that is deemed to be problem specific rather than general purpose.

### Output

- All solution output must be to `stdout`.
- For each problem, the system must output a distinguished string indicating what solution has been found or that no conclusion has been reached. Systems must use the SZS ontology and standards [83] for this. For example

  ```
  SZS status Theorem for SYN075+1
  ```

  or

  ```
  SZS status GaveUp for SYN075+1
  ```

- When outputting a solution, the start and end of the solution must be delimited by distinguished strings. Systems must use the SZS ontology and standards for this. For example

  ```
  SZS output start CNFRefutation for SYN075-1.p
    ...
  SZS output end CNFRefutation for SYN075-1.p
  ```

  The string specifying the problem status must be output before the start of a solution. Use of the TPTP format for proofs and finite interpretations [116] is encouraged.
- Solutions may not have irrelevant output (e.g., from other threads running in parallel) interleaved in the solution.

### Resource Usage

- Systems must be interruptible by a `SIGXCPU` signal so that CPU time limits can be imposed, and interruptible by a `SIGALRM` signal so that wall clock time limits can be imposed. For systems that create multiple processes the signal is sent first to the process at the top of the hierarchy, then one second later to all processes in the hierarchy. The default action on receiving these signals is to exit (thus complying with the time limit, as required), but systems may catch the signals and exit of their own accord. If a system runs past a time limit this is noticed in the timing data, and the system is considered to have not solved the problem.
- If a system terminates of its own accord it may not leave any temporary or intermediate output files. If a system is terminated by a `SIGXCPU` or `SIGALRM` it may not leave any temporary or intermediate files anywhere other than in `/tmp`.
- For practical reasons excessive output from an ATP system is not allowed. A limit, dependent on the disk space available, is imposed on the amount of output that can be produced.

# 8 The ATP Systems

These system descriptions were written by the entrants.

## 8.1 Beagle 0.9.47

Peter Baumgartner
Data61, Australia

### Architecture

Beagle [8] is an automated theorem prover for sorted first-order logic with equality over built-in theories. The theories currently supported are integer arithmetic, linear rational arithmetic and linear real arithmetic. It accepts formulas in the FOF and TFF formats of the TPTP syntax, and formulas in the SMT-LIB version 2 format.

Beagle first converts the input formulas into clause normal form. Pure arithmetic (sub-)formulas are treated by eager application of quantifier elimination. The core reasoning component implements the Hierarchic Superposition Calculus with Weak Abstraction (HSPWA) [9]. Extensions are a splitting rule for clauses that can be divided into variable disjoint parts, and a partial instantiation rule for variables with finite domain, and two kinds of background-sorted variables trading off completeness vs. search space.

The HSPWA calculus generalizes the superposition calculus by integrating theory reasoning in a black-box style. For the theories mentioned above, Beagle combines quantifier elimination procedures and other solvers to dispatch proof obligations over these theories. The default solvers are an improved version of Cooper's algorithm for linear integer arithmetic, and the CVC4 SMT solver for linear real/rational arithmetic. Non-linear integer arithmetic is treated by partial instantiation and additional lemmas.

### Strategies

Beagles uses the Discount loop for saturating a clause set under the calculus' inference rules. Simplification techniques include standard ones, such as subsumption deletion, demodulation by ordered unit equations, and tautology deletion. It also includes theory specific simplification rules for evaluating ground (sub)terms, and for exploiting cancellation laws and properties of neutral elements, among others. In the competition an aggressive form of arithmetic simplification is used, which seems to perform best in practice.

Beagle uses strategy scheduling by trying (at most) three flag settings sequentially.

### Implementation

Beagle is implemented in Scala. It is a full implementation of the HSPWA calculus. It uses a simple form of indexing, essentially top-symbol hashes, stored with each term and computed in a lazy way. Fairness is achieved through a combination of measuring clause weights and their derivation-age. It can be fine-tuned with a weight-age ratio parameter, as in other provers. Beagle's web site is

`https://bitbucket.org/peba123/beagle`

### Expected Competition Performance

Beagle 0.9.47 is the CASC-J8 TFN winner.

## 8.2 CSE 1.6

Feng Cao
JiangXi University of Science and Technology, China

### Architecture

CSE 1.6 is a developed prover based on the last version - CSE 1.5. It is an automated theorem prover for first-order logic without equality, based mainly on a novel inference mechanism called Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction (S-CS) [141]. S-CS is able to handle multiple (two or more) clauses dynamically in a synergized way in one deduction step, while binary resolution is a special case. CSE 1.6 also adopts conventional factoring, equality resolution (ER rule), and variable renaming. Some pre-processing techniques, including pure literal deletion and simplification based on the distance to the goal clause, and a number of standard redundancy criteria for pruning the search space: tautology deletion, subsumption (forward and backward), are applied as well. CSE 1.6 has been improved compared with CSE 1.5, mainly from the following aspects:

1. A multi-clause dynamic deduction algorithm based on full use of clauses is added by a different type of clause adequacy evaluation method.
2. A multi-clause synergized deduction algorithm with full use of synergized clauses is added, which can make the substitution of candidate literals involved in deduction from simple to complex.

Internally CSE 1.6 works only with clausal normal form. The E prover [60] is adopted with thanks for clausification of full first-order logic problems during preprocessing.

### Strategies

CSE 1.6 inherited most of the strategies in CSE 1.5. The main new strategies are:

- Clause Activity and Complexity Strategy. A measure and calculation method of clause activity and complexity is added by analyzing the properties of the variable terms and the structure of function terms of clauses, which can evaluate clauses with different term structures.
- Literal Matching Degree Strategy. A literal measurement and calculation method based on term matching degree is added, which can evaluate the complexity of candidate literals with different term structures participating in multi-clause deduction.

### Implementation

CSE 1.6 is implemented mainly in C++, and Java is used for batch problem running implementation. A shared data structure is used for constants and shared variables storage. In addition, special data structure is designed for property description of clause, literal and term,

so that it can support the multiple strategy mode. E prover is used for clausification of FOF problems, and then TPTP4X is applied to convert the CNF format into TPTP format.

### Expected Competition Performance

CSE 1.6 has made some improvements compared to CSE 1.5, and so we expect a better performance in this year's competition.

**Acknowledgement:** Development of CSE 1.6 has been partially supported by the General Research Project of Jiangxi Education Department (Grant No. GJJ200818).

## 8.3 CSE_E 1.5

Peiyao Liu
Southwest Jiaotong University, China

### Architecture

CSE_E 1.5 is an automated theorem prover for first-order logic by combining CSE 1.6 and E 3.0, where CSE 1.6 is based on the Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction (S-CS) [141] and E is mainly based on superposition. The combination mechanism is like this: E and CSE are applied to the given problem sequentially. If either prover solves the problem, then the proof process completes. If neither CSE nor E can solve the problem, some inferred clauses with no more than two literals, especially unit clauses, by CSE will be fed to E as lemmas, along with the original clauses, for further proof search. This kind of combination is expected to take advantage of both CSE and E, and produce a better performance. Concretely, CSE is able to generate a good number of unit clauses, based on the fact that unit clauses are helpful for proof search and equality handling. On the other hand, E has a good ability on equality handling.

### Strategies

The CSE part of CSE_E 1.5 takes almost the same strategies as in that in CSE 1.6 standalone, e.g., clause/literal selection, strategy selection, and CSC strategy. The only difference is that equality handling strategies of CSE part of the combined system are blocked. The main new strategies for the combined systems are:

- Lemma filtering mainly based on deduction weight of binary clauses.
- Fine-grained dynamic time allocation scheme in different run stages.

### Implementation

CSE_E 1.5 is implemented mainly in C++, and JAVA is used for batch problem running implementation. The job dispatch between CSE and E is implemented in C++.

### Expected Competition Performance

We expect CSE_E 1.5 to solve some hard problems that E cannot solve and have a satisfying performance.

**Acknowledgement:** Development of CSE_E 1.5 has been partially supported by the National Natural Science Foundation of China (NSFC) (Grant No. 61976130). Stephan Schulz for his kind permission on using his E prover that makes CSE_E possible.

## 8.4 cvc5 1.0

Andrew Reynolds
University of Iowa, USA

### Architecture

cvc5 is the successor of CVC4 [7]. It is an SMT solver based on the CDCL(T) architecture [41] that includes built-in support for many theories, including linear arithmetic, arrays, bit vectors, datatypes, finite sets and strings. It incorporates approaches for handling universally quantified formulas. For problems involving free function and predicate symbols, cvc5 primarily uses heuristic approaches based on conflict-based instantiation and E-matching for theorems, and finite model finding approaches for non-theorems. Like other SMT solvers, cvc5 treats quantified formulas using a two-tiered approach. First, quantified formulas are replaced by fresh Boolean predicates and the ground theory solver(s) are used in conjunction with the underlying SAT solver to determine satisfiability. If the problem is unsatisfiable at the ground level, then the solver answers "unsatisfiable". Otherwise, the quantifier instantiation module is invoked, and will either add instances of quantified formulas to the problem, answer "satisfiable", or return unknown. Finite model finding in cvc5 targets problems containing background theories whose quantification is limited to finite and uninterpreted sorts. In finite model finding mode, cvc5 uses a ground theory of finite cardinality constraints that minimizes the number of ground equivalence classes, as described in [51]. When the problem is satisfiable at the ground level, a candidate model is constructed that contains complete interpretations for all predicate and function symbols. It then adds instances of quantified formulas that are in conflict with the candidate model, as described in [51]. If no instances are added, it reports "satisfiable".

cvc5 has native support for problems in higher-order logic, as described in [6]. It uses a pragmatic approach for HOL, where lambdas are eliminated eagerly via lambda lifting. The approach extends the theory solver for quantifier-free uninterpreted functions (UF) and E-matching. For the former, the theory solver for UF in cvc5 now handles equalities between functions using an extensionality inference. Partial applications of functions are handle using a (lazy) applicative encoding where some function applications are equated to the applicative encoding. For the latter, several of the data structures for E-matching have been modified to incorporate matching in the presence of equalities between functions, function variables, and partial function applications.

### Strategies

For handling theorems, cvc5 primarily uses conflict-based quantifier instantiation [50, 5], numerative instantiation [49] and E-matching. vc5 uses a handful of orthogonal trigger se-

lection strategies for E-matching, and several orthogonal ordering heuristics for enumerative instantiation. For handling non-theorems, cvc5 primarily uses finite model finding techniques. Since cvc5 with finite model finding is also capable of establishing unsatisfiability, it is used as a strategy for theorems as well.

### Implementation

cvc5 is implemented in C++. The code is available from

`https://github.com/cvc5/cvc5`

### Expected Competition Performance

cvc5 1.0 is the CASC-J11 TFA winner.

## 8.5 cvc5 1.0.5

Andrew Reynolds
University of Iowa, USA

### Architecture

cvc5 [4] is the successor of CVC4 [7]. It is an SMT solver based on the CDCL(T) architecture [41] that includes built-in support for many theories, including linear arithmetic, arrays, bit vectors, datatypes, finite sets and strings. It incorporates approaches for handling universally quantified formulas. For problems involving free function and predicate symbols, cvc5 primarily uses heuristic approaches based on conflict-based instantiation and E-matching for theorems, and finite model finding approaches for non-theorems.

Like other SMT solvers, cvc5 treats quantified formulas using a two-tiered approach. First, quantified formulas are replaced by fresh Boolean predicates and the ground theory solver(s) are used in conjunction with the underlying SAT solver to determine satisfiability. If the problem is unsatisfiable at the ground level, then the solver answers "unsatisfiable". Otherwise, the quantifier instantiation module is invoked, and will either add instances of quantified formulas to the problem, answer "satisfiable", or return unknown. Finite model finding in cvc5 targets problems containing background theories whose quantification is limited to finite and uninterpreted sorts. In finite model finding mode, cvc5 uses a ground theory of finite cardinality constraints that minimizes the number of ground equivalence classes, as described in [51]. When the problem is satisfiable at the ground level, a candidate model is constructed that contains complete interpretations for all predicate and function symbols. It then adds instances of quantified formulas that are in conflict with the candidate model, as described in [51]. If no instances are added, it reports "satisfiable".

cvc5 has native support for problems in higher-order logic, as described in [6]. It uses a pragmatic approach for HOL, where lambdas are eliminated eagerly via lambda lifting. The approach extends the theory solver for quantifier-free uninterpreted functions (UF) and E-matching. For the former, the theory solver for UF in cvc5 now handles equalities between functions using an extensionality inference. Partial applications of functions are handle using a (lazy) applicative encoding where some function applications are equated to the applicative encoding. For the latter, several of the data structures for E-matching have been modified to

incorporate matching in the presence of equalities between functions, function variables, and partial function applications.

#### Strategies

For handling theorems, cvc5 primarily uses conflict-based quantifier instantiation [50, 5], enumerative instantiation [49], and E-matching. cvc5 uses a handful of orthogonal trigger selection strategies for E-matching, and several orthogonal ordering heuristics for enumerative instantiation. For handling non-theorems, cvc5 primarily uses finite model finding techniques. Since cvc5 with finite model finding is also capable of establishing unsatisfiability, it is used as a strategy for theorems as well.

#### Implementation

cvc5 is implemented in C++. The code is available from

`https://github.com/cvc5/cvc5`

#### Expected Competition Performance

The first-order theorem proving, finite model finding and proof generation [4], in cvc5 have undergone minor improvements in the past year. Additionally, this year cvc5 will be using a translator from TPTP to smt2 as a preprocess step and new faster parser for smt2. Overall, cvc5 will perform similar to previous years.

### 8.6 Drodi 3.5.1

Oscar Contreras
Amateur Programmer, Spain

#### Architecture

Drodi 3.5.1 is a very basic and lightweight automathed theorem prover. It implements ordered resolution and equality paramodulation inferences as well as demodulation and some other standard simplifications. It also includes its own basic implementations of clausal normal form conversion [42], AVATAR architecture with a SAT solver [133], Limited Resource Strategy [52], discrimination trees as well as KBO, non recursive and lexicographic reduction orderings. Drodi produces a (hopefully) verifiable proof in TPTP format.

The following features have been added since last CASC competition:

- Improved literal selection including lookahead as in [30].
- SinE distance for clauses and symbols as in [65].
- Layered clause selection as in [28].
- Stochastic strategy inspired in [66].

### Strategies

Drodi 3.5.1 has a fair number of selectable strategies including but not limited to the following:

- Otter, Discount and Limited Resource Strategy [52] saturation algorithms.
- A basic implementation of AVATAR architecture [133].
- Several literal and term reduction orderings.
- Several literal selection options [30].
- Several layered clause selection heuristics with adjustable selection ratios [28].
- Classical clause relevancy pruning.
- Some strategies are run a second time with a previously applied randomization to the problem [66].
- Drodi can generate a learning file from successful proofs and use the file to guide clause selection strategy. It is based in the enhanced ENIGMA method. However, unlike ENIGMA, the learning file is completely general and can be used with any kind of problems. This generality allows the use of the same learning file in both FOF and UEQ CASC competition divisions. The learning file is generated over a set of TPTP problems before the CASC competition using built-in Drodi functions that include a L2 Support Vector Machine. Drodi integrated learning functions are a generalization of ENIGMA [32, 33]. Literals polarity, equality, skolem and variable occurrences are stored in clause feature vectors. Unlike ENIGMA, instead of storing the specific functions and predicates themselves only the SinE distance and arity of functions and non equality predicates are stored in clause feature vectors with different features assigned to predicates and functions.

### Implementation

Drodi 3.5.1 is implemented in C. It includes discrimination trees and hashing indexing. All the code is original, without special code libraries or code taken from other sources.

### Expected Competition Performance

Several new features have been added to Drodi 3.5.1 and it is expected a significant improvement in FOF with respect to last year CASC and a more modest improvement in UEQ.

## 8.7 Duper 1.0

Joshua Clune
Carnegie Mellon University, USA

### Architecture

Duper is a superposition-based theorem prover for dependent type theory, designed to prove theorems in the proof assistant Lean 4. To solve problems in TPTP format, Duper first translates them into Lean goals, using a shallow embedding of first-order and higher-order logic into dependent type theory. Translation is currently possible for FOF, TFF, and THF problems without arithmetic. When run in Lean, Duper produces proof terms which get verified by Lean's kernel. When run as a standalone executable for the purposes of this competition, Duper still produces Lean proof terms but they are not checked by Lean's kernel.

Duper's core architecture is based on saturation with a set of inference and simplification rules that operate on dependently-typed terms but primarily perform first-order and some higher-order reasoning. The calculus is closely based on Zipperposition's [10] though it has been adapted to be sound in Lean's type theory. The unification procedure extends [135] to dependent type theory, and remains complete for the higher order fragment of Lean. Duper uses streams to represent conclusions of inference rules and interleaves between unification and inference, as in [10]. ¡/p¿

### Strategies

Currently, Duper has only a single strategy without time slicing and does not take advantage of multiple cores. The strategy is simple and does not tune its heuristics based on the input problem. Duper uses a KBO term ordering with a basic precedence and weight heuristic.

### Implementation

The prover is implemented in Lean 4 as a Lean tactic. The script provided to solve TPTP problems parses the given file, converts it to a Lean 4 goal, then attempts to solve said goal with the Duper tactic. Duper's source code can be found at

`https://github.com/leanprover-community/duper`

### Expected Competition Performance

Duper is still in an early stage of development and is not expected to compete with matured tools.

## 8.8 E 3.0

Stephan Schulz
DHBW Stuttgart, Germany

### Architecture

E [56, 60, 61] is a purely equational theorem prover for many-sorted first-order logic with equality, and for monomorphic higher-order logic. It consists of an (optional) clausifier for preprocessing full first-order formulae into clausal form, and a saturation algorithm implementing an instance of the superposition calculus with negative literal selection and a number of redundancy elimination techniques, optionally with higher-order extensions [134]. E is based on the DISCOUNT-loop variant of the given-clause algorithm, i.e., a strict separation of active and passive facts. No special rules for non-equational literals have been implemented. Resolution is effectively simulated by paramodulation and equality resolution. As of E 2.1, PicoSAT [18] can be used to periodically check the (on-the-fly grounded) proof state for propositional unsatisfiability. For the LTB divisions, a control program uses a SInE-like analysis to extract reduced axiomatizations that are handed to several instances of E. E will not use on-the-fly learning this year.

### Strategies

Proof search in E is primarily controlled by a literal selection strategy, a clause selection heuristic, and a simplification ordering. The prover supports a large number of pre-programmed literal selection strategies. Clause selection heuristics can be constructed on the fly by combining various parameterized primitive evaluation functions, or can be selected from a set of predefined heuristics. Clause evaluation heuristics are based on symbol-counting, but also take other clause properties into account. In particular, the search can prefer clauses from the set of support, or containing many symbols also present in the goal. Supported term orderings are several parameterized instances of Knuth-Bendix-Ordering (KBO) and Lexicographic Path Ordering (LPO), which can be lifted in different ways to literal orderings.

For CASC-J11, E implements a multi-core strategy-scheduling automatic mode. The total CPU time available is broken into several (unequal) time slices. For each time slice, the problem is classified into one of several classes, based on a number of simple features (number of clauses, maximal symbol arity, presence of equality, presence of non-unit and non-Horn clauses, possibly presence of certain axiom patterns...). For each class, a schedule of strategies is greedily constructed from experimental data as follows: The first strategy assigned to a schedule is the the one that solves the most problems from this class in the first time slice. Each subsequent strategy is selected based on the number of solutions on problems not already solved by a preceding strategy.

About 140 different strategies have been thoroughly evaluated on all untyped first-order problems from TPTP 7.3.0. We have also explored some parts of the heuristic parameter space with a short time limit of 5 seconds. This allowed us to test about 650 strategies on all TPTP problems, and an extra 7000 strategies on UEQ problems from TPTP 7.2.0. About 100 of these strategies are used in the automatic mode, and about 450 are used in at least one schedule.

### Implementation

E is build around perfectly shared terms, i.e. each distinct term is only represented once in a term bank. The whole set of terms thus consists of a number of interconnected directed acyclic graphs. Term memory is managed by a simple mark-and-sweep garbage collector. Unconditional (forward) rewriting using unit clauses is implemented using perfect discrimination trees with size and age constraints. Whenever a possible simplification is detected, it is added as a rewrite link in the term bank. As a result, not only terms, but also rewrite steps are shared. Subsumption and contextual literal cutting (also known as subsumption resolution) is supported using feature vector indexing [59]. Superposition and backward rewriting use fingerprint indexing [58], a new technique combining ideas from feature vector indexing and path indexing. Finally, LPO and KBO are implemented using the elegant and efficient algorithms developed by Bernd Löchner in [36, 37]. The prover and additional information are available at

`https://www.eprover.org`

### Expected Competition Performance

E 3.0 is the CASC-J11 SLH winner.

## 8.9 E 3.1

Stephan Schulz
DHBW Stuttgart, Germany

### Architecture

E [56, 60, 61] is a purely equational theorem prover for many-sorted first-order logic with equality, and for monomorphic higher-order logic. It consists of an (optional) clausifier for preprocessing full first-order formulae into clausal form, and a saturation algorithm implementing an instance of the superposition calculus with negative literal selection and a number of redundancy elimination techniques, optionally with higher-order extensions [134, 137]. E is based on the DISCOUNT-loop variant of the given-clause algorithm, i.e., a strict separation of active and passive facts. No special rules for non-equational literals have been implemented. Resolution is effectively simulated by paramodulation and equality resolution. As of E 2.1, PicoSAT [18] can be used to periodically check the (on-the-fly grounded) proof state for propositional unsatisfiability.

### Strategies

Proof search in E is primarily controlled by a literal selection strategy, a clause selection heuristic, and a simplification ordering. The prover supports a large number of pre-programmed literal selection strategies. Clause selection heuristics can be constructed on the fly by combining various parameterized primitive evaluation functions, or can be selected from a set of predefined heuristics. Clause evaluation heuristics are based on symbol-counting, but also take other clause properties into account. In particular, the search can prefer clauses from the set of support, or containing many symbols also present in the goal. Supported term orderings are several parameterized instances of Knuth-Bendix-Ordering (KBO) and Lexicographic Path Ordering (LPO), which can be lifted in different ways to literal orderings.

For CASC-29, E implements a two-stage multi-core strategy-scheduling automatic mode. The total CPU time available is broken into several (unequal) time slices. For each time slice, the problem is classified into one of several classes, based on a number of simple features (number of clauses, maximal symbol arity, presence of equality, presence of non-unit and non-Horn clauses, possibly presence of certain axiom patterns...). For each class, a schedule of strategies is greedily constructed from experimental data as follows: The first strategy assigned to a schedule is the the one that solves the most problems from this class in the first time slice. Each subsequent strategy is selected based on the number of solutions on problems not already solved by a preceding strategy. The strategies are then scheduled onto the available cores and run in parallel.

About 140 different strategies have been thoroughly evaluated on all untyped first-order problems from TPTP 7.3.0. We have also explored some parts of the heuristic parameter space with a short time limit of 5 seconds. This allowed us to test about 650 strategies on all TPTP problems, and an extra 7000 strategies on UEQ problems from TPTP 7.2.0. About 100 of these strategies are used in the automatic mode, and about 450 are used in at least one schedule.

### Implementation

E is build around perfectly shared terms, i.e. each distinct term is only represented once in a term bank. The whole set of terms thus consists of a number of interconnected directed acyclic graphs. Term memory is managed by a simple mark-and-sweep garbage collector. Unconditional (forward) rewriting using unit clauses is implemented using perfect discrimination trees with size and age constraints. Whenever a possible simplification is detected, it is added as a rewrite link in the term bank. As a result, not only terms, but also rewrite steps are shared. Subsumption and contextual literal cutting (also known as subsumption resolution) is supported using feature vector indexing [59]. Superposition and backward rewriting use fingerprint indexing [58], a new technique combining ideas from feature vector indexing and path indexing. Finally, LPO and KBO are implemented using the elegant and efficient algorithms developed by Bernd Löchner in [36, 37]. The prover and additional information are available at

`https://www.eprover.org`

### Expected Competition Performance

E 3.1 is basically E 3.0 made more robust and somewhat more efficient. We have not yet been able to evaluate and integrate new search strategies making full use of all new features. As a result, we expect performance to be similar to last year's version. The system is expected to perform well in most proof classes, but will at best complement top systems in the disproof classes.

## 8.10 GKC 0.8

Tanel Tammet
Tallinn University of Technology, Estonia

### Architecture

GKC [128] is a resolution prover optimized for search in large knowledge bases. The GKC version 0.8 running at CASC-29 is a marginally improved version of the GKC 0.7 running in two

previous CASCs. Almost all of the GKC development effort this year has gone to the common-sense superstructure GK (¡A HREF="https://logictools.org/gk/"¿https://logictools.org/gk/¡/A¿) and the natural language reasoning pipeline (¡A HREF="https://github.com/tammet/nlpsolver"¿https://github.co [131].

GKC is used as a foundation (GK Core) for building a common-sense reasoner GK. In particular, GK can handle inconsistencies and perform probabilistic and nonmonotonic reasoning [129, 130].

The WASM version of the previous GKC 0.6 is used as the prover engine in the educational ¡A HREF="http://logictools.org"¿http://logictools.org¡/A¿ system. It can read and output proofs in the TPTP, simplified TPTP and JSON format, the latter compatible with JSON-LD [132].

GKC only looks for proofs and does not try to show non-provability. These standard inference rules have been implemented in GKC:

- Binary resolution with optionally the set of support strategy, negative or positive ordered resolution or unit restriction.
- Hyperresolution.
- Factorization.
- Paramodulation and demodulation with the Knuth-Bendix ordering.

GKC includes an experimental implementation of propositional inferencing and instance generation, which we do not plan to use during the current CASC.

### Strategies

GKC uses multiple strategies run sequentially, with the time limit starting at 0.1 seconds for each, increased 10 or 5 times once the whole batch has been performed. The strategy selections takes into consideration the basic properties of the problem: the presence of equality and the approximate size of the problem.

We perform the selection of a given clause by using several queues in order to spread the selection relatively uniformly over these categories of derived clauses and their descendants: axioms, external axioms, assumptions and goals. The queues are organized in two layers. As a first layer we use the common ratio-based algorithm of alternating between selecting $n$ clauses from a weight-ordered queue and one clause from the FIFO queue with the derivation order. As a second layer we use four separate queues based on the derivation history of a clause. Each queue in the second layer contains the two sub-queues of the first layer.

### Implementation

GKC is implemented in C. The data representation machinery is built upon a shared memory graph database ¡A HREF="http://whitedb.org"¿Whitedb¡/A¿ enabling it to solve multiple different queries in parallel processeses without a need to repeatedly parse or load the large parsed knowledge base from the disk. An interesting aspect of GKC is the pervasive use of hash indexes, feature vectors and fingerprints, while no tree indexes are used.

GKC can be obtained from

```
https://github.com/tammet/gkc/
```

### Expected Competition Performance

We expect GKC to be in the middle of the final ranking for FOF and below the average in UEQ. We expect GKC to perform well on very large problems.

## 8.11 iProver 3.8

Konstantin Korovin
University of Manchester, United Kingdom

None provided

## 8.12 Lash 1.13

Cezary Kaliszyk
University of Innsbruck, Austria

### Architecture

Lash [21] is a higher-order automated theorem prover created as a fork of the theorem prover Satallax. The basic underlying calculus of Satallax is a ground tableau calculus whose rules only use shallow information about the terms and formulas taking part in the rule.

### Strategies

There are about 113 flags that control Lash's behavior, most of them inherited from Satallax. A mode is a collection of flag values. Starting from 10 Satallax modes, Grackle was used to derive 61 modes automatically, and grouped into two schedules of 15 modes each.

### Implementation

Lash uses new, efficient C representations of vital structures and operations. Most importantly, Lash uses a C representation of (normal) terms with perfect sharing along with a C implementation of normalizing substitutions. Lash's version 1.12 additionally includes a new term enumeration scheme, and Grackle-based strategy schedule.

### Expected Competition Performance

Comparable to Satallax.

## 8.13 LEO-II 1.7.0

Alexander Steen
University of Greifswald, Germany

### Architecture

LEO-II [13], the successor of LEO [12], is a higher-order ATP system based on extensional higher-order resolution. More precisely, LEO-II employs a refinement of extensional higher-order RUE resolution [11]. LEO-II is designed to cooperate with specialist systems for fragments of higher-order logic. By default, LEO-II cooperates with the first-order ATP system E [55]. LEO-II is often too weak to find a refutation amongst the steadily growing set of clauses on its own. However, some of the clauses in LEO-II's search space attain a special status: they are first-order clauses modulo the application of an appropriate transformation function. Therefore, LEO-II launches a cooperating first-order ATP system every n iterations of its (standard) resolution proof search loop (e.g., 10). If the first-order ATP system finds a refutation, it communicates its success to LEO-II in the standard SZS format. Communication between LEO-II and the cooperating first-order ATP system uses the TPTP language and standards.

### Strategies

LEO-II employs an adapted "Otter loop". Moreover, LEO-II uses some basic strategy scheduling to try different search strategies or flag settings. These search strategies also include some different relevance filters.

### Implementation

LEO-II is implemented in OCaml 4, and its problem representation language is the TPTP THF language [14]. In fact, the development of LEO-II has largely paralleled the development of the TPTP THF language and related infrastructure [113]. LEO-II's parser supports the TPTP THF0 language and also the TPTP languages FOF and CNF.

Unfortunately the LEO-II system still uses only a very simple sequential collaboration model with first-order ATPs instead of using the more advanced, concurrent and resource-adaptive OANTS architecture [15] as exploited by its predecessor LEO.

The LEO-II system is distributed under a BSD style license, and it is available from

`http://www.leoprover.org`

### Expected Competition Performance

LEO-II is not actively being developed anymore, hence there are no expected improvements to last year's CASC results.

## 8.14 Leo-III 1.7.8

Alexander Steen
University of Greifswald, Germany

### Architecture

Leo-III [63], the successor of LEO-II [13], is a higher-order ATP system based on extensional higher-order paramodulation with inference restrictions using a higher-order term ordering. The calculus contains dedicated extensionality rules and is augmented with equational simplification routines that have their intellectual roots in first-order superposition-based theorem proving. The saturation algorithm is a variant of the given clause loop procedure inspired by the first-order ATP system E.

Leo-III cooperates with external first-order ATPs that are called asynchronously during proof search; a focus is on cooperation with systems that support typed first-order (TFF) input. For this year's CASC, CVC4 [7] and E [56, 60] are used as external systems. However, cooperation is in general not limited to first-order systems. Further TPTP/TSTP-compliant external systems (such as higher-order ATPs or counter model generators) may be included using simple command-line arguments. If the saturation procedure loop (or one of the external provers) finds a proof, the system stops, generates the proof certificate and returns the result.

### Strategies

Leo-III comes with several configuration parameters that influence its proof search by applying different heuristics and/or restricting inferences. These parameters can be chosen manually by the user on start-up. Leo-III implements a very naive time slicing approach in which at most three different parameter configurations are used, one after each other. In practice, this hardly ever happens and Leo-III will just run with its default parameter setting.

### Implementation

¡style type="text/css"¿ .leopard font-variant: small-caps; ¡/style¿ Leo-III utilizes and instantiates the associated ¡span class="leopard"¿ LeoPARD¡/span¿ system platform [140] for higher-order (HO) deduction systems implemented in Scala (currently using Scala 2.13 and running on a JVM with Java 8). The prover makes use of ¡span class="leopard"¿LeoPARD¡/span¿'s data structures and implements its own reasoning logic on top. A hand-crafted parser is provided that supports all TPTP syntax dialects. It converts its produced concrete syntax tree to an internal TPTP AST data structure which is then transformed into polymorphically typed lambda terms. As of version 1.1, Leo-III supports all common TPTP dialects (CNF, FOF, TFF, THF) as well as their polymorphic variants [20, 34]. Since version 1.6.X (X ¿= 0) Leo-III also accepts non-classical problem input represented in non-classical TPTP, see ...

```
https://tptp.org/NonClassicalLogic
```

The term data structure of Leo-III uses a polymorphically typed spine term representation augmented with explicit substitutions and De Bruijn-indices. Furthermore, terms are perfectly shared during proof search, permitting constant-time equality checks between alpha-equivalent terms.

Leo-III's saturation procedure may at any point invoke external reasoning tools. To that end, Leo-III includes an encoding module which translates (polymorphic) higher-order clauses to polymorphic and monomorphic typed first-order clauses, whichever is supported by the external system. While LEO-II relied on cooperation with untyped first-order provers, Leo-III exploits the native type support in first-order provers (TFF logic) for removing clutter during translation and, in turn, higher effectivity of external cooperation.

Leo-III is available on GitHub:

```
https://github.com/leoprover/Leo-III
```

#### Expected Competition Performance

Version 1.7.8 is, for all intents and purposes of CASC, equivalent to the version from previous year except that support for reasoning in various quantified non-classical logics were added and/or improved. We do not expect Leo-III to be strongly competitive against more recent higher-order provers as Leo-III does not implement several standard features of effective systems (including time slicing and proper axiom selection). For the first time, a native LLVM build of Leo-III is used instead of the JVM-based set-up (at least for the SLH division).

### 8.15 Princess 230619

Philipp Rümmer¡br¿ University of Regensburg, Germany

#### Architecture

Princess [54] is a theorem prover for first-order logic modulo linear integer arithmetic. The prover uses a combination of techniques from the areas of first-order reasoning and SMT solving. The main underlying calculus is a free-variable tableau calculus, which is extended with constraints to enable backtracking-free proof expansion, and positive unit hyper-resolution for lightweight instantiation of quantified formulae. Linear integer arithmetic is handled using a set of built-in proof rules resembling the Omega test, which altogether yields a calculus that is complete for full Presburger arithmetic, for first-order logic, and for a number of further fragments. Princess also contains theory modules for, among others, non-linear arithmetic, rationals, bit-vectors, arrays, heaps, algebraic data-types, strings.

The list of contributors is available on

```
https://github.com/uuverifiers/princess/blob/master/AUTHORS
```

#### Strategies

For CASC, Princess will run a fixed schedule of configurations for each problem (portfolio method). Configurations determine, among others, the mode of proof expansion (depth-first, breadth-first), selection of triggers in quantified formulae, clausification, and the handling of functions. The portfolio was chosen based on training with a random sample of problems from the TPTP library.

The following options are used in the competition:

```
-portfolio=casc +threads +printProof -inputFormat=tptp
```

The version 230619 submitted to CASC-29 is the standard version of Princess. This is in contrast to CASC-26 (2017), when Princess was participating the last time, when a version tailor-made for CASC was used.

### Implementation

Princess is entirely written in Scala and runs on any recent Java virtual machine; besides the standard Scala and Java libraries, only the Cup parser library is used.

Sources and binaries are available from

`https://github.com/uuverifiers/princess`

### Expected Competition Performance

The system has not been competing at CASC in the years since 2017. Most of the implementation effort since then has focused on theory solvers, not on the core first-order proof procedure, so that the results should be similar as in 2017. Like in 2017, there is still limited support for proof generation in the context of quantifiers (free variables used in the constructed tableau), so that Princess will lose points here.

## 8.16 Prover9 1109a

Bob Veroff, of behalf of Bill McCune
University of New Mexico, USA

### Architecture

Prover9, Version 2009-11A, is a resolution/paramodulation prover for first-order logic with equality. Its overall architecture is very similar to that of Otter-3.3 [40]. It uses the "given clause algorithm", in which not-yet-given clauses are available for rewriting and for other inference operations (sometimes called the "Otter loop").

Prover9 has available positive ordered (and nonordered) resolution and paramodulation, negative ordered (and nonordered) resolution, factoring, positive and negative hyperresolution, UR-resolution, and demodulation (term rewriting). Terms can be ordered with LPO, RPO, or KBO. Selection of the "given clause" is by an age-weight ratio.

Proofs can be given at two levels of detail: (1) standard, in which each line of the proof is a stored clause with detailed justification, and (2) expanded, with a separate line for each operation. When FOF problems are input, proof of transformation to clauses is not given.

Completeness is not guaranteed, so termination does not indicate satisfiability.

### Strategies

Prover9 has available many strategies; the following statements apply to CASC.

Given a problem, Prover9 adjusts its inference rules and strategy according to syntactic properties of the input clauses such as the presence of equality and non-Horn clauses. Prover9 also does some preprocessing, for example, to eliminate predicates.

For CASC Prover9 uses KBO to order terms for demodulation and for the inference rules, with a simple rule for determining symbol precedence.

For the FOF problems, a preprocessing step attempts to reduce the problem to independent subproblems by a miniscope transformation; if the problem reduction succeeds, each subproblem is clausified and given to the ordinary search procedure; if the problem reduction fails, the original problem is clausified and given to the search procedure.

### Implementation

Prover9 is coded in C, and it uses the LADR libraries. Some of the code descended from EQP [39]. (LADR has some AC functions, but Prover9 does not use them). Term data structures are not shared (as they are in Otter). Term indexing is used extensively, with discrimination tree indexing for finding rewrite rules and subsuming units, FPA/Path indexing for finding subsumed units, rewritable terms, and resolvable literals. Feature vector indexing [57] is used for forward and backward nonunit subsumption. Prover9 is available from

`http://www.cs.unm.edu/~mccune/prover9/`

### Expected Competition Performance

Prover9 is the CASC fixed point, against which progress can be judged. Each year it is expected do worse than the previous year, relative to the other systems.

## 8.17 Satallax 3.4

Cezary Kaliszyk
Universität Innsbruck, Austria

### Architecture

Satallax 3.4 [22] is an automated theorem prover for higher-order logic. The particular form of higher-order logic supported by Satallax is Church's simple type theory with extensionality and choice operators. The SAT solver MiniSat [27] is responsible for much of the proof search. The theoretical basis of search is a complete ground tableau calculus for higher-order logic [24] with a choice operator [3]. Problems are given in the THF format.

Proof search: A branch is formed from the axioms of the problem and the negation of the conjecture (if any is given). From this point on, Satallax tries to determine unsatisfiability or satisfiability of this branch. Satallax progressively generates higher-order formulae and corresponding propositional clauses [23]. These formulae and propositional clauses correspond to instances of the tableau rules. Satallax uses the SAT solver MiniSat to test the current set of propositional clauses for unsatisfiability. If the clauses are unsatisfiable, then the original branch is unsatisfiable. Optionally, Satallax generates lambda-free higher-order logic (lfHOL) formulae in addition to the propositional clauses [136]. If this option is used, then Satallax periodically calls the theorem prover E [59] to test for lfHOL unsatisfiability. If the set of lfHOL formulae is unsatisfiable, then the original branch is unsatisfiable. Upon request, Satallax attempts to reconstruct a proof which can be output in the TSTP format.

### Strategies

There are about 150 flags that control the order in which formulae and instantiation terms are considered and propositional clauses are generated. Other flags activate some optional extensions to the basic proof procedure (such as whether or not to call the theorem prover E). A collection of flag settings is called a mode. Approximately 500 modes have been defined and tested so far. A strategy schedule is an ordered collection of modes with information about how much time the mode should be allotted. Satallax tries each of the modes for a certain amount of time sequentially. Before deciding on the schedule to use, Satallax parses the problem and determines if it is big enough that a SInE-based premise selection algorithm [31] should be used. If SInE is not activated, then Satallax uses a strategy schedule consisting of 37 modes. If SInE is activated, than Satallax is run with a SInE-specific schedule consisting of 58 modes with different SInE parameter values selecting different premises. Each mode is tried for time limits ranging from less than a second to just over 1 minute.

### Implementation

Satallax is implemented in OCaml, making use of the external tools MiniSat (via a foreign function interface) and E. Satallax is available at:

`http://cl-informatik.uibk.ac.at/~mfaerber/satallax.html`

### Expected Competition Performance

Satallax 3.4 was the CASC-27 THF winner.

## 8.18 Toma 0.4

Teppei Saito
Japan Advanced Institute of Science and Technology, Japan

### Architecture

Toma 0.4 is an automatic equational theorem prover. It proves unsatisfiability of a UEQ problem as follows: A given problem is transformed into a word problem whose validity entails unsatisfiability of the original problem. The word problem is solved by a new variant of maximal (ordered) completion [139, 29].

### Strategies

Toma performs ordered completion in the following way: (1) Given an equational system E, the tool constructs a lexicographic path order ¿ that maximizes reducibility of the ordered rewrite system (E, >) [139]. (2) Using the order >, the tool runs ordered completion [2] on E without the deduce rule (critical pair generation). Such a run eventually ends with an inter-reduced version (E', >). (3) Then the tool checks joinability of the goal. If the goal is joinable in (E', >), the tool outputs the proof and terminates. Otherwise, assigning the union of E' and a set of critical pairs of (E', >) to E, the tool goes back to the first step (1). After a certain

number of iterations of the loop, the tool skips the order generation step (1) and repeats only (2)(3) with a fixed lexicographic path order.

Toma is written in Haskell. Z3 is used to solve maximization problems of reducibility. The source code is available at:

`https://www.jaist.ac.jp/project/maxcomp/`

#### Expected Competition Performance

Toma is still in the experimental stage and unable to compete with matured tools. The tool would solve several easy problems from the categories BOO, GRP and RNG.

### 8.19 Twee 2.4.1

Nick Smallbone
Chalmers University of Technology, Sweden

#### Architecture

Twee [62] is a theorem prover for unit equality problems based on unfailing completion [2]. It implements a DISCOUNT loop, where the active set contains rewrite rules (and unorientable equations) and the passive set contains critical pairs. The basic calculus is not goal-directed, but Twee implements a transformation which improves goal direction for many problems.

Twee features ground joinability testing [38] and a connectedness test [1], which together eliminate many redundant inferences in the presence of unorientable equations. The ground joinability test performs case splits on the order of variables, in the style of [38], and discharges individual cases by rewriting modulo a variable ordering.

#### Strategies

Twee's strategy is simple and it does not tune its heuristics or strategy based on the input problem. The term ordering is always KBO; by default, functions are ordered by number of occurrences and have weight 1. The proof loop repeats the following steps:

- Select and normalise the lowest-scored critical pair, and if it is not redundant, add it as a rewrite rule to the active set.
- Normalise the active rules with respect to each other.
- Normalise the goal with respect to the active rules.

Each critical pair is scored using a weighted sum of the weight of both of its terms. Terms are treated as DAGs when computing weights, i.e., duplicate subterms are only counted once per term.

For CASC, to take advantage of multiple cores, several versions of Twee run in parallel using different parameters (e.g., with the goal-directed transformation on or off).

### Implementation

Twee is written in Haskell. Terms are represented as array-based flatterms for efficient unification and matching. Rewriting uses a perfect discrimination tree. The passive set is represented compactly (12 bytes per critical pair) by only storing the information needed to reconstruct the critical pair, not the critical pair itself. Because of this, Twee can run for an hour or more without exhausting memory.

Twee uses an LCF-style kernel: all rules in the active set come with a certified proof object which traces back to the input axioms. When a conjecture is proved, the proof object is transformed into a human-readable proof. Proof construction does not harm efficiency because the proof kernel is invoked only when a new rule is accepted. In particular, reasoning about the passive set does not invoke the kernel. The translation from Horn clauses to equations is not yet certified.

Twee can be downloaded as open source from:

`http://nick8325.github.io/twee`

### Expected Competition Performance

Twee 2.4.1 is the CASC-J11 UEQ winner.

## 8.20 Twee 2.4.2

Nick Smallbone
Chalmers University of Technology, Sweden

### Architecture

Twee 2.4.2 [62] is a theorem prover for unit equality problems based on unfailing completion [2]. It implements a DISCOUNT loop, where the active set contains rewrite rules (and unorientable equations) and the passive set contains critical pairs. The basic calculus is not goal-directed, but Twee implements a transformation which improves goal direction for many problems.

Twee features ground joinability testing [38] and a connectedness test [1], which together eliminate many redundant inferences in the presence of unorientable equations. The ground joinability test performs case splits on the order of variables, in the style of [38], and discharges individual cases by rewriting modulo a variable ordering.

### Strategies

Twee's strategy is simple and it does not tune its heuristics or strategy based on the input problem. The term ordering is always KBO; by default, functions are ordered by number of occurrences and have weight 1. The proof loop repeats the following steps:

- Select and normalise the lowest-scored critical pair, and if it is not redundant, add it as a rewrite rule to the active set.
- Normalise the active rules with respect to each other.
- Normalise the goal with respect to the active rules.

Each critical pair is scored using a weighted sum of the weight of both of its terms. Terms are treated as DAGs when computing weights, i.e., duplicate subterms are counted only once per term.

For CASC, to take advantage of multiple cores, several versions of Twee run in parallel using different parameters (e.g., with the goal-directed transformation on or off).

**Implementation**

Twee is written in Haskell. Terms are represented as array-based flatterms for efficient unification and matching. Rewriting uses a perfect discrimination tree.

The passive set is represented compactly (12 bytes per critical pair) by storing only the information needed to reconstruct the critical pair, not the critical pair itself. Because of this, Twee can run for an hour or more without exhausting memory.

Twee uses an LCF-style kernel: all rules in the active set come with a certified proof object which traces back to the input axioms. When a conjecture is proved, the proof object is transformed into a human-readable proof. Proof construction does not harm efficiency because the proof kernel is invoked only when a new rule is accepted. In particular, reasoning about the passive set does not invoke the kernel.

Twee can be downloaded as open source from:

`https://nick8325.github.io/twee/`

**Expected Competition Performance**

Either slightly better or slightly worse than Twee 2.4.1.

## 8.21 Vampire 4.7

Giles Reger
University of Manchester, United Kingdom

There are only small changes between Vampire 4.7 and Vampire 4.6 in the tracks relevant to CASC. As TFA did not run in 2021, the updates related to the paper "Making Theory Reasoning Simpler" [45] that were present last year should have an impact this year. This work introduces a new set of rules for the evaluation and simplification of theory literals. We have also added some optional preprocessing steps inspired by Twee (see "Twee: An Equational Theorem Prover" [62]) but these have not been fully incorporated into our strategy portfolio so are unlikely to make a significant impact.

**Architecture**

Vampire [35] is an automatic theorem prover for first-order logic with extensions to theory-reasoning and higher-order logic. Vampire implements the calculi of ordered binary resolution and superposition for handling equality. It also implements the Inst-gen calculus and a MACE-style finite model builder [46]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [133, 44]. A number of standard redundancy criteria and simplification techniques are used for pruning the search space: subsumption, tautology deletion, subsumption resolution and rewriting by ordered unit equalities. The reduction ordering is the Knuth-Bendix Ordering. Substitution

tree and code tree indexes are used to implement all major operations on sets of terms, literals and clauses. Internally, Vampire works only with clausal normal form. Problems in the full first-order logic syntax are clausified during preprocessing [47]. Vampire implements many useful preprocessing transformations including the SinE axiom selection algorithm. When a theorem is proved, the system produces a verifiable proof, which validates both the clausification phase and the refutation of the CNF.

**Strategies**

Vampire 4.7 provides a very large number of options for strategy selection. The most important ones are:

- Choices of saturation algorithm:
  - Limited Resource Strategy [52]
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [133]
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals [30].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection. This has been extended with a layered clause selection approach [28].
- Set-of-support strategy with extensions for theory reasoning.
- For theory-reasoning:
  - Ground equational reasoning via congruence closure.
  - Addition of theory axioms and evaluation of interpreted functions [45].
  - Use of Z3 with AVATAR to restrict search to ground-theory-consistent splitting branches [44].
  - Specialised theory instantiation and unification [48].
  - Extensionality resolution with detection of extensionality axioms
- For higher-order problems:
  - Translation to polymorphic first-order logic using applicative form and combinators
  - A superposition calculus [16] utilising a KBO-like ordering [17] for orienting combinator equations. The calculus introduces an inference, narrow, for rewriting with combinator equations.

- Proof search heuristics targeting the growth of clauses resulting from narrowing.
- An extension of unification with abstraction to deal with functional and boolean extensionality.
- Various inferences to deal with booleans

**Implementation**

Vampire 4.7 is implemented in C++. It makes use of minisat and z3 (the tagged GitHub commit details which z3 commit). See the website for more information and access to the GitHub repository.:

`https://vprover.github.io/`

**Expected Competition Performance**

Vampire 4.7 is the CASC-J11 FOF and FNT winner.

## 8.22 Vampire 4.8

Michael Rawson
TU Wien, Austria

There have been a number of changes and improvements since Vampire 4.7, although it is still the same beast. Most significant from a competition point of view are long-awaited refreshed strategy schedules. As a result, several features present in previous competitions will now come into full force, including new rules for the evaluation and simplification of theory literals. A large number of completely new features and improvements also landed this year: highlights include a significant refactoring of the substitution tree implementation, the arrival of encompassment demodulation to Vampire, and support for parametric datatypes.

Vampire's higher-order support has also been re-implemented from the ground up. The new implementation is still at an early stage and its theoretical underpinnings are being developed. There is currently no documentation of either.

**Architecture**

Vampire [35] is an automatic theorem prover for first-order logic with extensions to theory-reasoning and higher-order logic. Vampire implements the calculi of ordered binary resolution, and superposition for handling equality. It also implements the Inst-gen calculus and a MACE-style finite model builder [46]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [133, 44]. A number of standard redundancy criteria and simplification techniques are used for pruning the search space: subsumption, tautology deletion, subsumption resolution and rewriting by ordered unit equalities. The reduction ordering is the Knuth-Bendix Ordering. Substitution tree and code tree indexes are used to implement all major operations on sets of terms, literals and clauses. Internally, Vampire works only with clausal normal form. Problems in the full first-order logic syntax are clausified during preprocessing [47]. Vampire implements many useful preprocessing transformations including the SinE axiom selection algorithm. When a theorem is proved, the system produces a verifiable proof, which validates both the clausification phase and the refutation of the CNF.

### Strategies

Vampire 4.8 provides a very large number of options for strategy selection. The most important ones are:

- Choices of saturation algorithm:
  - Limited Resource Strategy [52]
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [133]
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals [30].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection. This has been extended with a layered clause selection approach [28].
- Set-of-support strategy with extensions for theory reasoning.
- For theory-reasoning:
  - Ground equational reasoning via congruence closure.
  - Addition of theory axioms and evaluation of interpreted functions [45].
  - Use of Z3 with AVATAR to restrict search to ground-theory-consistent splitting branches [44].
  - Specialised theory instantiation and unification [48].
  - Extensionality resolution with detection of extensionality axioms

The schedule for the new HOL implementation was developed using Snake, a strategy schedule construction tool described in more detail last year. The Snake schedule will this year fully embrace Vampire randomisation support [66] and, in particular, every strategy will independently shuffle the input problem, to nullify (in expectation) the effect of problem scrambling done by the organisers.

### Implementation

Vampire 4.8 is implemented in C++. It makes use of fixed versions of Minisat and Z3. See the website

`https://vprover.github.io/`

for more information and access to the GitHub repository.

### Expected Competition Performance

Vampire 4.8 should be an improvement on the previous version. A reasonably strong performance across all divisions is therefore expected. In the higher-order divisions, performance should be better than the previous year, but not yet on par with category-leading solvers.

## 8.23 Zipperposition 2.1.999

Jasmin Blanchette
Vrije Universiteit Amsterdam, The Netherlands

### Architecture

Zipperposition is a superposition-based theorem prover for typed first-order logic with equality and for higher-order logic. It is a pragmatic implementation of a complete calculus for full higher-order logic [10]. It features a number of extensions that include polymorphic types, user-defined rewriting on terms and formulas ("deduction modulo theories"), a lightweight variant of AVATAR for case splitting [26], and Boolean reasoning [138]. The core architecture of the prover is based on saturation with an extensible set of rules for inferences and simplifications. Zipperposition uses a full higher-order unification algorithm that enables efficient integration of procedures for decidable fragments of higher-order unification [135]. The initial calculus and main loop were imitations of an earlier version of E [56]. With the implementation of higher-order superposition, the main loop had to be adapted to deal with possibly infinite sets of unifiers [134].

### Strategies

The system uses various strategies in a portfolio. The strategies are run in parallel, making use of all CPU cores available. We designed the portfolio of strategies by manual inspection of TPTP problems. Zipperposition's heuristics are inspired by efficient heuristics used in E. Various calculus extensions are used by the strategies [134]. The portfolio mode distinguishes between first-order and higher-order problems. If the problem is first-order, all higher-order prover features are turned off. In particular, the prover uses standard first-order superposition calculus and disables collaboration with the backend prover (described below). Other than that, the portfolio is static and does not depend on the syntactic properties of the problem.

### Implementation

The prover is implemented in OCaml. Term indexing is done using fingerprints for unification, perfect discrimination trees for rewriting, and feature vectors for subsumption. Some inference rules such as contextual literal cutting make heavy use of subsumption. For higher-order problems, some strategies use the E prover as an end-game backend prover.

Zipperposition's code can be found at

`https://github.com/sneeuwballen/zipperposition`

and is entirely free software (BSD-licensed).

Zipperposition can also output graphic proofs using graphviz. Some tools to perform type inference and clausification for typed formulas are also provided, as well as a separate library for dealing with terms and formulas [25].

### Expected Competition Performance

Zipperposition 2.1.999 is the CASC-J11 THF winner.

## 8.24 Zipperposition 2.1.999

Jasmin Blanchette
Ludwig-Maximilians-Universität München, Germany

### Architecture

Zipperposition is a superposition-based theorem prover for typed first-order logic with equality and for higher-order logic. It is a pragmatic implementation of a complete calculus for full higher-order logic [10]. It features a number of extensions that include polymorphic types, user-defined rewriting on terms and formulas ("deduction modulo theories"), a lightweight variant of AVATAR for case splitting [26], and Boolean reasoning [138]. The core architecture of the prover is based on saturation with an extensible set of rules for inferences and simplifications. Zipperposition uses a full higher-order unification algorithm that enables efficient integration of procedures for decidable fragments of higher-order unification [135]. The initial calculus and main loop were imitations of an earlier version of E [56]. With the implementation of higher-order superposition, the main loop had to be adapted to deal with possibly infinite sets of unifiers [134].

### Strategies

The system uses various strategies in a portfolio. The strategies are run in parallel, making use of all CPU cores available. We designed the portfolio of strategies by manual inspection of TPTP problems. Zipperposition's heuristics are inspired by efficient heuristics used in E. Various calculus extensions are used by the strategies [134]. The portfolio mode distinguishes between first-order and higher-order problems. If the problem is first-order, all higher-order prover features are turned off. In particular, the prover uses standard first-order superposition calculus and disables collaboration with the backend prover (described below). Other than that, the portfolio is static and does not depend on the syntactic properties of the problem.

### Implementation

The prover is implemented in OCaml. Term indexing is done using fingerprints for unification, perfect discrimination trees for rewriting , and feature vectors for subsumption. Some inference rules such as contextual literal cutting make heavy use of subsumption. For higher-order problems, some strategies use the E prover as an end-game backend prover.

Zipperposition's code can be found at

```
https://github.com/sneeuwballen/zipperposition
```

and is entirely free software (BSD-licensed).

Zipperposition can also output graphic proofs using graphviz. Some tools to perform type inference and clausification for typed formulas are also provided, as well as a separate library for dealing with terms and formulas [25].

**Expected Competition Performance**

The prover is expected to perform well on THF, about as well as last year's version. We expect to beat E. In the SLH division, we expect respectable performance, but E will probably win.

# 9 Conclusion

The CADE-29 ATP System Competition was the twenty-eighth large scale competition for classical logic ATP systems. The organizers believe that CASC fulfills its main motivations: evaluation of relative capabilities of ATP systems, stimulation of research, motivation for improving implementations, and providing an exciting event. Through the continuity of the event and consistency in the reporting of the results, performance comparisons with previous and future years are easily possible. The competition provides exposure for system builders both within and outside of the community, and provides an overview of the implementation state of running, fully automatic, classical logic ATP systems.

# References

[1] L. Bachmair and N. Dershowitz. Critical Pair Criteria for Completion. *Journal of Symbolic Computation*, 6(1):1–18, 1988.

[2] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion Without Failure. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, pages 1–30. Academic Press, 1989.

[3] J. Backes and C.E. Brown. Analytic Tableaux for Higher-Order Logic with Choice. *Journal of Automated Reasoning*, 47(4):451–479, 2011.

[4] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In D. Fisman and G. Rosu, editors, *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 13243 in Lecture Notes in Computer Science, pages 415–442. Springer, 2022.

[5] H. Barbosa, P. Fontaine, and A. Reynolds. Congruence Closure with Free Variables. In A. Legay and T. Margaria, editors, *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 10205 in Lecture Notes in Computer Science, pages 2134–230. Springer-Verlag, 2017.

[6] H. Barbosa, A. Reynolds, D. El Ouraoui, C. Tinelli, and C. Barrett. Extending SMT Solvers to Higher-Order Logic. In P. Fontaine, editor, *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 35–54. Springer-Verlag, 2019.

[7] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification*, number 6806 in Lecture Notes in Computer Science, pages 171–177. Springer-Verlag, 2011.

[8] P. Baumgartner, J. Bax, and U. Waldmann. Beagle - A Hierarchic Superposition Theorem Prover. In A. Felty and A. Middeldorp, editors, *Proceedings of the 25th International Conference on Automated Deduction*, number 9195 in Lecture Notes in Computer Science, pages 285–294. Springer-Verlag, 2015.

[9] P. Baumgartner and U. Waldmann. Hierarchic Superposition With Weak Abstraction. In M.P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction*, number 7898 in Lecture Notes in Artificial Intelligence, pages 39–57. Springer-Verlag, 2013.

[10] A. Bentkamp, J. Blanchette, S. Tourret, and P. Vukmirović. Superposition for Full Higher-order Logic. In A. Platzer and G. Sutcliffe, editors, *Proceedings of the 28th International Conference on Automated Deduction*, number 12699 in Lecture Notes in Computer Science, pages 396–412. Springer-Verlag, 2021.

[11] C. Benzmüller. Extensional Higher-order Paramodulation and RUE-Resolution. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 399–413. Springer-Verlag, 1999.

[12] C. Benzmüller and M. Kohlhase. LEO - A Higher-Order Theorem Prover. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 139–143. Springer-Verlag, 1998.

[13] C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. LEO-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic. In P. Baumgartner, A. Armando, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 162–170. Springer-Verlag, 2008.

[14] C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0 - The Core TPTP Language for Classical Higher-Order Logic. In P. Baumgartner, A. Armando, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 491–506. Springer-Verlag, 2008.

[15] C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Combined Reasoning by Automated Cooperation. *Journal of Applied Logic*, 6(3):318–342, 2008.

[16] A. Bhayat and G. Reger. A Combinator-based Superposition Calculus for Higher-Order Logic. In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, number 12166 in Lecture Notes in Artificial Intelligence, pages 278–296, 2020.

[17] A. Bhayat and G. Reger. A Knuth-Bendix-Like Ordering for Orienting Combinator Equations. In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, number 12166 in Lecture Notes in Artificial Intelligence, pages 259–277, 2020.

[18] A. Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[19] J. Blanchette, M. Haslbeck, D. Matichuk, and T. Nipkow. Mining the Archive of Formal Proofs. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Proceedings of the 8th Conference on Intelligent Computer Mathematics*, number 9150 in Lecture Notes in Computer Science, pages 3–17, 2015.

[20] J. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-order Form with Rank-1 Polymorphism. In M.P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction*, number 7898 in Lecture Notes in Artificial Intelligence, pages 414–420. Springer-Verlag, 2013.

[21] C. Brown and C. Kaliszyk. Lash 1.0 System Description. In J. Blanchette, L. Kovacs, and D. Pattinson, editors, *Proceedings of the 11th International Joint Conference on Automated Reasoning*, number 13385 in Lecture Notes in Artificial Intelligence, pages 350–358, 2022.

[22] C.E. Brown. Satallax: An Automated Higher-Order Prover (System Description). In B. Gramlich, D. Miller, and U. Sattler, editors, *Proceedings of the 6th International Joint Conference on Automated Reasoning*, number 7364 in Lecture Notes in Artificial Intelligence, pages 111–117, 2012.

[23] C.E. Brown. Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems. *Journal of Automated Reasoning*, 51(1):57–77, 2013.

[24] C.E. Brown and G. Smolka. Analytic Tableaux for Simple Type Theory and its First-Order Fragment. *Logical Methods in Computer Science*, 6(2), 2010.

[25] S. Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond*. PhD thesis, Ecole Polytechnique, Paris, France, 2015.

[26] G. Ebner, J. Blanchette, and S. Tourret. A Unifying Splitting Framework. In A. Platzer and G. Sutcliffe, editors, *Proceedings of the 28th International Conference on Automated Deduction*, number 12699 in Lecture Notes in Computer Science, pages 344–360. Springer-Verlag, 2021.

[27] N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, number 2919 in Lecture Notes in Computer Science, pages 502–518. Springer-Verlag, 2004.

[28] B. Gleiss and M. Suda. Layered Clause Selection for Theory Reasoning. In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, number 12166 in Lecture Notes in Computer Science, pages 402–409, 2020.

[29] N. Hirokawa. Completion and Reduction Orders. In N. Kobayashi, editor, *Proceedings of the 6th International Conference on Formal Structures for Computation and Deduction*, number 195 in Leibniz International Proceedings in Informatics, pages 2:1–2:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[30] K. Hoder, G. Reger, M. Suda, and A. Voronkov. Selecting the Selection. In N. Olivetti and A. Tiwari, editors, *Proceedings of the 8th International Joint Conference on Automated Reasoning*, number 9706 in Lecture Notes in Artificial Intelligence, pages 313–329, 2016.

[31] K. Hoder and A. Voronkov. Sine Qua Non for Large Theory Reasoning. In V. Sofronie-Stokkermans and N. Bjœrner, editors, *Proceedings of the 23rd International Conference on Automated Deduction*, number 6803 in Lecture Notes in Artificial Intelligence, pages 299–314. Springer-Verlag, 2011.

[32] J. Jakubuv and J. Urban. ENIGMA: Efficient Learning-based Inference Guiding Machine. In H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, editors, *Proceedings of the 10th International Conference on Intelligent Computer Mathematics*, number 10383 in Lecture Notes in Artificial Intelligence, pages 292–302. Springer-Verlag, 2017.

[33] J. Jakubuv and J. Urban. Enhancing ENIGMA Given Clause Guidance. In F. Rabe, W. Farmer, G. Passmore, and A. Youssef, editors, *Proceedings of the 11th International Conference on Intelligent Computer Mathematics*, number 11006 in Lecture Notes in Artificial Intelligence, pages 118–124. Springer-Verlag, 2018.

[34] C. Kaliszyk, G. Sutcliffe, and F. Rabe. TH1: The TPTP Typed Higher-Order Form with Rank-1 Polymorphism. In P. Fontaine, S. Schulz, and J. Urban, editors, *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning*, number 1635 in CEUR Workshop Proceedings, pages 41–55, 2016.

[35] L. Kovacs and A. Voronkov. First-Order Theorem Proving and Vampire. In N. Sharygina and H. Veith, editors, *Proceedings of the 25th International Conference on Computer Aided Verification*, number 8044 in Lecture Notes in Artificial Intelligence, pages 1–35. Springer-Verlag, 2013.

[36] B. Loechner. Things to Know When Implementing KBO. *Journal of Automated Reasoning*, 36(4):289–310, 2006.

[37] B. Loechner. Things to Know When Implementing LBO. *Journal of Artificial Intelligence Tools*, 15(1):53–80, 2006.

[38] U. Martin and T. Nipkow. Ordered Rewriting and Confluence. In M.E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, number 449 in Lecture Notes in Artificial Intelligence, pages 366–380. Springer-Verlag, 1990.

[39] W.W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[40] W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.

[41] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[42] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 335–367. Elsevier Science,

2001.

[43] L. Paulson and J. Blanchette. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In G. Sutcliffe, E. Ternovska, and S. Schulz, editors, *Proceedings of the 8th International Workshop on the Implementation of Logics*, number 2 in EPiC Series in Computing, pages 1–11. EasyChair Publications, 2010.

[44] G. Reger, N. Bjørner, M. Suda, and A. Voronkov. AVATAR Modulo Theories. In C. Benzmüller, G. Sutcliffe, and R. Rojas, editors, *Proceedings of the 2nd Global Conference on Artificial Intelligence*, number 41 in EPiC Series in Computing, pages 39–52. EasyChair Publications, 2016.

[45] G. Reger, J. Schoisswohl, and A. Voronkov. Making Theory Reasoning Simpler. In J. Groote and K. Larsen, editors, *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 12652 in Lecture Notes in Computer Science, pages 164–180. Springer-Verlag, 2021.

[46] G. Reger, M. Suda, and A. Voronkov. Finding Finite Models in Multi-Sorted First Order Logic. In N. Creignou and D. Le Berre, editors, *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*, number 9710 in Lecture Notes in Computer Science, pages 323–341. Springer-Verlag, 2016.

[47] G. Reger, M. Suda, and A. Voronkov. New Techniques in Clausal Form Generation. In C. Benzmüller, G. Sutcliffe, and R. Rojas, editors, *Proceedings of the 2nd Global Conference on Artificial Intelligence*, number 41 in EPiC Series in Computing, pages 11–23. EasyChair Publications, 2016.

[48] G. Reger, M. Suda, and A. Voronkov. Unification with Abstraction and Theory Instantiation in Saturation-based Reasoning. In D. Beyer and M. Huisman, editors, *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 10805 in Lecture Notes in Computer Science, pages 3–22. Springer-Verlag, 2018.

[49] A. Reynolds, H. Barbosa, and P. Fontaine. Revisiting Enumerative Instantiation. In D. Beyer and M. Huisman, editors, *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 10805 in Lecture Notes in Computer Science, pages 112–131. Springer-Verlag, 2018.

[50] A. Reynolds, C. Tinelli, and L. de Moura. Finding Conflicting Instances of Quantified Formulas in SMT. In K. Claessen and V. Kuncak, editors, *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, pages 195–202, 2014.

[51] A. Reynolds, C. Tinelli, A. Goel, S. Krstic, M. Deters, and C. Barrett. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In M.P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction*, number 7898 in Lecture Notes in Artificial Intelligence, pages 377–391. Springer-Verlag, 2013.

[52] A. Riazanov and A. Voronkov. Limited Resource Strategy in Resolution Theorem Proving. *Journal of Symbolic Computation*, 36(1-2):101–115, 2003.

[53] O. Roussel. Controlling a Solver Execution with the runsolver Tool. *Journal of Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, 2011.

[54] P. Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 5330 in Lecture Notes in Artificial Intelligence, pages 274–289. Springer-Verlag, 2008.

[55] S. Schulz. A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae. In S. Haller and G. Simmons, editors, *Proceedings of the 15th International FLAIRS Conference*, pages 72–76. AAAI Press, 2002.

[56] S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.

[57] S. Schulz. System Abstract: E 0.81. In M. Rusinowitch and D. Basin, editors, *Proceedings of the 2nd International Joint Conference on Automated Reasoning*, number 3097 in Lecture Notes in Artificial Intelligence, pages 223–228. Springer-Verlag, 2004.

[58] S. Schulz. Fingerprint Indexing for Paramodulation and Rewriting. In B. Gramlich, D. Miller, and U. Sattler, editors, *Proceedings of the 6th International Joint Conference on Automated Reasoning*, number 7364 in Lecture Notes in Artificial Intelligence, pages 477–483. Springer-Verlag, 2012.

[59] S. Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In M.P. Bonacina and M. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, number 7788 in Lecture Notes in Artificial Intelligence, pages 45–67. Springer-Verlag, 2013.

[60] S. Schulz. System Description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 8312 in Lecture Notes in Computer Science, pages 477–483. Springer-Verlag, 2013.

[61] S. Schulz, S. Cruanes, and P. Vukmirović. Faster, Higher, Stronger: E 2.3. In P. Fontaine, editor, *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 495–507. Springer-Verlag, 2019.

[62] N. Smallbone. Twee: An Equational Theorem Prover (System Description). In A. Platzer and G. Sutcliffe, editors, *Proceedings of the 28th International Conference on Automated Deduction*, number 12699 in Lecture Notes in Computer Science, pages 602–613. Springer-Verlag, 2021.

[63] A. Steen and C. Benzmüller. Extensional Higher-Order Paramodulation in Leo-III. *Journal of Automated Reasoning*, 65(6):775–807, 2021.

[64] A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: a Cross-Community Infrastructure for Logic Solving. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Proceedings of the 7th International Joint Conference on Automated Reasoning*, number 8562 in Lecture Notes in Artificial Intelligence, pages 367–373, 2014.

[65] M. Suda. Aiming for the Goal with SInE. In C. Benzmüller, C. Lisetti, and M. Theobald, editors, *Proceedings of the 5th and 6th Vampire Workshops*, number 71 in EPiC Series in Computing, page 38–44. EasyChair Publications, 2019.

[66] M. Suda. Vampire Getting Noisy: Will Random Bits Help Conquer Chaos? (System Description). In J. Blanchette, L. Kovacs, and D. Pattinson, editors, *Proceedings of the 11th International Joint Conference on Automated Reasoning*, number 13385 in Lecture Notes in Artificial Intelligence, pages 659–667, 2022.

[67] G. Sutcliffe. Proceedings of the CADE-16 ATP System Competition. Trento, Italy, 1999.

[68] G. Sutcliffe. Proceedings of the CADE-17 ATP System Competition. Pittsburgh, USA, 2000.

[69] G. Sutcliffe. The CADE-16 ATP System Competition. *Journal of Automated Reasoning*, 24(3):371–396, 2000.

[70] G. Sutcliffe. Proceedings of the IJCAR ATP System Competition. Siena, Italy, 2001.

[71] G. Sutcliffe. The CADE-17 ATP System Competition. *Journal of Automated Reasoning*, 27(3):227–250, 2001.

[72] G. Sutcliffe. Proceedings of the CADE-18 ATP System Competition. Copenhagen, Denmark, 2002.

[73] G. Sutcliffe. Proceedings of the CADE-19 ATP System Competition. Miami, USA, 2003.

[74] G. Sutcliffe. Proceedings of the 2nd IJCAR ATP System Competition. Cork, Ireland, 2004.

[75] G. Sutcliffe. Proceedings of the CADE-20 ATP System Competition. Tallinn, Estonia, 2005.

[76] G. Sutcliffe. The IJCAR-2004 Automated Theorem Proving Competition. *AI Communications*, 18(1):33–40, 2005.

[77] G. Sutcliffe. Proceedings of the 3rd IJCAR ATP System Competition. Seattle, USA, 2006.

[78] G. Sutcliffe. The CADE-20 Automated Theorem Proving Competition. *AI Communications*, 19(2):173–181, 2006.

[79] G. Sutcliffe. Proceedings of the CADE-21 ATP System Competition. Bremen, Germany, 2007.

[80] G. Sutcliffe. The 3rd IJCAR Automated Theorem Proving Competition. *AI Communications*, 20(2):117–126, 2007.

[81] G. Sutcliffe. Proceedings of the 4th IJCAR ATP System Competition. Sydney, Australia, 2008.

[82] G. Sutcliffe. The CADE-21 Automated Theorem Proving System Competition. *AI Communications*, 21(1):71–82, 2008.

[83] G. Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 38–49, 2008.

[84] G. Sutcliffe. Proceedings of the CADE-22 ATP System Competition. Montreal, Canada, 2009.

[85] G. Sutcliffe. The 4th IJCAR Automated Theorem Proving System Competition - CASC-J4. *AI Communications*, 22(1):59–72, 2009.

[86] G. Sutcliffe. Proceedings of the 5th IJCAR ATP System Competition. Edinburgh, United Kingdom, 2010.

[87] G. Sutcliffe. The CADE-22 Automated Theorem Proving System Competition - CASC-22. *AI Communications*, 23(1):47–60, 2010.

[88] G. Sutcliffe. Proceedings of the CADE-23 ATP System Competition. Wroclaw, Poland, 2011.

[89] G. Sutcliffe. The 5th IJCAR Automated Theorem Proving System Competition - CASC-J5. *AI Communications*, 24(1):75–89, 2011.

[90] G. Sutcliffe. Proceedings of the 6th IJCAR ATP System Competition. Manchester, England, 2012.

[91] G. Sutcliffe. The CADE-23 Automated Theorem Proving System Competition - CASC-23. *AI Communications*, 25(1):49–63, 2012.

[92] G. Sutcliffe. Proceedings of the 24th CADE ATP System Competition. Lake Placid, USA, 2013.

[93] G. Sutcliffe. The 6th IJCAR Automated Theorem Proving System Competition - CASC-J6. *AI Communications*, 26(2):211–223, 2013.

[94] G. Sutcliffe. Proceedings of the 7th IJCAR ATP System Competition. Vienna, Austria, 2014.

[95] G. Sutcliffe. The CADE-24 Automated Theorem Proving System Competition - CASC-24. *AI Communications*, 27(4):405–416, 2014.

[96] G. Sutcliffe. Proceedings of the CADE-25 ATP System Competition. Berlin, Germany, 2015. http://www.tptp.org/CASC/25/Proceedings.pdf.

[97] G. Sutcliffe. The 7th IJCAR Automated Theorem Proving System Competition - CASC-J7. *AI Communications*, 28(4):683–692, 2015.

[98] G. Sutcliffe. Proceedings of the 8th IJCAR ATP System Competition. Coimbra, Portugal, 2016. http://www.tptp.org/CASC/J8/Proceedings.pdf.

[99] G. Sutcliffe. The 8th IJCAR Automated Theorem Proving System Competition - CASC-J8. *AI Communications*, 29(5):607–619, 2016.

[100] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[101] G. Sutcliffe. Proceedings of the 26th CADE ATP System Competition. Gothenburg, Sweden, 2017. http://www.tptp.org/CASC/26/Proceedings.pdf.

[102] G. Sutcliffe. The CADE-26 Automated Theorem Proving System Competition - CASC-26. *AI Communications*, 30(6):419–432, 2017.

[103] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[104] G. Sutcliffe. Proceedings of the 9th IJCAR ATP System Competition. Oxford, United Kingdom, 2018. http://www.tptp.org/CASC/J9/Proceedings.pdf.

[105] G. Sutcliffe. The 9th IJCAR Automated Theorem Proving System Competition - CASC-J9. *AI Communications*, 31(6):495–507, 2018.

[106] G. Sutcliffe. Proceedings of the CADE-27 ATP System Competition. Natal, Brazil, 2019. http://www.tptp.org/CASC/27/Proceedings.pdf.

[107] G. Sutcliffe. Proceedings of the 10th IJCAR ATP System Competition. Online, 2020. http://www.tptp.org/CASC/J10/Proceedings.pdf.

[108] G. Sutcliffe. The CADE-27 Automated Theorem Proving System Competition - CASC-27. *AI Communications*, 32(5-6):373–389, 2020.

[109] G. Sutcliffe. Proceedings of the CADE-28 ATP System Competition. Online, 2021. http://www.tptp.org/CASC/28/Proceedings.pdf.

[110] G. Sutcliffe. The 10th IJCAR Automated Theorem Proving System Competition - CASC-J10. *AI Communications*, 34(2):163–177, 2021.

[111] G. Sutcliffe. Proceedings of the 11th IJCAR ATP System Competition. Online, 2022. http://www.tptp.org/CASC/J11/Proceedings.pdf.

[112] G. Sutcliffe. Proceedings of the CADE-29 ATP System Competition. Online, 2023. http://www.tptp.org/CASC/29/Proceedings.pdf.

[113] G. Sutcliffe and C. Benzmüller. Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. *Journal of Formalized Reasoning*, 3(1):1–27, 2010.

[114] G. Sutcliffe and M. Desharnais. The CADE-28 Automated Theorem Proving System Competition - CASC-28. *AI Communications*, 34(4):259–276, 2022.

[115] G. Sutcliffe and M. Desharnais. The 11th IJCAR Automated Theorem Proving System Competition - CASC-J11. *AI Communications*, 36(2):73–91,, 2023.

[116] G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 67–81. Springer, 2006.

[117] G. Sutcliffe and C. Suttner. The CADE-14 ATP System Competition. Technical Report 98/01, Department of Computer Science, James Cook University, Townsville, Australia, 1998.

[118] G. Sutcliffe and C. Suttner. The CADE-18 ATP System Competition. *Journal of Automated Reasoning*, 31(1):23–32, 2003.

[119] G. Sutcliffe and C. Suttner. The CADE-19 ATP System Competition. *AI Communications*, 17(3):103–182, 2004.

[120] G. Sutcliffe, C. Suttner, and F.J. Pelletier. The IJCAR ATP System Competition. *Journal of Automated Reasoning*, 28(3):307–320, 2002.

[121] G. Sutcliffe and C.B. Suttner. Special Issue: The CADE-13 ATP System Competition. *Journal of Automated Reasoning*, 18(2), 1997.

[122] G. Sutcliffe and C.B. Suttner. The Procedures of the CADE-13 ATP System Competition. *Journal of Automated Reasoning*, 18(2):163–169, 1997.

[123] G. Sutcliffe and C.B. Suttner. Proceedings of the CADE-15 ATP System Competition. Lindau, Germany, 1998.

[124] G. Sutcliffe and C.B. Suttner. The CADE-15 ATP System Competition. *Journal of Automated Reasoning*, 23(1):1–23, 1999.

[125] G. Sutcliffe and C.B. Suttner. Evaluating General Purpose Automated Theorem Proving Systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.

[126] G. Sutcliffe and J. Urban. The CADE-25 Automated Theorem Proving System Competition - CASC-25. *AI Communications*, 29(3):423–433, 2016.

[127] C.B. Suttner and G. Sutcliffe. The CADE-14 ATP System Competition. *Journal of Automated Reasoning*, 21(1):99–134, 1998.

[128] T. Tammet. GKC: a Reasoning System for Large Knowledge Bases. In P. Fontaine, editor, *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 538–549. Springer-Verlag, 2019.

[129] T. Tammet, D. Draheim, and P. Järv. Confidences for Commonsense Reasoning. In A. Platzer and G. Sutcliffe, editors, *Proceedings of the 28th International Conference on Automated Deduction*, number 12699 in Lecture Notes in Computer Science, pages 507–524. Springer-Verlag, 2021.

[130] T. Tammet, D. Draheim, and P. Järv. Gk: Implementing Full First Order Default Logic for Commonsense Reasoning (System Description). In J. Blanchette, L. Kovacs, and D. Pattinson, editors, *Proceedings of the 11th International Joint Conference on Automated Reasoning*, number 13385 in Lecture Notes in Artificial Intelligence, pages 300–309, 2022.

[131] T. Tammet, P. Järv, M. Verrev, and D. Draheim. An Experimental Pipeline for Automated Reasoning in Natural Language. In B. Pientka and C. Tinelli, editors, *Proceedings of the 29th International Conference on Automated Deduction*, number 14132 in Lecture Notes in Computer Science, page To appear. Springer-Verlag, 2023.

[132] T. Tammet and G. Sutcliffe. Combining JSON-LD with First Order Logic. In E. Marx and T. Soru, editors, *Proceedings of the 15th IEEE International Conference on Semantic Computing*, pages 256–261, 2021.

[133] A. Voronkov. AVATAR: The New Architecture for First-Order Theorem Provers. In A. Biere and R. Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification*, number 8559 in Lecture Notes in Computer Science, pages 696–710, 2014.

[134] P. Vukmirović, A. Bentkamp, J. Blanchette, S. Cruanes, V. Nummelin, and S. Tourret. Making Higher-order Superposition Work. In A. Platzer and G. Sutcliffe, editors, *Proceedings of the 28th International Conference on Automated Deduction*, number 12699 in Lecture Notes in Computer Science, pages 415–432. Springer-Verlag, 2021.

[135] P. Vukmirović, A. Bentkamp, and V. Nummelin. Efficient Full Higher-order Unification. In Z.M. Ariola, editor, *Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction*, number 167 in Leibniz International Proceedings in Informatics, pages 5:1–5:20. Dagstuhl Publishing, 2020.

[136] P. Vukmirović, J. Blanchette, S. Cruanes, and S. Schulz. Extending a Brainiac Prover to Lambda-Free Higher-Order Logic. In T. Vojnar and L. Zhang, editors, *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 11427 in Lecture Notes in Computer Science, pages 192–210. Springer-Verlag, 2019.

[137] P. Vukmirović, J. Blanchette, and S. Schulz. Extending a High-Performance Prover to Higher-Order Logic. In S. Sankaranarayanan and N. Sharygina, editors, *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 13994 in Lecture Notes in Computer Science, page 111–129. Springer-Verlag, 2023.

[138] P. Vukmirović and V. Nummelin. Boolean Reasoning in a Higher-Order Superposition Prover. In P. Fontaine, P. Rümmer, and S. Tourret, editors, *Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning*, number 2752 in CEUR Workshop Proceedings, pages 148–166, 2020.

[139] S. Winkler and G. Moser. MaedMax: A Maximal Ordered Completion Tool. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Proceedings of the 9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 388–404, 2018.

[140] M. Wisniewski, A. Steen, and C. Benzmüller. LeoPARD - A Generic Platform for the Implementation of Higher-Order Reasoners. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Proceedings of the International Conference on Intelligent Computer Mathematics*, number 9150 in Lecture Notes in Computer Science, pages 325–330. Springer-Verlag, 2015.

[141] Y. Xu, J. Liu, S. Chen, X. Zhong, and X. He. Contradiction Separation Based Dynamic Multi-clause Synergized Automated Deduction. *Information Sciences*, 462:93–113, 2018.