CASC-J10

CASC-J10
CASC-J10
CASC-J10

Proceedings of the 10th IJCAR ATP System Competition (CASC-J10)

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 10th IJCAR ATP System Competition (CASC-J10) was held on 2th July 2020. 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-J10 was held on 2nd July 2020, as part of the 10th International Joint Conference on Automated Reasoning (IJCAR 2020)[1], Online, Earth. It was the twenty-fifth competition in the CASC series [139, 145, 142, 95, 97, 138, 136, 137, 102, 104, 106, 108, 111, 113, 115, 117, 119, 121, 123, 144, 125, 127, 130, 132].

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

Twenty-five ATP system versions, listed in Table 1 and 2, entered into the various competition and demonstration divisions. The winners of the CASC-27 (the previous CASC) divisions were automatically entered into the corresponding demonstration divisions, to provide benchmarks against which progress can be judged (the competition archive provides access to the systems' executables and source code).

The design and procedures of this CASC evolved from those of previous CASCs [139, 140, 135, 141, 93, 94, 96, 98, 99, 100, 101, 103, 105, 107, 110, 112, 114, 116, 118, 120, 122, 124, 126, 129, 131]. Important changes for CASC-J10 were:

[1] CADE was a constituent conference of IJCAR, hence CASC-"J10".

ATP System	Divisions	Entrant (Associates)	Entrant's Affiliation
ATPBoost 1.0	LTB (demo)	Bartosz Piotrowski (Josef Urban, Stephan Schulz)	University of Warsaw & Czech Technical University in Prague
CSE 1.3	FOF	Feng Cao (Yang Xu, Jian Zhong, Peiyao Liu, Guanfeng Wu, Jun Liu, Shuwei Chen, Xingxing He, Peng Xu, Qinghua Liu, Huimin Fu, Zhenming Song)	Southwest Jiaotong University
CSE_E 1.2	FOF	Feng Cao (Yang Xu, Stephan Schulz, Peiyao Liu, Jian Zhong, Jun Liu, Zhihui Li, Shuwei Chen, Guanfeng Wu, Keming Wang, Xiaodong Guan, Zhenming Song)	Southwest Jiaotong University
CVC4 1.8	THF TFA FOF FNT	Andrew Reynolds (Haniel Barbosa, Cesare Tinelli, Clark Barrett)	University of Iowa
E 2.4	UEQ (demo)	CASC	CASC-27 winner
E 2.5	FOF FNT UEQ LTB	Stephan Schulz	DHBW Stuttgart
Enigma 0.5.1	FOF	Jan Jakubuv (Josef Urban, Mirek Olšák, Karel Chvalovský, Stephan Schulz)	Czech Technical University in Prague
Etableau 0.2	FOF UEQ	John Hester	University of Florida
GKC 0.5	FOF UEQ LTB	Tanel Tammet	Tallinn University of Technology
iProver 3.3	FOF FNT EPR UEQ LTB	Konstantin Korovin (André Duarte, Edvard Holden, Julio Cesar Lopez Hernandez)	University of Manchester
lazyCoP 0.1	FOF UEQ	Michael Rawson	University of Manchester
leanCoP 2.2	FOF	Jens Otten	University of Oslo
LEO-II 1.7.0	THF	Alexander Steen (Christoph Benzmüller)	University of Luxembourg

Table 1: The ATP systems and entrants

ATP System	Divisions	Entrants (Associates)	Entrant's Affiliation
Leo-III 1.4	LTB	CASC	CASC-27 winner
Leo-III 1.5	THF LTB	Alexander Steen (Christoph Benzmüller)	University of Luxembourg
MaLARea 0.9	LTB	Josef Urban (Jan Jakubuv, Cezary Kaliszyk, Bartosz Piotrowski, Stephan Schulz, Mirek Olšák)	Czech Technical University in Prague
nanoCoP 1.1	FOF	Jens Otten	University of Oslo
Prover9 1109a	FOF (demo)	CASC (William McCune, Bob Veroff)	CASC fixed point
PyRes 1.3	FOF FNT EPR	Stephan Schulz	DHBW Stuttgart
Satallax 3.4	THF	CASC	CASC-27 winner
Satallax 3.5	THF	Michael Färber	Inria, LSV, ENS Paris-Saclay
Twee 2.2.1	FOF UEQ	Nick Smallbone (Koen Claessen)	Chalmers University of Technology
Vampire 4.4	TFA FOF FNT (demo)	CASC	CASC-27 winner
Vampire 4.5	THF TFA FOF FNT UEQ	Giles Reger (Martin Suda, Andrei Voronkov,Evgeny Kotelnikov, Laura Kovacs, Martin Riener, Michael Rawson, Bernhard Gleiss, Jakob Rath, Ahmed Bhayat, Petra Hozzova, Johannes Schoisswohl)	University of Manchester
Zipperposition 2.0	THF FOF LTB	Petar Vukmirovic (Alexander Bentkamp, Simon Cruanes, Sophie Tourret, Visa Nummelin)	Vrije Universiteit Amsterdam

Table 2: The ATP systems and entrants, continued

- All divisions used a wall clock time limit, to promote use of all the cores on the CPU.
- The EPR division was placed on hiatus.
- A new variant of the SotAC measure was adopted, to allow comparison of SotAC values between CASC editions.

The competition organizer was Geoff Sutcliffe. CASC is overseen by a panel of knowledgeable researchers who are not participating in the event. The CASC-J10 panel members were Pascal Fontaine, Andre Platzer, and Christoph Weidenbach. The competition was run on computers provided by StarExec at the University of Miami. The CASC-J10 web site provides access to resources used before, during, and after the event: `http://www.tptp.org/CASC/J10`

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 two problem categories:

- The **TFI** category: TFA with only Integer 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 **UEQ** division: Unit EQuality clause normal form theorems (unsatisfiable clause sets).

The **LTB** division: Theorems (axioms with a provable conjecture) from Large Theories, presented in Batches. A large theory has many functions and predicates, and many axioms of which typically only a few are required for the proof of a theorem. The problems in a batch are given to an ATP system all at once, and typically have a common core set of axioms. The batch presentation allows the ATP systems to load and preprocess the common core set of axioms just once, and to share logical and control results between proof searches. Each problem category might be accompanied by a set of training problems and their solutions, taken from the same source as the competition problems. The training data can be used for ATP system tuning during (typically at the start of) the competition. In CASC-J10 the LTB division had one problem category:

- The **HL4** category: Problems exported from HOL4 [91].

Eight versions of each problem were provided - two FOF versions, two TF0 versions, one TF1 version, two TH0 versions, and one TH1 version. Systems could attempt as many of the versions as they want, in any order including in parallel, and a solution to any version counted as a solution to the problem. Some of the HL4 problems had less than 100 axioms, and some had over 100000 axioms.

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

2.2 The Demonstration Division

ATP systems that cannot run in the competition divisions for any reason (e.g., the system requires special hardware, the system is a previous winner, or the entrant is an organizer) can be entered into the demonstration division. Demonstration division systems can run on the competition computers, or the computers can be supplied by the entrant. 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 trophies or prizes are awarded.

3 Infrastructure

3.1 Computers

The competition computers had:

- Two octa-core Intel(R) Xeon(R) E5-2667, 3.20GHz CPUs
- 256GB memory
- The CentOS Linux release 7.4.1708 operating system, kernel 3.10.0-957.12.2.el7.x86_64.

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

3.2 Problems

3.2.1 Problem Selection

The problems for the THF, TFA, FOF, FNT, and UEQ divisions were taken from the TPTP Problem Library [128], version v7.4.0. 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 are syntactically non-propositional.
- The TPTP uses system performance data in the Thousands of Solutions from Theorem Provers (TSTP) solution library to compute problem difficulty ratings in the range 0.00 (easy) to 1.00 (unsolved) [143]. 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 [95].
- 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 LTB division are taken from various sources, with each problem category being based on one source. Entrants are expected to honestly not use publicly available problem sets for tuning before the competition. The process for selecting problems depends on the problem source.

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 [31] (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 limit imposed on each solution attempt - see Section 3.3. The numbers of problems to be used in each division of the competition are determined from the number of computers available, the time allocated to the competition, the number of ATP systems to be run on the competition computers over the divisions, and the time limit impoased on each solution attempt, according to the following relationship:

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

It is a lower bound on the number of problems because it assumes that every system uses all of the time limit for each problem. Since some solution attempts succeed before the time limit

is reached, more problems can be used. The numbers of problems used in the categories in the various divisions are (roughly) proportional to the numbers of eligible problems, after taking into account the limitation on very similar problems, determined according to the judgement of the competition organizers.

In the LTB division the number of problems in each problem category is determined by the number of problems in the corresponding problem set. In CASC-J10 the HL4 problem category had 10000 problems (with eight versions of each problem).

3.2.3 Problem Preparation

The problems are given to the ATP systems in TPTP format, with `include` directives. In order to ensure that no system receives an advantage or disadvantage due to the specific presentation of the problems in the TPTP, the problems in the TPTP-based divisions are obfuscated by:

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

In the In the LTB division the formulae are not obfuscated, thus allowing the ATP systems to take advantage of natural structure that occurs in the problems.

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

3.2.4 Batch Specification Files

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

- A problem category line of the form
  `division.category LTB.`*category_mnemonic*
- 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. The language of a solution might not be the same as the language of the problem, e.g., a proof to a THF problem might be written in FOF, or the proof of a TFF problem might be written in THF.

Each batch specification consists of:

- A header line `% SZS start BatchConfiguration`
- A specification of whether or not the problems in the batch must be attempted in order is given, in a line of the form
  `execution.order` *ordered/unordered*
  If the batch is ordered the ATP systems may not start any attempt on a problem, including reading the problem file, before ending the attempt on the preceding problem. For CASC-J10 it was
  `execution.order unordered`
- A specification of what output is required from the ATP systems for each problem, in a line of the form
  `output.required` *space_separated_list*
  where the available list values are the SZS values `Assurance`, `Proof`, `Model`, and `Answer`. For CASC-J10 it was
  `output.required Proof`.
- The wall clock time limit per problem, in a line of the form
  `limit.time.problem.wc` *limit_in_seconds*
  A value of zero indicates no per-problem limit. For CASC-J10 it was
  `limit.time.problem.wc 0`
- The overall wall clock time limit for the batch, in a line of the form
  `limit.time.overall.wc` *limit_in_seconds*
- A terminator line `% SZS end BatchConfiguration`
- A header line `% SZS start BatchIncludes`
- `include` directives that are used in every problem. All the problems in the batch have these `include` directives, and can also have other `include` directives that are not listed here. For CASC-J10, see the additional notes below.
- A terminator line `% SZS end BatchIncludes`
- A header line `% SZS start BatchProblems`
- Pairs of problem file names (relative to the directory holding the batch specification file), and output file names where the output for the problem must be written. The output files must be written in the directory specified as the second argument to the `starexec_run` script (the first argument is the name of the batch specification file). For CASC-J10, see the additional notes below.
- A terminator line `% SZS end BatchProblems`

Additional Notes for CASC-J10

- In the `BatchProblems` section, the multiple versions of each problem are specified using UNIX `*` globbing, e.g., `HL400001*.p`. The versions of each problem have extensions as follows: the first FOF version uses `+4`, the second FOF version uses `+5`, the first TF0 version uses `_4`, the second TF0 version uses `_5`, the only TF1 version uses `_7`, the first TH0 version uses `^4`, the second TH0 version uses `^5`, and the only TH1 version uses `^7`.
- Proof output must identify which version of the problem was solved - see Section 6.1. The proof may not have irrelevant output (e.g., from other threads running attempts on other versions of the problem) interleaved in the proof.
- In the `BatchIncludes` section (not in problem files), multiple versions of included axiom files may be specified using UNIX `*` globbing, e.g., `include('Axioms/HL4001*.ax')` could refer to all of `HL4002+4.ax`, `HL4002_4.ax`, `HL4002^4.ax`, `HL4002+5.ax`, `HL4002_5.ax`,

`HL4002^5.ax`, `HL4002_7.ax`, `HL4002^7.ax`. For a given problem, systems should use only the axiom files whose version matches that of the problem file (there may be none), e.g., if the problem version is `+4` then use only the axiom files with the version `+4`. Using any other versions could lead to weird results.

3.3 Resource Limits

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

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

4 System Evaluation

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

The systems are ranked in the competition divisions according to the number of problems solved with an acceptable proof/model output. Ties are broken according to the average wall clock time taken over problems solved. Trophies are awarded to the competition divisions' 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 that use translations from one form to another, e.g., translation of FOF problems to CNF, the translations must be adequately documented.
- Derivations must show only relevant inference steps.
- Inference steps must document the parent formulae, the inference rule used, and the inferred formula.
- Inference steps must be reasonably fine-grained.
- An unsatisfiable set of ground instances of clauses is acceptable for establishing the unsatisfiability of a set of clauses.

- Models must be complete, documenting the domain, function maps, and predicate maps. The domain, function maps, and predicate maps may be specified by explicit ground lists (of mappings), or by any clear, terminating algorithm.

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

- The *state-of-the-art contribution* (SotAC) quantifies the unique abilities of each system. For each problem solved by a system, its SotAC for the problem is

$$1 - FractionOfSystemsThatSolvedTheProblem$$

  and a system's overall SotAC is its average for the problems it solves but which are not solved by all the systems.
- The *core usage* is the average of the ratios of CPU time to wall clock time used, over the problems solved. This measures the extent to which the systems take advantage of multiple cores.
- The *efficiency* measure combines the number of problems solved with the time taken. It is the average of the inverses of the times taken for problems solved, multiplied by the fraction of problems solved. This can be interpreted intuitively as the average of the solution rates for problems solved, multiplied by the fraction of problems solved. Efficiency is computed for both CPU time and wall clock time, to measure how efficiently the systems use one core and how efficiently systems use 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 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 demonstration divisions, to provide benchmarks against which progress can be judged. Prover9 1109a

is automatically entered into the FOF division, to provide a fixed-point against which progress can be judged.

5.1 System Descriptions

A system description has to be provided for each ATP system entered, using the HTML schema supplied on the CASC web site. (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: `SEU140+2`
- FNT: `NLP042+1` and `SWV017+1`
- UEQ: `BOO001-1`
- LTB: For CASC-J10, all of the variants of `HL400001` that the system can attempt.

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, UEQ, 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 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
```

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

```
% 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 [109] 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. Tthe 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 [134] is encouraged.

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 ATPBoost 1.0

Bartosz Piotrowski
University of Warsaw, Poland

Architecture

ATPBoost 1.0 [69], is a metasystem for ATP in large theories relying on consistent naming of symbols and formulae. It learns premise selection from examples of successful proofs using several machine learning methods (k-nearest neighbors, gradient boosted trees, recurrent neural networks, and graph neural networks). The underlying deductive system is (currently) E prover. The system addresses the fact of a multiplicity of possible proofs. The learning process is reinforced by a feedback loop between the learners and the prover during which new training proofs may be found and wrong classifications of premises may be corrected. The version for CASC J10 will (likely) use non-neural machine learning methods only (due to time and hardware limitations, as the neural methods are time-consuming and often require GPU for efficient training).

Strategies

The main strategy of ATPBoost 1.0 for learning premise selection is to run multiple iterations of the learning-proving feedback loop. During the learning part, machine learning models are trained on the available proofs to select relevant axioms. During the proving part, the prover is run multiple times with axioms advised by the trained models. Newly found proofs give

a feedback signal to the learners: highly-ranked premises which were not used in any of the proofs are returned as negative training examples, and new positive examples are extracted from the new proofs. The machine learning models trained in such a way provide premise-selection advice for ATP attempts for new conjectures. For each conjecture, the prover is run with several axiom limits.

Implementation

The metasystem is implemented in Python. It uses several machine learning libraries (XGBoost, OpenNMT, TensorFlow, Scikit-learn).

ATPBoost is available from

`https://github.com/BartoszPiotrowski/ATPboost`

Expected Competition Performance

ATPBoost is most useful when the number of available axioms is very large but the problems itself are not very difficult provided relevant axioms are selected. ATPBoost has good performance on the MPTP Challenge. The performance on the type of problems in the current LTB division is unknown!

7.2 CSE 1.3

Feng Cao
Southwest Jiaotong University, China

Architecture

CSE 1.3 is a developed prover based on the last version of CSE 1.2. It is an automated theorem prover for first-order logic without equality mainly based on a novel inference mechanism, called as Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction (S-CS) [156], which is able to handle multiple (two or more) clauses dynamically in a synergized way in one deduction step, while binary resolution is its special case. CSE 1.3 also adopts conventional factoring, equality resolution (ER rule), and variable renaming. Some preprocessing techniques, including pure literal deletion and simplification based on the distance to the goal clause, and a number of standard redundancy criteria for pruning the search space: tautology deletion, subsumption (forward and backward) are applied as well.

CSE 1.3 has been improved, compared with CSE 1.2, mainly from the following aspects:

1. Optimization of contradiction separation algorithm based on full usage of clauses, which is able to increase the sufficiency of clauses participating in deduction.

2. Optimization of contradiction separation algorithm based on optimized deduction path, which is able to effectively control the unifier complexity in the deduction process.

3. Dynamic adjustment of clause and literal weight update in the deduction process.

Internally, CSE 1.3 works only with clausal normal form. E prover [88] is adopted with thanks for clausification of full first-order logic problems during preprocessing.

Strategies

CSE 1.3 inherited most of the strategies in CSE 1.2. The main new strategies are:

- SCSs control strategy. Control the number of literals in SCSs generated in the deduction process, and the SCSs meet the set contradictions will be retained.
- Clause weight update strategy. Different update methods are applied in different clause deduction stages.
- Literal weight update strategy. Flexible update methods are used for different literal deduction stages.

Implementation

CSE 1.3 is implemented mainly in C++, and JAVA is used for batch problem running implementation. Shared data structure is used for constants and shared variables storage. In addition, special data structure is designed for property description of clause, literal and term, so that it can support the multiple strategy mode. E prover is used for clausification of FOF problems, and then TPTP4X is applied to convert the CNF format into TPTP format.

Expected Competition Performance

CSE 1.3 has made some improvements compared to CSE 1.2, and so we expect a better performance in this year's competition.

Acknowledgement:
Development of CSE 1.3 has been partially supported by the National Natural Science Foundation of China (NSFC) (Grant No.61673320).

7.3 CSE_E 1.2

Feng Cao
Southwest Jiaotong University, China

Architecture

CSE_E 1.2 is an automated theorem prover for first-order logic by combining CSE 1.3 and E 2.4, where CSE is based on the Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction (S-CS) [156] and E is mainly based on superposition. The combination mechanism is like this: E and CSE are applied to the given problem sequentially. If either prover solves the problem, then the proof process completes. If neither CSE nor E can solve the problem, some inferred clauses, especially unit clauses, by CSE will be fed to E as lemmas, along with the original clauses, for further proof search. This kind of combination is expected to take advantage of both CSE and E, and produce a better performance. Concretely, CSE is able to generate a good number of unit clauses, based on the fact that unit clauses are helpful for proof search and equality handling. On the other hand, E has a good ability on equality handling.

Strategies

The strategies of CSE part of CSE_E 1.2 take the same strategies as in CSE 1.3 standalone, e.g., clause/literal selection, strategy selection, and CSC strategy. The main new strategies for the combined systems are:

- Lemma filtering mainly based on variable appearance.
- Dynamic time allocation scheme in different run stages.

Implementation

CSE_E 1.2 is implemented mainly in C++, and JAVA is used for batch problem running implementation. The job dispatch between CSE and E is implemented in JAVA.

Expected Competition Performance

We expect CSE_E 1.2 to solve some hard problems and have a satisfying performance.

Acknowledgement:
Development of CSE 1.3 has been partially supported by the National Natural Science Foundation of China (NSFC) (Grant No.61673320). Stephan Schulz for his kind permission on using his E prover that makes CSE_E possible.

7.4 CVC4 1.8

Andrew Reynolds
University of Iowa, USA

Architecture

CVC4 [7] is an SMT solver based on the DPLL(T) architecture [57] 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 conflict-based instantiation and 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 [76].

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 [78]. 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 [79]. If no instances are added, it reports "satisfiable".

CVC4 has native support for problems in higher-order logic, as described in recent work [6]. It uses a pragmatic approach for HOL, where lambdas are eliminated eagerly via lambda lifting. The approach extends the theory solver for quantifier-free uninterpreted functions (UF) and E-matching. For the former, the theory solver for UF in CVC4 now handles equalities between functions using an extensionality inference. Partial applications of functions are handle using a (lazy) applicative encoding where some function applications are equated to the applicative encoding. For the latter, several of the data structures for E-matching have been modified to incorporate matching in the presence of equalities between functions, function variables, and partial function applications.

Strategies

For handling theorems, CVC4 primarily uses conflict-based quantifier instantiation [77, 5], enumerative instantiation [74] 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 [75], which select relevant quantifier instantiations based on models for counterexamples to quantified formulas. At the quantifier-free level, CVC4 uses standard decision procedures for linear arithmetic and uninterpreted functions, as well as heuristic approaches for handling non-linear arithmetic [80].

Implementation

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

`https://github.com/CVC4`

Expected Competition Performance

The first-order theorem proving and finite model finding capabilities of CVC4 have not changed much in the past year. Its heuristics for linear and non-linear arithmetic have changed slightly, which should impact TFA. Its higher-order capabilities have undergone some minor improvements, which should lead to slightly better performance in THF.

7.5 E 2.4

Stephan Schulz
DHBW Stuttgart, Germany

Architecture

E 2.4pre [84, 88, 89] is a purely equational theorem prover for many-sorted first-order logic with equality. It consists of an (optional) clausifier for pre-processing full first-order formulae into clausal form, and a saturation algorithm implementing an instance of the superposition calculus with negative literal selection and a number of redundancy elimination techniques. E

is based on the DISCOUNT-loop variant of the given-clause algorithm, i.e., a strict separation of active and passive facts. No special rules for non-equational literals have been implemented. Resolution is effectively simulated by paramodulation and equality resolution. As of E 2.1, PicoSAT [17] can be used to periodically check the (on-the-fly grounded) proof state for propositional unsatisfiability.

For the LTB divisions, a control program uses a SInE-like analysis to extract reduced axiomatizations that are handed to several instances of E. E will not use on-the-fly learning this year.

Strategies

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

For CASC-27, 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 230 different strategies have been thoroughly evaluated on all untyped first-order problems from TPTP 7.2.0. In addition, we have explored some parts of the heuristic parameter space with a short time limit of 5 seconds. This allowed us to test about 650 strategies on all TPTP problems, and an extra 7000 strategies on all 1193 UEQ problems from TPTP 7.2.0. About 100 of these strategies are used in the automatic mode, and about 450 are used in at least one schedule.

Implementation

E is build around perfectly shared terms, i.e. each distinct term is represented only 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 [87]. Superposition and backward rewriting use fingerprint indexing [86], 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 [52, 53]. The prover and additional information are available at

`https://www.eprover.org`

Expected Competition Performance

E 2.4 was the CASC-27 UEQ winner.

7.6 E 2.5

Stephan Schulz
DHBW Stuttgart, Germany

Architecture

E 2.5pre [84, 88, 89] is a purely equational theorem prover for many-sorted first-order logic with equality. It consists of an (optional) clausifier for pre-processing full first-order formulae into clausal form, and a saturation algorithm implementing an instance of the superposition calculus with negative literal selection and a number of redundancy elimination techniques. E is based on the DISCOUNT-loop variant of the *given-clause* algorithm, i.e., a strict separation of active and passive facts. No special rules for non-equational literals have been implemented. Resolution is effectively simulated by paramodulation and equality resolution. As of E 2.1, PicoSAT [17] can be used to periodically check the (on-the-fly grounded) proof state for propositional unsatisfiability.

For the LTB divisions, a control program uses a SInE-like analysis to extract reduced axiomatizations that are handed to several instances of E. E will not use on-the-fly learning this year.

Strategies

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

For CASC-J10, E implements a strategy-scheduling automatic mode. The total CPU time available is broken into several (unequal) time slices. For each time slice, the problem is classified into one of several classes, based on a number of simple features (number of clauses, maximal symbol arity, presence of equality, presence of non-unit and non-Horn clauses, possibly presence of certain axiom patterns, ...). For each class, a schedule of strategies is greedily constructed from experimental data as follows: The first strategy assigned to a schedule is the the one that solves the most problems from this class in the first time slice. Each subsequent strategy is selected based on the number of solutions on problems not already solved by a preceding strategy.

About 130 different strategies have been thoroughly evaluated on all untyped first-order problems from TPTP 7.3.0. We have also explored some parts of the heuristic parameter space

with a short time limit of 5 seconds. This allowed us to test about 650 strategies on all TPTP problems, and an extra 7000 strategies on UEQ problems from TPTP 7.2.0. About 100 of these strategies are used in the automatic mode, and about 450 are used in at least one schedule.

Implementation

E is build around perfectly shared terms, i.e. each distinct term is only represented once in a term bank. The whole set of terms thus consists of a number of interconnected directed acyclic graphs. Term memory is managed by a simple mark-and-sweep garbage collector. Unconditional (forward) rewriting using unit clauses is implemented using perfect discrimination trees with size and age constraints. Whenever a possible simplification is detected, it is added as a rewrite link in the term bank. As a result, not only terms, but also rewrite steps are shared. Subsumption and contextual literal cutting (also known as subsumption resolution) is supported using feature vector indexing [87]. Superposition and backward rewriting use fingerprint indexing [86], 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 [52, 53]. The prover and additional information are available at

`https://www.eprover.org`

Expected Competition Performance

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

7.7 Enigma 0.5.1

Jan Jakubuv
Czech Technical University in Prague, Czech Republic

Architecture

ENIGMA (Efficient learNing-based Inference Guiding MAchine) [42, 43, 44, 30, 22, 40] is an efficient implementation of learning-based guidance for given clause selection in saturation-based automated theorem provers. Clauses from many proof searches are classified as positive and negative based on their participation in the proofs. An efficient classification model is trained on this data, using fast feature-based characterization of the clauses. The learned model is then tightly linked with the core prover and used as a basis of a parameterized evaluation heuristic that provides fast ranking of all generated clauses. ENIGMA 0.4 uses E as its underlying prover. The CASC-27 FOF competition version will most likely use a classifier based on gradient-boosted trees (XGBoost or LightGBM) and clause features that abstract from symbol and formula/clause names. We may also include neural classifiers based on clause structure. The system will be pre-trained on the latest public TPTP library and also use a portfolio of strategies pre-trained for TPTP with BliStr(Tune) [149, 41]. The system may also include strategies for large TPTP problems used previously in the E.T. system.

Strategies

The core system is a modification of E adding learning-based evaluation of the generated clauses. The system is pre-trained on TPTP using abstracted characterization of clauses, and the trained classifier is used during the competition as an additional clause evaluation heuristic that is combined in various ways with the core E strategies.

Implementation

The system modifies E's implementation by adding various features extraction methods and linking E with efficient learning-based classifiers (tree-based, linear and neural). ENIGMA is available at:

`https://github.com/ai4reason/enigmatic`

Expected Competition Performance

ENIGMA should be able to improve on E's auto schedule in FOF.

7.8 Etableau 0.2

John Hester
University of Florida, USA

Architecture

Etableau 0.2 is an automated theorem prover built alongside E [88], using it as a library. Etableau at a basic level implements the clausal connection calculus, with enforced regularity, folding up, local unification, and other improvements. The greatest distinguishing feature of Etableau is a novel calculus using E's saturation to close local branches of tableaux. If a branch is local (shares no variables with other branches of the tableau) and cannot be closed by other means, the branch is saturated as an independent problem. If a contradiction is found, this local branch can now be marked as closed. If a local branch is found to be satisfiable, the problem is satisfiable.

Strategies

Etableau uses iterative deepening to control search through the space of tableaux. At a given maximum depth, tableaux to be expanded on are chosen based on the ratio of branches that have been closed. In other words, tableaux that have many closed branches and a small number of remaining open branches are processed first. In situations where attempts to close branches by saturation are made, E is used with its auto heuristic.

Implementation

Etableau is written in C, using many data structures and functions from E and with many added. Etableau is available at this github repository:

`https://github.com/hesterj/E-TAB`

Expected Competition Performance

Etableau is expected to perform well on problems with smaller numbers of axioms, and struggle on problems with many axioms.

7.9 GKC 0.5

Tanel Tammet
Tallinn University of Technology, Estonia

Architecture

GKC [146] is a resolution prover optimized for search in large knowledge bases. It is used as a foundation (GK Core) for building a common-sense reasoner GK. In particular, GK can handle inconsistencies and perform probabilistic and nonmonotonic reasoning, see [147]. We envision natural language question answering systems as the main potential application for these specialized methods.

These standard inference rules have been implemented in GKC:

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

GKC does not currently implement any propositional inferences or instance generation.

GKC splits the multiple strategies it decides to try between several forked instances. For the competition the plan is to use eight forks. Each fork runs a subset of strategies sequentially.

Strategies

For CASC GKC uses multiple strategies run sequentially, with the time limit starting at one second for each, increased 5 times once the whole batch has been perforfmed. The strategy selection takes into consideration the basic properties of the problem, in particular its approximate size. There is no interplay between different strategies.

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

GKC only looks for proofs and does not try to show non-provability.

Implementation

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

GKC can be obtained from

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

Expected Competition Performance

Compared to the performance in previous CASC, GKC 0.5 should perform somewhat better. In particular, more search strategies have been implemented and the selection of search strategies is wider and more varied. The core algorithms and data structures remain the same. We expect GKC to be in the middle of the final ranking for FOF and below the average in UEQ and LTB. We expect GKC to perform well on very large problems.

7.10 iProver 3.3

Konstantin Korovin
University of Manchester, United Kingdom

Architecture

iProver [48] is an automated theorem prover based on an instantiation calculus Inst-Gen [49, 27], which is complete for first-order logic. iProver approximates first-order clauses using propositional abstractions which are solved using MiniSAT [26] and refined using model-guided instantiations. iProver combines instantiation, resolution and superposition [25] and is extended with a general abstraction-refinement framework for under-and over-approximations [32, 33].

Recent features in iProver include:

- Superposition calculus together with simplifications such as demodulation, light normalisation, subsumption, subsumption resolution and global subsumption. iProver's simplification set up [25] is tunable via command line options and generalises common architectures such as Discount or Otter.
- Heuristics used in iProver are learnt using dynamic clustering and hyper-parameter optimisation using SMAC [39] as described in [38, 37].
- In the LTB division iProver combines abstraction-refinement with axiom selection based on the SinE algorithm [36] as implemented in Vampire [50]. iProver will run in parallel most promising learnt heuristics.

Strategies

iProver has around 100 options to control the proof search including options for literal selection, passive clause selection, frequency of calling the SAT solver, simplifications and options for combination of instantiation with resolution and superposition. By default 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. In the competition we will also use a selection of learnt strategies.

Implementation

iProver is implemented in OCaml and for the ground reasoning uses MiniSat [26]. iProver accepts FOF, TFF and CNF formats. Vampire [50, 34] and E prover [88] are used for proof-producing clausification of FOF/TFF problems, Vampire is also used for SInE axiom selection [36] in the LTB division. iProver is available at:

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

Expected Competition Performance

We expect improvement in performance compared to the previous year due to improved implementation of superposition, simplifications and heuristic selection.

7.11 lazyCoP 0.1

Michael Rawson
University of Manchester, United Kingdom

Architecture
lazyCoP 0.1 is a connection-tableaux system for first-order logic with equality. It implements the lazy paramodulation calculus described in [68], with some additional inferences such as "shortcut" strict rules and equality lemmas. The system implements well-known refinements of the predicate connection calculus, such as tautology elimination and strong regularity, and these are lifted to equalities where appropriate. The resulting system appears to be complete, but we make no theoretical claim one way or another.

The system was originally conceived to efficiently accommodate a machine-learned heuristic guidance system: this system is not yet guided in this way, but learned heuristics are intended for a future version.

Strategies

The system explores a tableaux-level search space using the classic A* informed-search algorithm. The (admissible) heuristic function is the number of open branches. Typical connection systems explore via some kind of iterative deepening: A* search is a necessity for future learned guidance, and is not as catastrophic in memory consumption as might be expected. No form of strategy scheduling is yet implemented and the system will run for the entire time allowed on all available cores.

Implementation

A finite tree of inference rules forms a search space. To expand a selected leaf node, the system traverses from root to leaf, applying each rule to a new empty tableau. Possible inferences from the resulting tableau are added to the leaf and the resulting nodes are enqueued.

The system does not yet include a custom clausification routine: a recent build of Vampire is employed for this purpose. lazyCoP is implemented entirely in the Rust programming language, allowing tight control over memory allocation and layout while avoiding some classes of memory- and thread- safety bugs. The source code (likely to be incomplete and/or buggy up to and including the competition!) is available at:

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

Expected Competition Performance

Performance on problems without equality is hoped to be comparable with other connection systems, if slightly slower. Problems requiring a modest amount of equational reasoning (or problems requiring no equational reasoning but containing extraneous equality axioms) are not expected to perform well, but should not cause catastrophic blowup either. Pure-equality problems (such as UEQ) are not the intended domain and do not perform well, but the first author remains hopeful for At Least One Problem.

7.12 leanCoP 2.2

Jens Otten
University of Oslo, Norway

Architecture

leanCoP [67, 58] is an automated theorem prover for classical first-order logic with equality. It is a very compact implementation of the connection (tableau) calculus [16, 51].

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 [59]. leanCoP uses an optimized structure-preserving transformation into clausal form [59] and a fixed strategy schedule that is controlled by a shell script.

Implementation

leanCoP is implemented in Prolog. The source code of the core prover 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.

leanCoP can read formulae in leanCoP syntax and in TPTP first-order syntax. Equality axioms and axioms to support distinct objects are automatically added if required. The leanCoP core prover returns a very compact connection proof, which is then translated into a more comprehensive output format, e.g., into a lean (TPTP-style) connection proof or into a readable text proof.

The source code of leanCoP 2.2 is available under the GNU general public license. It can be downloaded from the leanCoP website at:

`http://www.leancop.de`

The website also contains information about ileanCoP [58] and MleanCoP [61, 62], two versions of leanCoP for first-order intuitionistic logic and first-order modal logic, respectively.

Expected Competition Performance

As the prover has not changed, the performance of leanCoP 2.2 is expected to be the same as last year.

7.13 LEO-II 1.7.0

Alexander Steen
University of Luxembourg, Luxembourg

Architecture

LEO-II [11], the successor of LEO [10], is a higher-order ATP system based on extensional higher-order resolution. More precisely, LEO-II employs a refinement of extensional higher-order RUE resolution [9]. LEO-II is designed to cooperate with specialist systems for fragments of higher-order logic. By default, LEO-II cooperates with the first-order ATP system E [83]. LEO-II is often too weak to find a refutation amongst the steadily growing set of clauses on its own. However, some of the clauses in LEO-II's search space attain a special status: they are first-order clauses modulo the application of an appropriate transformation function. Therefore, LEO-II launches a cooperating first-order ATP system every n iterations of its (standard) resolution proof search loop (e.g., 10). If the first-order ATP system finds a refutation, it communicates its success to LEO-II in the standard SZS format. Communication between LEO-II and the cooperating first-order ATP system uses the TPTP language and standards.

Strategies

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

Implementation

LEO-II is implemented in OCaml 4, and its problem representation language is the TPTP THF language [12]. In fact, the development of LEO-II has largely paralleled the development of the TPTP THF language and related infrastructure [133]. LEO-II's parser supports the TPTP THF0 language and also the TPTP languages FOF and CNF.

Unfortunately the LEO-II system still uses only a very simple sequential collaboration model with first-order ATPs instead of using the more advanced, concurrent and resource-adaptive OANTS architecture [13] as exploited by its predecessor LEO.

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

`http://www.leoprover.org`

Expected Competition Performance

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

7.14 Leo-III 1.4

Alexander Steen
University of Luxembourg, Luxembourg

Architecture

Leo-III [92] [SB18], the successor of LEO-II [11] is a higher-order ATP system based on extensional higher-order paramodulation with inference restrictions using a higher-order term ordering. The calculus contains dedicated extensionality rules and is augmented with equational simplification routines that have their intellectual roots in first-order superposition-based theorem proving. The saturation algorithm is a variant of the given clause loop procedure inspired by the first-order ATP system E.

Leo-III cooperates with external first-order ATPs which are called asynchronously during proof search; a focus is on cooperation with systems that support typed first-order (TFF) input. For this year's CASC, CVC4 [7] and E [84, 88] are used as external systems. However, cooperation is in general not limited to first-order systems. Further TPTP/TSTP-compliant external systems (such as higher-order ATPs or counter model generators) may be included using simple command-line arguments. If the saturation procedure loop (or one of the external provers) finds a proof, the system stops, generates the proof certificate and returns the result.

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

Strategies

Leo-III comes with several configuration parameters that influence its proof search by applying different heuristics and/or restricting inferences. These parameters can be chosen manually by the user on start-up. Leo-III currently does not offer portfolio scheduling (time slicing) nor automatic selection of configuration parameters that seems somehow beneficial for the reasoning problem input at hand. Strategies and strategy scheduling will be addressed in further upcoming versions.

Implementation

Leo-III utilizes and instantiates the associated LeoPARD system platform [155] for higher-order (HO) deduction systems implemented in Scala (currently using Scala 2.12 and running on a JVM with Java 8). The prover makes use of LeoPARD's sophisticated data structures and implements its own reasoning logic on top. 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) as well as its polymorphic variants [18, 45].

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

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

Leo-III is available on GitHub:

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

Expected Competition Performance

Leo 1.4 was the CASC-27 LTB winner.

7.15 Leo-III 1.5

Alexander Steen
University of Luxembourg, Luxembourg

Architecture

Leo-III [92], the successor of LEO-II [11], is a higher-order ATP system based on extensional higher-order paramodulation with inference restrictions using a higher-order term ordering. The calculus contains dedicated extensionality rules and is augmented with equational simplification routines that have their intellectual roots in first-order superposition-based theorem proving. The saturation algorithm is a variant of the given clause loop procedure inspired by the first-order ATP system E. Leo-III cooperates with external first-order ATPs that are called asynchronously during proof search; a focus is on cooperation with systems that support typed first-order (TFF) input. For this year's CASC, CVC4 [7] and E [84, 88] are used as external systems. However, cooperation is in general not limited to first-order systems. Further TPTP/TSTP-compliant external systems (such as higher-order ATPs or counter model generators) may be included using simple command-line arguments. If the saturation procedure loop (or one of the external provers) finds a proof, the system stops, generates the proof certificate and returns the result.

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

Strategies

Leo-III comes with several configuration parameters that influence its proof search by applying different heuristics and/or restricting inferences. These parameters can be chosen manually by the user on start-up. Leo-III implements a naive time slicing approach of some of these strategies for this CASC.

Implementation

¡style type="text/css"¿ .leopard font-variant: small-caps; ¡/style¿ Leo-III utilizes and instantiates the associated ¡SPAN CLASS="leopard"¿LeoPARD¡/SPAN¿ system platform [155] for higher-order (HO) deduction systems implemented in Scala (currently using Scala 2.13 and running on a JVM with Java 8). The prover makes use of ¡span class="leopard"¿LeoPARD¡/span¿'s sophisticated data structures and implements its own reasoning logic on top. 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) as well as its polymorphic variants [18, 45].

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

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

Leo-III is available on GitHub:

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

Expected Competition Performance

Version 1.5 only marginally improves the previous release by fixing some bugs. As CASC is using wall clock (WC) time instead of CPU time usage in all divisions, the Java VM version of Leo-III is used in the competition (as opposed to a native build used last year). We hope that the JRE performs, after a slow start-up, quite well on longer (wrt. WC time) runs. We expect a similar performance as in last year's CASC.

Also in the LTB mode, there are no major novelties: only some timing parameters have been changed compared to last year. Stemming from Leo-III's support for polymorphic HOL reasoning, we expect a reasonable performance compared to the other systems. On the other hand, Leo-III's LTB mode does not do any learning and/or analysis of the learning samples.

7.16 MaLARea 0.9

Josef Urban
Czech Technical University in Prague, Czech Republic

Architecture

MaLARea 0.8 [148, 150, 47] 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 based on symbol-based similarity, model-based similarity, term-based similarity, and obviously previous successful proofs. The version for CASC-27 will use the E prover with the BliStr(Tune) [149, 41]

large-theory strategies, possibly also Prover9, Mace and Paradox. The premise selection methods will likely also use the distance-weighted k-nearest neighbor [46], ATPBoost [69], 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 time limits 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.

7.17 nanoCoP—1.1

Jens Otten
University of Oslo, Norway

Architecture

nanoCoP [63, 64] is an automated theorem prover for classical first-order logic with equality. It is a very compact implementation of the non-clausal connection calculus [60].

Strategies

An additional decomposition rule is added to the clausal connection calculus and the extension rule is generalized to non-clausal formulae. 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, restricted backtracking, and a fixed strategy schedule that is controlled by a shell script [66].

Implementation

nanoCoP is implemented in Prolog. The source code of the core prover 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.

nanoCoP can read formulae in leanCoP/nanoCoP syntax and in TPTP first-order syntax. Equality axioms are automatically added if required. The nanoCoP core prover returns a compact non-clausal connection proof.

The source code of nanoCoP 1.1 is available under the GNU general public license. It can be downloaded from the nanoCoP website at:

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

The provers nanoCoP-i and nanoCoP-M are version of nanoCoP for first-order intuitionistic logic and first-order modal logic, respectively. They are based on an adapted non-clausal connection calculus for non-classical logics [65].

Expected Competition Performance

nanoCoP is expected to have a better performance than leanCoP on formulae that have a nested (non-clausal) structure.

7.18 Prover9 1109a

Bob Veroff on behalf of William McCune
University of New Mexico, USA

Architecture

Prover9, Version 2009-11A, is a resolution/paramodulation prover for first-order logic with equality. Its overall architecture is very similar to that of Otter-3.3 [56]. 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 [55]. (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 [85] 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.

7.19 PyRes 1.3

Stephan Schulz
DHBW Stuttgart, Germany

Architecture

PyRes [90] is a simple resolution-style theorem prover for first-order logic, implemented in very clear and well-commented Python. It has been written as a pedagogical tool to illustrate the architecture and basic algorithms of a saturation-style theorem prover. The prover consists of a parser for (most of) TPTP-3 format, a simple clausifier to convert full first-order format into clause normal form, and a saturation core trying to derive the empty clause from the resulting clause set.

The saturation core is based on the DISCOUNT-loop variant of the *given-clause* algorithm, i.e., a strict separation of active and passive facts. It implements simple binary resolution and factoring [82], optionally with selection of negative literals [3]. Redundancy elimination is restricted to forward and backward subsumption and tautology deletion. There are no inference rules for equality - if equality is detected, the necessary axioms are added.

Strategies

The prover supports several negative literal selection strategies, as well as selection of the given clause from a set of differently weighted priority queues in the style of E [89]. In the competition, it will always select the syntactically largest literal, and will use weight-age interleaved clause selection with a pick-given ratio of 5 to 1.

Implementation

The prover is implemented in Python 3, with maximal emphasis on clear and well-documented code. Terms are represented as nested lists (equivalent to LISP style s-expressions). Literals, clauses, and formulas are implemented as classes using an object-oriented style.

One of the changes compared to last years version is the introduction of simple indices for unification (which now uses top symbol hashing) and subsumption. Subsumption now uses a new, simple technique we call *predicate abstraction indexing*, which represents a clause as an ordered sequence of the predicate symbols of its literals. PyRes builds a proof object on the fly, and can print a TPTP-3 style proof or saturation derivation.

The system source is available at:

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

Expected Competition Performance

Performance is expected to be better than last year, but still mediocre for non-equational problems, and abysmal for problems with equality. However, per CASC rules, PyRes will still be assumed superior to any non-participating prover.

7.20 Satallax 3.4

Michael Färber
ENS Paris-Saclay, France

Architecture

Satallax 3.4 [19] 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 [21] with a choice operator [4]. 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 [20]. These formulae and propositional clauses correspond to instances of the tableau rules. Satallax uses the SAT solver MiniSat to test the current set of propositional clauses for unsatisfiability. If the clauses are unsatisfiable, then the original branch is unsatisfiable. Optionally, Satallax generates lambda-free higher-order logic (lfHOL) formulae in addition to the propositional clauses [153]. If this option is used, then Satallax periodically calls the theorem prover E [87] to test for lfHOL unsatisfiability. If the set of lfHOL formulae is unsatisfiable, then the original branch is unsatisfiable. Upon request, Satallax attempts to reconstruct a proof which can be output in the TSTP format.

Strategies

There are about 150 flags that control the order in which formulae and instantiation terms are considered and propositional clauses are generated. Other flags activate some optional extensions to the basic proof procedure (such as whether or not to call the theorem prover E). A collection of flag settings is called a mode. Approximately 500 modes have been defined and tested so far. A strategy schedule is an ordered collection of modes with information about how much time the mode should be allotted. Satallax tries each of the modes for a certain amount of time sequentially. Before deciding on the schedule to use, Satallax parses the problem and determines if it is big enough that a SInE-based premise selection algorithm [36] should be used. If SInE is not activated, then Satallax uses a strategy schedule consisting of 37 modes. If SInE is activated, than Satallax is run with a SInE-specific schedule consisting of 58 modes with different SInE parameter values selecting different premises. Each mode is tried for time limits ranging from less than a second to just over 1 minute.

Implementation

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

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

Expected Competition Performance

Satallax 3.4 was the CASC-27 THF winner.

7.21 Satallax 3.5

Michael Färber
ENS Paris-Saclay, France

Architecture

Satallax [19] 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 [21] with a choice operator [4]. Problems are given in the THF format. 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 [20]. These formulae and propositional clauses correspond to instances of the tableau rules. Satallax uses the SAT solver MiniSat to test the current set of propositional clauses for unsatisfiability. If the clauses are unsatisfiable, then the original branch is unsatisfiable. Optionally, Satallax generates lambda-free higher-order logic (lfHOL) formulae in addition to the propositional clauses [153]. If this option is used, then Satallax periodically calls the theorem prover E [87] to test for lfHOL unsatisfiability. If the set of lfHOL formulae is unsatisfiable, then the original branch is unsatisfiable. Upon request, Satallax attempts to reconstruct a proof which can be output in the TSTP format.

Strategies

There are about 150 flags that control the order in which formulae and instantiation terms are considered and propositional clauses are generated. Other flags activate some optional extensions to the basic proof procedure (such as whether or not to call the theorem prover E). A collection of flag settings is called a mode. Approximately 500 modes have been defined and tested so far. A strategy schedule is an ordered collection of modes with information about how much time the mode should be allotted. Satallax tries each of the modes for a certain amount of time sequentially. Before deciding on the schedule to use, Satallax parses the problem and determines if it is big enough that a SInE-based premise selection algorithm [36] should be used. If SInE is not activated, then Satallax uses a strategy schedule consisting of 37 modes. If SInE is activated, than Satallax is run with a SInE-specific schedule consisting of 58 modes with different SInE parameter values selecting different premises. Each mode is tried for time limits ranging from less than a second to just over 1 minute.

Implementation

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

Satallax is available at:

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

Expected Competition Performance

The main change from Satallax 3.4 is the switch to a branch of E that fixes a higher-order unsoundness. The performance of Satallax 3.5 should be comparable to Satallax 3.4.

7.22 Twee 2.2.1

Nick Smallbone
Chalmers University of Technology, Sweden

Architecture

Twee 2.2.1 is an equational prover based on unfailing completion [2]. It features ground joinability testing [54] and a connectedness test [1], which together eliminate many redundant inferences in the presence of unorientable equations.

Twee's implementation of ground joinability testing performs case splits on the order of variables, in the style of [54], and discharges individual cases by rewriting modulo a variable ordering. The case splitting strategy chooses only useful case splits, which prevents the number of cases from blowing up.

Horn clauses are encoded as equations as described in [23]. The CASC version of Twee "handles" non-Horn clauses by discarding them.

Strategies

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

The main loop is a DISCOUNT loop. The active set contains rewrite rules and unorientable equations, which are used for rewriting, and the passive set contains unprocessed critical pairs. Twee often interreduces the active set, and occasionally simplifies the passive set with respect to the active set. Each critical pair is scored using a weighted sum of the weight of both of its terms. Terms are treated as DAGs when computing weights, i.e., duplicate subterms are only counted once per term. The weights of critical pairs that correspond to Horn clauses are adjusted by the heuristic described in [23], section 5.

For CASC, to take advantage of multiple cores, several versions of Twee run in parallel using different parameters.

Implementation

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

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

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

Twee can be downloaded from:

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

Expected Competition Performance

Twee is quite strong at UEQ, but will be at a disadvantage compared to provers with smart timeslicing (such as E and Vampire) which can make better use of the multiple cores. Prediction: Twee will solve more problems than last year but finish below E and Vampire. It could still solve some problems that no-one else does, though. Twee will do badly in FOF since it throws away all non-Horn clauses. It may get lucky and solve a few hard problems, especially if some mostly-equational problems show up.

7.23 Vampire 4.4

Giles Reger
University of Manchester, United Kingdom

There are no major changes to the main part of Vampire since 4.4, beyond some new proof search heuristics and new default values for some options. The biggest addition is support for higher-order reasoning via translation to applicative form and combinators, addition of axioms and extra inference rules, and a new form of combinatory unification.

Architecture

Vampire [50] 4.4 is an automatic theorem prover for first-order logic with extensions to theory-reasoning and higher-order logic. Vampire implements the calculi of ordered binary resolution and superposition for handling equality. It also implements the Inst-gen calculus and a MACE-style finite model builder [71]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [151, 70]. 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.

Vampire 4.4 provides a very large number of options for strategy selection. The most important ones are:

- Choices of saturation algorithm:
  - Limited Resource Strategy [81]
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [151]
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals [35].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection.
- Set-of-support strategy.
- For theory-reasoning:
  - Ground equational reasoning via congruence closure.
  - Addition of theory axioms and evaluation of interpreted functions.
  - Use of Z3 with AVATAR to restrict search to ground-theory-consistent splitting branches [70].
  - Specialised theory instantiation and unification [73].

  - Extensionality resolution with detection of extensionality axioms
- For higher-order problems:
  - Translation to applicative and combinator form.
  - Addition of combinator axioms.
  - Addition of shortcut inference rules that encode axioms.
  - Proof search heuristics targetting the growth of combinator axioms.
  - Restricted combinatory unification [73].

Implementation

Vampire 4.4 is implemented in C++. It makes use of Minisat and Z3.

Expected Competition Performance

Vampire 4.4 was the CASC-27 TFA, FOF, and FNT winner.

7.24 Vampire 4.5

Giles Reger
University of Manchester, United Kingdom

Architecture

Vampire [50] 4.5 is an automatic theorem prover for first-order logic with extensions to theory-reasoning and higher-order logic. Vampire implements the calculi of ordered binary resolution and superposition for handling equality. It also implements the Inst-gen calculus and a MACE-style finite model builder [71]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [151, 70]. 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 [72]. Vampire implements many useful preprocessing transformations including the SinE axiom selection algorithm. When a theorem is proved, the system produces a verifiable proof, which validates both the clausification phase and the refutation of the CNF.

Strategies

Vampire 4.5 provides a very large number of options for strategy selection. The most important ones are:

- Choices of saturation algorithm:
  - Limited Resource Strategy [81]

  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [151]
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals [35].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection. This has been extended with a layered clause selection approach [28].
- Set-of-support strategy with extensions for theory reasoning.
- For theory-reasoning:
  - Ground equational reasoning via congruence closure.
  - Addition of theory axioms and evaluation of interpreted functions.
  - Use of Z3 with AVATAR to restrict search to ground-theory-consistent splitting branches [70].
  - Specialised theory instantiation and unification [73].
  - Extensionality resolution with detection of extensionality axioms
- For higher-order problems:
  - Translation to polymorphic first-order logic using applicative form and combinators.
  - A new superposition calculus [14] utilising a KBO-like ordering [15] for orienting combinator equations. The calculus introduces an inference, narrow, for rewriting with combinator equations.
  - Proof search heuristics targeting the growth of clauses resulting from narrowing.
  - An extension of unification with abstraction to deal with functional and boolean extensionality.
- Various inferences to deal with booleans

Implementation

Vampire 4.5 is implemented in C++. It makes use of minisat and Z3. See the website

`https://vprover.github.iohttps://vprover.github.io/}</A>`

for more information and access to the GitHub repository.

Expected Competition Performance

There are four areas of improvement in Vampire 4.5. Firstly, a new layered clause selection approach [28] gives Vampire more fine-grained control over clause selection, in particular the way in which clauses involving theory axioms are selected. Secondly, theory evaluation and instantiation methods have been overhauled. Thirdly, a new subsumption demodulation rule [29] improves support for reasoning with conditional equalities. Finally, higher-order reasoning (introduced in Vampire 4.4) has been rewritten based on a new superposition calculus [14] utilising a KBO-like ordering [15] for orienting combinator equations. Vampire 4.5 should be an improvement on Vampire 4.4.

7.25 Zipperposition 2.0

Petar Vukmirović
Vrije Universiteit Amsterdam, The Netherlands

Architecture

Zipperposition is a superposition-based theorem prover for typed first-order logic with equality and higher-order logic. It is a pragmatic implementation of a complete calculus for Boolean-free higher-order logic [8]. It features a number of extensions that include polymorphic types; user-defined rewriting on terms and formulas ("deduction modulo theories"); a lightweight variant of AVATAR for case splitting; boolean reasoning [154]. The core architecture of the prover is based on saturation with an extensible set of rules for inferences and simplifications. Zipperposition uses a recently developed full higher-order unification algorithm that enables efficient integration of procedures for decidable fragments of higher-order unification [152]. The initial calculus and main loop were imitations of an old version of E [84], but there are many more rules nowadays. A summary of the calculus for integer arithmetic and induction can be found in [24].

Strategies

The system uses various strategies in a portfolio. The strategies are run in parallel, making use of all CPU cores available. We designed the portfolio of strategies by manual inspection of different TPTP problems. Heuristics used in Zipperposition are inspired by efficient heuristics used in E. Portfolio mode differentiates higher-order problems from the first-order ones. If the problem is first-order all higher-order prover features are turned off. Other than that, the portfolio is static and does not depend on the syntactic properties of the problem.

Implementation

The prover is implemented in OCaml, and has been around for eight years. Term indexing is done using fingerprints for unification, perfect discrimination trees for rewriting, and feature vectors for subsumption. Some inference rules such as contextual literal cutting make heavy use of subsumption. For higher-order problems some strategies use E prover, running in lambda-free higher-order mode, as an end-game backend prover. The code can be found at

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

and is entirely free software (BSD-licensed).

Zipperposition can also output graphic proofs using graphviz. Some tools to perform type inference and clausification for typed formulas are also provided, as well as a separate library for dealing with terms and formulas [24].

Expected Competition Performance

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

8 Conclusion

The 10th IJCAR ATP System Competition was the twenty-fifth 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] L. Bachmair and N. Dershowitz. Critical Pair Criteria for Completion. *Journal of Symbolic Computation*, 6(1):1–18, 1988.

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

[3] L. Bachmair, H. Ganzinger, D. McAllester, and C. Lynch. Resolution Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 19–99. Elsevier Science, 2001.

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

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

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

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

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

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

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

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

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

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

[14] A. Bhayat and G. Reger. A Combinator-Based Superposition Calculus for Higher-Order Logic. In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, page To appear, 2020.

[15] A. Bhayat and G. Reger. A Knuth-Bendix-Like Ordering for Orienting Combinator Equations. In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, page To appear, 2020.

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

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

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

[19] 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.

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

[21] 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.

[22] K. Chvalovsky, J. Jakubuv, M. Suda, and J. Urban. ENIGMA-NG: Efficient Neural and Gradient-Boosted Inference Guidance for E. In P. Fontaine, editor, *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 197–215. Springer-Verlag, 2019.

[23] K. Claessen and N. Smallbone. Efficient Encodings of First-Order Horn Formulas in Equational Logic. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Proceedings of the 9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 388–404, 2018.

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

[25] A. Duarte and K. Korovin. Implementing Superposition in iProver. In N. Peltier and V. Sofronie-

Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, page To appear, 2020.

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

[29] L. Gleiss, B. Kovacs and J. Rath. Subsumption Demodulation in First-Order Theorem Proving. In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science, page To appear, 2020.

[30] Z. Goertzel, J. Jakubuv, and J. Urban. ENIGMAWatch: ProofWatch Meets ENIGMA. In S. Cerrito and A. Popescu, editors, *Proceedings of the 28th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, number 11714 in Lecture Notes in Artificial Intelligence, page To appear. Springer-Verlag, 2019.

[31] 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.

[32] J. Hernandez and K. Korovin. An Abstraction-Refinement Framework for Reasoning with Large Theories. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Proceedings of the 9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 663–679, 2018.

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

[34] 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.

[35] 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.

[36] 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.

[37] E. Holden and K. Korovin. Experiments with Selection of Theorem Proving Heuristics. In A. Bolotov and F. Kammueller, editors, *Proceedings of the 26th Automated Reasoning Workshop*, page To appear, 2019.

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

[39] F. Hutter, H. Hoos, and K. Leyton-Brown. Parallel Algorithm Configuration. In Y. Hamadi and M. Schoenauer, editors, *Proceedings of the 6th International Conference on Learning and Intelligent Optimization*, number 7219 in Lecture Notes in Computer Science, pages 55–70. Springer-Verlag, 2012.

[40] J. Jakubuv, M. Chvalovský, K. Olsák, B. Piotrowski, M. Suda, and J. Urban. ENIGMA Anony-

mous: Symbol-Independent Inference Guiding Machine. arXiv:2002.05406, 2020.

[41] 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.

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

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

[44] J. Jakubuv and J. Urban. Hammering Mizar by Learning Clause Guidance. In *Proceedings of the 10th International Conference on Interactive Theorem Proving*, Leibniz International Proceedings in Informatics, page To appear. Dagstuhl Publishing, 2019.

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

[46] 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.

[47] 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. EasyChair Publications, 2015.

[48] 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.

[49] 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.

[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] R. Letz and G. Stenz. System Description: DCTP - A Disconnection Calculus Theorem Prover. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artificial Intelligence, pages 381–385. Springer-Verlag, 2001.

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

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

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

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

276, 1997.

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

[57] 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.

[58] 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.

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

[60] 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.

[61] J. Otten. Implementing Connection Calculi for First-order Modal Logics. In K. Korovin and S. Schulz, editors, *Proceedings of the 9th International Workshop on the Implementation of Logics*, number 22 in EPiC Series in Computing, pages 18–32. EasyChair Publications, 2012.

[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. nanoCoP: Natural Non-clausal Theorem Proving. In C. Sierra, editor, *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 4924–4928. Morgan-Kaufmann, 2017.

[65] 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*, number 10501 in Lecture Notes in Artificial Intelligence, pages 209–227. Springer-Verlag, 2017.

[66] J. Otten. Proof Search Optimizations for Non-clausal Connection Calculi. In B. Konev, P. Rümmer, and J. Urban, editors, *Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning*, number 2162 in CEUR Workshop Proceedings, pages 49–57, 2018.

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

[68] A. Paskevich. Connection Tableaux with Lazy Paramodulation. *Journal of Automated Reasoning*, 40(2-3):179–194, 2008.

[69] B. Piotrowski and J. Urban. ATPboost: Learning Premise Selection in Binary Setting with ATP Feedback. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Proceedings of the 9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 566–574, 2018.

[70] 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.

[71] 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.

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

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

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

[75] 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.

[76] A. Reynolds, T. King, and V. Kuncak. Solving Quantified Linear Arithmetic by Counterexample-guided Instantiation. *Formal Methods in System Design*, 51(3):500–532, 2017.

[77] 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.

[78] 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.

[79] 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.

[80] 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*, number 10483 in Lecture Notes in Computer Science, pages 22–40. Springer-Verlag, 2017.

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

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

[83] 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.

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

[85] 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.

[86] 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.

[87] 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.

[88] 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.

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

[90] S. Schulz and A. Pease. Teaching Automated Theorem Proving by Example: PyRes 1.2 (system description). In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science, page To appear, 2020.

[91] K. Slind and M. Norrish. A Brief Overview of HOL4. In O. Mohamed, C. Munoz, and S. Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, number 5170 in Lecture Notes in Computer Science, pages 28–32. Springer-Verlag, 2008.

[92] A. Steen and C. Benzmüller. The Higher-Order Prover Leo-III. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Proceedings of the 8th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Artificial Intelligence, pages 108–116, 2018.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[109] 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.

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

[111] G. Sutcliffe. The 4th IJCAR Automated Theorem Proving System Competition - CASC-J4. *AI*

Communications, 22(1):59–72, 2009.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[133] 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.

[134] 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.

[135] 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.

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

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

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

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

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

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

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

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

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

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

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

[147] T. Tammet. Extending Automated Deduction for Commonsense Reasoning. arXiv:2003.13159, 2020.

[148] 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.

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

[150] 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.

[151] 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.

[152] P. Vukmirovic, A. Bentkamp, and V. Nummelin. Efficient Full Higher-order Unification. In Z.M. Ariola, editor, *Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics, page To appear. Dagstuhl Publishing, 2020.

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

[154] P. Vukmirovic and V. Nummelin. Boolean Reasoning in a Higher-Order Superposition Prover. In P. Fontaine, P. Rümmer, and S. Tourret, editors, *Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning*, page To appear, 2020.

[155] M. Wisniewski, A. Steen, and C. Benzmüller. LeoPARD - A Generic Platform for the Imple-

mentation 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.

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