CASC-28

CASC-28

CASC-28

CASC-28

Proceedings of CASC-28 – the CADE-28 ATP System Competition

Geoff Sutcliffe

University of Miami, USA

Abstract

The CADE ATP System Competition (CASC) evaluates the performance of sound, fully automatic, classical logic, ATP systems. The evaluation is in terms of the number of problems solved, the number of acceptable proofs and models produced, and the average runtime for problems solved, in the context of a bounded number of eligible problems chosen from the TPTP problem library and other useful sources of test problems, and specified time limits on solution attempts. The CADE-28 ATP System Competition (CASC-28) was held on 13th July 2021. The design of the competition and its rules, and information regarding the competing systems, are provided.

1 Introduction

The CADE and IJCAR conferences are the major forums for the presentation of new research in all aspects of automated deduction. In order to stimulate ATP research and system development, and to expose ATP systems within and beyond the ATP community, the CADE ATP System Competition (CASC) is held at each CADE and IJCAR conference. CASC-28 was held on 13th July 2021, as part of the 28th International Conference on Automated Deduction (CADE-28). It was the twenty-sixth competition in the CASC series [131, 137, 134, 85, 87, 130, 128, 129, 92, 94, 96, 98, 101, 103, 105, 107, 109, 111, 113, 136, 115, 117, 120, 123, 124].

CASC evaluates the performance of sound, fully automatic, classical logic, ATP systems. The evaluation is in terms of:

- the number of problems solved,
- the number of acceptable proofs and models produced, and
- the average runtime for problems solved;

in the context of:

- a bounded number of eligible problems, chosen from the TPTP problem library [118] and other useful sources of test problems, and
- specified time limits on solution attempts.

Twenty-two ATP system versions, listed in Tables 1 and 2, entered into the various competition and demonstration divisions. The winners of the CASC-J10 (the previous CASC) divisions, and the Prover9 1109a system, were automatically entered into the corresponding demonstration divisions, to provide benchmarks against which progress can be judged (the competition archive provides access to the systems' executables and source code).

The design and procedures of CASC-28 evolved from those of previous CASCs [131, 132, 127, 133, 83, 84, 86, 88, 89, 90, 91, 93, 95, 97, 100, 102, 104, 106, 108, 110, 112, 114, 116, 119, 121, 122]. Important changes for CASC-28 were:

ATP System	Divisions	Entrant (Associates)	Entrant's Affiliation
CSE 1.4	FOF	Feng Cao (Yang Xu, Peiyao Liu, Jun Liu, Shuwei Chen, Guoyan Zeng, Jian Zhong, Guanfeng Wu, Xingxing He, Peng Xu, Qinghua Liu, Huimin Fu, Zhenming Song)	Southwest Jiaotong University
CSE_E 1.3	FOF	Peiyao Liu (Yang Xu, Feng Cao, Stephan Schulz, Jun Liu, Shuwei Chen, Guoyan Zeng, Jian Zhong, Guanfeng Wu, Xingxing He, Peng Xu, Qinghua Liu Huimin Fu, Zhenming Song) Xiaodong Guan, Zhenming Song)	Southwest Jiaotong University
CSE-F 1.0	FOF	Peiyao Liu (Yang Xu, Feng Cao, Jun Liu, Shuwei Chen, Guoyan Zeng, Jian Zhong, Guanfeng Wu, Xingxing He, Peng Xu, Qinghua Liu, Huimin Fu, Zhenming Song)	Southwest Jiaotong University
cvc5 1.0	THF FOF FNT SLH	Andrew Reynolds (Haniel Barbosa, Cesare Tinelli, Clark Barrett)	University of Iowa
Drodi 3.1.5	FOF UEQ	Oscar Contreras	Amateur Programmer
E 2.5	UEQ LTB (demo)	CASC	CASC-J10 winner
E 2.6	FOF FNT UEQ LTB	Stephan Schulz	DHBW Stuttgart
Ehoh 2.7	THF SLH	Petar Vukmirović (Stephan Schulz)	Vrije Universiteit Amsterdam
Etableau 0.67	FOF FNT UEQ	John Hester	University of Florida
GKC 0.7	FOF UEQ LTB	Tanel Tammet	Tallinn University of Technology
iProver 3.5	FOF FNT UEQ LTB	Konstantin Korovin (André Duarte, Edvard Holden)	University of Manchester
JavaRes 1.3.0	FOF	Adam Pease (Stephan Schulz)	Articulate Software
LEO-II 1.7.0	THF	Alexander Steen (Christoph Benzmüller)	University of Luxembourg

Table 1: The ATP systems and entrants

ATP System	Divisions	Entrants (Associates)	Entrant's Affiliation
Leo-III 1.6	THF SLH LTB	Alexander Steen (Christoph Benzmüller)	University of Luxembourg
Prover9 1109a	FOF (demo)	CASC (William McCune, Bob Veroff)	CASC fixed point
RPx 1.0	FOF FNT	Anders Schlichtkrull (Jasmin Blanchette, Dmitriy Traytel)	Aalborg University Copenhagen
SATCoP 0.1	FOF	Michael Rawson (Giles Reger)	University of Manchester
Twee 2.4	FOF UEQ	Nick Smallbone (Koen Claessen)	Chalmers University of Technology
Vampire 4.5	FOF FNT (demo)	CASC	CASC-J10 winner
Vampire 4.6	THF FOF FNT UEQ SLH LTB	Giles Reger (Martin Suda, Andrei Voronkov, Evgeny Kotelnikov, Laura Kovacs, Martin Riener, Michael Rawson, Bernhard Gleiss, Jakob Rath, Ahmed Bhayat, Johannes Schoisswohl, Petra Hozzova, Marton Hajdu)	University of Manchester
Zipperposition 2.0	THF (demo)	CASC	CASC-J10 winner
Zipperposition 2.1	THF FOF SLH LTB (demo)	Petar Vukmirović (Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, Sophie Tourret)	Vrije Universiteit Amsterdam

Table 2: The ATP systems and entrants, continued

- The TFA division was placed on hiatus.
- The "Slammer Hammer" (SLH) division was added, with a prize provided by Jasmin Blanchette's Matryoshka project[1].
- Only one proof by contradictory axioms, for each contradictory set, counted for the ranking in the LTB division.

The competition organizer was Geoff Sutcliffe, assisted by Martin Desharnais for the SLH and LTB divisions. CASC is overseen by a panel of knowledgeable researchers who are not participating in the event. The CASC-28 panel members were Peter Baumgartner, Pascal Fontaine, and Christoph Weidenbach. The competition was run on computers provided by StarExec at the University of Miami.[2] The CASC-28 web site provides access to resources used before, during, and after the event: `http://www.tptp.org/CASC/28`

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 web pages 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.

> *A Tense Note:* Attentive readers will notice changes between use of the past to present tenses in this document. Many parts of CASC are established and stable - these are described in the present tense (the rules are the rules). Aspects that are particular to CASC-28 are described in the past tense, so they make sense when reading this after the event.

2 Divisions

CASC is divided into divisions according to problem and system characteristics. There are competition divisions in which systems are explicitly ranked, and a demonstration division in which systems demonstrate their abilities without being ranked. Some divisions are further divided into problem categories, which makes it possible to analyse, at a more fine grained level, which systems work well for what types of problems. The competition rankings are made at only the division level.

2.1 The Competition Divisions

The competition divisions are open to ATP systems that meet the required system properties, described in Section 6.1. Each division uses problems that have certain logical, language, and syntactic characteristics, so that the ATP systems that compete in the division are, in principle, able to attempt all the problems in the division.

The **THF** division: Typed (monomorphic) Higher-order Form theorems (axioms with a provable conjecture). The THF division has two problem categories:

- The **TNE** category: THF with No Equality
- The **TEQ** category: THF with EQuality

[1] `https://matryoshka-project.github.io`

[2] http://starexec.ccs.miami.edu

The **FOF** division: First-Order Form theorems (axioms with a provable conjecture). The FOF division has two problem categories:

- The **FNE** category: FOF with No Equality
- The **FEQ** category: FOF with EQuality

The **FNT** division: First-order form Non-Theorems (axioms with a countersatisfiable conjecture, and satisfiable axiom sets). The FNT division has two problem categories:

- The **FNN** category: FNT with No equality
- The **FNQ** category: FNT with eQuality

The **UEQ** division: Unit EQuality clause normal form theorems (unsatisfiable clause sets).

The **SLH** division: Typed (monomorphic) higher-order theorems without arithmetic (axioms with a provable conjecture), generated by Isabelle's SledgeHammer system [55].

The **LTB** division: Theorems (axioms with a provable conjecture) from Large Theories, presented in Batches. A large theory has many functions and predicates, and many axioms of which typically only a few are required for the proof of a theorem. The problems in a batch are given to an ATP system all at once, and typically have a common core set of axioms. The batch presentation allows the ATP systems to load and preprocess the common core set of axioms just once, and to share logical and control results between proof searches. Each problem category might be accompanied by a set of training problems and their solutions, taken from the same source as the competition problems. The training data can be used for ATP system tuning during (typically at the start of) the competition. In CASC-28 the LTB division had one problem category:

- The **JJT** category: Problems generated by Isabelle's SledgeHammer system [55], from the "Jinja with Threads" entry in Isabelle's Archive of Formal Proofs (AFP).[3]

Five versions of each JJT problem were provided: a first-order form (FOF) version, a monomorphic typed first-order form (TF0) version, a polymorphic typed first-order form (TF1) version, a monomorphic typed higher-order form (TH0) version, and a polymorphic typed higher-order form (TH1) version. Systems could attempt as many of the versions as they wanted, in any order including in parallel, and a solution to any version counted as a solution to the problem. The FOF problems had 57 to 6613 axioms, the TF0 problems 51 to 6385 axioms, the TF1 problems 1031 to 1680 axioms, the TH0 problems 14 to 1149 axioms, and the TH1 problems 1019 to 1666 axioms. There were no common core sets of axioms.

Section 3.2 explains what problems are eligible for use in each division and category. Section 4 explains how the systems are ranked in each division.

2.2 The Demonstration Division

ATP systems that cannot run in the competition divisions for any reason (e.g., the system requires special hardware, the system is a previous winner, or the entrant is an organizer) can be entered into the demonstration division. Demonstration division systems can run on the competition computers, or the computers can be supplied by the entrant. The demonstration division results are presented along with the competition divisions' results, but might not be comparable with those results. The systems are not ranked, and no trophies or prizes are awarded.

[3] https://www.isa-afp.org/entries/JinjaThreads.html

3 Infrastructure

3.1 Computers

The competition computers had:

- Two octa-core Intel(R) Xeon(R) E5-2667, 3.20GHz CPUs
- 256GB memory
- The CentOS Linux release 7.4.1708 (Core) operating system, with
  Linux kernel 3.10.0-693.el7.x86_64.

One ATP system runs on one CPU at a time, with access to half (128GB) the memory. Systems can use all the cores on the CPU, which is advantageous in the divisions where a wall clock time limit was used.

3.2 Problems

3.2.1 Problem Selection

The problems for the THF, FOF, FNT, and UEQ divisions were taken from the TPTP Problem Library [118], version v7.5.0. The TPTP version used for CASC is released only after the competition has started, so that new problems have not been seen by the entrants. The problems have to meet certain criteria to be eligible for use. The problems used are randomly selected from the eligible problems based on a seed supplied by the competition panel:

- The TPTP tags problems that designed specifically to be suited or ill-suited to some ATP system, calculus, or control strategy as *biased*, and they are excluded from the competition.
- The problems are 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) [135]. Difficult problems with a rating in the range 0.21 to 0.99 are eligible. Problems of lesser and greater ratings might also be eligible in some divisions 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 for computing the problem ratings.
- The selection is constrained so that no division or category contains an excessive number of very similar problems [85].
- 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, 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 for the SLH division were were generated by Isabelle's SledgeHammer system. 100 problems were sliced out from the largest theory (measured by the number of proof goals) in each of 50 randomly selected sessions in Isabelle's Archive of Formal Proofs (AFP), providing 5000 problems that could be used. 720 appropriately difficult problems were chosen based on performance data similar to that in the TSTP.

The problems for the LTB division are taken from various sources, chosen for each CASC. Each problem category is based on one source. Entrants are expected to honestly not use publicly available problem sets for tuning before the competition. The process for selecting problems depends on the problem source. The JJT category problems were generated by Isabelle's SledgeHammer system from the "Jinja with Threads" entry in the AFP. The axioms for each problem were selected from the HOL library and relevant AFP entries using the MeSh filter that combines the MePo [52] and MaSh [46] filters. The generation aimed to export problems with 1024 axioms, but more or less axioms occur in the problems because sometimes the axiom selection fails to find enough relevant axioms, and sometimes monomorphization [17] yields many new axioms. A total of 18474 problems were exported in each of the five versions, and appropriately difficult problems were chosen based on performance data similar to that in the TSTP.

3.2.2 Number of Problems

In the TPTP-based divisions, the minimal numbers of problems that must be used in each division and category to ensure sufficient confidence in the competition results are determined from the numbers of eligible problems in each division and category [29]. The competition organizers have to ensure that there are sufficient computers available to run the ATP systems on this minimal number of problems. The minimal numbers of problems are used in determining the time limits imposed on solution attempts - see Section 3.3. The numbers of problems to be used in each division of the competition are determined from the number of computers available, the time allocated to the competition, the number of ATP systems to be run on the competition computers over the divisions, and the time limits imposed on solution attempts, according to the following relationship:

$$NumberOfProblems = \frac{NumberOfComputers * TimeAllocated}{NumberOfATPSystems * TimeLimit}$$

It is a lower bound on the number of problems because it assumes that every system uses all of the time limit for each problem. Since some solution attempts succeed before the time limit is reached, more problems can be used. The numbers of problems used in the categories in the various divisions are (roughly) proportional to the numbers of eligible problems, after taking into account the limitation on very similar problems, determined according to the judgement of the competition organizers.

In the SLH division the number of problems was determined in consultation with Jasmin Blanchette (who sponsored the prize for the division). Calculations similar to those used for the TPTP-based divisions were used.

In the LTB division the number of problems in each problem category is determined by the number of problems in the corresponding problem set. In CASC-28 the JJT problem category had 10000 problems (with five versions of each problem).

3.2.3 Problem Preparation

The problems are given to the ATP systems in TPTP format, with `include` directives. 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 in the TPTP-based divisions are obfuscated by:

- stripping out all comment lines, including the problem header
- randomly reordering the formulae/clauses (the `include` directives are left before the formulae, type declarations and definitions are kept before the symbols' uses)
- randomly swapping the arguments of associative connectives, and randomly reversing implications

- randomly reversing equalities

In the non-TPTP-based divisions the formulae are not obfuscated, thus allowing the ATP systems to take advantage of natural structure that occurs in the problems.

In the TPTP-based divisions the problems are given in increasing order of TPTP difficulty rating. In the SLH division the problems were given in a roughly estimated order of difficulty. In the LTB division the problems in each batch are given in their natural order in the problem source.

3.2.4 Batch Specification Files

The problems for each problem category of the LTB division are listed in a *batch specification* file, containing containing global data lines and one or more batch specifications. The global data lines are:

- A problem category line of the form
  `division.category LTB.`*category_mnemonic*
  For CASC-28 it was
  `division.category LTB.JJT`
- The name of a `.tgz` file (relative to the directory holding the batch specification file) that contains training data in the form of problems in TPTP format and one or more solutions to each problem in TSTP format, in a line of the form
  `division.category.training_data` *tgz_file_name*
  For CASC-28 it was
  `division.category.training_data TrainingData/TrainingData.JJT.tgz`
  The `.tgz` file expands in place to three directories: `Axioms`, `Problems`, and `Solutions`. `Axioms` contains all the axiom files that are used in the training and competition problems. `Problems` contains the training problems. `Solutions` contains a subdirectory for each of the `Problems`, containing TPTP format solutions to the problem. The language of a solution might not be the same as the language of the problem, e.g., a proof to a THF problem might be written in FOF, or the proof of a TFF problem might be written in THF.

Each batch specification consists of:

- A header line `% SZS start BatchConfiguration`
- A specification of whether or not the problems in the batch must be attempted in order is given, in a line of the form
  `execution.order` *ordered/unordered*
  If the batch is ordered the ATP systems may not start any attempt on a problem, including reading the problem file, before ending the attempt on the preceding problem. For CASC-28 it was
  `execution.order unordered`
- A specification of what output is required from the ATP systems for each problem, in a line of the form
  `output.required` *space_separated_list*
  where the available list values are the SZS values `Assurance`, `Proof`, `Model`, and `Answer`. For CASC-28 it was
  `output.required Proof`.
- The wall clock time limit per problem, in a line of the form
  `limit.time.problem.wc` *limit_in_seconds*

A value of zero indicates no per-problem limit. For CASC-28 it was
`limit.time.problem.wc 0`
- The overall wall clock time limit for the batch, in a line of the form
`limit.time.overall.wc` *limit_in_seconds*
- A terminator line `% SZS end BatchConfiguration`
- A header line `% SZS start BatchIncludes`
- `include` directives that are used in every problem. All the problems in the batch have these `include` directives, and can also have other `include` directives that are not listed here. In CASC-28 there were no included axiom files.
- A terminator line `% SZS end BatchIncludes`
- A header line `% SZS start BatchProblems`
- Pairs of problem file names (relative to the directory holding the batch specification file), and output file names where the output for the problem must be written. The output files must be written in the directory specified as the second argument to the `starexec_run` script (the first argument is the name of the batch specification file). For CASC-28, see the additional notes below.
- A terminator line `% SZS end BatchProblems`

Additional Notes for CASC-28

- In the `BatchProblems` section, the multiple versions of each problem were specified using UNIX `*` globbing, e.g., `JJT00001*.p`. The versions of each problem had extensions as follows: the FOF version used `+1`, the TF0 version used `_1`, the TF1 version used `_2`, the TH0 version used `^1`, and the TH1 version used `^2`.
- Proof output had to identify which version of the problem was solved - see Section 6.1.
- In the `BatchIncludes` section (not in problem files), multiple versions of included axiom files could be specified using UNIX `*` globbing. For a given problem, systems could use only the axiom files whose version matched that of the problem file (there might be none). Using any other versions could lead to weird results.

3.3 Resource Limits

In the TPTP-based divisions, a wall clock time limit was imposed for each problem. The minimal time limit for each problem is 120s. The maximal time limit for each problem is determined using the relationship used for determining the number of problems, with the minimal number of problems as the $NumberOfProblems$. The time limit is chosen as a reasonable value within the range allowed, and is announced at the competition. There were no CPU time limits (i.e., using all cores on the CPU made sense).

In the SLH division, a CPU time limit was imposed for each problem. The minimal time limit per problem was 15s and the maximal time limit per problem was 90s. The time limit was chosen as a reasonable value within the range allowed, and was announced at the competition.

In the LTB division, wall clock time limits are imposed. For each batch there might be a wall clock time limit for each problem, provided in the configuration section at the start of each batch. If there is a wall clock time limit for each problem, the minimal limit for each problem is 15s, and the maximal limit for each problem is 90s. For each batch there is an overall wall clock time limit, provided in the configuration section at the start of each batch. The overall limit is proportional to the number of problems in the batch, e.g., (but not necessarily) the batch's per-problem time limit multiplied by the number of problems in the batch. Time spent before starting the first problem of a batch (e.g., preloading and analysing the batch axioms), and times spent between the end of an attempt on a problem and the starting of the next (e.g., learning from a proof just found), are not part of the times taken on the individual problems, but are part of the overall time taken. There are no CPU time limits.

4 System Evaluation

For each ATP system, for each problem, four items of data are recorded: whether or not the problem was solved, the CPU time taken, the wall clock time taken, and whether or not a solution (proof or model) was output.

The systems are ranked in the competition divisions according to the number of problems solved with an acceptable solution 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 criteria, 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

$$1 - FractionOfSystemsThatSolvedTheProblem$$

  and a system's overall SotAC is its average for the problems it solves but that are not solved by all the systems.
- The *core usage* is the average of the ratios of CPU time to wall clock time used, over the problems solved. This measures the extent to which the systems take advantage of multiple cores.
- The *efficiency* measure combines the number of problems solved with the time taken. It is the average of the inverses of the times taken for problems solved, multiplied by the fraction of problems solved. This can be interpreted intuitively as the average of the solution rates for problems solved, 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.

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 (see Section 6.1 regarding soundness checking before the competition). 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 (of divisions ranked by the numbers of solutions output) 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.

5 System Entry

To be entered into CASC systems must be registered 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.

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 division winners from the previous CASC are automatically entered into their demonstration divisions, to provide benchmarks against which progress can be judged. Prover9 1109a is automatically entered into the FOF division, to provide a fixed-point against which progress can be judged.

5.1 System Descriptions

A system description has to be provided for each ATP system entered, 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 6.1).
- 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 7).

5.2 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 solutions are acceptable (see Section 4).

Proof/model samples are required as follows:

- THF: `SET014^4`
- FOF: `SEU140+2`
- FNT: `NLP042+1` and `SWV017+1`
- UEQ: `BOO001-1`

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

6 System Requirements

6.1 System Properties

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. At some time before the competition all the systems in the competition divisions are tested for soundness. Non-theorems are submitted to the systems in the THF, FOF, UEQ, SLH, and LTB divisions, and theorems are submitted to the systems in the FNT division. Finding 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.
- 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. (It's OK to store information about LTB training problems.) 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 or their solutions. 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

- In all divisions except LTB the solution output must be to `stdout`. In the LTB division the solution output must be to the named output file for each problem, in the directory specified as the second argument to the `starexec_run` script. If multiple attempts are made on a problem in an unordered batch, each successive output file must overwrite the previous one.
- In the LTB division the systems must print SZS notification lines to `stdout` when starting and ending work on a problem (including any cleanup work, such as deleting temporary files). For example

  ```
  % SZS status Started for CSR075+2.p
    ... (system churns away, progress output to file)
  % SZS status GaveUp for CSR075+2.p
  % SZS status Ended for CSR075+2.p
  ```

  ... and later in another attempt on that problem ...

  ```
  % SZS status Started for CSR075+2.p
    ... (system churns away, result and solution overwrites file)
  % SZS status Theorem for CSR075+2.p
  % SZS status Ended for CSR075+2.p
  ```

- 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 [99] for this. For example

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

  or

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

  In the LTB division this line must be the last line output before the ending notification line. The line must also be output to the output file.
- 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 [126] 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.

6.2 System Delivery

Entrants must email a StarExec installation package to the competition organizers by the system delivery deadline. The installation package must be a .tgz file containing only the components necessary for running the system (i.e., not including source code, etc.). The entrants must also email a .tgz file containing the source code and any files required for building the StarExec installation package to the competition organizers by the system delivery deadline.

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.

After the competition all competition division systems' source code is made publicly available on the CASC web site. In the demonstration division the entrant specifies whether or not the source code is placed on the CASC web site. An open source license is encouraged.

6.3 System Execution

Execution of the ATP systems is controlled by StarExec. The jobs are queued onto the computers so that each CPU is running one job at a time. All attempts at the Nth problems in all the divisions and categories are started before any attempts at the (N+1)th problems.

A system has solved a problem iff it outputs its termination string within the time limit, and a system has produced a proof/model iff it outputs its end-of-solution string within the time limit. The result and timing data is used to generate an HTML file, and a web browser is used to display the results.

The execution of the demonstration division systems is supervised by their entrants.

7 The ATP Systems

These system descriptions were written by the entrants.

7.1 CSE 1.4

Feng Cao
Southwest Jiaotong University, China

Architecture

CSE 1.4 is a developed prover based on the last version - CSE 1.3. 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) [147]. 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 1.4 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 1.4 has been improved compared with CSE 1.3, mainly from the following aspects:

1. Optimization of the contradiction separation algorithm based on full usage of clauses, which is able to evaluate whether the input clause has been fully used in the deduction process.

2. Optimization of the contradiction separation algorithm based on optimized deduction path, which is able to increase the deduction usage of original clauses.

Internally CSE 1.4 works only with clausal normal form. The E prover [78] is adopted with thanks for clausification of full first-order logic problems during preprocessing.

Strategies

CSE 1.4 inherited most of the strategies in CSE 1.3. The main new strategies are:

- Deduction control strategy. Dynamically evaluate the complexity of deduction based on the unified clause structure.

- CSCs control strategy. Evaluate whether the CSC generated in the deduction process are retained according its term structure.

- Clause selection strategy. Select different types of clauses to participate in S-CS deduction according to the dynamic sorting method.

Implementation

CSE 1.4 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 1.4 has made some improvements compared to CSE 1.3, and so we expect a better performance in this year's competition.

Acknowledgement: Development of CSE 1.4 has been partially supported by the National Natural Science Foundation of China (NSFC) (Grant No.61673320) and The General Research Project of Jiangxi Education Department (Grant No. GJJ200818).

7.2 CSE_E 1.3

Peiyao Liu
Southwest Jiaotong University, China

Architecture

CSE_E 1.3 is an automated theorem prover for first-order logic, combining CSE-F 1.0 and E 2.5, where CSE-F is based on the Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction (S-CS) [147] and E is mainly based on superposition. The combination mechanism is like this: E and CSE-F are applied to the given problem sequentially. If either prover solves the problem, then the proof process completes. If neither CSE-F nor E can solve the problem, some inferred clauses with no more than two literals, especially unit clauses, from CSE-F are 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-F and E, and produce a better performance. Concretely, CSE-F 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-F part of CSE_E 1.3 uses almost the same strategies as in the CSE-F 1.0 standalone, e.g., clause/literal selection, strategy selection, and CSC strategy. The only difference is that equality handling strategies of CSE-F 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 1.3 is implemented mainly in C++, and Java is used for batch problem running implementation. The job dispatch between CSE-F and E is implemented in C++.

Expected Competition Performance

We expect CSE_E 1.3 to solve some hard problems that E cannot solve and have a satisfying performance.

Acknowledgement: Development of CSE-F 1.0 has been partially supported by the National Natural Science Foundation of China (NSFC) (Grant No.61673320). Acknowledgements to Prof. Stephan Schulz for his kind permission on using his E prover that makes CSE_E possible.

7.3 CSE-F 1.0

Peiyao Liu
Southwest Jiaotong University, China

Architecture

CSE-F 1.0 is an automated theorem prover for first-order logic without equality mainly based on a novel inference mechanism, called as Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction (S-CS) [147], which is able to handle multiple (two or more) clauses dynamically in a synergized way in one deduction step, while binary resolution is its special case.

CSE-F 1.0 is the successor of CSE 1.3, and retains the advantages of CSE 1.3. At the same time, it has improved the deduction framework, and implements a new S-CS inference algorithm. With this new inference algorithm, binary clauses are fully used until no binary clause is added to the contradiction during each deduction epoch, given the fact that unit clauses are helpful for proof search. Based on this, CSE-F 1.0 can solve some hard problems that CSE 1.3 cannot.

CSE-F 1.0 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.

Internally, CSE-F 1.0 works only with clausal normal form. The E prover [78] is adopted with thanks for clausification of full first-order logic problems during preprocessing.

Strategies

CSE-F 1.0 inherited most of the strategies in CSE 1.3. It introduces some new strategies as well, e.g., pre-processing, clause (literal) selection and clause filtering. At the same time, some strategies in CSE 1.3 were optimized. The main new strategies are:

- Pre-processing strategy. Mark clauses containing independent literals that can form complementary pairs with certain unit clause for subsequent pre-processing.

- Clause(literal) selection strategy. This strategy mainly considers the comprehensive weight involving complexity, function nesting level, deduction distance of clause(literal).

- Clause filtering strategy. Filter out the binary clauses that do not meet deduction conditions.

The main optimized strategies include:

- Clause deduction distance. Dynamically update clause deduction distance in the process of constructing contradictions in order to better guide selection of clauses, rather than just as a pre-processing strategy.
- Contradiction separation clause (CSC) strategy. Strictly limit the number of literals in CSC. Initially limit the number of literals in CSC to zero, and gradually expand the number of literals in CSC during the deduction process.

Implementation

CSE-F 1.0 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 supports 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-F 1.0 adopts a new algorithm framework, and so we expect a better performance than CSE 1.3 in this year's competition.

Acknowledgement: Development of CSE-F 1.0 has been partially supported by the National Natural Science Foundation of China (NSFC) (Grant No.61673320).

7.4 cvc5 1.0

Andrew Reynolds
University of Iowa, USA

Architecture

cvc5 is the successor of CVC4 [6]. It is an SMT solver based on the CDCL(T) architecture [53] 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 [64]. 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 [65]. If no instances are added, it reports "satisfiable".

cvc5 has native support for problems in higher-order logic, as described in [5]. 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 [63, 4], numerative instantiation [62] and E-matching. vc5 uses a handful of orthogonal trigger selection strategies for E-matching, nd 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 has support for fine-grained proofs, which will be generated in solutions this year. Due to performance overhead in generating proofs, we expect the performance of cvc5 to degrade slightly with respect to CVC4 last year. Otherwise, the first-order theorem proving and finite model finding capabilities of cvc5 have undergone minor improvements, particularly to enumerative instantiation. Hence, cvc5 will perform comparably to previous versions of CVC4.

7.5 Drodi 3.1.5

Oscar Contreras
Amateur programmer, Spain

Architecture

Drodi 3.1 is a very basic and lightweight automated 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 [54], AVATAR architecture with a SAT solver [142], Limited Resource Strategy [66], discrimination trees as well as KBO, non recursive and lexicographic reduction orderings. Drodi produces a (hopefully) verifiable proof in TPTP format.

Strategies

Drodi 3.1 has a fair number of selectable strategies including, but not limited to, the following:

- Otter, Discount and Limited Resource Strategy [66] saturation algorithms.
- A basic implementation of AVATAR architecture [142].
- Several literal and term reduction orderings.
- Several literal selection options.
- Several clause selection heuristics with adjustable selection ratios, including several types of clause weight queues and one age queue.
- Classical clause relevancy pruning.
- 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.

Drodi's integrated learning functions are a generalization of ENIGMA [39, 40]. It uses a general learning file applicable to any kind of problems during CASC competition. Literal 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 general properties of functions and non equality predicates are stored in clause feature vectors. Predicates are differentiated from functions. In addition the following properties are also stored:

- Predicate or function is in the conjecture.
- Predicate or function is in the problem file but it is not in the conjecture.
- Predicate or function is only in axiom files.

Implementation

Drodi 3.1 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

This is the first time that Drodi participates in CASC. It will enter the FOF and UEQ divisions. Program tests with 2020 CASC-J10 problems indicate that Drodi will score in the second half of the score table, probably in the last or next to last position.

7.6 E 2.5

Stephan Schulz
DHBW Stuttgart, Germany

Architecture

E 2.5pre [74, 78, 79] is a purely equational theorem prover for many-sorted first-order logic with equality. It consists of an (optional) clausifier for pre-processing 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. 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 [16] 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-J10, E implements a 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 130 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 [77]. Superposition and backward rewriting use fingerprint indexing [76], 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 [47, 48]. The prover and additional information are available at

```
https://www.eprover.org
```

Expected Competition Performance

The inference core of E 2.5pre has been slightly modified since last years pre-release. We have also been able to evaluate some more different search strategies. As a result, we expect performance to be somewhat better than in the last years, especially in UEQ. The system is expected to perform well in most proof classes, but will at best complement top systems in the disproof classes.

7.7 E 2.6

Stephan Schulz
DHBW Stuttgart, Germany

Architecture

E [74, 78, 79] is a purely equational theorem prover for many-sorted first-order logic with equality, with some extensions for higher-order logic. It consists of an (optional) clausifier for pre-processing 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. 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 [16] 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-28, E implements a 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 [77]. Superposition and backward rewriting use fingerprint indexing [76], 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 [47, 48]. The prover and additional information are available at

`https://www.eprover.org`

Expected Competition Performance

The inference core of E 2.6 has been slightly modified since last year. We have also been able to evaluate some more different search strategies. As a result, we expect performance to be somewhat better than in the last years, especially in UEQ. The system is expected to perform well in most proof classes, but will at best complement top systems in the disproof classes.

7.8 Ehoh 2.7

Petar Vukmirović
Vrije Universiteit Amsterdam, The Netherlands

Architecture

Ehoh is a higher-order superposition-based theorem prover implementing lambda-free higher-order superposition [7]. Recently, Ehoh has been extended to support not only lambda-free, but full higher-order syntax. Internally, Ehoh unfolds all definitions of predicate symbols, lifts lambdas and removes all Boolean subterms through a FOOL-like [44] preprocessing transformation. After these steps are performed, the problem lies in the lambda-free fragment and the standard lambda-free superposition applies. Ehoh also supports TFX `$ite` and `$let` syntax. On the reasoning side, modest additions to the calculus have been made: We implemented rules NegExt, PosExt and Ext-* family of rules described by Bentkamp et al. [8]. Full support for lambda-terms and calculus-level treatment of Boolean terms is expected in the next version of Ehoh.

Strategies

The system uses exactly the same portfolio of strategies as E 2.7, with the only difference that rules NegExt, PosExt and Ext-* family rules are turned on regardless of the chosen strategy.

Implementation

Ehoh 2.7 shares the codebase of E 2.7: It is a version of E prover compiled with compile-time option ENABLE_LFHO enabled. Ehoh is available from

`https://github.com/eprover/eprover`

which includes more details on Ehoh's compilation and installation.

Expected Competition Performance

The prover is expected to have poor performance on THF problems, slightly worse than CVC4 1.8. On SLH problems, it is expected to perform better, on a par with Zipperposition.

7.9 Etableau 0.67

John Hester
University of Florida, USA

Architecture

Etableau is a theorem prover for first order logic based on combining the strong connection calculus and the superposition calculus. Etableau centers the idea of local variables in tableau proof search. Branches that are local (contain only local variables) are sent to the core proof procedure of E. Saturating along branches allows the automatic generation of unit lemmata.

Strategies

Etableau uses a depth first branch selection function, and maintains a small number of distinct tableaux in memory simultaneously. During superposition proof search on local branches, E's "–auto" mode is used. Etableau can backtrack when proof search fails, and remembers previous attempts at using superposition search on branches so that the search does not have to repeat itself.

Implementation

Etableau is implemented in C and compiled alongside E, using E as a library and orthogonal prover. This allows Etableau to use the clause and formula datatypes of E, facilitating directly calling the proof search functions of E with clauses from the tableau rather than starting a new process for every time an attempt to saturate a branch is made. Etableau also uses the clausification and preprocessing of E. Etableau can be obtained from

`https://github.com/hesterj/Etableau`

Expected Competition Performance

Etableau will solve fewer problems than E, but may solve some that others cannot. Etableau will solve significantly more problems than last year.

7.10 GKC 0.7

Tanel Tammet
Tallinn University of Technology, Estonia

Architecture

GKC [138] is a resolution prover optimized for search in large knowledge bases. It 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, see [139]. We envision natural language question answering systems as the main potential application for these specialized methods.

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, see [140].

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 does not currently implement any propositional inferences or instance generation. It only looks for proofs and does not try to show non-provability.

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 Whitedb 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

Compared to the performance in previous CASC, GKC 0.7 should perform somewhat better. In particular, more search strategies have been implemented and the selection of search strategies is wider and more varied. The core algorithms and data structures remain the same. We expect GKC to be in the middle of the final ranking for FOF and below the average in UEQ and LTB. We expect GKC to perform well on very large problems.

7.11 iProver 3.5

Konstantin Korovin
University of Manchester, United Kingdom

Architecture

iProver interleaves instantiation calculus Inst-Gen [43, 42, 26] with ordered resolution and superposition calculi [22]. iProver approximates first-order clauses using propositional abstractions which are solved using MiniSAT [25] and refined using model-guided instantiations. iProver also implements a general abstraction-refinement framework for under-and over-approximations of first-order clauses [31, 32]. First-order clauses are exchanged between calculi during the proof search.

Recent features in iProver include:

- AC joinability and AC normalisation [23].

- Support for quantified linear and non-linear arithmetic.
- Superposition calculus with simplifications including: demodulation, light normalisation, subsumption, subsumption resolution and global subsumption. iProver's simplification set up [22] is tunable via command line options and generalises common architectures such as Discount or Otter.
- HOS-ML framework for learning heuristics using combination of hyper-parameter optimisation and dynamic clustering together with schedule optimisation using constraint solving [37, 36]

Strategies

iProver has around 100 options to control the proof search including options for literal selection, passive clause selection, frequency of calling the SAT solver, simplifications and options for combination of instantiation with resolution and superposition. For the competition HOS-ML [37] was used to build a multi-core schedule from heuristics learnt over a sample of FOF problems.

Implementation

iProver is implemented in OCaml and for the ground reasoning uses MiniSat [25] and Z3 [21]. iProver accepts FOF, TFF and CNF formats. Vampire [45, 33] and E prover [78] are used for proof-producing clausification of FOF/TFF problems. Vampire is also used for SInE axiom selection [35] in the LTB division. iProver is available at:

```
http://www.cs.man.ac.uk/~korovink/iprover/
```

Expected Competition Performance

We expect improvement in performance compared to the previous year due to improvements in superposition, AC reasoning, simplifications and heuristic selection. Heuristic tuning is still work in progress and in particular we reused heuristics trained for FOF in the LTB division which might be not ideal as the nature of the problems is quite different.

7.12 JavaRes 1.3.0

Adam Pease
Articulate Software, USA

Architecture

JavaRes is a simple, resolution-based theorem prover, primarily created for teaching theorem proving. It implements the basic calculus from Robinson's seminal paper [67], extended with negative literal selection and some redundancy elimination as described by Bachmair and Ganziner [3]. The core is a given-clause based clausal saturation algorithm. The system also supports full first-order input via clausification, and equality handling via automatic addition of equality axioms.

Strategies

JavaRes includes all the optimization strategies in PyRes. For clause selection it implements two methods, which are combined. The most basic is a first-in-first-out (FIFO) strategy that will eventually try every clause. A symbol-counting strategy picks the clause with the fewest symbols. This results in a strong bias to smaller clauses while ensuring that all clauses will eventually be tried. JavaRes supports indexing for subsumption and resolution. Subsumption removes clauses from the set of clauses to be processed (called “forward subsumption”) and from the set already processed (”backwards subsumption”) thereby decreasing the problem search space. More general clauses subsume more specific ones. Indexing is used and employs records with signs and predicate symbols only, so that potential clauses can be accepted or rejected more rapidly than attempting unification. JavaRes also implements PyRes’ approach to literal selection. Largest literal selection is the default strategy. For large theories, JavaRes has implemented the SInE algorithm [35] although performance on the LTB problems is so poor for this simple prover, compared to modern provers such as E and Vampire that we do not enter JavaRes in that division.

Implementation

JavaRes is largely a re-implementation of PyRes, but in Java, and with additional features that are not part of the core inference algorithm. Additional features include implementation of SInE, parsing of SUO-KIF syntax, graphical proof output using GraphViz. The implementation is designed to be straightforward and doesn’t include any of the newer Java language features such as lambda expressions. It is available from

`https://github.com/ontologyportal/JavaRes`

Expected Competition Performance

JavaRes is faster than PyRes simply due to the implementation language. It solves a few more problems than PyRes but is significantly faster on problems that both provers solve. It is inferior compared to most modern superposition-based provers. It is expected to perform reasonably well on problems without equality.

7.13 LEO-II 1.7.0

Alexander Steen
University of Luxembourg, Luxembourg

Architecture

LEO-II [11], the successor of LEO [10], 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 [9]. 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 [73]. 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 [12]. In fact, the development of LEO-II has largely paralleled the development of the TPTP THF language and related infrastructure [125]. 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 [13] 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.

7.14 Leo-III 1.6

Alexander Steen
University of Luxembourg, Luxembourg

Architecture

Leo-III [81], the successor of LEO-II [11], 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 on 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 which 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 [6] and E [74, 78] 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.

For the LTB division, Leo-III is augmented by an external Python3 driver which schedules Leo-III on the batches.

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 naive time slicing approach of some of these strategies since last CASC.

Implementation

Leo-III utilizes and instantiates the associated LeoPARD system platform [146] 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 LeoPARD'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 its polymorphic variants [18, 41].

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 that 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 1.6 only marginally improves the previous release by fixing some bugs; also the old (and very slow) ANTLR-based parser was replaced by a new (hand-crafted) TPTP parser. As CASC is using wall clock (WC) time instead of CPU time usage in all divisions (except for SLH), the Java VM version of Leo-III is used in the competition (as opposed to a native build used last year). We hope that the JRE performs - after a slow start-up - quite well on longer runs (wrt. WC time). We do not expect Leo-III to be competitive in the SLH division as it imposes strong CPU time limits that Leo-III's JRE will quickly exceed. For LTB and THF, we expect a similar performance as in last year's CASC.

In the LTB mode, Leo-III is testing a preliminary SinE-based axiom selection. Stemming from Leo-III's support for polymorphic HOL reasoning, we expect a reasonable performance.

On the other hand Leo-III's performance for reasoning with a large number of axioms is quite poor. Leo-III's LTB mode does not do any learning and/or analysis of the learning samples.

7.15 Prover9 1109a

Bob Veroff on behalf of William 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 [51]. 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 [50]. (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 [75] 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.

7.16 RPx 1.0

Anders Schlichtkrull
Aalborg University Copenhagen, Denmark

Architecture

RPx 1.0 [70, 68] implements the nondeterministic ordered resolution prover by Bachmair and Ganzinger [3]. It therefore uses their ordered resolution rule together with their definitions of tautology deletion, subsumption rules and reduction rules. The ordered resolution rule is restricted to binary resolution.

Strategies

The prover loop is loosely modelled after the one described by Voronkov [142]. It applies the ordered resolution rule together with tautology deletion, subsumption and reduction. This strategy is applied to all problems.

Implementation

The prover is built in Isabelle/HOL [RPXISA] as a data refinement from the calculus down to a fully executable program. This goes through the following refinement layers: (1) the nondeterministic ordered resolution prover by Bachmair and Ganzinger [72, 71, 69, 3], (2) a nondeterministic ordered resolution prover that enforces fairness, (3) a deterministic prover that represents clauses and the clauses database as lists, and commits to a strategy for assigning priorities to clauses, and (4) a fully executable program with a concrete datatype for atoms and executable definitions for most general unifiers, clause subsumption, and the order on atoms. Each layer's prover's soundness and completeness are proved by a refinement from the soundness and completeness of the previous layer. From the fourth layer Standard ML code is extracted using Isabelle's code generator [30]. The fourth layer uses several formalizations from IsaFoR [82, 141]. RPx employs the TPTP parser and clausifier of Metis [38]. Coupling Metis and the generated RPX code together required a simple conversion between their very similar datatypes for clauses. To be able to solve problems with equality in the competition, RPx uses a script written by Geoff Sutcliffe that uses the TPTP4X tool to add the axioms of equality into problems with equality. RPx is available at

```
https://github.com/anderssch/RPx
```

Expected Competition Performance

RPx competes in the divisions FOF and FNT. RPx uses a magnificent calculus but the data structures are mediocre. A benchmark done when developing RPx concluded that it is not a competitive prover. This was concluded by comparing its performance with that of Vampire, E and Metis. Vampire and E were far ahead. Metis also performed better, but by a smaller margin [70].

7.17 SATCoP 0.1

Michael Rawson
University of Manchester, United Kingdom

Architecture

SATCoP 0.1 [56] implements a typical connection-tableau system with a SAT twist: first-order clauses (partially) instantiated while building tableaux are continuously grounded and fed to a boolean satisfiability routine. When the growing set of propositional clauses becomes unsatisfiable, we have found a proof. In the meantime, ground information can influence search. Satisfying assignments focus SATCoP somewhat: goal literals are only attempted if their ground abstraction is assigned true. Ground information can also be used to control a combination of pseudo-random shuffling and iterative deepening. We note that this system has been developed slightly since [56]: the system is very new and there are still many directions and optimisations to explore.

Strategies

There are no specially-designed strategies in SATCoP. However, some parts of the system employ a pseudo-random number generator (PRNG), which can change proof search significantly. Therefore, to make use of multiple cores, we launch an appropriate number of threads, each using a different seed for their PRNG. Each thread runs to the time limit uninterrupted

Implementation

The system is implemented compactly in a few thousand lines of Rust. The internal SAT routine is by far the biggest bottleneck: we first try cheap stochastic local search, then fall back to PicoSAT [16] if we fail to find a satisfying assignment quickly.

See the website

`https://github.com/MichaelRawson/satcop/`

Expected Competition Performance

Based on the 2020 competition, we hope to prove at least 150 of the 2021 FOF problems. SATCoP can in principle attempt unit equality problems, but is not very good at them. It cannot show non-theorems, or deal with other logics.

7.18 Twee 2.4

Nick Smallbone
Chalmers University of Technology, Sweden

Architecture

Twee [80] 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 [49] 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 [49], and discharges individual cases by rewriting modulo a variable ordering.

Horn clauses are encoded as equations as described in [19]. For CASC, Twee accepts non-Horn problems but throws away all the non-Horn clauses.

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. The weights of critical pairs that correspond to Horn clauses are adjusted by the heuristic described in [CS18], section 5.

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 is quite strong at UEQ, and ought to compete with the top provers. It should perform better than last year, thanks to the goal-directed transformation mentioned above and some other performance improvements. It may suffer in COL (because its handling of existential goals is mediocre) and in RNG (where many problems are best solved with LPO or RPO). As Twee only supports Horn clauses it will do badly in FOF. It may get lucky and solve a few hard problems, especially if some mostly-equational problems show up.

7.19 Vampire 4.5

Giles Reger
University of Manchester, United Kingdom

Architecture

Vampire [45] 4.5 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 [59]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [142, 57]. 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 [60]. 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 4.5 provides a very large number of options for strategy selection. The most important ones are:

- Choices of saturation algorithm:
  - Limited Resource Strategy [66]
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [142]

- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals [34].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection. This has been extended with a layered clause selection approach [27].
- 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.
  - Use of Z3 with AVATAR to restrict search to ground-theory-consistent splitting branches [57].
  - Specialised theory instantiation and unification [61].
  - Extensionality resolution with detection of extensionality axioms
- For higher-order problems:
  - Translation to polymorphic first-order logic using applicative form and combinators.
  - A new superposition calculus [14] utilising a KBO-like ordering [15] 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 4.5 is implemented in C++. It makes use of minisat and Z3. See the website

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

for more information and access to the GitHub repository.

Expected Competition Performance

There are four areas of improvement in Vampire 4.5. Firstly, a new layered clause selection approach [27] gives Vampire more fine-grained control over clause selection, in particular the way in which clauses involving theory axioms are selected. Secondly, theory evaluation and instantiation methods have been overhauled. Thirdly, a new subsumption demodulation rule [28] improves support for reasoning with conditional equalities. Finally, higher-order reasoning (introduced in Vampire 4.4) has been rewritten based on a new superposition calculus [14] utilising a KBO-like ordering [15] for orienting combinator equations. Vampire 4.5 should be an improvement on Vampire 4.4.

7.20 Vampire 4.6

Giles Reger
University of Manchester, United Kingdom

There are only small changes between Vampire 4.5 and Vampire 4.6 in the tracks relevant to CASC. Most of our efforts have been spent on theory reasoning (which are not relevant as TFA is not running) and efforts to parallelise Vampire which are too immature for CASC this year. One significant engineering effort has been to incorporate higher-order and polymorphic reasoning into the "main branch" such that a single executable is used for all divisions.

Architecture

Vampire [45] 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 [59]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [142, 57].

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 [60]. 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 4.6 provides a very large number of options for strategy selection. The most important ones are:

- Choices of saturation algorithm:
  - Limited Resource Strategy [66]
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [142]
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals [34].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection. This has been extended with a layered clause selection approach [27].
- 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 [58].
  - Use of Z3 with AVATAR to restrict search to ground-theory-consistent splitting branches [57].
  - Specialised theory instantiation and unification [61].
  - 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 [14] utilising a KBO-like ordering [15] 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 4.6 is implemented in C++. It makes use of minisat and z3. See the website for more information and access to the GitHub repository:

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

Expected Competition Performance

Vampire 4.6 should be roughly the same as Vampire 4.5.

7.21 Zipperposition 2.0

Petar Vukmirović
Vrije Universiteit Amsterdam, The Netherlands

Architecture

Zipperposition is a superposition-based theorem prover for typed first-order logic with equality and higher-order logic. It is a pragmatic implementation of a complete calculus for Boolean-free higher-order logic [8]. 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; boolean reasoning [145]. The core architecture of the prover is based on saturation with an extensible set of rules for inferences and simplifications. Zipperposition uses a recently developed full higher-order unification algorithm that enables efficient integration of procedures for decidable fragments of higher-order unification [144]. The initial calculus and main loop were imitations of an old version of E [74], but there are many more rules nowadays. A summary of the calculus for integer arithmetic and induction can be found in [20].

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 different TPTP problems. Heuristics used in Zipperposition are inspired by efficient heuristics used in E. Portfolio mode differentiates higher-order problems from the first-order ones. If the problem is first-order all higher-order prover features are turned off. 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, and has been around for eight years. 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 E prover, running in lambda-free higher-order mode, as an end-game backend prover. The 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 [20].

Expected Competition Performance

The prover is expected to have average performance on FOF, similar to Prover9, and a good performance on THF, at the level of last-year's CASC winner.

7.22 Zipperposition 2.1

Petar Vukmirović
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 [7]. 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 [24]; pragmatic boolean reasoning [145]. 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 [144]. The initial calculus and main loop were imitations of an old version of E [74]. With the implementation of higher-order superposition, the main loop had to be adapted to deal with possibly infinite sets of unifiers [143]. A summary of the calculus for integer arithmetic and induction can be found in [20].

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 different TPTP problems. Heuristics used in Zipperposition are inspired by efficient heuristics used in E. A detailed overview of various calculus extensions used by the strategies is available [143]. Portfolio mode differentiates higher-order problems from the first-order ones. 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. 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, and has been around for nine years. 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 E prover, running in lambda-free higher-order mode, as an end-game backend prover. The 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 [20].

The 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 [20].

Expected Competition Performance

The prover is expected to have average performance on FOF. It is expected to perform well at THF, at least as good as last-year's version. In the SLH and LTB divisions we expect reasonable performance, on a par with Ehoh.

8 Conclusion

The CADE-28 ATP System Competition was the twenty-sixth large scale competition for classical logic ATP systems. The organizer believes 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] L. Bachmair, H. Ganzinger, D. McAllester, and C. Lynch. Resolution Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 19–99. Elsevier Science, 2001.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] A. Bentkamp, J. Blanchette, P. Vukmirovic, and U. Waldmann. Superposition with Lambdas. In P. Fontaine, editor, *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 55–73. Springer-Verlag, 2019.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] J. Blanchette. *Automatic Proofs and Refutations for Higher-Order Logic*. PhD thesis, Technische Universität München Lehrstuhl für Logik und Verifikation, Munich, Germany, 2015.

[18] 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.

[19] K. Claessen and N. Smallbone. Efficient Encodings of First-Order Horn Formulas in Equational Logic. 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.

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

[21] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. Ramakrishnan and J. Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 4963 in Lecture Notes in Artificial Intelligence, pages 337–340. Springer-Verlag, 2008.

[22] A. Duarte and K. Korovin. Implementing Superposition in iProver. In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, number 12167 in Lecture Notes in Artificial Intelligence, pages 388–397, 2020.

[23] A. Duarte and K. Korovin. AC Simplifications and Closure Redundancies in the Superposition Calculus. In A. Das and S. Negri, editors, *Proceedings of the 30th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, Lecture Notes in Artificial Intelligence, page To appear. Springer-Verlag, 2021.

[24] 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.

[25] 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.

[26] H. Ganzinger and K. Korovin. New Directions in Instantiation-based Theorem Proving. In P. Kolaitis, editor, *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*, pages 55–64. IEEE Press, 2003.

[27] 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.

[28] L. Gleiss, B. Kovacs and J. Rath. Subsumption Demodulation in First-Order Theorem Proving. 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 297–315, 2020.

[29] M. Greiner and M. Schramm. A Probablistic Stopping Criterion for the Evaluation of Benchmarks. Technical Report I9638, Institut für Informatik, Technische Universität München, München, Germany, 1996.

[30] F. Haftmann and T. Nipkow. Code Generation via Higher-Order Rewrite Systems. In M. Blume, N. Kobayashi, and G Vidal, editors, *Proceedings of the 10th International Symposium on Functional and Logic Programming*, number 6009 in Lecture Notes in Computer Science, pages 103–117. Springer-Verlag, 2010.

[31] J. Hernandez and K. Korovin. An Abstraction-Refinement Framework for Reasoning with Large Theories. 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 663–679, 2018.

[32] J. Hernandez and K. Korovin. Towards an Under-Approximation Abstraction-Refinement for Reasoning with Large Theories. In A. Bolotov and F. Kammueller, editors, *Proceedings of the 26th Automated Reasoning Workshop*, page To appear, 2019.

[33] K. Hoder, Z. Khasidashvili, K. Korovin, and A. Voronkov. Preprocessing Techniques for First-Order Clausification. In G. Cabodi and S. Singh, editors, *Proceedings of the Formal Methods in Computer-Aided Design 2012*, pages 44–51. IEEE Press, 2012.

[34] 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.

[35] 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.

[36] E. Holden and K. Korovin. SMAC and XGBoost your Theorem Prover. In T. Hales, C. Kaliszyk, R. Kumar, S. Schulz, and J. Urban, editors, *Proceedings of the 4th Conference on Artificial Intelligence and Theorem Proving*, pages 93–95, 2019.

[37] E. Holden and K. Korovin. Heterogeneous Heuristic Optimisation and Scheduling for First-Order Theorem Proving. In F. Kamareddine and C. Sacerdoti, editors, *Proceedings of the 14th International Conference on Intelligent Computer Mathematics*, Lecture Notes in Artificial Intelligence, page To appear. Springer-Verlag, 2021.

[38] J. Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In M. Archer, B. Di Vito, and C. Munoz, editors, *Proceedings of the 1st International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.

[39] 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.

[40] 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.

[41] 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.

[42] K. Korovin. iProver - An Instantiation-based Theorem Prover for First-order Logic (System Description). 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 292–298, 2008.

[43] K. Korovin. Inst-Gen - A Modular Approach to Instantiation-based Automated Reasoning. In A. Voronkov and C. Weidenbach, editors, *Programming Logics, Essays in Memory of Harald Ganzinger*, number 7797 in Lecture Notes in Computer Science, pages 239–270. Springer-Verlag, 2013.

[44] E. Kotelnikov, L. Kovacs, G. Reger, and A. Voronkov. The Vampire and the FOOL. In J. Avigad and A. Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 37–48. ACM, 2016.

[45] 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.

[46] D. Külwein, J. Blanchette, C. Kaliszyk, and J. Urban. Machine Learning for Sledgehamme. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Proceedings of the 4th International Conference on Interactive Theorem Proving*, number 7998 in Lecture Notes in Computer Science, pages 35–50. Springer-Verlag, 2013.

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

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

[49] 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.

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

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

[52] J. Meng and L. Paulson. Lightweight Relevance Filtering for Machine-generated Resolution Problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[53] 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.

[54] 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.

[55] 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.

[56] M. Rawson and G. Reger. Eliminating Models during Model Elimination. In A. Das and S. Negri, editors, *Proceedings of the 30th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, Lecture Notes in Artificial Intelligence, page To appear. Springer-Verlag, 2021.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] A. Reynolds, C. Tinelli, A. Goel, and S. Krstic. Finite Model Finding in SMT. In N. Sharygina and H. Veith, editors, *Proceedings of the 25th International Conference on Computer Aided Verification*, number 8044 in Lecture Notes in Computer Science, pages 640–655. Springer-Verlag, 2013.

[65] 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.

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

[67] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[68] A. Schlichtkrull, J. Blanchette, and D. Traytel. A Verified Functional Implementation of Bachmair and Ganzinger's Ordered Resolution Prover. *Archive of Formal Proofs*, pages 1–65, 2018.

[69] A. Schlichtkrull, J. Blanchette, and D. Traytel. Formalization of Bachmair and Ganzinger's Ordered Resolution Prover. *Archive of Formal Proofs*, pages 1–118, 2018.

[70] A. Schlichtkrull, J. Blanchette, and D. Traytel. A Verified Prover Based on Ordered Resolution. In A. Mahboubi and M. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 152–165. ACM Press, 2019.

[71] A. Schlichtkrull, J. Blanchette, D. Traytel, and U. Waldmann. Formalizing Bachmair and Ganzinger's Ordered Resolution Prover. 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 88–107, 2018.

[72] A. Schlichtkrull, J. Blanchette, D. Traytel, and U. Waldmann. Formalizing Bachmair and Ganzinger's Ordered Resolution Prover. *Journal of Automated Reasoning*, 64(7):1169–1195, 2020.

[73] 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.

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

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] S. Schulz, S. Cruanes, and P. Vukmirovic. 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.

[80] 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.

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

[82] C. Sternagel and R. Thiemann. Formalizing Knuth-Bendix Orders and Knuth-Bendix Completion. In F. Raamsdonk, editor, *Proceedings of the 24th International Conference on Rewriting Techniques and Applications*, number 21 in Leibniz International Proceedings in Informatics, pages 287–302. Dagstuhl Publishing, 2013.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[99] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[118] 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.

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

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

[121] G. Sutcliffe. Proceedings of the CADE-27 ATP System Competition. Natal, Brazil, 2019.

http://www.tptp.org/CASC/27/Proceedings.pdf.

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

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

[124] G. Sutcliffe. The 10th IJCAR Automated Theorem Proving System Competition - CASC-J10. *AI Communications*, page To appear, 2021.

[125] 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.

[126] 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, 2006.

[127] 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.

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

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

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

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

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

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

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

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

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

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

[138] 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.

[139] 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.

[140] 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.

[141] R. Thiemann and C. Sternagel. Certification of Termination Proofs Using CeTA. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, number 5674 in Lecture Notes in Computer Science, pages 452–468, 2009.

[142] 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.

[143] 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.

[144] P. Vukmirovic, 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.

[145] P. Vukmirovic 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.

[146] 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.

[147] 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.