CASC-J11

CASC-J11
CASC-J11
CASC-J11

Proceedings of the 11th IJCAR ATP System Competition (CASC-J11)

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 time taken 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 11th IJCAR ATP System Competition (CASC-J11) was held on 9th August 2022. The design of the competition, its rules, and information regarding the competing systems, are provided.

1 Introduction

The CADE and IJCAR conferences are the major forums for the presentation of new research in all aspects of automated deduction. In order to stimulate ATP research and system development, and to expose ATP systems within and beyond the ATP community, the CADE ATP System Competition (CASC) is held at each CADE and IJCAR conference. CASC-J11 was held on 9th August 2022, as part of the 11th International Joint Conference on Automated Reasoning (IJCAR 2022)[1], which in turn was part of the Federated Logic Conference 2022, in Haifa, Israel. It was the twenty-seventh competition in the CASC series [132, 138, 135, 84, 86, 131, 129, 130, 91, 93, 95, 97, 100, 102, 104, 106, 108, 110, 112, 137, 114, 116, 119, 122, 124, 126].

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 time taken for problems solved;

in the context of:

- a bounded number of eligible problems, chosen from the TPTP problem library [117] and other useful sources of test problems, and
- specified time limits on solution attempts.

Twenty-four ATP systems and system variants competed in the various divisions. The winners of the CASC-28 (the previous CASC) divisions, and the Prover9 1109a system, were automatically entered into the corresponding demonstration divisions to provide benchmarks against which progress can be judged (the competition archive provides access to the systems' executables and source code). The systems, the divisions in which they were entered, and their entrants, are listed in Tables 1 and 2.

The design and procedures of CASC-J11 evolved from those of previous CASCs [132, 133, 128, 134, 82, 83, 85, 87, 88, 89, 90, 92, 94, 96, 99, 101, 103, 105, 107, 109, 111, 113, 115, 118, 120, 121, 123].

[1] CADE was a constituent conference of the 11th IJCAR, hence CASC-"J11".

ATP System	Divisions	Entrant (Associates)	Entrant's Affiliation
CSE 1.5	FOF	Feng Cao (Yang Xu, Peiyao Liu, Jun Liu, Shuwei Chen, Guoyan Zeng, Jian Zhong, Guanfeng Wu, Xingxing He, Peng Xu)	JiangXi University of Science and Technology
CSE_E 1.4	FOF	Peiyao Liu (Yang Xu, Feng Cao, Stephan Schulz, Jun Liu, Shuwei Chen, Guoyan Zeng, Jian Zhong, Guanfeng Wu, Xingxing He, Peng Xu)	Southwest Jiaotong University
cvc5 1.0	THF TFA FOF FNT SLH	Andrew Reynolds (Haniel Barbosa, Cesare Tinelli, Clark Barrett)	University of Iowa
Drodi 3.3.3	FOF UEQ	Oscar Contreras	Amateur Programmer
E 3.0	THF FOF FNT UEQ SLH LTB	Stephan Schulz (Petar Vukmirović, Jasmin Blanchette)	DHBW Stuttgart
Ehoh 2.7	SLH (demo)	CASC	CASC-28 winner
Etableau 0.67	FOF	John Hester	Imandra
GKC 0.7	FOF UEQ	Tanel Tammet	Tallinn University of Technology
Goéland 1.0.0	FOF	Julie Cailler (Johann Rosain, David Delahaye, Simon Robillard, Hinde Lilia Bouziane)	Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier
iProver 3.6	TFA FOF FNT UEQ LTB	Konstantin Korovin (André Duarte, Edvard Holden)	University of Manchester
Lash 1.12	THF	Cezary Kaliszyk (Chad Brown, Jan Jakubuv)	University of Innsbruck
LEO-II 1.7.0	THF	Alexander Steen (Christoph Benzmüller)	University of Greifswald
Leo-III 1.7.0	THF SLH LTB	Alexander Steen (Christoph Benzmüller)	University of Greifswald

Table 1: The ATP systems and entrants

ATP System	Divisions	Entrants (Associates)	Entrant's Affiliation
Prover9 1109a	FOF (demo)	CASC (William McCune, Bob Veroff)	CASC fixed point
Statallax 3.4	THF	Cezary Kaliszyk (Chad Brown, Michael Färber)	University of Innsbruck
SnakeForV4.7 1.0	TFA FOF FNT UEA (demo)	Martin Suda (Filip Bartek)	Czech Technical University
Toma 0.2	UEQ	Teppei Saito (Nao Hirokawa)	Japan Advanced Institute of Science and Technology
Twee 2.4	UEQ	CASC	CASC-28 winner
Twee 2.4.1	UEQ	Nick Smallbone	Chalmers University of Technology
Vampire 4.5	TFA (demo)	CASC	CASC-28 winner
Vampire 4.6	FOF FNT LTB (demo)	CASC	CASC-28 winner
Vampire 4.7	THF TFA FOF FNT SLH LTB	Giles Reger (Martin Suda, Andrei Voronkov, Laura Kovacs, Ahmed Bhayat, Bernhard Gleiss, Marton Hajdu, Petra Hozzova, Evgeny Kotelnikov, Jakob Rath, Michael Rawson, Martin Riener, Johannes Schoisswohl)	University of Manchester
Zipperposition 2.1	THF (demo)	CASC	CASC-28 winner
Zipperposition 2.1.999	THF FOF SLH (demo)	Jasmin Blanchette (Petar Vukmirović Alexander Bentkamp, Simon Cruanes, Visa Nummelin, Sophie Tourret)	Vrije Universiteit Amsterdam

Table 2: The ATP systems and entrants, continued

Important changes for CASC-J11 were:

- The TFA division returned from hiatus.
- The LTB division did not have FOF versions of the problems, but had TXF versions.

The competition organizer was Geoff Sutcliffe, assisted by Martin Desharnais for the SLH and LTB divisions. CASC is overseen by a panel of knowledgeable researchers who are not participating in the event. The CASC-J11 panel members were Pascal Fontaine, Cláudia Nalon, and Christoph Weidenbach. The competition was run on computers provided by StarExec at the University of Miami.[2] The CASC-J11 web site provides access to resources used before, during, and after the event: `http://www.tptp.org/CASC/J11`

The CASC rules, specifications, and deadlines are absolute. Only the panel has the right to make exceptions. It is assumed that all entrants have read the web pages related to the competition, and have complied with the competition rules. Non-compliance with the rules can lead to disqualification. A "catch-all" rule is used to deal with any unforeseen circumstances: *No cheating is allowed*. The panel is allowed to disqualify entrants due to unfairness, and to adjust the competition rules in case of misuse.

> *A Tense Note:* Attentive readers will notice changes between use of the past to present tenses in this document. Many parts of CASC are established and stable - these are described in the present tense (the rules are the rules). Aspects that are particular to CASC-J11 are described in the past tense, so they make sense when reading this after the event.

2 Divisions

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

2.1 The Competition Divisions

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

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

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

The **TFA** division: Typed (monomorphic) 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

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

- 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 **SLH** division: Typed (monomorphic) higher-order theorems without arithmetic (axioms with a provable conjecture), generated by Isabelle's Sledgehammer system [61].

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 a 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-J11 the LTB division had one problem category:

- The **VBT** category: Problems generated by Isabelle's Sledgehammer system, from the "van Emde Boas Trees" entry in Isabelle's Archive of Formal Proofs (AFP) [1]. The VBT category was accompanied by training data.

Six versions of each VBT problem were provided: a monomorphic typed first-order form (TF0) version, a polymorphic typed first-order form (TF1) version, a monomorphic typed extended first-order form (TX0) version, a polymorphic typed extended first-order form (TX1) version, a monomorphic typed higher-order form (TH0) version, and a polymorphic typed higher-order form (TH1) version.

- The THF versions did not use `$ite`, `$let`, or tuples.
- There were no common include files.
- The TF0 problems had upto 12792 axioms, the TF1 problems had upto 12528 axioms, the TX0 problems had upto 12579 axioms, the TX1 problems had upto 12488 axioms, the TH0 problems had upto 10296 axioms, and the TH1 problems had upto 10078 axioms.
- Not all of the problems were theorems.

Systems could attempt as many of the versions as they wanted, and could attempt the problems and versions in any order including in parallel. A solution to any version counted as a solution to the problem.

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 do not run in the competition divisions for any reason (e.g., the system requires special hardware, the system is a previous winner, or the entrant is an organizer) can be entered into the demonstration division. Demonstration division systems can run on the competition computers, or the computers can be supplied by the entrant. The demonstration division results are presented along with the competition divisions' results, but might not be comparable with those results.

3 Infrastructure

3.1 Computers

The competition computers had:

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

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

3.2 Problems

3.2.1 Problem Selection

The problems for the THF, FOF, FNT, and UEQ divisions were taken from the TPTP Problem Library [117], version v8.1.0. The TPTP version used for CASC is released only after the competition has started, so that new problems have not been seen by the entrants. The problems have to meet certain criteria to be eligible for use. The problems used are randomly selected from the eligible problems based on a seed supplied by the competition panel:

- The TPTP tags problems that designed specifically to be suited or ill-suited to some ATP system, calculus, or control strategy as *biased*. 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) [136]. 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 [84].
- The selection is biased to select problems that are new in the TPTP version used, until 50% of the problems in each problem category have been selected, after which random selection (from old and new problems) continues. The number of new problems used depends on how many new problems are eligible and the limitation on very similar problems.

The problems for the SLH division were were generated by Isabelle's Sledgehammer system. 100 problems were sliced out from the largest theory (measured by the number of proof goals) in each of 50 randomly selected sessions in Isabelle's Archive of Formal Proofs, providing 5000 problems that could be used. 720 appropriately difficult problems were chosen based on performance data similar to that in the TSTP.

The problems for the LTB division are taken from various sources, chosen for each CASC. Each problem category is based on one source. Entrants are expected to honestly not use publicly available problem sets for tuning before the competition. The process for selecting problems depends on the problem source. The VBT problems were generated by Isabelle's (version d76b150efdc2) Sledgehammer system from the AFP's (version 04d347968c92) "van Emde Boas Trees" entry [1]. For each problem, the generation used the deterministic MePo fact filter [58] to heuristically select up to 8192 relevant axioms from the surrounding context. For the monomorphic versions (i.e., TF0, TX0, TH0), the monomorphization pass [18] could introduce up to 2048 differently typed variants of the selected axioms. A total of 8989 problems were exported in each of the six versions, and 8000 appropriately difficult problems were chosen based on performance data similar to that in the TSTP.

3.2.2 Number of Problems

In the TPTP-based divisions, the minimal numbers of problems that must be used in each division and category to ensure sufficient confidence in the competition results are determined from the numbers of eligible problems in each division and category [37]. The competition organizers have to ensure that there are sufficient computers available to run the ATP systems on this minimal number of problems. The minimal numbers of problems are used in determining the time limits imposed on solution attempts - see Section 3.3. The numbers of problems to be used in each division of the competition are determined from the number of computers available, the time allocated to the competition, the number of ATP systems to be run on the computers over the divisions, and the time limits imposed on solution attempts, according to the following relationship:

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

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

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

In the LTB division the number of problems in each problem category is determined by the number of problems in the corresponding problem set. The VBT problem category had 8000 problems (with six 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 left before the symbols' uses)
- randomly swapping the arguments of associative connectives, and randomly reversing implications
- randomly reversing equalities

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

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

3.2.4 Batch Specification Files

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

- A problem category line of the form
  `division.category LTB.`*category_mnemonic*
  For CASC-J11 it was
  `division.category LTB.VBT`
- The name of a `.tgz` file (relative to the directory holding the batch specification file) that contains training data in the form of problems in TPTP format and one or more solutions to each problem in TSTP format, in a line of the form
  `division.category.training_data` *tgz_file_name*
  For CASC-J11 it was
  `division.category.training_data TrainingData/TrainingData.VBT.tgz`
  The `.tgz` file expands in place to three directories: `Axioms`, `Problems`, and `Solutions`. `Axioms` contains all the axiom files that are used in the training and competition problems. `Problems` contains the training problems. `Solutions` contains a subdirectory for each of the `Problems`, containing TPTP format solutions to the problem. The language of a solution might not be the same as the language of the problem, e.g., a proof to a THF problem might be written in FOF, or the proof of a TFF problem might be written in THF.

Each batch specification consists of:

- A header line `% SZS start BatchConfiguration`
- A specification of whether or not the problems in the batch must be attempted in order is given, in a line of the form
  `execution.order` *ordered/unordered*
  If the batch is ordered the ATP systems may not start any attempt on a problem, including reading the problem file, before ending the attempt on the preceding problem. For CASC-J11 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-J11 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-J11 it was
`limit.time.problem.wc 0`
- The overall wall clock time limit for the batch, in a line of the form
`limit.time.overall.wc` *limit_in_seconds*
- A terminator line `% SZS end BatchConfiguration`
- A header line `% SZS start BatchIncludes`
- `include` directives that are used in every problem. All the problems in the batch have these `include` directives, and can also have other `include` directives that are not listed here. In CASC-J11 there were no included axiom files.
- A terminator line `% SZS end BatchIncludes`
- A header line `% SZS start BatchProblems`
- Pairs of problem file names (relative to the directory holding the batch specification file), and output file names where the output for the problem must be written. The output files must be written in the directory specified as the second argument to the `starexec_run` script (the first argument is the name of the batch specification file). For CASC-J11, see the additional notes below.
- A terminator line `% SZS end BatchProblems`

Additional Notes for CASC-J11

- In the `BatchProblems` section, the multiple versions of each problem were specified using UNIX `*` globbing, e.g., `VBT00001*.p`. The versions of each problem had extensions as follows: the TF0 version used `_1`, the TF1 version used `_2`, the TX0 version used `_3`, the TX1 version used `_4`, the TH0 version used `^1`, and the TH1 version used `^2`.
- Proof output had to identify which version of the problem was solved - see Section 6.1.
- In the `BatchIncludes` section (not in problem files), multiple versions of included axiom files could be specified using UNIX `*` globbing. For a given problem, systems could use only the axiom files whose version matched that of the problem file (there might be none). Using any other versions could lead to weird results.

3.3 Resource Limits

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

In the SLH division a CPU time limit is imposed for each problem. The minimal time limit per problem was 15s and the maximal time limit per problem was 90s. The time limit was chosen as a reasonable value within the range allowed, and was announced at the competition. In CASC-J11 a wall clock time limit was imposed for each problem, and there were no CPU time limits (i.e., using all cores made sense).

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

4 System Evaluation

For each ATP system, for each problem, four items of data are recorded: whether or not the problem was solved, the CPU time taken, the wall clock time taken, and whether or not a solution (proof or model) was output. The systems are ranked in the competition divisions according to the number of problems solved with an acceptable solution output. Ties are broken according to the average time taken over problems solved. Trophies are awarded to the competition divisions' winners. In the demonstration division the systems are not ranked, and no trophies or prizes are awarded.

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

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

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

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

$$1 - FractionOfSystemsThatSolvedTheProblem$$

  and a system's overall SotAC is its average for the problems it solves but that are not solved by all the systems.
- The *core usage* measures the extent to which the systems take advantage of multiple cores. It is the average of the ratios of CPU time used to wall clock time used, over the problems solved.

- The *efficiency* measure combines the number of problems solved with the time taken. It is the average of the inverses of the times taken for problems solved, multiplied by the fraction of problems solved. This can be interpreted intuitively as the average of the solution rates for problems solved, multiplied by the fraction of problems solved. Efficiency is computed for both CPU time and wall clock time, to measure how efficiently the systems use one core and multiple cores respectively.

At some time after the competition all high ranking systems in each division are tested over the entire TPTP. This provides a final check for soundness (see Section 6.1 regarding soundness checking before the competition). If a system is found to be unsound during or after the competition, but before the competition report is published, and it cannot be shown that the unsoundness did not manifest itself in the competition, then the system is retrospectively disqualified. At some time after the competition, the solutions from the winners are checked by the panel. If any of the solutions are unacceptable, i.e., they are sufficiently worse than the samples provided, then that system is retrospectively disqualified. All disqualifications are explained in the competition report.

5 System Entry

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

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

The division winners from the previous CASC are automatically entered into their demonstration divisions, to provide benchmarks against which progress can be judged. Prover9 1109a is automatically entered into the FOF demonstration 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, using the HTML schema supplied on the CASC web site. The schema has the following sections:

- Architecture. This section introduces the ATP system, and describes the calculus and inference rules used.
- Strategies. This section describes the search strategies used, why they are effective, and how they are selected for given problems. Any strategy tuning that is based on specific problems' characteristics must be clearly described (and justified in light of the tuning restrictions described in Section 6.1).
- Implementation. This section describes the implementation of the ATP system, including the programming language used, important internal data structures, and any special code libraries used. The availability of the system is also given here.

- Expected competition performance. This section makes some predictions about the performance of the ATP system for each of the divisions and categories in which it is competing.
- References.

The system description has to be emailed to the competition organizers by the system description deadline. The system descriptions form part of the competition proceedings (Section 7).

5.2 Sample Solutions

For systems in the divisions that require solution output, representative sample solutions must be emailed to the competition organizers by the sample solutions deadline. Use of the TPTP format for proofs and finite interpretations is encouraged. The competition panel decides whether or not solutions are acceptable (see Section 4).

Proof/model samples are required as follows:

- THF: `SET014^4`
- TFA: `DAT013=1`
- FOF: `SEU140+2`
- FNT: `NLP042+1` and `SWV017+1`
- UEQ: `BOO001-1`

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

6 System Requirements

6.1 System Properties

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

Execution, Soundness, and Completeness

- Systems must be fully automatic, i.e., all command line switches have to be the same for all problems in each division.
- Systems' performances must be reproducible by running the system again.
- Systems must be sound. At some time before the competition all the systems in the competition divisions are tested for soundness. Non-theorems are submitted to the systems in the THF, TFA, FOF, UEQ, SLH, and LTB divisions, and theorems are submitted to the systems in the FNT division. Finding a proof of a non-theorem or a disproof of a theorem indicates unsoundness. If a system fails the soundness testing it must be repaired by the unsoundness repair deadline or be withdrawn.
- Systems do not have to be complete in any sense, including calculus, search control, implementation, or resource requirements.
- All techniques used must be general purpose, and expected to extend usefully to new unseen problems. The precomputation and storage of information about individual problems that might appear in the competition or their solutions is not allowed. (It's OK to

store information about LTB training problems.) Strategies and strategy selection based on individual problems or their solutions are not allowed. If machine learning procedures are used to tune a system, the learning must ensure that sufficient generalization is obtained so that no there is no specialization to individual problems. The system description must explain any such tuning or training that has been done. The competition panel may disqualify any system that is deemed to be problem specific rather than general purpose.

Output

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

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

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

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

- For each problem, the system must output a distinguished string indicating what solution has been found or that no conclusion has been reached. Systems must use the SZS ontology and standards [98] 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 `Ended` notification line. The line must also be output to the output file.
- When outputting a solution, the start and end of the solution must be delimited by distinguished strings. Systems must use the SZS ontology and standards for this. For example

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

  The string specifying the problem status must be output before the start of a solution. Use of the TPTP format for proofs and finite interpretations [127] is encouraged.
- Solutions may not have irrelevant output (e.g., from other threads running in parallel) interleaved in the solution.

Resource Usage

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

6.2 System Delivery

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

For systems running on entrant supplied computers in the demonstration division, entrants must email a `.tgz` file containing the source code and any files required for building the executable system to the competition organizers by the system delivery deadline.

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

6.3 System Execution

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

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

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

7 The ATP Systems

These system descriptions were written by the entrants.

7.1 CSE 1.5

Feng Cao
JiangXi University of Science and Technology, China

Architecture

CSE 1.5 is a developed prover based on the last version - CSE 1.4. It is an automated theorem prover for first-order logic without equality, based mainly on a novel inference mechanism called Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction (S-CS) [150]. S-CS is able to handle multiple (two or more) clauses dynamically in a synergized way in one deduction step, while binary resolution is a special case. CSE 1.5 also adopts conventional factoring, equality resolution (ER rule), and variable renaming. Some pre-processing techniques, including pure literal deletion and simplification based on the distance to the goal clause, and a number of standard redundancy criteria for pruning the search space: tautology deletion, subsumption (forward and backward), are applied as well.

CSE 1.5 has been improved compared with CSE 1.4, mainly from the following aspects:

- Optimization of the contradiction separation algorithm based on full usage of clauses, the number of repetitions of the input clause used in the deduction process is calculated by the evaluation function.
- In order to avoid the local optimum caused by multi-clause contradiction separation deduction path search according to the given rules, a restart mechanism is added.

Internally CSE 1.5 works only with clausal normal form. The E prover [77] is adopted with thanks for clausification of full first-order logic problems during preprocessing.

Strategies

CSE 1.5 inherited most of the strategies in CSE 1.4. The main new strategies are:

- Deduction control strategy. In the process of proof search, different contradiction separation algorithms can be used interchangeably.
- Clause selection strategy. Add the priority order of long clauses participating in deduction.

Implementation

CSE 1.5 is implemented mainly in C++, and Java is used for batch problem running implementation. A shared data structure is used for constants and shared variables storage. In addition, special data structure is designed for property description of clause, literal and term, so that it can support the multiple strategy mode. E prover is used for clausification of FOF problems, and then TPTP4X is applied to convert the CNF format into TPTP format.

Expected Competition Performance

CSE 1.5 has made some improvements compared to CSE 1.4, and so we expect a better performance in this year's competition.

Acknowledgement: Development of CSE 1.5 has been partially supported by the General Research Project of Jiangxi Education Department (Grant No. GJJ200818).

7.2 CSE_E 1.4

Peiyao Liu
Southwest Jiaotong University, China

Architecture

CSE_E 1.4 is an automated theorem prover for first-order logic by combining CSE 1.4 and E 2.6, where CSE 1.4 is based on the Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction (S-CS) [150] and E is mainly based on superposition. The combination mechanism is like this: E and CSE are applied to the given problem sequentially. If either prover solves the problem, then the proof process completes. If neither CSE nor E can solve the problem, some inferred clauses with no more than two literals, especially unit clauses, by CSE will be fed to E as lemmas, along with the original clauses, for further proof search.

This kind of combination is expected to take advantage of both CSE and E, and produce a better performance. Concretely, CSE is able to generate a good number of unit clauses, based on the fact that unit clauses are helpful for proof search and equality handling. On the other hand, E has a good ability on equality handling.

Strategies

The CSE part of CSE_E 1.4 takes almost the same strategies as in that in CSE 1.4 standalone, e.g., clause/literal selection, strategy selection, and CSC strategy. The only difference is that equality handling strategies of CSE part of the combined system are blocked. The main new strategies for the combined systems are:

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

Implementation

CSE_E 1.4 is implemented mainly in C++, and JAVA is used for batch problem running implementation. The job dispatch between CSE and E is implemented in C++.

Expected Competition Performance

We expect CSE_E 1.4 to solve some hard problems that E cannot solve and have a satisfying performance.

Acknowledgement: Development of CSE_E 1.4 has been partially supported by the National Natural Science Foundation of China (NSFC) (Grant No. 61976130). Stephan Schulz for his kind permission on using his E prover that makes CSE_E possible.

7.3 cvc5 1.0

Andrew Reynolds
University of Iowa, USA

Architecture

cvc5 is the successor of CVC4 [7]. It is an SMT solver based on the CDCL(T) architecture [59] that includes built-in support for many theories, including linear arithmetic, arrays, bit vectors, datatypes, finite sets and strings. It incorporates approaches for handling universally quantified formulas. For problems involving free function and predicate symbols, cvc5 primarily uses heuristic approaches based on conflict-based instantiation and E-matching for theorems, and finite model finding approaches for non-theorems. Like other SMT solvers, cvc5 treats quantified formulas using a two-tiered approach. First, quantified formulas are replaced by fresh Boolean predicates and the ground theory solver(s) are used in conjunction with the underlying SAT solver to determine satisfiability. If the problem is unsatisfiable at the ground level, then the solver answers "unsatisfiable". Otherwise, the quantifier instantiation module is invoked, and will either add instances of quantified formulas to the problem, answer "satisfiable", or return unknown. Finite model finding in cvc5 targets problems containing background theories whose quantification is limited to finite and uninterpreted sorts. In finite model finding mode, cvc5 uses a ground theory of finite cardinality constraints that minimizes the number of ground equivalence classes, as described in [70]. 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 [70]. If no instances are added, it reports "satisfiable".

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

Strategies

For handling theorems, cvc5 primarily uses conflict-based quantifier instantiation [69, 5], numerative instantiation [68] and E-matching. cvc5 uses a handful of orthogonal trigger selection strategies for E-matching, and several orthogonal ordering heuristics for enumerative instantiation. For handling non-theorems, cvc5 primarily uses finite model finding techniques. Since cvc5 with finite model finding is also capable of establishing unsatisfiability, it is used as a strategy for theorems as well.

Implementation

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

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

Expected Competition Performance

cvc5 has support for fine-grained proofs, which will be generated in solutions this year. The first-order theorem proving and finite model finding capabilities of cvc5 have undergone minor improvements in the past year. Hence, cvc5 will perform comparably to its performance the previous year.

7.4 Drodi 3.3.3

Oscar Contreras
Amateur Programmer, Spain

Architecture

Drodi is a very basic and lightweight automated theorem prover. It implements ordered resolution and equality paramodulation inferences as well as demodulation and some other standard simplifications. It also includes its own basic implementations of clausal normal form conversion [60], AVATAR architecture with a SAT solver [143], Limited Resource Strategy [71], discrimination trees as well as KBO, non-recursive and lexicographic reduction orderings. Drodi produces a verifiable proof in TPTP format.

Strategies

Drodi 3.3 has a fair number of selectable strategies including but not limited to the following:

- Otter, Discount and Limited Resource Strategy [71] saturation algorithms.
- A basic implementation of AVATAR architecture [143].
- Several literal and term reduction orderings.
- Several literal selection options.
- Several clause selection heuristics with adjustable selection ratios, including several types of clause weight queues and one age queue.
- Classical clause relevancy pruning.
- Drodi can generate a learning file from successful proofs and use the file to guide clause selection strategy.

Drodi integrated learning functions are a generalization of ENIGMA [46, 47]. It will use a general learning file applicable to any kind of problems during CASC competition. However, unlike ENIGMA, the learning file is completely general and can be used with any kind of problems. This generality allows the use of the same learning file in both FOF and UEQ CASC competition divisions. Literals' polarity, equality, skolem and variable occurrences are stored in clause feature vectors. Unlike ENIGMA, instead of storing the specific functions and predicates themselves, only the general properties of functions and non-equality predicates are stored in clause feature vectors. Predicates are differentiated from functions. In addition the following properties are also stored:

- Predicate is in a selected or unselected literal.

- Predicate or function is in the conjecture.
- Predicate or function is only in input formulas that are not the conjecture nor axioms.
- Predicate or function is only in axiom input formulas.

Each problem is classified in one of eight possible types according to its general characteristics (horn, ground, with equalities, with unit clauses...). There are 120 different strategy settings that may affect the learning characteristics (clause selection, literal selection, ordering and saturation algorithm combinations). There is a learning vector for each problem type/strategy setting combination. Therefore a learning file may contain up to 960 different learning vectors. Each learning vector has 2471 components that are double precision float numbers (8 bytes). Therefore the learning file size may be up to 19 MB.

Implementation

Drodi 3.3 is implemented in C. It includes discrimination trees and hashing indexing. All the code is original, without special code libraries or code taken from other sources.

Expected Competition Performance

Drodi has been slightly improved during last year. Competition performance is expected to be similar to last CASC competition: second half of the score table in FOF division and last or next to last in UEQ division.

7.5 E 3.0

Stephan Schulz
DHBW Stuttgart, Germany

Architecture

E [73, 77, 78] is a purely equational theorem prover for many-sorted first-order logic with equality, and for monomorphic higher-order logic. It consists of an (optional) clausifier for preprocessing full first-order formulae into clausal form, and a saturation algorithm implementing an instance of the superposition calculus with negative literal selection and a number of redundancy elimination techniques, optionally with higher-order extensions [144]. 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-J11, E implements a multi-core strategy-scheduling automatic mode. The total CPU time available is broken into several (unequal) time slices. For each time slice, the problem is classified into one of several classes, based on a number of simple features (number of clauses, maximal symbol arity, presence of equality, presence of non-unit and non-Horn clauses, possibly presence of certain axiom patterns...). For each class, a schedule of strategies is greedily constructed from experimental data as follows: The first strategy assigned to a schedule is the the one that solves the most problems from this class in the first time slice. Each subsequent strategy is selected based on the number of solutions on problems not already solved by a preceding strategy.

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

Implementation

E is build around perfectly shared terms, i.e. each distinct term is only represented once in a term bank. The whole set of terms thus consists of a number of interconnected directed acyclic graphs. Term memory is managed by a simple mark-and-sweep garbage collector. Unconditional (forward) rewriting using unit clauses is implemented using perfect discrimination trees with size and age constraints. Whenever a possible simplification is detected, it is added as a rewrite link in the term bank. As a result, not only terms, but also rewrite steps are shared. Subsumption and contextual literal cutting (also known as subsumption resolution) is supported using feature vector indexing [76]. Superposition and backward rewriting use fingerprint indexing [75], 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 [53, 54]. The prover and additional information are available at

`https://www.eprover.org`

Expected Competition Performance

The inference core of E 3.0 has been extended to higher-order logic, and now supports multi-core scheduling. However, we have not yet been able to evaluate and integrate new search strategies making full use of these features. As a result, we expect performance to be only slightly better than in the last year. The system is expected to perform well in most proof classes, but will at best complement top systems in the disproof classes.

7.6 Ehoh 2.7

Petar Vukmirović
Vrije Universiteit Amsterdam, The Netherlands

Architecture

Ehoh is a higher-order superposition-based theorem prover implementing lambda-free higher-order superposition [8]. Recently, Ehoh has been extended to support not only lambda-free, but full higher-order syntax. Internally, Ehoh unfolds all definitions of predicate symbols, lifts lambdas and removes all Boolean subterms through a FOOL-like [51] preprocessing transformation. After these steps are performed, the problem lies in the lambda-free fragment and the standard lambda-free superposition applies. Ehoh also supports TFX `$ite` and `$let` syntax. On the reasoning side, modest additions to the calculus have been made: We implemented rules NegExt, PosExt and Ext-* family of rules described by Bentkamp et al. [9]. Full support for lambda-terms and calculus-level treatment of Boolean terms is expected in the next version of Ehoh.

Strategies

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

Implementation

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

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

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

Expected Competition Performance

Ehoh 2.7 is the CASC-28 SLH division winner.

7.7 Etableau 0.67

John Hester
University of Florida, USA

Architecture

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

Strategies

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

Implementation

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

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

Expected Competition Performance

Etableau will solve fewer problems than E, but may solve some that others cannot.

7.8 Goéland 1.0.0

Julie Cailler
Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, France

Architecture

Goéland [24] is a first-order concurrent automated theorem prover. It implements the free variable tableau method with equality (using rigid E-unification [28]) and Deduction Modulo Theory (DMT) [29], with some variations like preskolemization or polarized DMT [38]. It produces human readable proofs.

Strategies

Goéland is based on concurrency and message exchanges. Branches are explored by different processes, each returning a local solution, i.e., a substitution that closes that branch. The parent process then has to select a solution on which all the branches can agree, or perform backtracking if no such substitution can be found. This concurrent approach helps limit backtracking and avoids some forms of unfairness in proof search related to the order in which branches are explored. Iterative deepening is used to guard against unfairness caused by unbounded free variable introductions.

Implementation

Goéland is implemented in Go. This language fits well with the core idea of concurrency behind the prover, which is provided by means of the notion of goroutine. Goroutines are actually light threads and a large number of them can be launched simultaneously at runtime.

In Goéland, each goroutine contains a non-branching step of the proof, which interacts with other goroutines. The implementation of Goéland can be found here:

`https://github.com/GoelandProver/Goeland`

Expected Competition Performance

Development of Goéland started one year ago, and functionalities such as DMT or equality reasoning are even more recent. For this first participation at CASC, we do not have any particular expectations. Our goal is mainly to present our approach to the community (feel free to read our article accepted at IJCAR 2022 [24]). Good luck, Goéland!

7.9 GKC 0.7

Tanel Tammet
Tallinn University of Technology, Estonia

Architecture

GKC [139] is a resolution prover optimized for search in large knowledge bases. The GKC version running at CASC-J11 is exactly the same as was run at the previous CASC-28. Almost all of the GKC development effort this year has gone to the commonsense superstructure GK (`https://logictools.org/gk/`) and several associated systems.

GKC is used as a foundation (GK Core) for building a common-sense reasoner GK. In particular, GK can handle inconsistencies and perform probabilistic and nonmonotonic reasoning, see [140, 141]. We are working on a natural language question answering system and envision NLP question answering as the main potential application for these specialized methods. The WASM version of the previous GKC 0.6 is used as the prover engine in the educational `http://logictools.org` system. It can read and output proofs in the TPTP, simplified TPTP and JSON format, the latter compatible with JSON-LD, see [142].

These standard inference rules have been implemented in GKC:

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

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

Strategies

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

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

Implementation

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

GKC can be obtained from

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

Expected Competition Performance

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

7.10 iProver 3.6

Konstantin Korovin
University of Manchester, United Kingdom

Architecture

iProver interleaves instantiation calculus Inst-Gen [50, 49, 35] with ordered resolution and superposition calculi [30]. iProver approximates first-order clauses using propositional abstractions that are solved using MiniSAT [34] or Z3 [27] and refined using model-guided instantiations. iProver also implements a general abstraction-refinement framework for under-and over-approximations of first-order clauses [39, 40]. First-order clauses are exchanged between calculi during the proof search.

Recent features in iProver include:

- Ground joinability and connectedness in the superposition calculus [32].
- Support for quantified linear arithmetic.
- AC joinability and AC normalisation [31].
- Superposition calculus with simplifications including: demodulation, light normalisation, subsumption, subsumption resolution and global subsumption. iProver's simplification set up [30] is tunable via command line options and generalises common architectures such as Discount or Otter.

- HOS-ML framework for learning heuristics using combination of hyper-parameter optimisation and dynamic clustering together with schedule optimisation using constraint solving [45, 44]

Strategies

iProver has around 100 options to control the proof search including options for literal selection, passive clause selection, frequency of calling the SAT/SMT solvers, simplifications, and options for combination of instantiation with resolution and superposition. For the competition HOS-ML [45] was used to build a multi-core schedule from heuristics learnt over a sample of FOF problems. Some theories and fragments are recognised such as EPR, UEQ, Horn, groups, rings and lattices for which options are adapted accordingly.

Implementation

iProver is implemented in OCaml and for the ground reasoning uses MiniSat [34] and Z3 [27]. iProver accepts FOF, TFF and CNF formats. Vampire [52, 64] and E prover [77] are used for proof-producing clausification of FOF/TFF problems. Vampire is also used for SInE axiom selection [43] in the LTB division and for theory axioms in the TFA division. iProver is available at:

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

Expected Competition Performance

We expect better performance compared to the previous year due to new methods: joinability, connectedness, AC reasoning, simplifications and heuristic selection. Heuristic tuning is still work in progress and heuristics are not fully tuned for the new options.

7.11 Lash 1.12

Cezary Kaliszyk
University of Innsbruck, Austria

Architecture

Lash [20] is a higher-order automated theorem prover created as a fork of the theorem prover Satallax. The basic underlying calculus of Satallax is a ground tableau calculus whose rules only use shallow information about the terms and formulas taking part in the rule.

Strategies

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

Implementation

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

Expected Competition Performance

Comparable to Satallax.

7.12 LEO-II 1.7.0

Alexander Steen
University of Greifswald, Germany

Architecture

LEO-II [12], the successor of LEO [11], 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 [10]. 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 [72]. 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 [13]. In fact, the development of LEO-II has largely paralleled the development of the TPTP THF language and related infrastructure [125]. LEO-II's parser supports the TPTP THF0 language and also the TPTP languages FOF and CNF.

Unfortunately the LEO-II system still uses only a very simple sequential collaboration model with first-order ATPs instead of using the more advanced, concurrent and resource-adaptive OANTS architecture [14] as exploited by its predecessor LEO.

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

```
http://www.leoprover.org
```

Expected Competition Performance

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

7.13 Leo-III 1.7.0

Alexander Steen
University of Greifswald, Germany

Architecture

Leo-III [80], the successor of LEO-II [12], 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 [73, 77] are used as external systems. However, cooperation is in general not limited to first-order systems. Further TPTP/TSTP-compliant external systems (such as higher-order ATPs or counter model generators) may be included using simple command-line arguments. If the saturation procedure loop (or one of the external provers) finds a proof, the system stops, generates the proof certificate and returns the result.

Strategies

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

Implementation

Leo-III utilizes and instantiates the associated LeoPARD system platform [149] for higher-order (HO) deduction systems implemented in Scala (currently using Scala 2.13 and running on a JVM with Java 8). The prover makes use of LeoPARD's data structures and implements its own reasoning logic on top. A hand-crafted parser is provided that supports all TPTP syntax dialects. It converts its produced concrete syntax tree to an internal TPTP AST data structure which is then transformed into polymorphically typed lambda terms. As of version 1.1, Leo-III supports all common TPTP dialects (CNF, FOF, TFF, THF) as well as their polymorphic variants [19, 48]. Since version 1.6.X (X ¿= 0) Leo-III also accepts non-classical problem input represented in non-classical TPTP, see ...

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

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

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

Leo-III is available on GitHub:

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

Expected Competition Performance

Version 1.7 is essentially (for the purposes of CASC) equivalent to 1.6, except that support for reasoning in various non-classical logics were added. We do not expect Leo-III to be strongly competitive against more recent higher-order provers as Leo-III does not implement several standard features of effective systems (including time slicing), and also the known bottlenecks have not been worked on.

7.14 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 [57]. 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 [56]. (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 [74] 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.

7.15 Satallax 3.4

Cezary Kaliszyk
Universität Innsbruck, Austria

Architecture

Satallax 3.4 [21] 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 [34] is responsible for much of the proof search. The theoretical basis of search is a complete ground tableau calculus for higher-order logic [23] 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 [22]. 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 [146]. If this option is used, then Satallax periodically calls the theorem prover E [76] 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 [43] 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.

7.16 SnakeForV4.7 1.0

Martin Suda
Czech Technical University in Prague, Czech Republic

Architecture

Snake is a strategy discovery and schedule construction tool applied - in this submission - to Vampire 4.7. It combines a simple strategy space sampling (ssss) with *singleton-focused stochastic strategy sieving* (ssss) to build a database of strategies. This is followed up by a *heuristic instruction-limited schedule selection* (hiss). The stochastic aspect, as well as the instruction limiting, loosely build on previous work [81].

Singleton-focused means that, apriori, each discovered strategy is optimized for a single (hard) problem. However, the *sieving* strips down (i.e., makes default) all options but those necessary for the observed success. This, together with the fact that Vampire's strategy description language is relatively crude, helps prevent unwanted overfitting.

The heuristic schedule selection phase uses *(probabilistic) weighted greedy cover*, where the weight of improving the schedule's expected number of solved problems by ΔE while scheduling a strategy S for I additional instructions is $\Delta E/I$.

Strategies

The strategies avaliable for discovery to SnakeForV4.7 are those of Vampire 4.7. (However standalone Vampire 4.7 will run its own schedules.) Should Vampire 4.7 rely on its schedules from last years, SnakeForV4.7 could highlight features added to Vampire more recently, such as layered clause selection [36] or new arithmetic reasoning rules [63]. SnakeForV4.7 also makes use of various randomization and shuffling options [81], which allow it to approach theorem proving probabilistically. For example, Snake could decide to schedule two independently seeded copies of the same strategy in order to lower the chance of not solving some key problems.

While the schedules (one for each entered division) are constructed to maximize the expected performace on the respective known TPTP problems, the final schedules are monolithic and do not branch based on specific problems' characteristics.

Implementation

While Snake itself is a suite of Python and bash scripts, operated semi-manually in several phases, and is not ready for release at this point, the version of Vampire with SnakeForV4.7 schedules will be available at:

`https://github.com/vprover/vampire/tree/shuffling`

Expected Competition Performance

SnakeForV4.7 has a chance of being better than Vampire 4.7 proper. This could mean a (demonstrated) win in FOF, TFA, and FNT, and a medal place in UEQ.

7.17 Toma 0.2

Teppei Saito
Japan Advanced Institute of Science and Technology, Japan

Architecture

Toma 0.2 is an automatic equational theorem prover. It proves unsatisfiability of a UEQ problem as follows: A given problem is transformed into a word problem whose validity entails unsatisfiability of the original problem. The word problem is solved by a new variant of maximal (ordered) completion [148, 41].

Strategies

Toma performs ordered completion in the following way: Given an equational system E, the tool constructs a lexicographic path order $>$ that maximizes reducibility of the ordered rewrite system (E, $>$) [148]. Using the order $>$, the tool runs ordered completion [3] on E without the deduce rule (critical pair generation). Especially, redundant equations are deleted by Martin and Nipkow's ground joinability testing [55]. Such a run eventually ends with an inter-reduced version (E', $>$). The tool checks joinability of the goal. If the goal is joinable in (E', $>$), the tool outputs the proof and terminates. Otherwise, assigning the union of E' and a set of critical pairs of (E', $>$) to E, the tool goes back to the first step.

Implementation

Toma is written in Haskell. To solve maximization problems of reducibility, it uses Z3. The source code is available at:

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

Expected Competition Performance

Toma is still in the experimental stage and unable to compete with matured tools.

7.18 Twee 2.4

Nick Smallbone
Chalmers University of Technology, Sweden

Architecture

Twee [79] is a theorem prover for unit equality problems based on unfailing completion [3]. It implements a DISCOUNT loop, where the active set contains rewrite rules (and unorientable equations) and the passive set contains critical pairs. The basic calculus is not goal-directed, but Twee implements a transformation which improves goal direction for many problems.

Twee features ground joinability testing [55] and a connectedness test [2], which together eliminate many redundant inferences in the presence of unorientable equations. The ground joinability test performs case splits on the order of variables, in the style of [55], and discharges individual cases by rewriting modulo a variable ordering.

Horn clauses are encoded as equations as described in [25]. For CASC, Twee accepts non-Horn problems but throws away all the non-Horn clauses.

Strategies

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

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

Each critical pair is scored using a weighted sum of the weight of both of its terms. Terms are treated as DAGs when computing weights, i.e., duplicate subterms are only counted once per term. The weights of critical pairs that correspond to Horn clauses are adjusted by the heuristic described in [25], section 5.

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

Implementation

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

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

Twee can be downloaded as open source from:

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

Expected Competition Performance

Twee 2.4 is the CASC-28 UEQ division winner.

7.19 Twee 2.4.1

Nick Smallbone
Chalmers University of Technology, Sweden

Architecture

Twee [79] is a theorem prover for unit equality problems based on unfailing completion [3]. It implements a DISCOUNT loop, where the active set contains rewrite rules (and unorientable equations) and the passive set contains critical pairs. The basic calculus is not goal-directed, but Twee implements a transformation which improves goal direction for many problems.

Twee features ground joinability testing [55] and a connectedness test [2], which together eliminate many redundant inferences in the presence of unorientable equations. The ground joinability test performs case splits on the order of variables, in the style of [55], and discharges individual cases by rewriting modulo a variable ordering.

Strategies

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

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

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

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

Implementation

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

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

Twee can be downloaded as open source from:

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

Expected Competition Performance

Similar to Twee 2.4, the current champion in UEQ.

7.20 Vampire 4.5

Giles Reger
University of Manchester, United Kingdom

Architecture

Vampire [52] 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 [65]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [143, 62]. 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 [66]. 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 [71]
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [143]
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals [42].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection. This has been extended with a layered clause selection approach [36].
- 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 [62].
  - Specialised theory instantiation and unification [67].
  - 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 [15] utilising a KBO-like ordering [16] for orienting combinator equations. The calculus introduces an inference, narrow, for rewriting with combinator equations.
  - Proof search heuristics targeting the growth of clauses resulting from narrowing.
  - An extension of unification with abstraction to deal with functional and boolean extensionality.
- Various inferences to deal with booleans

Implementation

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

`https://vprover.github.io`

for more information and access to the GitHub repository.

Expected Competition Performance

Vampire 4.5 is the CASC-27 TFA division winner.

7.21 Vampire 4.6

Giles Reger
University of Manchester, United Kingdom

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

Architecture

Vampire [52] 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 [65]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [143, 62].

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 [66]. Vampire implements many useful preprocessing transformations including the SinE axiom selection algorithm.

When a theorem is proved, the system produces a verifiable proof, which validates both the clausification phase and the refutation of the CNF.

Strategies

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

- Choices of saturation algorithm:
  - Limited Resource Strategy [71]
  - DISCOUNT loop

  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [143]
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals [42].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection. This has been extended with a layered clause selection approach [36].
- 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 [63].
  - Use of Z3 with AVATAR to restrict search to ground-theory-consistent splitting branches [62].
  - Specialised theory instantiation and unification [67].
  - Extensionality resolution with detection of extensionality axioms
- For higher-order problems:
  - Translation to polymorphic first-order logic using applicative form and combinators
  - A superposition calculus [15] utilising a KBO-like ordering [16] for orienting combinator equations. The calculus introduces an inference, narrow, for rewriting with combinator equations.
  - Proof search heuristics targeting the growth of clauses resulting from narrowing.
  - An extension of unification with abstraction to deal with functional and boolean extensionality.
  - Various inferences to deal with booleans

Implementation

Vampire 4.6 is implemented in C++. It makes use of minisat and z3. See the website for more information and access to the GitHub repository:

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

Expected Competition Performance

Vampire 4.6 is the CASC-28 FOF, FNT, and LTB division winner.

7.22 Vampire 4.7

Giles Reger
University of Manchester, United Kingdom

There are only small changes between Vampire 4.7 and Vampire 4.6 in the tracks relevant to CASC. As TFA did not run in 2021, the updates related to the paper "Making Theory Reasoning Simpler" [63] that were present last year should have an impact this year. This work introduces a new set of rules for the evaluation and simplification of theory literals. We have also added some optional preprocessing steps inspired by Twee (see "Twee: An Equational Theorem Prover" [79]) but these have not been fully incorporated into our strategy portfolio so are unlikely to make a significant impact.

Architecture

Vampire [52] 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 [65]. Splitting in resolution-based proof search is controlled by the AVATAR architecture which uses a SAT or SMT solver to make splitting decisions [143, 62]. 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 [66]. Vampire implements many useful preprocessing transformations including the SinE axiom selection algorithm. When a theorem is proved, the system produces a verifiable proof, which validates both the clausification phase and the refutation of the CNF.

Strategies

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

- Choices of saturation algorithm:
  - Limited Resource Strategy [71]
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR [143]
- A variety of optional simplifications.
- Parameterized reduction orderings.

- A number of built-in literal selection functions and different modes of comparing literals [42].
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection. This has been extended with a layered clause selection approach [36].
- 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 [63].
  - Use of Z3 with AVATAR to restrict search to ground-theory-consistent splitting branches [62].
  - Specialised theory instantiation and unification [67].
  - Extensionality resolution with detection of extensionality axioms
- For higher-order problems:
  - Translation to polymorphic first-order logic using applicative form and combinators
  - A superposition calculus [15] utilising a KBO-like ordering [16] for orienting combinator equations. The calculus introduces an inference, narrow, for rewriting with combinator equations.
  - Proof search heuristics targeting the growth of clauses resulting from narrowing.
  - An extension of unification with abstraction to deal with functional and boolean extensionality.
  - Various inferences to deal with booleans

Implementation

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

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

Expected Competition Performance

Vampire 4.7 should be roughly the same as Vampire 4.6. The only place we might see an improvement is in TFA.

7.23 Zipperposition 2.1

Petar Vukmirović
Vrije Universiteit Amsterdam, The Netherlands

Architecture

Zipperposition is a superposition-based theorem prover for typed first-order logic with equality and for higher-order logic. It is a pragmatic implementation of a complete calculus for full higher-order logic [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 [33]; pragmatic boolean reasoning [147]. The core architecture of the prover is based on saturation with an extensible set of rules for inferences and simplifications. Zipperposition uses a full higher-order unification algorithm that enables efficient integration of procedures for decidable fragments of higher-order unification [145]. The initial calculus and main loop were imitations of an old version of E [73]. With the implementation of higher-order superposition, the main loop had to be adapted to deal with possibly infinite sets of unifiers [144]. A summary of the calculus for integer arithmetic and induction can be found in [26].

Strategies

The system uses various strategies in a portfolio. The strategies are run in parallel, making use of all CPU cores available. We designed the portfolio of strategies by manual inspection of different TPTP problems. Heuristics used in Zipperposition are inspired by efficient heuristics used in E. A detailed overview of various calculus extensions used by the strategies is available [144]. Portfolio mode differentiates higher-order problems from the first-order ones. If the problem is first-order all higher-order prover features are turned off. In particular, the prover uses standard first-order superposition calculus and disables collaboration with the backend prover. Other than that, the portfolio is static and does not depend on the syntactic properties of the problem.

Implementation

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

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

and is entirely free software (BSD-licensed).

Zipperposition can also output graphic proofs using graphviz. Some tools to perform type inference and clausification for typed formulas are also provided, as well as a separate library for dealing with terms and formulas [26].

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 [26].

Expected Competition Performance

Zipperposition 2.1 is the CASC-28 THF division winner.

7.24 Zipperposition 2.1.999

Jasmin Blanchette
Vrije Universiteit Amsterdam, The Netherlands

Architecture

Zipperposition is a superposition-based theorem prover for typed first-order logic with equality and for higher-order logic. It is a pragmatic implementation of a complete calculus for full higher-order logic [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 [33], and Boolean reasoning [147]. The core architecture of the prover is based on saturation with an extensible set of rules for inferences and simplifications. Zipperposition uses a full higher-order unification algorithm that enables efficient integration of procedures for decidable fragments of higher-order unification [145]. The initial calculus and main loop were imitations of an earlier version of E [73]. With the implementation of higher-order superposition, the main loop had to be adapted to deal with possibly infinite sets of unifiers [144].

Strategies

The system uses various strategies in a portfolio. The strategies are run in parallel, making use of all CPU cores available. We designed the portfolio of strategies by manual inspection of TPTP problems. Zipperposition's heuristics are inspired by efficient heuristics used in E. Various calculus extensions are used by the strategies [144]. The portfolio mode distinguishes between first-order and higher-order problems. If the problem is first-order, all higher-order prover features are turned off. In particular, the prover uses standard first-order superposition calculus and disables collaboration with the backend prover (described below). Other than that, the portfolio is static and does not depend on the syntactic properties of the problem.

Implementation

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

Zipperposition's code can be found at

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

and is entirely free software (BSD-licensed).

Zipperposition can also output graphic proofs using graphviz. Some tools to perform type inference and clausification for typed formulas are also provided, as well as a separate library for dealing with terms and formulas [26].

Expected Competition Performance

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

8 Conclusion

The 11th IJCAR ATP System Competition was the twenty-sixth large scale competition for classical logic ATP systems. The organizers believe that CASC fulfills its main motivations: evaluation of relative capabilities of ATP systems, stimulation of research, motivation for improving implementations, and providing an exciting event. Through the continuity of the event and consistency in the reporting of the results, performance comparisons with previous and future years are easily possible. The competition provides exposure for system builders both within and outside of the community, and provides an overview of the implementation state of running, fully automatic, classical logic ATP systems.

References

[1] T. Ammer and P. Lammich. van Emde Boas Trees. *Archive of Formal Proofs*, 2021. http://isa-afp.org/entries/Van_Emde_Boas_Trees.html.

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

[3] 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.

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

[9] A. Bentkamp, J. Blanchette, S. Tourret, P. Vukmirović, 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

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

[19] 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.

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

[21] 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.

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

[23] 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.

[24] J. Cailler, J. Rosain, D. Delahaye, S. Robillard, and H. Bouziane. Goéland: A Concurrent Tableau-Based Theorem Prover. In J. Blanchette, L. Kovacs, and D. Pattinson, editors, *Proceedings of the 11th International Joint Conference on Automated Reasoning*, number 13385 in Lecture Notes in Artificial Intelligence, page To appear, 2022.

[25] 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.

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

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

337–340. Springer-Verlag, 2008.

[28] A. Degtyarev and A. Voronkov. What You Always Wanted To Know About Rigid E-unification. *Journal of Automated Reasoning*, 20(1-2):47–80, 1998.

[29] D. Delahaye, D. Doligez, F. Gibert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. 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 274–290. Springer-Verlag, 2013.

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

[31] A. Duarte and K. Korovin. AC Simplifications and Closure Redundancies in the Superposition Calculus. In A. Das and S. Negri, editors, *Proceedings of the 30th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, number 12842 in Lecture Notes in Artificial Intelligence, pages 200–217. Springer-Verlag, 2021.

[32] A. Duarte and K. Korovin. Ground Joinability and Connectedness in the Superposition Calculus. In J. Blanchette, L. Kovacs, and D. Pattinson, editors, *Proceedings of the 11th International Joint Conference on Automated Reasoning*, number 13385 in Lecture Notes in Artificial Intelligence, page To appear, 2022.

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

[34] 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.

[35] 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.

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

[37] 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.

[38] O. Hermant. Polarized Rewriting and Tableaux in B Set Theory. In M. Cristia, D. Delahaye, and C. Dubois, editors, *Proceedings of the 3rd International Workshop on Sets and Tools*, number 2199 in CEUR Workshop Proceedings, pages 67–72, 2018.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] E. Holden and K. Korovin. Heterogeneous Heuristic Optimisation and Scheduling for First-Order Theorem Proving. In F. Kamareddine and C. Sacerdoti, editors, *Proceedings of the 14th International Conference on Intelligent Computer Mathematics*, number 12833 in Lecture Notes in Computer Science, pages 107–123. Springer-Verlag, 2021.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

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

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

[55] 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.

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

[57] W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne

National Laboratory, Argonne, USA, 2003.

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

[59] 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.

[60] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 335–367. Elsevier Science, 2001.

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

[62] 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.

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

[64] G. Reger, M. Suda, and A. Voronkov. Playing with AVATAR. In A. Felty and A. Middeldorp, editors, *Proceedings of the 25th International Conference on Automated Deduction*, number 9195 in Lecture Notes in Computer Science, pages 399–415. Springer-Verlag, 2015.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

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

[72] 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.

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

[74] 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.

[75] 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.

[76] 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.

[77] 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.

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

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

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

[81] M. Suda. Vampire Getting Noisy: Will Random Bits Help Conquer Chaos? (System Description). EasyChair Preprint no. 7719, 2022.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[98] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[117] 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.

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

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

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

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

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

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

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

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

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

[127] 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.

[128] 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.

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

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

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

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

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

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

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

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

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

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

[139] 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.

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

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

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

[143] 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.

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

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

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

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

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

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

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