Proceedings of CASC-26 – the CADE-26 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-26 ATP System Competition (CASC-26) was held on 9th August 2017. The design of the competition and its rules, and information regarding the competing systems, are provided in this report.

1 Introduction

The CADE and IJCAR conferences are the major forum 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-26 was held on 9th August 2017, as part of the 26th International Conference on Automated Deduction (CADE-26), in Gothenburg, Sweden. It was the twenty-second competition in the CASC series [130, 136, 133, 91, 93, 129, 127, 128, 98, 100, 102, 104, 107, 109, 112, 114, 116, 118, 120, 135, 122].

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 [123] and other useful sources of test problems, and
- specified time limits on solution attempts.

Twenty-one ATP system versions, listed in Table 1, entered into the various competition and demonstration divisions. The winners of the CASC-J8 (the previous CASC) divisions were automatically entered into those 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 this CASC evolved from those of previous CASCs [130, 131, 126, 132, 89, 90, 92, 94, 95, 96, 97, 99, 101, 103, 106, 108, 111, 113, 115, 117, 119, 121]. Important

ATP System	Divisions	Entrant (Associates)	Entrant's Affiliation
CVC4 1.5.5	TFA FOF FNT SLH	Andrew Reynolds (Clark Barrett, Cesare Tinelli)	University of Iowa
E 2.1	FOF FNT EPR SLH LTB	Stephan Schulz	DHBW Stuttgart
ET 2.0	SLH	Josef Urban (Jan Jakubuv, Cezary Kaliszyk, Stephan Schulz)	Czech Technical University in Prague
iProver 2.5	EPR	CASC	CASC-J8 winner
iProver 2.6	FOF FNT EPR SLH LTB	Konstantin Korovin	University of Manchester
iProverModulo 2.5-0.1	FOF SLH	Guillaume Burel	University Paris-Saclay
Isabelle 2016	THF	Jasmin Blanchette (Lawrence Paulson, Tobias Nipkow, Makarius Wenzel)	Vrije Universiteit Amsterdam
lean-nanoCoP 1.0	FOF	Jens Otten	University of Oslo
LEO-II 1.7.0	THF	Alexander Steen (Max Wisniewski Christoph Benzmüller)	Freie Universität Berlin
Leo-III 1.1	THF FOF SLH (demo)	Alexander Steen (Max Wisniewski Christoph Benzmüller)	Freie Universität Berlin
MaLARea 0.6	LTB	Josef Urban (Jan Jakubuv, Cezary Kaliszyk, Stephan Schulz)	Czech Technical University in Prague
Princess 170717	TFA	Philipp Rümmer	Uppsala University
Prover9 2009-11A	FOF	CASC (William McCune, Bob Veroff)	CASC fixed point
Satallax 3.0	THF	CASC	CASC-J8 winner
Satallax 3.2	THF	Michael Färber (Chad Brown)	Universität Innsbruck
Scavenger EP-0.1	EPR	Bruno Woltzenlogel Paleo (Daniyar Itegulov, Uladzislau Padtsiolkin)	Australian National University
Scavenger EP-0.2	FOF FNT EPR	Bruno Woltzenlogel Paleo (Daniyar Itegulov, Uladzislau Padtsiolkin)	Australian National University
Vampire 4.0	FOF LTB	CASC	CASC-J8 winner
Vampire 4.1	TFA FNT	CASC	CASC-J8 winner
Vampire 4.2	TFA FOF FNT EPR SLH LTB	Giles Reger (Martin Suda, Andrei Voronkov Evgeny Kotelnikov, Laura Kovacs)	University of Manchester
Zipperposition 1.1	THF TFA FOF SLH	Simon Cruanes	Inria Nancy

Table 1: The ATP systems and entrants

changes for this CASC were:

- The TFN division was put into a hiatus state.
- The SLH division was added.

The competition organizer was Geoff Sutcliffe. CASC is overseen by a panel of knowledgeable researchers who are not participating in the event. The CASC-26 panel members were Pascal Fontaine, Moa Johansson, and Dejan Jovanović. The competition was run on computers provided by StarExec at the University of Iowa. The CASC-26 web site provides access to resources used before, during, and after the event: `http://www.tptp.org/CASC/26`

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.

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 problem categories have no effect on the competition rankings, which 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 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

The **TFA** division: Typed First-order with Arithmetic theorems (axioms with a provable conjecture). The TFA division has three problem categories:

- The **TFI** category: TFA with only Integer arithmetic
- The **TFR** category: TFA with only Rational arithmetic
- The **TFE** category: TFA with only rEal arithmetic

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 **EPR** division: Effectively PRopositional clause normal form theorems and non-theorems (clause sets). *Effectively propositional* means that the problems are syntactically non-propositional but are known to be reducible to propositional problems, e.g., CNF problems that have no functions with arity greater than zero. The EPR division has two problem categories:

- The **EPT** category: Effectively Propositional Theorems (unsatisfiable clause sets)
- The **EPS** category: Effectively Propositional non-theorems (Satisfiable clause sets)

The **SLH** division: Typed first-order theorems without arithmetic (axioms with a provable conjecture), generated by Isaballe's SLedgeHammer system [67] and submitted to the SystemOnTPTP [110] service.

The **LTB** division: First-order form theorems (axioms with a provable conjecture) from Large Theories, presented in Batches. A large theory has many functors and predicates, and many axioms of which typically only a few are required for the proof of a theorem. Problems in a batch all use a common core set of axioms, and the problems in a batch are given to the ATP system all at once. Each problem category is accompanied by a set of training problems and their solutions, taken from the same source as the competition problems, that can be used for tuning and training during (typically at the start of) the competition. In CASC-26 the LTB division had one problem category, which remained a secret until the day of CASC (to ensure there was no pre-tuning).

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, 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. Computers supplied by the entrant may be brought to CASC, or may be accessed via the internet. 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 prizes are awarded.

3 Infrastructure

3.1 Computers

The computers had:

- Four (a quad-core chip) Intel(R) Xeon(R) E5-2609, 2.40GHz CPUs
- 128GB memory
- The Red Hat Enterprise Linux Server release 7.2 (Maipo) operating system, kernel 3.10.0-327.10.1.el7.x86_64.

One ATP system runs on one CPU at a time. Systems can use all the cores on the CPU (which is advantageous in the divisions where a wall clock time limit is used).

3.2 Problems

3.2.1 Problem Selection

The problems for the THF, TFA, FOF, FNT, and EPR divisions were taken from the TPTP Problem Library [123], version v7.0.0. The TPTP version used for CASC is released 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 selection. 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 must be syntactically non-propositional.
- The TPTP uses system performance data in the Thousands of Solutions from Theorem Provers (TSTP) solution library to compute problem difficulty ratings in the range 0.00 (easy) to 1.00 (unsolved) [134]. 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 [91].
- 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 collected from submissions from Isabelle's Sledgehammer subsystem to the SystemOnTPTP service. The problems were collected over a long period with sampling that ensures diversity. 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, with each problem category being based on one source. The process for selecting problems depends on the problem source. Entrants are expected to honestly not use publicly available problem sets for tuning before the competition.

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 each solution attempt - see Section 3.3.

In the TPTP-based and SLH divisions, the lower bound on the total number of problems to be used is 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 all the divisions, and the per-problem time limit, according to the following relationship:

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

It is a lower bound on the total 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 LTB division the number of problems in each problem category is determined by the number of problems in the corresponding problem set. In CASC-26, the one problem category had 1500 problems.

3.2.3 Problem Preparation

The problems are 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 preprocessed to:

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

In the SLH and LTB divisions the formulae are not preprocessed, 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 are given in a roughly estimated order of difficulty. In the LTB division the problems are given in the natural order of their creation for the problem sets, e.g., export from an ITP system.

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*
- 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*
  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.

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-26 it is
  `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-26 it is
  `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-26 it is
  `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. Problems in the batch have all these `include` directives, and can also have other `include` directives that are not listed here.
- 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).
- A terminator line `% SZS end BatchProblems`

3.3 Resource Limits

3.3.1 TPTP-based divisions

CPU and wall clock time limits are imposed for each problem. The minimal CPU time limit per problem is 240s. The maximal CPU time limit per problem is determined using the relationship used for determining the number of problems, with the minimal number of problems as the $NumberOfProblems$. The CPU time limit is chosen as a reasonable value within the range allowed, and is announced at the competition. The wall clock time limit is imposed in addition to the CPU time limit, to limit very high memory usage that causes swapping. The wall clock time limit per problem is double the CPU time limit. An additional memory limit is imposed, depending on the computers' memory.

3.3.2 SLH division

In the SLH division, a wall clock time limit is imposed for each problem. The minimal wall clock time limit per problem is 15s, and the maximal wall clock time limit per problem is 90s. The wall clock time limit is chosen as a reasonable value within the range allowed, based on performance data similar to that in the TSTP, and is announced at the competition. There are no CPU time limits (i.e., using all cores makes sense).

3.3.3 LTB division

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. The minimal wall clock time limit per problem is 15s, and the maximal wall clock time limit per 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., the batch's per-problem time limit multiplied by the number of problems in the batch. There are no CPU time limits.

Time spent before starting the first problem in the batch, e.g., learning from the training set and pre-loading the common axioms, and times spent between ending a problem and starting the next, e.g., learning from previous proofs, were not part of the time taken on problems. However, time taken on such tasks was part of the overall time taken for the batch.

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 proof or model was output.

The systems are ranked in the competition divisions, from the performance data. The THF, TFA, FOF, FNT, and LTB divisions are ranked according to the number of problems solved with an acceptable proof/model output. The EPR and SLH divisions are ranked according to the number of problems solved, but not necessarily accompanied by a proof or model (but systems that do output proofs/models are highlighted in the presentation of results). Ties are broken according to the average time taken over problems solved (CPU time or wall clock time, depending on the type of limit in the division). Trophies are awarded to the division winners.

The competition panel decides whether or not the systems' proofs and models 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 proofs of FOF problems by CNF refutation, the conversion from FOF to CNF 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 the reciprocal of the number of systems that solved the problem, so that if a system is the only one to solve a problem then its SoTAC for the problem is 1.00, and if all the systems solve a problem their SoTAC for the problem is the inverse of the number of systems. A system's overall SoTAC is its average SoTAC over the problems it solved.
- 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 competition ran on quad-core computers, thus the maximal core usage was 4.0.
- The *efficiency* measure combines the number of problems solved with the time taken. It is the average solution rate over the problems solved (the solution rate for one problem is the inverse of the time taken to solve it), 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. 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 proofs and models from the winners (of divisions ranked by the numbers of proofs/models output) are checked by the panel. If any of the proofs or models are unacceptable, i.e., they are significantly 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 entered, an entrant must be nominated to handle all issues (e.g., installation and execution difficulties) arising before and during 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 competition 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 divisions, to provide benchmarks against which progress can be judged. Prover9 2009-11A is automatically entered into the FOF division, to provide a fixed-point against which progress can be judged.

5.1 System Description

A system description has to be provided for each ATP system entered, using the HTML schema supplied on the CASC web site. (See Section 7 for these descriptions.) 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 in 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.

5.2 Sample Solutions

For systems in the divisions that require proof/model 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 proofs and models are acceptable.

Proof/model samples are required as follows:

- THF: `SET014^4`
- TFA: `DAT013=1`
- FOF and LTB: `SEU140+2`
- FNT: `NLP042+1` and `SWV017+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' performance 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, TFA, FOF, EPR, SLH, and LTB divisions, and theorems are submitted to the systems in the FNT and EPR divisions. 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. For systems running on entrant supplied computers in the demonstration division, the entrant must perform the soundness testing and report the results to the competition organizers.
- 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 and axiom sets is not allowed. Strategies and strategy selection based on individual problems is not allowed. If machine learning procedures are used, the learning must ensure that sufficient generalization is obtained so that no there is no specialization to individual problems or their solutions.
- 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 fully 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 all solution output must be to `stdout`. In the LTB division all 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, result and solution output to file)
% SZS status GaveUp for CSR075+2.p
% SZS status Ended for CSR075+2.p</PRE>
  ... and later in another attempt on that problem ...<PRE>
```

```
% SZS status Started for CSR075+2.p
  ... (system churns away, result and solution appended to 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 [105] 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 proofs/models, the start and end of the proof/model 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 proof/model. Use of the TPTP format for proofs and finite interpretations is encouraged [125].

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 that 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-proof/model 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 CVC4 1.5.2

Andrew Reynolds
University of Iowa, USA

Architecture

CVC4 [3] is an SMT solver based on the DPLL(T) architecture [56] 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, CVC4 primarily uses heuristic approaches based on E-matching for theorems, and finite model finding approaches for non-theorems. For problems in pure arithmetic, CVC4 uses techniques for counterexample-guided quantifier instantiation [71].

Like other SMT solvers, CVC4 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 CVC4 targets problems containing background theories whose quantification is limited to finite and uninterpreted sorts. In finite model finding mode, CVC4 uses a ground theory of finite cardinality constraints that minimizes the number of ground equivalence classes, as described in [73]. 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 [74]. If no instances are added, it reports "satisfiable".

Strategies

For handling theorems, CVC4 primarily uses conflict-based quantifier instantiation [72] and E-matching. CVC4 uses a handful of orthogonal trigger selection strategies for E-matching. For handling non-theorems, CVC4 primarily uses finite model finding techniques. Since CVC4 with finite model finding is also capable of establishing unsatisfiability, it is used as a strategy for theorems as well. For problems in pure arithmetic, CVC4 uses variations of counterexample-guided quantifier instantiation [71], which select relevant quantifier instantiations based on models for counterexamples to quantified formulas. CVC4 relies on this method both for theorems in TFA and non-theorems in TFN.

Implementation

CVC4 is implemented in C++. The code is available from:

`https://github.com/CVC4`

Expected Competition Performance

For TFA, CVC4 should perform better than last year due to its use of new heuristic techniques for non-linear real and integer arithmetic [75]. For FOF, it should perform slightly better due to improvements in the implementation of E-matching and several optimizations related to conflict-based instantiation [2]. It should perform roughly the same in the FNT division as last year.

7.2 E 2.1

Stephan Schulz
DHBW Stuttgart, Germany

Architecture

E 2.1 [81, 86] 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.

For the SLH and LTB divisions, a control program uses a SInE-like analysis to extract reduced axiomatizations that are handed to several instances of E. E will probably 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).

For CASC-26, 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,...). 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 220 different strategies have been evaluated on all untyped first-order problems from TPTP 6.4.0. About 90 of these strategies are used in the automatic mode, and about 210 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 [82]. Superposition and backward rewriting use fingerprint indexing [84], 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 [51, 52]. The prover and additional information are available at

`http://www.eprover.org`

Expected Competition Performance

E 2.1 has slightly better strategies than previous versions, and has some minor improvements in clausification and Set-of-Support implementation. The system is expected to perform well in most proof classes, but will at best complement top systems in the disproof classes.

7.3 ET 0.2

Josef Urban
Czech Technical University in Prague, Czech Republic

Architecture

ET [37] 0.2 is a metasystem using E prover with specific strategies [138, 41, 36] and preprocessing tools [40, 39] that are targeted mainly at problems with many redundant axioms. Its design is motivated by the recent experiments in the Large-Theory Batch division [43] and on the Flyspeck, Mizar and Isabelle datasets, however, ET does no learning from related proofs.

Strategies

We characterize formulas by the symbols and terms that they contain, normalized in various ways. Then we run various algorithms that try to remove the redundant axioms and use special strategies on such problems.

Implementation

The metasystem is implemented in ca. 1000 lines of Perl. It uses a number of external programs, some of them based on E's code base, some of them independently implemented in C++.

Expected Competition Performance

ET can solve some problems that E 1.8 cannot prove, and even some TPTP problems with rating 1.00. The CASC performance should not be much worse than that of E, possibly better, depending on problem selection.

7.4 iProver 2.5

Kontantin Korovin
University of Manchester, United Kingdom

Architecture

iProver is an automated theorem prover based on an instantiation calculus Inst-Gen [27, 46] which is complete for first-order logic. iProver combines first-order reasoning with ground reasoning for which it uses MiniSat [26] and optionally PicoSAT [13] (only MiniSat will be used at this CASC). iProver also combines instantiation with ordered resolution; see [45, 46] for the implementation details. The proof search is implemented using a saturation process based on the given clause algorithm. iProver uses non-perfect discrimination trees for the unification indexes, priority queues for passive clauses, and a compressed vector index for subsumption and subsumption resolution (both forward and backward). The following redundancy eliminations are implemented: blocking non-proper instantiations; dismatching constraints [28, 45]; global subsumption [45]; resolution-based simplifications and propositional-based simplifications. A compressed feature vector index is used for efficient forward/backward subsumption and subsumption resolution. Equality is dealt with (internally) by adding the necessary axioms of equality. Recent changes in iProver include improved preprocessing and incremental finite model finding; support of the AIG format for hardware verification and model-checking (implemented with Dmitry Tsarkov).

In the LTB division, iProver uses axiom selection based on the Sine algorithm [34] as implemented in Vampire [50], i.e., axiom selection is done by Vampire and proof attempts are done by iProver.

Some of iProver features are summarised below.

- proof extraction for both instantiation and resolution [48],
- model representation, using first-order definitions in term algebra [48],
- answer substitutions,
- semantic filtering,
- incremental finite model finding,
- sort inference, monotonic [21] and non-cyclic [47] sorts,
- predicate elimination [44].

Sort inference is targeted at improving finite model finding and symmetry breaking. Semantic filtering is used in preprocessing to eliminated irrelevant clauses. Proof extraction is challenging due to simplifications such global subsumption which involve global reasoning with the whole clause set and can be computationally expensive.

Strategies

iProver has around 60 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. At CASC iProver will execute a small number of fixed schedules of selected options depending on general syntactic properties such as Horn/non-Horn, equational/non-equational, and maximal term depth. For the LTB and FNT divisions several strategies are run in parallel.

Implementation

Prover is implemented in OCaml and for the ground reasoning uses MiniSat [26]. iProver accepts FOF and CNF formats. Vampire [50, 32] and E prover [86] are used for proof-producing clausification of FOF problems, Vampire is also used for axiom selection [34] in the LTB division.

iProver is available at:

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

Expected Competition Performance

iProver 2.5 is the CASC-J8 THF division winner.

7.5 iProver 2.6

Konstantin Korovin
University of Manchester, United Kingdom

Architecture

iProver is an automated theorem prover based on an instantiation calculus Inst-Gen [27, 46] which is complete for first-order logic. iProver combines first-order reasoning with ground reasoning for which it uses MiniSat [26] and optionally PicoSAT [13] (only MiniSat will be used at this CASC). iProver also combines instantiation with ordered resolution; see [45, 46] for the implementation details. The proof search is implemented using a saturation process based on the given clause algorithm. iProver uses non-perfect discrimination trees for the unification indexes, priority queues for passive clauses, and a compressed vector index for subsumption and subsumption resolution (both forward and backward). The following redundancy eliminations are implemented: blocking non-proper instantiations; dismatching constraints [28, 45]; global subsumption [45]; resolution-based simplifications and propositional-based simplifications. A compressed feature vector index is used for efficient forward/backward subsumption and subsumption resolution. Equality is dealt with (internally) by adding the necessary axioms of equality. Recent changes in iProver include improved preprocessing and incremental finite model finding; support for the TFF format restricted to clauses; the AIG format for hardware verification and QBF reasoning.

In the LTB and SLH divisions, iProver combines an abstraction-refinement framework [31] with axiom selection based on the SinE algorithm [34] as implemented in Vampire [50], i.e., axiom selection is done by Vampire and proof attempts are done by iProver.

Some of iProver features are summarised below.

- proof extraction for both instantiation and resolution [48],
- model representation, using first-order definitions in term algebra [48],
- answer substitutions,
- semantic filtering,
- incremental finite model finding,
- sort inference, monotonic [21] and non-cyclic [47] sorts,
- support for the TFF format restricted to clauses,
- predicate elimination [44].

Sort inference is targeted at improving finite model finding and symmetry breaking. Semantic filtering is used in preprocessing to eliminated irrelevant clauses. Proof extraction is challenging due to simplifications such global subsumption which involve global reasoning with the whole clause set and can be computationally expensive.

Strategies

iProver has around 60 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. At CASC iProver will execute a small number of fixed schedules of selected options depending on general syntactic properties such as Horn/non-Horn, equational/non-equational, and maximal term depth. For the LTB, SLH and FNT divisions several strategies are run in parallel.

Implementation

iProver is implemented in OCaml and for the ground reasoning uses MiniSat [26]. iProver accepts FOF, TFF and CNF formats. Vampire [50, 32] and E prover [86] are used for proof-producing clausification of FOF/TFF problems, Vampire is also used for axiom selection [HV11] in the LTB/SLH divisions.

iProver is available at:

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

Expected Competition Performance

Compared to the last year, we integrated an abstraction-refinement framework [31] which we expect to improve performance in the LTB and SLH divisions. There are a several general improvements that should positively affect overall performance.

7.6 iProverModulo 2.5-0.1

Guillaume Burel
ENSIIE, University ParisSaclay, France

Architecture

iProverModulo [20] is an extension of iProver [45] to integrate Polarized resolution modulo [25]. Polarized resolution modulo consists in presenting the theory in which the problem has to be solved by means of polarized rewriting rules instead of axioms. It can also be seen as a combination of the set-of-support strategy and selection of literals.

iProverModulo consists of two tools: First, autotheo is a theory preprocessor that converts the axioms of the input into rewriting rules that can be used by Polarized resolution modulo. Second, these rewriting rules are handled by a patched version of iProver 2.5 that integrates Polarized resolution modulo. The integration of polarized resolution modulo in iProver only affects its ordered resolution calculus, so that the instantiation calculus is untouched.

iProverModulo 2.5+0.1 outputs a proof that is made of two parts: First, autotheo prints a derivation of the transformation of the axioms into rewriting rules. This derivation is in TSTP format and includes the CNF conversions obtained from E. Second, the modified version of iProver outputs a proof in TSTP format from this set of rewriting rules and the other input formulas.

Strategies

Autotheo is first run to transform the formulas of the problem whose role is “axiom” into polarized rewriting rules. Autotheo offers a set of strategies to that purpose. For the competition, the Equiv and the ClausalAll strategies will be used. The former strategy orients formulas intuitively depending of their shape. It may be incomplete, so that the prover may give up in certain cases. However, it shows interesting results on some problems. The second strategy should be complete, at least when equality is not involved. The rewriting system for the first strategy is tried for half the time given for the problem, then the prover is restarted with the second strategy if no proof has been found.

The patched version of iProver is run on the remaining formulas modulo the rewriting rules produced by autotheo. No scheduling is performed. To be compatible with Polarized resolution modulo, literals are selected only when they are maximal w.r.t. a KBO ordering, and orphans are not eliminated. To take advantage of Polarized resolution modulo, the resolution calculus is triggered more often than the instantiation calculus, on the contrary to the original iProver.

Normalization of clauses w.r.t. the term rewriting system produced by autotheo is performed by transforming these rules into an OCaml program, compiling this program, and dynamically linking it with the prover.

Implementation

iProverModulo is available as a patch to iProver. The most important additions are the plugin-based normalization engine and the handling of polarized rewriting rules. iProverModulo is available from

`http://www.ensiie.fr/~guillaume.burel/blackandwhite\_iProverModulo.html.en`

Since iProverModulo needs to compile rewriting rules, an OCaml compiler is also provided.

Autotheo is available independently from iProverModulo from

`http://www.ensiie.fr/~guillaume.burel/blackandwhite\_autotheo.html.en`

Autotheo uses E to compute clausal normal form of formula. The version of E it uses is very slightly modified to make it print the CNF derivation even if no proof is found.

Both of autotheo and iProver are written in OCaml.

For the SLD division, iProverModulo uses the CNF transformation tool provided with the Logtk library [22].

Expected Competition Performance

Although iProverModulo is now based on version 2.5 of iProver, no great improvement of performance is expected compared to CASC-25, since only the resolution part of iProver, which is relatively stable, has been modified.

7.7 Isabelle 2016

Jasmin Blanchette
Vrije Universiteit Amsterdam, Netherlands

Architecture

Isabelle/HOL 2016 [58] is the higher-order logic incarnation of the generic proof assistant Isabelle2016. Isabelle/HOL provides several automatic proof tactics, notably an equational reasoner [57], a classical reasoner [68], and a tableau prover [66]. It also integrates external first- and higher-order provers via its subsystem Sledgehammer [67, 14]. Isabelle includes a parser for the TPTP syntaxes CNF, FOF, TFF0, and THF0, due to Nik Sultana. It also includes TPTP versions of its popular tools, invokable on the command line as `isabelle tptp_`*`tool max_secs file.p`*. For example:

```
isabelle tptp\_isabelle\_hot 100 SEU/SEU824\verb|^|3.p
```

Isabelle is available in two versions. The HOT version (which is not participating in CASC-J8) includes LEO-II [8] and Satallax [17] as Sledgehammer backends, whereas the competition version leaves them out.

Strategies

The *Isabelle* tactic submitted to the competition simply tries the following tactics sequentially:

`sledgehammer`
Invokes the following sequence of provers as oracles via Sledgehammer:

- `satallax` - Satallax 2.7 [17] (*HOT version only*);
- `leo2` - LEO-II 1.6.2 [8] (*HOT version only*);
- `spass` - SPASS 3.8ds [15];
- `vampire` - Vampire 3.0 (revision 1435) [76];
- `e` - E 1.8 [83];

`nitpick`
For problems involving only the type `$o` of Booleans, checks whether a finite model exists using Nitpick [16]. `simp`
Performs equational reasoning using rewrite rules [57]. `blast`
Searches for a proof using a fast untyped tableau prover and then attempts to reconstruct the proof using Isabelle tactics [66]. `auto+spass`
Combines simplification and classical reasoning [68] under one roof; then invoke Sledgehammer with SPASS on any subgoals that emerge. `z3`
Invokes the SMT solver Z3 4.4.0 [24]. `cvc4`
Invokes the SMT solver CVC4 1.5pre [4]. `fast`
Searches for a proof using sequent-style reasoning, performing a depth-first search [68]. Unlike `blast`, it construct proofs directly in Isabelle. That makes it slower but enables it to work in the presence of the more unusual features of HOL, such as type classes and function unknowns. `best`

Similar to `fast`, except that it performs a best-first search. `force`

Similar to `auto`, but more exhaustive. `meson`

Implements Loveland's MESON procedure [53]. Constructs proofs directly in Isabelle. `fastforce`

Combines `fast` and `force`.

Implementation

Isabelle is a generic theorem prover written in Standard ML. Its meta-logic, Isabelle/Pure, provides an intuitionistic fragment of higher-order logic. The HOL object logic extends pure with a more elaborate version of higher-order logic, complete with the familiar connectives and quantifiers. Other object logics are available, notably FOL (first-order logic) and ZF (Zermelo-Fraenkel set theory).

The implementation of Isabelle relies on a small LCF-style kernel, meaning that inferences are implemented as operations on an abstract `theorem` datatype. Assuming the kernel is correct, all values of type `theorem` are correct by construction.

Most of the code for Isabelle was written by the Isabelle teams at the University of Cambridge and the Technische Universität München. Isabelle/HOL is available for all major platforms under a BSD-style license from

`http://www.cl.cam.ac.uk/research/hvg/Isabelle/`

Expected Competition Performance

I expect we will end up in second place (excluding proof output), behind Satallax, since we haven't upgraded the system since 2016. We will be back, hopefully in 2018!

7.8 lean-nanoCoP 1.0

Jens Otten
University of Oslo, Norway

Architecture

lean-nanoCoP is an automated theorem prover for classical first-order logic with equality. It combines the provers leanCoP [65, 59] and nanoCoP [63], which are very compact implementations of the clausal connection calculus [12] and the non-clausal connection calculus [61], respectively.

Strategies

The reduction rule of the connection calculus is applied before the extension rule. Open branches are selected in a depth-first way. Iterative deepening on the proof depth is performed in order to achieve completeness. Additional inference rules and techniques include regularity, lemmata, and restricted backtracking [60]. leanCoP uses an optimized structure-preserving transformation into clausal forms [60]. The fixed strategy scheduling, which is controlled by a shell script, invokes the leanCoP and nanoCoP core provers.

Implementation

leanCoP and nanoCoP are implemented in Prolog. The source code of the core provers consists only of a few lines of code. Prolog's built-in indexing mechanism is used to quickly find connections when the extension rule is applied.

lean-nanoCoP can read formulae in leanCoP syntax and in TPTP first-order syntax. The leanCoP and nanoCoP core provers return very compact connection proofs; leanCoP translates its proof into a more readable output format.

The source codes of leanCoP and nanoCoP are available under the GNU general public license. They can be downloaded from the leanCoP and nanoCoP websites at

`http://www.leancop.de`

and

`http://www.leancop.de/nanocop`

The leanCoP website also contains information about ileanCoP [59] and MleanCoP [62], two versions of leanCoP for first-order intuitionistic logic and several first-order modal logics, respectively. Recently, versions of nanoCoP for these logics have been developed as well [64].

Expected Competition Performance

For problems that are in a "strong" non-clausal form, the combination of leanCoP and nanoCoP are expected to perform better than the leanCoP prover by itself.

7.9 LEO-II 1.7.0

Alexander Steen
Freie Universität Berlin, Germany

Architecture

LEO-II [8], the successor of LEO [6], 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 [5]. 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 [80]. 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 [9]. In fact, the development of LEO-II has largely paralleled the development of the TPTP THF language and related infrastructure [124]. 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 [10] 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 ist not actively being developed anymore, hence there are no expected improvements to last year's CASC results.

7.10 Leo-III 1.1

Alexander Steen
Freie Universität Berlin, Germany

Architecture

Leo-III [88], the successor of LEO-II [8], is a higher-order ATP system based on higher-order paramodulation with inference restrictions using a higher-order term ordering.

Since Leo-III employs a agent-based blackboard architecture, multiple independent proof search approaches can be run in parallel as so-called agents. In version 1.1, each agent runs a sequential proof search based on the given-clause algorithm as known from E, each with different search strategy.

Leo-III heavily relies on cooperation with external (first-order) ATPs that are called asynchronously during proof search. At the moment, first-order cooperation is limited to typed first-order systems, where CVC4 [3] is used as default external system. Nevertheless, further external systems (also further higher-order systems or counter model generators) can be employed using command-line arguments. If either one of the saturation procedure loops or one of the external provers finds a proof, the system stops and returns the result.

Strategies

Leo-III runs multiple search strategies in parallel using its agent-based architecture. The search strategies differ in the employed relevance filter parameters, inference parameters, preprocessing techniques and hence the considered formula set. The available portfolio of strategies also contains incomplete approaches that might outperform default search strategies for some problem input classes.

Implementation

Leo-III exemplarily utilizes and instantiates the associated LeoPARD system platform [141] for higher-order (HO) deduction systems implemented in Scala (currently using Scala 2.12). The prover makes use of LeoPARD's sophisticated data structures and implements its own reasoning logic on top, e.g. as agents in LeoPARD's provided blackboard architecture [11].

A generic parser is provided that supports all TPTP syntax dialects. It is implemented using ANTLR4 and 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) including their polymorphic variants [7, 38].

The term data structure of Leo-III uses a 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.

As pointed out before, Leo-III's agents 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. While LEO-II relied on cooperation with untyped first-order provers, we hope to reduce clutter and therefore achieve better results using native type support in first-order provers.

Leo-III 1.1 will be available on GitHub after CASC-26:

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

Expected Competition Performance

In contrast to its last version 1.0 (competed at CASC-J8), Leo-III 1.1 has been improved in several aspects. Due to the novel cooperation schemes with typed first-order provers, we strongly expect better results compared to its predecessor LEO-II.

7.11 MaLARea 0.6

Josef Urban
Czech Technical University in Prague, Czech Republic

Architecture

MaLARea 0.6 [137, 139, 43] is a metasystem for ATP in large theories where symbol and formula names are used consistently. It uses several deductive systems (now E,SPASS,Vampire, Paradox, Mace), as well as complementary AI techniques like machine learning (the SNoW system) based on symbol-based similarity, model-based similarity, term-based similarity, and obviously previous successful proofs. The version for CASC-26 will mainly use the E prover with the BliStr(Tune) [138, 36] large-theory strategies, possibly also Prover9, Mace and Paradox. The premise selection methods will likely also use the distance-weighted k-nearest neighbor [42] and E's implementation of SInE.

Strategies

The basic strategy is to run ATPs on problems, then use the machine learner to learn axiom relevance for conjectures from solutions, and use the most relevant axioms for next ATP attempts. This is iterated, using different timelimits and axiom limits. Various features are used for learning, and the learning is complemented by other criteria like model-based reasoning, symbol and term-based similarity, etc.

Implementation

The metasystem is implemented in ca. 2500 lines of Perl. It uses many external programs - the above mentioned ATPs and machine learner, TPTP utilities, LADR utilities for work with models, and some standard Unix tools.

MaLARea is available at:

`https://github.com/JUrban/MPTP2/tree/master/MaLARea`

The metasystem's Perl code is released under GPL2.

Expected Competition Performance

Thanks to machine learning, MaLARea is strongest on batches of many related problems with many redundant axioms where some of the problems are easy to solve and can be used for learning the axiom relevance. MaLARea is not very good when all problems are too difficult (nothing to learn from), or the problems (are few and) have nothing in common. Some of its techniques (selection by symbol and term-based similarity, model-based reasoning) could however make it even there slightly stronger than standard ATPs. MaLARea has a very good performance on the MPTP Challenge, which is a predecessor of the LTB division, and on several previous LTB competitions.

7.12 Princess 170717

Philipp Rümmer
Uppsala University, Sweden

Architecture

Princess [78, 79] is a theorem prover for first-order logic modulo linear integer arithmetic. The prover uses a combination of techniques from the areas of first-order reasoning and SMT solving. The main underlying calculus is a free-variable tableau calculus, which is extended with constraints to enable backtracking-free proof expansion, and positive unit hyper-resolution for lightweight instantiation of quantified formulae. Linear integer arithmetic is handled using a set of built-in proof rules resembling the Omega test, which altogether yields a calculus that is complete for full Presburger arithmetic, for first-order logic, and for a number of further fragments. In addition, some built-in procedures for nonlinear integer arithmetic are available.

The internal calculus of Princess only supports uninterpreted predicates; uninterpreted functions are encoded as predicates, together with the usual axioms. Through appropriate translation of quantified formulae with functions, the e-matching technique common in SMT solvers can be simulated; triggers in quantified formulae are chosen based on heuristics similar to those in the Simplify prover.

Strategies

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

Implementation

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

Princess is available from:

`http://www.philipp.ruemmer.org/princess.shtml`

Expected Competition Performance

Princess should perform roughly as in the last years. Compared to last year, the support for outputting proofs was extended, and should now cover all relevant theory modules for CASC (but not yet all proof strategies).

7.13 Prover9 2009-11A

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 [55]. 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 [54]. (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 [83] 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.

7.14 Satallax 3.0

Michael Färber
Universität Innsbruck, Austria

Architecture

Satallax 3.0 [17] is an automated theorem prover for higher-order logic. The particular form of higher-order logic supported by Satallax is Church's simple type theory with extensionality and choice operators. The SAT solver MiniSat [26] is responsible for much of the proof search. The theoretical basis of search is a complete ground tableau calculus for higher-order logic [19] with a choice operator [1]. Problems are given in the THF format.

Proof search: A branch is formed from the axioms of the problem and the negation of the conjecture (if any is given). From this point on, Satallax tries to determine unsatisfiability or satisfiability of this branch. Satallax progressively generates higher-order formulae and corresponding propositional clauses [Bro13]. These formulae and propositional clauses correspond to instances of the tableau rules. Satallax uses the SAT solver MiniSat to test the current set of propositional clauses for unsatisfiability. If the clauses are unsatisfiable, then the original branch is unsatisfiable. Optionally, Satallax generates first-order formulae in addition to the propositional clauses. If this option is used, then Satallax periodically calls the first-order theorem prover E to test for first-order unsatisfiability. If the set of first-order formulae is unsatisfiable, then the original branch is unsatisfiable. Upon request, Satallax attempts to reconstruct a proof which can be output in the TSTP format.

Strategies

There are about 140 flags that control the order in which formulae and instantiation terms are considered and propositional clauses are generated. Other flags activate some optional extensions to the basic proof procedure (such as whether or not to call the theorem prover E). A collection of flag settings is called a mode. Approximately 500 modes have been defined and tested so far. A strategy schedule is an ordered collection of modes with information about how much time the mode should be allotted. Satallax tries each of the modes for a certain amount of time sequentially. Satallax 3.0 has a strategy schedule consisting of 54 modes (15 of which make use of E). Each mode is tried for time limits ranging from less than a second to about 90 seconds. The strategy schedule was determined through experimentation using the THF problems in version 6.3.0 of the TPTP library.

Implementation

Satallax is implemented in OCaml. A foreign function interface is used to interact with MiniSat 2.2.0 Satallax is available at:

`http://satallaxprover.com`

Expected Competition Performance

Satallax 3.0 is the CASC-J8 THF division winner.

7.15 Satallax 3.2

Michael Färber
Universität Innsbruck, Austria

Architecture

Satallax 3.2 [17] is an automated theorem prover for higher-order logic. The particular form of higher-order logic supported by Satallax is Church's simple type theory with extensionality and choice operators. The SAT solver MiniSat [26] is responsible for much of the proof search. The theoretical basis of search is a complete ground tableau calculus for higher-order logic [19] with a choice operator [1]. Problems are given in the THF format.

Proof search: A branch is formed from the axioms of the problem and the negation of the conjecture (if any is given). From this point on, Satallax tries to determine unsatisfiability or satisfiability of this branch. Satallax progressively generates higher-order formulae and corresponding propositional clauses [18]. These formulae and propositional clauses correspond to instances of the tableau rules. Satallax uses the SAT solver MiniSat to test the current set of propositional clauses for unsatisfiability. If the clauses are unsatisfiable, then the original branch is unsatisfiable. Optionally, Satallax generates first-order formulae in addition to the propositional clauses. If this option is used, then Satallax periodically calls the first-order theorem prover E [85] to test for first-order unsatisfiability. If the set of first-order formulae is unsatisfiable, then the original branch is unsatisfiable. Upon request, Satallax attempts to reconstruct a proof which can be output in the TSTP format. The proof reconstruction has been significantly changed since Satallax 3.0 in order to make proof reconstruction more efficient and thus less likely to fail within the time constraints.

Strategies

There are about 150 flags that control the order in which formulae and instantiation terms are considered and propositional clauses are generated. Other flags activate some optional extensions to the basic proof procedure (such as whether or not to call the theorem prover E). A collection of flag settings is called a mode. Approximately 500 modes have been defined and tested so far. A strategy schedule is an ordered collection of modes with information about how much time the mode should be allotted. Satallax tries each of the modes for a certain amount of time sequentially. Before deciding on the schedule to use, Satallax parses the problem and determines if it is big enough that a SInE-based premise selection algorithm [34] should be used. If SInE is not activated, then Satallax 3.2 uses a strategy schedule consisting of 37 modes. Each mode is tried for time limits ranging from less than a second to just over 1 minute. If SInE is activated, than Satallax is run with a SInE-specific schedule consisting of 20 possible SInE parameter values selecting different premises and some corresponding modes and time limits.

Implementation

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

`http://satallaxprover.com`

Expected Competition Performance

The addition of a SInE-like procedure for premise selection means Satallax should be able to solve some large problems that were previously out of reach. In addition, the changes to the way TSTP proofs are generated should mean that proofs are more likely to be constructed and reported after a proof has been found. We hope that this will be reflected in an improved performance over Satallax 3.0 from last year.

7.16 Scavenger EP-0.1 and EP-0.2

Bruno Woltzenlogel Paleo
Australian National University, Australia

Architecture

Scavenger [35] is a theorem prover based on the new Conflict Resolution calculus [87]. At the proof-theoretical level, Conflict Resolution (CR) is a generalization of the conflict-driven clause learning (CDCL) principle to first-order logic. CR derivations are isomorphic to implication graphs (as maintained by SAT-solvers): every unit-propagating resolution inference corresponds to a new propagated literal in the graph; every assumption/decision corresponds to a decision literal in the graph; and every conflict inference corresponds to a conflict in the graph. CR's clause learning inference learns a disjunction of negations of *instances* of the decision literals that are ancestors of the conflict, using the compositions of the unifiers on the paths from the decisions to the conflict. From a natural deduction point of view, CR's clause learning inference rule generalizes implication/negation introduction by taking unification into account and considering several assumptions at once [142]. In this sense, it does to Gentzen's implication/negation introduction what Robinson's resolution did to implication elimination (a.k.a. modus ponens).

The architecture of Scavenger attempts to be similar to the architecture of SAT-solvers, but data structures typically used in sat-solvers (e.g., Two-Watched-Literals) cannot be easily and efficiently generalized to first-order logic. Because of that, Scavenger's architecture also has a "taste" of saturation. For example, whereas in a SAT-solver propagation causes a literal to be assigned (either true or false), in Scavenger, propagation often requires generation of an instance of a literal, and it is this generated instance that is assigned.

Strategies

Proof search in the Conflict Resolution calculus presents unique challenges. For example, in contrast to what happens in the propositional case, unit-propagation may not terminate. Scavenger is an experimental prover, and such challenges have been dealt with in various ways [35]. Scavenger-EP-0.1 is one of the three versions evaluated in [35]. It simply ignores the non-termination of unit-propagation (and hence is incomplete). (Scavenger-TD-0.1 and Scavenger-PD-0.1 maintain completeness by iteratively deepening the propagation and making decisions eagerly. However, on TPTP problems they did not perform as well as Scavenger-EP-0.1, and therefore will not participate in CASC this year.)

Scavenger-EP-0.2 extends Scavenger-EP-0.1 from CNF without equality to FOF with equality. However, equality reasoning is done in a naive way: (instances of) equality axioms are added to the problem when needed. Scavenger-EP-0.2 also implements a VSIDS heuristic for decision literal selection and optimizes unification and some data structures.

Implementation

Scavenger is implemented in Scala and runs on the Java Virtual Machine. Terms and formulas are simply typed lambda terms. Clauses are two-sided sequents (pairs of lists of positive and negative atomic formulas). Inference rules are classes with assertions that ensure their soundness. A hashmap is used to allow quicker detection of propagating clauses (in an attempt to generalize two-watched-literals to first-order logic).

Scavenger is available at:

`http://https://gitlab.com/aossie/Scavenger/`

Expected Competition Performance

Both competing versions of Scavenger are expected to perform better on effectively propositional problems (where the non-termination of unit-propagation is not an issue) than on problems that are not in this fragment. Scavenger-EP-0.1 has been evaluated in [35], and similar performance is expected in CASC. Scavenger-EP-0.2 has not been thoroughly evaluated yet. It is hoped that it will perform better than Scavenger-EP-0.1.

7.17 Vampire 4.0

Giles Reger
University of Manchester, United Kingdom

Architecture

Vampire 4.0 is an automatic theorem prover for first-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. Splitting in resolution-based proof search is controlled by the AVATAR architecture, which uses a SAT solver to make splitting decisions. Both resolution and instantiation based proof search make use of global subsumption.

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

- Choices of saturation algorithm:
  - Limited Resource Strategy
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals.
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection.
- Set-of-support strategy.
- Ground equational reasoning via congruence closure.
- Evaluation of interpreted functions.
- Extensionality resolution with detection of extensionality axioms

Implementation

Vampire 4.0 is implemented in C++.

Expected Competition Performance

Vampire 4.0 is the CASC-J8 FOF and LTB division winner.

7.18 Vampire 4.1

Giles Reger
University of Manchester, United Kingdom

Architecture

Vampire [50] 4.1 is an automatic theorem prover for first-order logic. Vampire implements the calculi of ordered binary resolution and superposition for handling equality. It also implements the Inst-gen calculus [46] and a MACE-style finite model builder [70]. Splitting in resolution-based proof search is controlled by the AVATAR architecture [140] which uses a SAT or SMT solver to make splitting decisions. Both resolution and instantiation based proof search make use of global subsumption [46].

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

- Choices of saturation algorithm:
  - Limited Resource Strategy [77].
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR
- A variety of optional simplifications.
- Parameterized reduction orderings.

- A number of built-in literal selection functions and different modes of comparing literals.
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection.
- Set-of-support strategy.
- Ground equational reasoning via congruence closure.
- Addition of theory axioms and evaluation of interpreted functions.
- Use of Z3 [24] with AVATAR to restrict search to ground-theory-consistent splitting branches.
- Extensionality resolution [30] with detection of extensionality axioms.

Implementation

Vampire 4.1 is implemented in C++.

Expected Competition Performance

Vampire 4.0 is the CASC-J8 TFA and FNT division winner.

7.19 Vampire 4.2

Giles Reger
University of Manchester, United Kingdom

Architecture

Vampire [50] 4.2 is an automatic theorem prover for first-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 [70]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [140, 69]. Both resolution and instantiation based proof search make use of global subsumption.

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

- Choices of saturation algorithm:
  - Limited Resource Strategy
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [140]
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals [33].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection.
- Set-of-support strategy.
- 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 [69].
- Specialised theory instantiation and unification
- Extensionality resolution with detection of extensionality axioms

Implementation

Vampire 4.2 is implemented in C++. It makes use of minisat and z3.

Expected Competition Performance

Vampire 4.2 should be an improvement on Vampire 4.1, which won 4 divisions last year. Most changes have happened in parts relevant to TFA, some small changes in parts relevant to model building (EPR and FNT), and some general improvements to preprocessing that will effect all tracks.

7.20 Zipperposition 1.1

Simon Cruanes
Inria Nancy, France

Architecture

Zipperposition is a superposition-based theorem prover for typed first-order logic with equality. It features a number of extensions that include polymorphic types; linear arithmetic on integers and rationals using a specialized set of first-order inference rules; datatypes with structural induction; user-defined rewriting on terms and formulas ("deduction modulo theories"); a lightweight variant of AVATAR for boolean case splitting; first-class booleans [49]; and (very experimental) support for higher-order logic, extending first-order rules to higher-order terms using a customized variant of pattern unification. The core architecture of the prover is based on saturation with an extensible set of rules for inferences and simplifications. The initial calculus and main loop were imitations of an old version of E [81], but there are many more rules nowadays. A summary of the calculus for integer arithmetic and induction can be found in [23].

Strategies

The system does not feature advanced strategies: only one saturation loop with pick-given ratio and clause selection heuristics is used. No tuning specific to CASC was made.

Implementation

The prover is implemented in OCaml, and has been around for five years. Term indexing is done using discrimination trees (non perfect ones for unification, perfect ones for rewriting) as well as feature vectors for subsumption. Some inference rules such as contextual literal cutting make heavy use of subsumption. The code can be found at

`https://github.com/c-cube/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 [23].

Expected Competition Performance

The prover is expected to have decent performance on first-order theorems, hopefully beating prover9; decent performance in arithmetic (ignoring the lack of real arithmetic), but behind more sophisticated provers such as Vampire or CVC4; and poor performance on THF problems except those that only require first-class booleans.

8 Conclusion

The CADE-26 ATP System Competition was the twenty-second large scale competition for classical logic ATP systems. The organizer believes that CASC fulfills its main motivations: stimulation of research, motivation for improving implementations, evaluation of relative capabilities of ATP systems, and providing an exciting event. Through the continuity of the event and consistency in the 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] J. Backes and C.E. Brown. Analytic Tableaux for Higher-Order Logic with Choice. *Journal of Automated Reasoning*, 47(4):451–479, 2011.

[2] 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.

[3] 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.

[4] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification*, number 4590 in Lecture Notes in Computer Science, pages 298–302. Springer-Verlag, 2007.

[5] 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.

[6] 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.

[7] C. Benzmüller and L. Paulson. Quantified Multimodal Logics in Simple Type Theory. *Logica Universalis*, 7(1):7–20, 2013.

[8] 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.

[9] 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.

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

[11] C. Benzmüller, A. Steen, and M. Wisniewski. Leo-III Version 1.1 (System description). In T. Eiter, D. Sands, S. Schulz, J. Urban, G. Sutcliffe, and A. Voronkov, editors, *Proceedings of the IWIL Workshop and LPAR Short Presentations*, number 1 in Kalpa Publications in Computing, 2017.

[12] W. Bibel. *Automated Theorem Proving*. Vieweg and Sohn, 1987.

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

[14] J. Blanchette, S. Boehme, and L. Paulson. Extending Sledgehammer with SMT Solvers. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction*, number 6803 in Lecture Notes in Artificial Intelligence, pages 116–130. Springer-Verlag, 2011.

[15] J. Blanchette, A. Popescu, D. Wand, and C. Weidenbach. More SPASS with Isabelle. In L. Beringer and A. Felty, editors, *Proceedings of the 3rd International Conference on Interactive Theorem Proving*, number 7406 in Lecture Notes in Computer Science, pages 345–360. Springer-Verlag, 2012.

[16] S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In J. Giesl and R. Haehnle, editors, *Proceedings of the 5th International Joint Conference on Automated Reasoning*, number 6173 in Lecture Notes in Artificial Intelligence, pages 107–121, 2010.

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

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

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

[20] G. Burel. Experimenting with Deduction Modulo. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction*, number 6803 in Lecture Notes in Artificial Intelligence, pages 162–176. Springer-Verlag, 2011.

[21] K. Claessen, A. Lilliestrom, and N. Smallbone. Sort It Out with Monotonicity - Translating between Many-Sorted and Unsorted First-Order Logic. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction*, number 6803 in Lecture Notes in Artificial Intelligence, pages 207–221. Springer-Verlag, 2011.

[22] S. Cruanes. Logtk: A Logic ToolKit for Automated Reasoning and its Implementation. In S. Schulz, L. de Moura, and B. Konev, editors, *Proceedings of the 4th Workshop on the Practical Aspects of Automated Reasoning*, number 31 in EPiC Series in Computing, pages 39–49. EasyChair Publications, 2014.

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

[24] 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.

[25] G. Dowek. Polarized Resolution Modulo. In C. Calude and V. Sassone, editors, *Theoretical Computer Science*, IFIP Advances in Information and Communication Technology, pages 182–196. Springer-Verlag, 2010.

[26] 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.

[27] 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.

[28] H. Ganzinger and K. Korovin. Integrating Equational Reasoning into Instantiation-Based Theorem Proving. In J. Marcinkowski and A. Tarlecki, editors, *Proceedings of the 18th International*

Workshop on Computer Science Logic, 13th Annual Conference of the EACSL, number 3210 in Lecture Notes in Computer Science, pages 71–84. Springer-Verlag, 2004.

[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] A. Gupta, L. Kovacs, B. Kragl, and A. Voronkov. Extensional Crisis and Proving Identity. In F. Cassez and J-F. Franck, editors, *Proceedings of the 12th International Symposium on Automated Technology for Verification and Analysis*, number 8837 in Lecture Notes in Computer Science, pages 185–200, 2014.

[31] J. Hernandez and K. Korovin. Towards an Abstraction-Refinement Framework for Reasoning with Large Theories. In T. Eiter, D. Sands, S. Schulz, J. Urban, G. Sutcliffe, and A. Voronkov, editors, *Proceedings of the IWIL Workshop and LPAR Short Presentations*, number 1 in Kalpa Publications in Computing, 2017.

[32] 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.

[33] 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.

[34] 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.

[35] D. Itegulov, J. Slaney, and B. Woltzenlogel Paleo. Scavenger 0.1: A Theorem Prover Based on Conflict Resolution. In L. de Moura, editor, *Proceedings of the 26th International Conference on Automated Deduction*, Lecture Notes in Computer Science, page To appear. Springer-Verlag, 2017.

[36] J. Jakubuv and J. Urban. BliStrTune: Hierarchical Invention of Theorem Proving Strategies. In Y. Bertot and V. Vafeiadis, editors, *Proceedings of Certified Programs and Proofs 2017*, pages 43–52. ACM, 2017.

[37] C. Kaliszyk, S. Schulz, J. Urban, and J. Vyskocil. System Description: E.T. 0.1. In A. Felty and A. Middeldorp, editors, *Proceedings of the 25th International Conference on Automated Deduction*, number 9195 in Lecture Notes in Computer Science, pages 389–398. Springer-Verlag, 2015.

[38] 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 the Practical Aspects of Automated Reasoning*, number 1635 in CEUR Workshop Proceedings, pages 41–55, 2016.

[39] C. Kaliszyk and J. Urban. Lemma Mining over HOL Light. 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 503–517. Springer-Verlag, 2013.

[40] C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. arXiv:1310.2805, 2013.

[41] C. Kaliszyk and J. Urban. PRocH: Proof Reconstruction for HOL Light. In M.P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction*, number 7898 in Lecture Notes in Artificial Intelligence, pages 267–274. Springer-Verlag, 2013.

[42] C. Kaliszyk and J. Urban. Stronger Automation for Flyspeck by Feature Weighting and Strategy Evolution. In *Proceedings of the 3rd International Workshop on Proof Exchange for Theorem Proving*, page To appear. EasyChair Proceedings in Computing, 2013.

[43] C. Kaliszyk, J. Urban, and J. Vyskocil. Machine Learner for Automated Reasoning 0.4 and 0.5.

In S. Schulz, L. de Moura, and B. Konev, editors, *Proceedings of the 4th Workshop on Practical Aspects of Automated Reasoning*, number 31 in EPiC Series in Computing, pages 60–66, 2015.

[44] Z. Khasidashvili and K. Korovin. Predicate Elimination for Preprocessing in First-order Theorem Proving. 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 361–372. Springer-Verlag, 2016.

[45] 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.

[46] 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.

[47] K. Korovin. Non-cyclic Sorts for First-order Satisfiability. In P. Fontaine, C. Ringeissen, and R. Schmidt, editors, *Proceedings of the International Symposium on Frontiers of Combining Systems*, number 8152 in Lecture Notes in Computer Science, pages 214–228, 2013.

[48] K. Korovin and C. Sticksel. A Note on Model Representation and Proof Extraction in the First-order Instantiation-based Calculus Inst-Gen. In R. Schmidt and F. Papacchini, editors, *Proceedings of the 19th Automated Reasoning Workshop*, pages 11–12, 2012.

[49] E. Kotelnikov, L. Kovacsi, G. Reger, and A. Voronkov. The Vampire and thew 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.

[50] 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.

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

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

[53] D.W. Loveland. *Automated Theorem Proving : A Logical Basis*. Elsevier Science, 1978.

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

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

[56] 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.

[57] T. Nipkow. Equational Reasoning in Isabelle. *Science of Computer Programming*, 12(2):123–149, 1989.

[58] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[59] J. Otten. leanCoP 2.0 and ileancop 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic 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 283–291, 2008.

[60] J. Otten. Restricting Backtracking in Connection Calculi. *AI Communications*, 23(2-3):159–182, 2010.

[61] J. Otten. A Non-clausal Connection Calculus. In K. Brünnler and G. Metcalfe, editors, *Proceedings of the 20th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, number 6793 in Lecture Notes in Artificial Intelligence, pages 226–241. Springer-Verlag, 2011.

[62] J. Otten. MleanCoP: A Connection Prover for First-Order Modal Logic. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Proceedings of the 7th International Joint Conference on Automated Reasoning*, number 8562 in Lecture Notes in Artificial Intelligence, pages 269–276, 2014.

[63] J. Otten. nanoCoP: A Non-clausal Connection Prover. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Proceedings of the 7th International Joint Conference on Automated Reasoning*, number 8562 in Lecture Notes in Artificial Intelligence, pages 302–312, 2016.

[64] J. Otten. Non-clausal Connection Calculi for Non-classical Logics. In C. Nalon and R. Schmidt, editors, *Proceedings of the 26th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, Lecture Notes in Artificial Intelligence, page To appear. Springer-Verlag, 2017.

[65] J. Otten and W. Bibel. leanCoP: Lean Connection-Based Theorem Proving. *Journal of Symbolic Computation*, 36(1-2):139–161, 2003.

[66] L. Paulson. A Generic Tableau Prover and its Integration with Isabelle. *Artificial Intelligence*, 5(3):73–87, 1999.

[67] 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, 2010.

[68] L.C. Paulson and T. Nipkow. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[69] 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.

[70] 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.

[71] A. Reynolds, M. Deters, V. Kuncak, C. Barrett, and C. Tinelli. Counterexample Guided Quantifier Instantiation for Synthesis in CVC4. In D. Kroening and C. Pasareanu, editors, *Proceedings of the 27th International Conference on Computer Aided Verification*, number 9207 in Lecture Notes in Computer Science, pages 198–216. Springer-Verlag, 2015.

[72] 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.

[73] 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.

[74] 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.

[75] A. Reynolds, C. Tinelli, D. Jovanovic, and C. Barrett. Designing Theory Solvers with Extensions. In C. Dixon and M. Finger, editors, *Proceedings of the 11th International Symposium on Frontiers of Combining Systems*, Lecture Notes in Computer Science, page To appear. Springer-Verlag, 2017.

[76] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

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

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

[79] P. Rümmer. E-Matching with Free Variables. In N. Bjørner and A. Voronkov, editors, *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 7180 in Lecture Notes in Artificial Intelligence, pages 359–374. Springer-Verlag, 2012.

[80] 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.

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

[82] S. Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proceedings of the Workshop on Empirically Successful First Order Reasoning, 2nd International Joint Conference on Automated Reasoning*, 2004.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] J. Slaney and B. Woltzenlogel Paleo. Conflict Resolution: a First-Order Resolution Calculus with Decision Literals and Conflict-Driven Clause Learning. *Journal of Automated Reasoning*, page To appear, 2017.

[88] A. Steen, M. Wisniewski, and C. Benzmüller. Agent-Based HOL Reasoning. In G-M. Greuel, T. Koch, P. Paule, and Sommese A., editors, *Proceedings of the 5th International Congress on Mathematical Software*, number 9725 in Lecture Notes in Computer Science, pages 75–81. Springer-Verlag, 2016.

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

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

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

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

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

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

2002.

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

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

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

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

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

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

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

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

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

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

[105] 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.

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

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

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

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

[110] G. Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 6355 in Lecture Notes in Artificial Intelligence, pages 1–12. Springer-Verlag, 2010.

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

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

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

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

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

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

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

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

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

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

[121] G. Sutcliffe. Proceedings of the 8th IJCAR ATP System Competition. Coimbra, Portugal, 2016.

http://www.tptp.org/CASC/J8/Proceedings.pdf.

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

[123] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, page To appear, 2017.

[124] 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.

[125] 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.

[126] 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.

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

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

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

[130] G. Sutcliffe and C.B. Suttner, editors. *Special Issue: The CADE-13 ATP System Competition*, volume 18, 1997.

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

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

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

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

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

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

[137] J. Urban. MaLARea: a Metasystem for Automated Reasoning in Large Theories. In J. Urban, G. Sutcliffe, and S. Schulz, editors, *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories*, number 257 in CEUR Workshop Proceedings, pages 45–58, 2007.

[138] J. Urban. BliStr: The Blind Strategymaker. arXiv:1301.2683, 2013.

[139] J. Urban, G. Sutcliffe, P. Pudlak, and J. Vyskocil. MaLARea SG1: Machine Learner for Automated Reasoning with Semantic Guidance. 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 441–456. Springer-Verlag, 2008.

[140] 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.

[141] 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.

[142] B. Woltzenlogel Paleo. First-Order Conflict-Driven Clause Learning from a Proof-Theoretical Perspective. In G. Dowek, C. Dubois, B. Pientka, and F. Rabe, editors, *Universality of Proofs*, number 16421 in Dagstuhl Seminar, 2016.