CASC-J8

CASC-J8

CASC-J8

CASC-J8

Proceedings of the 8th IJCAR ATP System Competition (CASC-J8)

Geoff Sutcliffe

University of Miami, USA

Abstract

The CADE ATP System Competition (CASC) evaluates the performance of sound, fully automatic, classical logic, ATP systems. The evaluation is in terms of the number of problems solved, the number of acceptable proofs and models produced, and the average runtime for problems solved, in the context of a bounded number of eligible problems chosen from the TPTP problem library and other useful sources of test problems, and specified time limits on solution attempts. The 8th IJCAR ATP System Competition (CASC-J8) was held on 29th July 2016. The design of the competition and its rules, and information regarding the competing systems, are provided in this report.

1 Introduction

The CADE and IJCAR conferences are the major forum for the presentation of new research in all aspects of automated deduction. In order to stimulate ATP research and system development, and to expose ATP systems within and beyond the ATP community, the CADE ATP System Competition (CASC) is held at each CADE and IJCAR conference. CASC-J8 was held on 29th July 2016, as part of the 8th International Joint Conference on Automated Reasoning (IJCAR 2016)[1], in Coimbra, Portugal. It was the twenty-first competition in the CASC series [110, 116, 113, 74, 76, 109, 107, 108, 81, 83, 85, 87, 90, 93, 95, 97, 99, 101, 103, 115].

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

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

in the context of:

- a bounded number of eligible problems, chosen from the TPTP problem library [91] and other useful sources of test problems, and
- specified time limits on solution attempts.

Twenty-one ATP system versions, listed in Table 1, entered into the various competition and demonstration divisions. The winners of the CASC-25 (the previous CASC) divisions were automatically entered into those divisions, to provide benchmarks against which progress can be judged (the competition archive provides access to the systems' executables and source code).[2]

The design and procedures of this CASC evolved from those of previous CASCs [110, 111, 106, 112, 72, 73, 75, 77, 78, 79, 80, 82, 84, 86, 89, 92, 94, 96, 98, 100, 102]. Important changes for this CASC were:

[1]CADE was a constituent conference of IJCAR, hence CASC-"J8".

[2]As the LTB division had been suspended since CASC-24 in 2013, the CASC-24 UEQ winner was entered.

ATP System	Divisions	Entrant (Associates)	Entrant's Affiliation
Beagle 0.9.47	TFA TFN	Peter Baumgartner (Joshua Bax)	Data61
CVC4 1.5	TFN	CASC	CASC-25 winner
CVC4 1.5.1	TFA TFN FOF FNT	Andrew Reynolds (Clark Barrett, Cesare Tinelli, Tim King)	University of Iowa
E 2.0	FOF FNT EPR LTB	Stephan Schulz (Martin Möhrmann	DHBW Stuttgart
Geo-III 2016C	FOF FNT EPR	Hans de Nivelle	University of Wrocław
iProver 2.5	FOF FNT EPR LTB	Konstantin Korovin	University of Manchester
Isabelle 2015	THF	Jasmin Blanchette (Lawrence Paulson, Tobias Nipkow, Makarius Wenzel)	Inria Nancy
leanCoP 2.2	FOF	Jens Otten	University of Potsdam
LEO-II 1.7.0	THF	Max Wisniewski (Alexander Steen Christoph Benzmüller)	Freie Universität Berlin
LEO-III 1.0	THF	Max Wisniewski (Alexander Steen Christoph Benzmüller)	Freie Universität Berlin
LEO+III 1.0	THF	Max Wisniewski (Alexander Steen Christoph Benzmüller)	Freie Universität Berlin
MaLARea 0.6	LTB (demo)	Josef Urban (Jan Jakubuv, Cezary Kaliszyk, Stephan Schulz, Jiri Vyskocil)	Czech Technical University in Prague
Nitpick 2015	THN	Jasmin Blanchette	Inria Nancy
Princess 160606	TFA TFN	Philipp Rümmer	Uppsala University
Prover9 2009-11A	FOF	CASC (William McCune, Bob Veroff)	CASC fixed point
Prover9Plus 1.0	LTB (demo)	Bob Veroff (William McCune, Josef Urban	University of New Mexico
Refute 2015	THN	Jasmin Blanchette (Tjark Weber)	Inria Nancy
Satallax 2.8	THF	CASC	CASC-25 winner
Satallax 3.0	THF	Michael Färber (Chad Brown)	Universität Innsbruck
Vampire 4.0	FOF FNT EPR LTB	CASC	CASC-25 winner
Vampire 4.1	TFA FOF FNT EPR LTB	Giles Reger (Martin Suda, Andrei Voronkov Evgeny Kotelnikov, Laura Kovacs	University of Manchester
VampireZ3 1.0	TFA	CASC	CASC-25 winner

Table 1: The ATP systems and entrants

- The THN division was put into a hiatus state.
- Efficiency based on wall clock time was added to the results.

The competition organizer was Geoff Sutcliffe. The competition is overseen by a panel of knowledgeable researchers who are not participating in the event. The panel members were Pascal Fontaine, Aart Middeldorp, and Christoph Wernhard. The CASC rules, specifications, and deadlines are absolute. Only the panel has the right to make exceptions. The competition was run on computers provided by StarExec at the University of Iowa. The CASC-J8 web site provides access to resources used before, during, and after the event: `http://www.tptp.org/CASC/J8`

It was assumed that all entrants had read the web pages related to the competition, and have complied with the competition rules. Non-compliance with the rules could lead to disqualification. A "catch-all" rule was used to deal with any unforeseen circumstances: *No cheating is allowed*. The panel was allowed to disqualify entrants due to unfairness, and to adjust the competition rules in case of misuse.

2 Divisions

CASC is divided into divisions according to problem and system characteristics. There are *competition divisions* in which systems are explicitly ranked, and a *demonstration division* in which systems demonstrate their abilities without being ranked. Some divisions are further divided into problem categories, which makes it possible to analyse, at a more fine grained level, which systems work well for what types of problems. The problem categories have no effect on the competition rankings, which are made at only the division level.

2.1 The Competition Divisions

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

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

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

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

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

The **TFN** division: Typed First-order with arithmetic Non-theorems (axioms with a provable conjecture).

The **FOF** division: First-Order Form theorems (syntactically non-propositional, 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 (syntactically non-propositional, axioms with a countersatisfiable conjecture, and satisfiable axiom sets). The FNT division has two problem categories:

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

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

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

The **LTB** division: First-order form theorems (axioms with a provable conjecture) from Large Theories, presented in Batches. A large theory has many functors and predicates, and many axioms of which typically only a few are required for the proof of a theorem. Problems in a batch all use a common core set of axioms, and the problems in a batch are given to the ATP system all at once. 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. The LTB division's problem categories are accompanied by sets of training problems and their solutions, taken from the same exports as the competition problems, that can be used for tuning and training during (typically at the start of) the competition. The LTB division has three problem categories:

- The **AIM** category: Problems exported from the AIM loops project [34].
- The **CML** category: Problems exported from CakeML [40].
- The **HLL** category: Problems exported from the Flyspeck project in HOL Light [30].

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

2.2 The Demonstration Division

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

3 Infrastructure

3.1 Computers

The computers had:

- Two quad-core Intel(R) Xeon(R) E5-2609, 2.40GHz CPUs
- 128GB memory

- The Red Hat Enterprise Linux Workstation release 6.3 (Santiago) operating system, kernel 2.6.32-431.1.2.el6.x86_64

Each ATP system ran one job on one CPU at a time. Systems could use all the cores on the CPU (although this did not necessarily help in the non-LTB divisions, because a CPU time limit was imposed).

3.2 Problems

3.2.1 Problem Selection

Problems for the non-LTB divisions were taken from the TPTP Problem Library, version v6.4.0. The TPTP version used for CASC is released after the competition has started, so that new problems have not been seen by the entrants. The problems have to meet certain criteria to be eligible for selection. The problems used are randomly selected from the eligible problems based on a seed supplied by the competition panel.

- The TPTP tags problems that designed specifically to be suited or ill-suited to some ATP system, calculus, or control strategy as *biased*, and they are excluded from the competition.
- The problems are syntactically non-propositional.
- The TPTP uses system performance data in the Thousands of Solutions from Theorem Provers (TSTP) solution library to compute problem difficulty ratings in the range 0.00 (easy) to 1.00 (unsolved) [114]. Difficult problems with a rating in the range 0.21 to 0.99 are eligible for CASC. Problems of lesser and greater ratings might also be eligible in some divisions if there are not enough problems with the desired ratings. Performance data from systems submitted by the system submission deadline is used for computing the problem ratings for the TPTP version used for the competition.
- The selection is constrained so that no division or category contains an excessive number of very similar problems.
- The selection mechanism is biased to select problems that are new in the TPTP version used, until 50% of the problems in each category have been selected, after which random selection (from old and new problems) continues. The actual percentage of new problems used depends on how many new problems are eligible and the limitation on very similar problems.

Problems for the LTB division are taken from publicly available problem sets: the AIM problem category used the AIM2185 problem set[3]; the CML problem category used the CML10277 problem set[4]; the HLL problem category used the HH7150 problem set[5]; Entrants are expected to honestly not use any of the (publicly available) problems for tuning or training before the competition.

The problems in each category have a large number of common `include`d axiom files (the "common core set of axioms"). Systems can benefit from preloading and analyzing these common axioms once, in advance of problem solving.

In order to facilitate and promote learning from previous proofs, each problem category is accompanied by a set of training problems and their solutions, which can be used for tuning and training during (typically at the start of) the competition. The training problems are not used

[3]`http://www.tptp.org/CASC/J8/AIM2185.tgz`
[4]`http://grid02.ciirc.cvut.cz/~mptp/CML10227.tar.gz`
[5]`https://github.com/JUrban/HH7150`

in the competition. In order to support learning, the problems in each category have consistent symbol usage, and almost consistent axiom naming, between problems.

3.2.2 Number of Problems

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 [28] (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 is used in determining the time limits imposed on each solution attempt - see Section 3.3.

A lower bound on the total number of problems to be used is determined from the number of computers available, the time allocated to the competition, the number of ATP systems to be run on the competition computers over all the divisions, and the per-problem time limit, according to the following relationship:

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

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

The numbers of problems used in the categories in the various divisions are (roughly) proportional to the numbers of eligible problems, after taking into account the limitation on very similar problems. The numbers of problems used in each division and category are determined according to the judgement of the competition organizers.

3.2.3 Problem Preparation

The problems are in TPTP format, with `include` directives. The problems in each non-LTB division are given in increasing order of TPTP difficulty rating. The problems in the LTB division are given in the natural order of their export.

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

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

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` *division_mnemonic*`.`*category_mnemonic*

  For the LTB division it was

`division.category LTB.`*category_mnemonic* where the category mnemonics were `AIM`, `CML`, and `HLL`.
- The name of a directory 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_directory` *directory_name* The directory contains a file `TrainingData.`*ProblemCategory*`.tgz` that expands in place to three directories: `Axioms`, `Problems`, and `Solutions`. `Axioms` contains all the axiom files that can be used in the training and competition problems. `Problems` contains the training problems. `Solutions` contains a subdirectory for each of the `Problems`, containing TPTP format solutions to the problem.

Each batch specification consists of:

- A header line `% SZS start BatchConfiguration`
- A specification of whether or not the problems in the batch must be attempted in order is given, in a line of the form
  `execution.order` *ordered/unordered*
  For the LTB division it was
  `execution.order ordered`
  i.e., systems may not start any attempt on a problem, including reading the problem file, before ending the attempt on the preceding problem.
- 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 the LTB division 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.
- The overall wall clock time limit (for the batch), in a line of the form
  `limit.time.overall.wc` *limit_in_seconds*
- A terminator line `% SZS end BatchConfiguration`
- A header line `% SZS start BatchIncludes`
- `include` directives that are used in every problem. Problems in the batch have all these `include` directives, and can also have other `include` directives that are not listed here.
- A terminator line `% SZS end BatchIncludes`
- A header line `% SZS start BatchProblems`
- Pairs of absolute problem file names, and absolute output file names where the output for the problem must be written.
- A terminator line `% SZS end BatchProblems`

3.3 Resource Limits

3.3.1 Non-LTB divisions

CPU and wall clock time limits are imposed. The minimal CPU time limit per problem is 240s. The maximal CPU time limit per problem is determined using the relationship used for determining the number of problems, with the minimal number of problems as the

NumberOfProblems. The CPU time limit is chosen as a reasonable value within the range allowed, and is announced at the competition. The wall clock time limit is imposed in addition to the CPU time limit, to limit very high memory usage that causes swapping. The wall clock time limit per problem is double the CPU time limit. An additional memory limit is imposed, depending on the computers' memory. The time are imposed individually on each solution attempt.

In the demonstration division, each entrant can choose to use either a CPU or a wall clock time limit, whose value is the CPU time limit of the competition divisions.

3.3.2 LTB division

For each batch there is a wall clock time limit per problem, which is provided in the configuration section at the start of each batch. The minimal wall clock time limit per problem is 30s. For each problem category there is an overall wall clock time limit, which is provided in the configuration section at the start of each batch, and is also available as a command line parameter. The overall limit is the sum over the batches of the batch's per-problem 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 ending a problem and starting 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. In the LTB division, the wall clock time is the time from when the system reports starting on a problem and reports ending on a problem - the time spent before starting the first problem, and times spent between ending a problem and starting the next, are not part of the time taken on problems.

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

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

- Derivations must be complete, starting at formulae from the problem, and ending at the conjecture (for axiomatic proofs) or a $false$ formula (for proofs by contradiction, including CNF refutations).
- For proofs of FOF problems by CNF refutation, the conversion from FOF to CNF must be adequately documented.
- Derivations must show only relevant inference steps.
- Inference steps must document the parent formulae, the inference rule used, and the inferred formula.
- Inference steps must be reasonably fine-grained.

- An unsatisfiable set of ground instances of clauses is acceptable for establishing the unsatisfiability of a set of clauses.
- Models must be complete, documenting the domain, function maps, and predicate maps. The domain, function maps, and predicate maps may be specified by explicit ground lists (of mappings), or by any clear, terminating algorithm.

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

- The *state-of-the-art contribution* (SOTAC) quantifies the unique abilities of each system. For each problem solved by a system, its SOTAC for the problem is the inverse of the number of systems that solved the problem. A system's overall SOTAC is its average SOTAC over the problems it solves.
- The *core usage* is the average of the ratios of CPU time to wall clock time used, over the problems solved. This measures the extent to which the systems take advantage the multiple cores.
- The *efficiency measure* balances the number of problems solved with the time taken. It is the average of the inverses of the times for problems solved multiplied by the fraction of problems solved. This can be interpreted intuitively as the average of the solution rates for problems solved, multiplied by the fraction of problems solved. Efficiency is computed for both CPU time and wall clock time, to measure how efficiently the systems use one core and how efficiently systems use the multiple cores, respectively.

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

5 System Entry

To be entered into CASC, systems must be registered using the CASC system registration form. No registrations are accepted after the registration deadline. For each system entered, an entrant has to be nominated to handle all issues (including execution difficulties) arising before and during the competition. The nominated entrant must formally register for CASC. It is not necessary for entrants to physically attend the competition.

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

The division winners from the previous CASC are automatically entered into their divisions, to provide benchmarks against which progress can be judged. Prover9 2009-11A is automatically entered into the FOF division, to provide a fixed-point against which progress can be judged.

5.1 System Description

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

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

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

5.2 Sample Solutions

For systems in the proof/model classes, representative sample solutions must be emailed to the competition organizers by the sample solutions deadline. Use of the TPTP format for proofs and finite interpretations is encouraged. The competition panel decides whether or not proofs and models are acceptable.

Proof/model samples are required as follows:

- THF: `SET014`$\hat{4}$
- TFA: `DAT013=1`
- FOF and LTB: `SEU140+2`
- FNT: `NLP042+1` and `SWV017+1`

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

6 System Requirements

6.1 System Properties

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

Soundness and Completeness

- Systems must be sound. At some time before the competition all the systems in the competition divisions are tested for soundness. Non-theorems are submitted to the systems in the THF, TFA, FOF, EPR, and LTB divisions, and theorems are submitted to the systems in the TFN, FNT and EPR divisions. Finding a proof of a non-theorem or a disproof of a theorem indicates unsoundness. If a system fails the soundness testing it must be repaired by the unsoundness repair deadline or be withdrawn. For systems running on entrant supplied computers in the demonstration division, the entrant must perform the soundness testing and report the results to the competition organizers.

- Systems do not have to be complete in any sense, including calculus, search control, implementation, or resource requirements.
- All techniques used must be general purpose, and expected to extend usefully to new unseen problems. The precomputation and storage of information about individual TPTP problems and axiom sets is not allowed. Strategies and strategy selection based on individual TPTP problems is not allowed. If machine learning procedures are used, the learning must ensure that sufficient generalization is obtained so that no there is no specialization to individual problems or their solutions.
- The LTB training problems and solutions may be used for producing generally useful strategies that extend to other problems in the problem sets. Such strategies can rely on the consistent naming of symbols and formulas in the problem sets, and may use techniques for memorization and generalization of problems and solutions in the training set. The system description must fully explain any such tuning or training that has been done. Precomputation and storage of information about other problems in the LTB problem sets, or their solutions, is not allowed.
- The competition panel may disqualify any system whose tuning or training is deemed to be problem specific rather than general purpose.
- The system's performance must be reproducible by running the system again.

Execution

- Systems that cannot run on the competition computers can be entered into the demonstration division.
- Systems must be fully automatic, i.e., all command line switches have to be the same for all problems in each division.
- In the LTB division the systems must attempt the problems in the order given in the batch specification file. Systems may not start any attempt on a problem, including reading the problem file, before ending the attempt on the preceding problem.

Output

- In the non-LTB divisions all solution output must be to `stdout`. In the LTB division all solution output must be to the named output file for each problem.
- In the 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 /home/graph/tptp/TPTP/Problems/CSR/CSR075+2.p
  ... (system churns away, result and solution output to file)
% SZS status Theorem for /home/graph/tptp/TPTP/Problems/CSR/CSR075+2.p
% SZS status Ended for /home/graph/tptp/TPTP/Problems/CSR/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 [88] for this. For example

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

  or

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

  In the LTB division this line must be the last line output before the ending notification line (the line must also be output to the output file).
- When outputting proofs/models, the start and end of the proof/model must be delimited by distinguished strings. Systems must use the SZS ontology and standards for this. For example

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

  The string specifying the problem status must be output before the start of a proof/model. Use of the TPTP format for proofs and finite interpretations is encouraged [105].

Resource Usage

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

6.2 System Delivery

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

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

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

6.3 System Execution

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

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

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

7 The ATP Systems

These system descriptions were written by the entrants.

7.1 Beagle 0.9.47

Peter Baumgartner
Data61, Australia

Architecture

Beagle [4] is an automated theorem prover for sorted first-order logic with equality over built-in theories. The theories currently supported are integer arithmetic, linear rational arithmetic and linear real arithmetic. It accepts formulas in the FOF and TFF formats of the TPTP syntax, and formulas in the SMT-LIB version 2 format.

Beagle first converts the input formulas into clause normal form. Pure arithmetic (sub-)formulas are treated by eager application of quantifier elimination. The core reasoning component implements the Hierarchic Superposition Calculus with Weak Abstraction (HSPWA) [5]. Extensions are a splitting rule for clauses that can be divided into variable disjoint parts, and a partial instantiation rule for variables with finite domain, and two kinds of background-sorted variables trading off completeness vs. search space.

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

Strategies

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

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

Implementation

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

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

Expected Competition Performance

Beagle is implemented in a straightforward way and would benefit from optimized data structures. We do not expect it to come in among the first.

7.2 CVC4 1.5

Andrew Reynolds
EPFL, Switzerland

Architecture

CVC4 [2] is an SMT solver based on the DPLL(T) architecture [45] that includes built-in support for many theories, including linear arithmetic, arrays, bit vectors, datatypes and strings. It incorporates approaches for handling universally quantified formulas. CVC4 primarily uses heuristic approaches based on E-matching for theorems, and finite model finding approaches for non-theorems.

Like other SMT solvers, CVC4 treats quantified formulas using a two-tiered approach. First, quantified formulas are replaced by fresh Boolean predicates and the ground theory solver(s) are used in conjunction with the underlying SAT solver to determine satisfiability. If the problem is unsatisfiable at the ground level, then the solver answers "unsatisfiable". Otherwise, the quantifier instantiation module is invoked, and will either add instances of quantified formulas to the problem, answer "satisfiable", or return unknown.

Finite model finding in CVC4 targets problems containing background theories whose quantification is limited to finite and uninterpreted sorts. In finite model finding mode, CVC4 uses a ground theory of finite cardinality constraints that minimizes the number of ground equivalence classes, as described in [60]. 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 [61]. If no instances are added, it reports "satisfiable".

Strategies

For handling theorems, CVC4 primarily uses configurations that combine conflict-based quantifier instantiation [59] and E-matching. CVC4 uses a handful of orthogonal trigger selection strategies for E-matching.

For handling non-theorems, CVC4 primarily uses finite model finding techniques. These techniques can also be used for bounded integer quantification for non-theorems involving arithmetic [57]. Since CVC4 with finite model finding is also capable of establishing unsatisfiability, it is used as a strategy for theorems as well.

For problems in pure arithmetic, CVC4 uses techniques for counterexample-guided quantifier instantiation [58], which select relevant quantifier instantiations based on models for counterexamples to quantified formulas. CVC4 relies on this method both for theorems in TFA and non-theorems in TFN.

Implementation

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

```
https://github.com/CVC4
```

Expected Competition Performance

CVC4 1.5 is the CASC-25 TFN division winner.

7.3 CVC4 1.5

Andrew Reynolds
University of Iowa, USA

Architecture

CVC4 [2] is an SMT solver based on the DPLL(T) architecture [45] that includes built-in support for many theories, including linear arithmetic, arrays, bit vectors, datatypes, finite sets and strings. It incorporates approaches for handling universally quantified formulas. For problems involving free function and predicate symbols, CVC4 primarily uses heuristic approaches based on E-matching for theorems, and finite model finding approaches for non-theorems. For problems in pure arithmetic, CVC4 uses techniques for counterexample-guided quantifier instantiation [58].

Like other SMT solvers, CVC4 treats quantified formulas using a two-tiered approach. First, quantified formulas are replaced by fresh Boolean predicates and the ground theory solver(s) are used in conjunction with the underlying SAT solver to determine satisfiability. If the problem is unsatisfiable at the ground level, then the solver answers "unsatisfiable". Otherwise, the quantifier instantiation module is invoked, and will either add instances of quantified formulas to the problem, answer "satisfiable", or return unknown. Finite model finding in CVC4 targets problems containing background theories whose quantification is limited to finite and uninterpreted sorts. In finite model finding mode, CVC4 uses a ground theory of finite cardinality constraints that minimizes the number of ground equivalence classes, as described in [60]. 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 [61]. If no instances are added, it reports “satisfiable”.

Strategies

For handling theorems, CVC4 primarily uses configurations that combine conflict-based quantifier instantiation [59] and E-matching. CVC4 uses a handful of orthogonal trigger selection strategies for E-matching. For handling non-theorems, CVC4 primarily uses finite model finding techniques. Since CVC4 with finite model finding is also capable of establishing unsatisfiability, it is used as a strategy for theorems as well. For problems in pure arithmetic, CVC4 uses variations of counterexample-guided quantifier instantiation, which select relevant quantifier instantiations based on models for counterexamples to quantified formulas. CVC4 relies on this method both for theorems in TFA and non-theorems in TFN.

Implementation

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

```
https://github.com/CVC4
```

Expected Competition Performance

CVC4 should perform moderately better than last year in FOF and TFA. The main improvements have been a new implementation of counterexample-guided quantifier instantiation [58] for linear real and integer arithmetic, optimizations for ground theory combination and conflict-based quantifier instantiation, and the use of new strategies. It should perform roughly the same in FNT and TFN.

7.4 E 2.0

Stephan Schulz
DHBW Stuttgart, Germany

Architecture

E 2.0 [67, 71] is a purely equational theorem prover for many-sorted first-order logic with equality. It consists of an (optional) clausifier for pre-processing full first-order formulae into clausal form, and a saturation algorithm implementing an instance of the superposition calculus with negative literal selection and a number of redundancy elimination techniques. E is based on the DISCOUNT-loop variant of the given-clause algorithm, i.e., a strict separation of active and passive facts. No special rules for non-equational literals have been implemented. Resolution is effectively simulated by paramodulation and equality resolution.

For the 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).

For CASC-J8, E implements a strategy-scheduling automatic mode. The total CPU time available is broken into several (unequal) time slices. For each time slice, the problem is classified into one of several classes, based on a number of simple features (number of clauses, maximal symbol arity, presence of equality, presence of non-unit and non-Horn clauses, ...). For each class, a schedule of strategies is greedily constructed from experimental data as follows: The first strategy assigned to a schedule is the the one that solves the most problems from this class in the first time slice. Each subsequent strategy is selected based on the number of solutions on problems not already solved by a preceding strategy.

About 210 different strategies have been evaluated on all untyped first-order problems from TPTP 6.0.0, and about 180 of these strategies are used in the automatic mode.

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 [68]. Superposition and backward rewriting use fingerprint indexing [70], 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 [43, 42]. The prover and additional information are available at

`http://www.eprover.org`

Expected Competition Performance

E 2.0 has slightly better strategies than previous versions, and has some minor improvements in the inference engine. The system is expected to perform well in most proof classes, but will at best complement top systems in the disproof classes.

7.5 Geo-III 2016C

Hans de Nivelle
University of Wrocław, Poland

Architecture

Geo III is a theorem prover for Partial Classical Logic, based on reduction to Kleene Logic [23]. Currently, only Chapters 4 and 5 are implemented. Since Kleene logic generalizes 2-valued logic, Geo III can take part in CASC. Apart from being 3-valued, the main differences with earlier versions of Geo are (1) more sophisticated learning schemes, (2) improved proof logging, and (3) replacement of recursion by explicit use of a stack. The Geo family of provers uses exhaustive backtracking, combined with learning after failure. Earlier versions learned only conflict formulas. Geo III learns disjunctions of arbitrary width. Experiments show that this often results in shorter proofs.

If Geo will be ever embedded in proof assistants, these assistants will require proofs. In order to be able to provide these at the required level of detail, Geo III contains a hierarchy of proof rules that is independent of the rest of the system, and that can be modified independently. In order to be flexible in the main loop, recursion was replaced by an explicit stack. Using an explicit stack, it is easier to remove unused assumptions, or to rearrange the order of assumptions. Also, restarts are easier to implement with a stack.

Strategies

Geo uses breadth-first, exhaustive model search, combined with learning. In case of branching, the branches are explored in random order. Specially for CASC, a restart strategy was added, which ensures that proof search is always restarted after 4 minutes. This was done because Geo III has no indexing. After some time, proof search becomes so inefficient that it makes no sense to continue, so that it is better to restart.

Implementation

Geo III is written in C++-11. No features outside of the standard are used. It has been tested with g++ version 4.8.4 and with clang. Difference with previous year's version is that version 2016C uses sophisticated matching algorithms [24] for establishing if a geometric formula is false in an interpretation.

Expected Competition Performance

We expect that Geo 2016C will be better than Geo 2015E.

7.6 iProver 2.5

Kontantin Korovin
University of Manchester, United Kingdom

Architecture

iProver is an automated theorem prover based on an instantiation calculus Inst-Gen [26, 36] which is complete for first-order logic. iProver combines first-order reasoning with ground reasoning for which it uses MiniSat [25] and optionally PicoSAT [12] (only MiniSat will be used at this CASC). iProver also combines instantiation with ordered resolution; see [35, 36] for the implementation details. The proof search is implemented using a saturation process based on the given clause algorithm. iProver uses non-perfect discrimination trees for the unification indexes, priority queues for passive clauses, and a compressed vector index for subsumption and subsumption resolution (both forward and backward). The following redundancy eliminations are implemented: blocking non-proper instantiations; dismatching constraints [27, 35]; global subsumption [35]; resolution-based simplifications and propositional-based simplifications. A compressed feature vector index is used for efficient forward/backward subsumption and subsumption resolution. Equality is dealt with (internally) by adding the necessary axioms of equality. Recent changes in iProver include improved preprocessing and incremental finite model finding; support of the AIG format for hardware verification and model-checking (implemented with Dmitry Tsarkov).

In the LTB division, iProver uses axiom selection based on the Sine algorithm [32] as implemented in Vampire [39], i.e., axiom selection is done by Vampire and proof attempts are done by iProver.

Some of iProver features are summarised below.

- proof extraction for both instantiation and resolution [38],
- model representation, using first-order definitions in term algebra [38],
- answer substitutions,
- semantic filtering,
- incremental finite model finding,
- sort inference, monotonic [21] and non-cyclic [37] sorts,
- predicate elimination [33].

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

Strategies

iProver has around 60 options to control the proof search including options for literal selection, passive clause selection, frequency of calling the SAT solver, simplifications and options for combination of instantiation with resolution. At CASC iProver will execute a small number of fixed schedules of selected options depending on general syntactic properties such as

Horn/non-Horn, equational/non-equational, and maximal term depth. For the LTB and FNT divisions several strategies are run in parallel.

Implementation

Prover is implemented in OCaml and for the ground reasoning uses MiniSat [25]. iProver accepts FOF and CNF formats. Vampire [39, 31] and E prover [71] are used for proof-producing clausification of FOF problems, Vampire is also used for axiom selection [32] in the LTB division.
iProver is available at:

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

Expected Competition Performance

Compared to the last year, we restructured core datastructures aiming at flexibility to different extensions rather than performance. We also improved preprocessing, including predicated elimination. We expect a moderatly improved overall performance.

7.7 Isabelle 2015

Jasmin Blanchette
Inria Nancy, France

Architecture

Isabelle/HOL 2015 [47] is the higher-order logic incarnation of the generic proof assistant Isabelle2015. Isabelle/HOL provides several automatic proof tactics, notably an equational reasoner [46], a classical reasoner [55], and a tableau prover [53]. It also integrates external first- and higher-order provers via its subsystem Sledgehammer [54, 13]. Isabelle includes a parser for the TPTP syntaxes CNF, FOF, TFF0, and THF0, due to Nik Sultana. It also includes TPTP versions of its popular tools, invokable on the command line as `isabelle tptp_`*`tool max_secs file.p`*. For example:

```
isabelle tptp_isabelle_hot 100 SEU/SEU824^3.p
```

Isabelle is available in two versions. The HOT version (which is not participating in CASC-J8) includes LEO-II [8] and Satallax [18] as Sledgehammer backends, whereas the competition version leaves them out.

Strategies

The *Isabelle* tactic submitted to the competition simply tries the following tactics sequentially:
`sledgehammer`
Invokes the following sequence of provers as oracles via Sledgehammer:

- `satallax` - Satallax 2.7 [18] (*HOT version only*);
- `leo2` - LEO-II 1.6.2 [8] (*HOT version only*);

- `spass` - SPASS 3.8ds [15];
- `vampire` - Vampire 3.0 (revision 1435) [62];
- `e` - E 1.8 [69];

`nitpick`
For problems involving only the type `$o` of Booleans, checks whether a finite model exists using Nitpick [17].
`simp`
Performs equational reasoning using rewrite rules [46].
`blast`
Searches for a proof using a fast untyped tableau prover and then attempts to reconstruct the proof using Isabelle tactics [53].
`auto+spass`
Combines simplification and classical reasoning [55]
under one roof; then invoke Sledgehammer with SPASS on any subgoals that emerge.
`z3`
Invokes the SMT solver Z3 4.4.0 [22].
`cvc4`
Invokes the SMT solver CVC4 1.5pre [3].
`fast`
Searches for a proof using sequent-style reasoning, performing a depth-first search [55]. Unlike `blast`, it construct proofs directly in Isabelle. That makes it slower but enables it to work in the presence of the more unusual features of HOL, such as type classes and function unknowns.
`best`
Similar to `fast`, except that it performs a best-first search.
`force`
Similar to `auto`, but more exhaustive.
`meson`
Implements Loveland's MESON procedure [44]. Constructs proofs directly in Isabelle.
`fastforce`
Combines `fast` and `force`.

Implementation

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

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

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

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

Expected Competition Performance

Thanks to the addition of CVC4 and a new version of Vampire, Isabelle might have become now strong enough to take on Satallax and its various declensions. But we expect Isabelle to end in second or third place, to be honest.

7.8 leanCoP 2.2

Jens Otten
University of Potsdam, Germany

Architecture

leanCoP [52, 48] is an automated theorem prover for classical first-order logic with equality. It is a very compact implementation of the connection (tableau) calculus [11, 41].

Strategies

The reduction rule of the connection calculus is applied before the extension rule. Open branches are selected in a depth-first way. Iterative deepening on the proof depth is performed in order to achieve completeness. Additional inference rules and techniques include regularity, lemmata, and restricted backtracking [49]. leanCoP uses an optimized structure-preserving transformation into clausal form [49] and a fixed strategy scheduling, which is controlled by a shell script.

Implementation

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

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

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

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

The website also contains information about ileanCoP [48] and MleanCoP [50, 51], two versions of leanCoP for first-order intuitionistic logic and first-order modal logic, respectively.

Expected Competition Performance

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

7.9 LEO-II 1.7.0

Max Wisniewski
Freie Universität Berlin, Germany

Architecture

LEO-II [8], the successor of LEO [7], 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 [6]. 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 [66]. LEO-II is often too weak to find a refutation amongst the steadily growing set of clauses on its own. However, some of the clauses in LEO-II's search space attain a special status: they are first-order clauses modulo the application of an appropriate transformation function. Therefore, LEO-II launches a cooperating first-order ATP system every n iterations of its (standard) resolution proof search loop (e.g., 10). If the first-order ATP system finds a refutation, it communicates its success to LEO-II in the standard SZS format. Communication between LEO-II and the cooperating first-order ATP system uses the TPTP language and standards.

Strategies

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

Implementation

LEO-II is implemented in OCaml 4, and its problem representation language is the TPTP THF language [9]. In fact, the development of LEO-II has largely paralleled the development of the TPTP THF language and related infrastructure [104]. LEO-II's parser supports the TPTP THF0 language and also the TPTP languages FOF and CNF.

Unfortunately the LEO-II system still uses only a very simple sequential collaboration model with first-order ATPs instead of using the more advanced, concurrent and resource-adaptive OANTS architecture [10] as exploited by its predecessor LEO.

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

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

Expected Competition Performance

Leo-II 1.7.0 differs from last years CASC version only wrt to some proof generation aspects and some other minor modifications. These changes are not expected to improve LEO-II's performance at CASC over the previous version.

7.10 Leo-III 1.0

Max Wisniewski
Freie Universität Berlin, Germany

Architecture

Leo-III [120], the successor of LEO-II [8], is a higher-order ATP system based on ordered higher-order paramodulation employing an agent-based blackboard architecture. In its first version, Leo-III is using multiple, adapted sequential DISCOUNT loops, each with different search strategies. In addition, similar to LEO-II, each sequential loop will call non-blockingly an external ATP every n iterations of the sequential loop. In the current version, the called ATPs have to understand THF syntax and return the result in standard SZS format. In the competition mode only our own prover LEO-II will be used as a cooperation prover. If either one of the paramodulation loops or one of the external provers finds a proof, the system stops and returns the result.

Strategies

Leo-III runs multiple search strategies in parallel. These strategies containing some incomplete versions, that are outperforming the complete versions for some problem inputs. The search also differs in the employed relevance filters, preprocessing techniques and hence the considered formula set.

Ultimately, Leo-III is in its first version an enhancement of LEO-II. The main improvement in comparison to its predecessor is the better equational handling with the new calculus, and the multi-search of the agent architecture.

Implementation

Leo-III is implemented in Scala. Its natural problem representation is the TPTP THF language [9], but it can process every language of the TPTP including TFF and FOF. Leo-III is available from:

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

Expected Competition Performance

In its first version Leo-III is not yet tuned for performance, but more of a straight-forward implementation of the calculus itself. Due to its cooperation with LEO-II it will work at least as good as LEO-II, but it will most probably not be able to compete with the main competitors.

7.11 Leo+III 1.0

Max Wisniewski
Freie Universität Berlin, Germany

Architecture

Leo-III [120], the successor of LEO-II [8], is a higher-order ATP system based on ordered higher-order paramodulation employing an agent-based blackboard architecture. In its first version, Leo-III is using multiple, adapted sequential DISCOUNT loops, each with different search strategies. In addition, similar to LEO-II, each sequential loop will call non-blockingly an external ATP every n iterations of the sequential loop. In the current version, the called ATPs have to understand THF syntax and return the result in standard SZS format. In the competition mode only our own prover LEO-II will be used as a cooperation prover. If either one of the paramodulation loops or one of the external provers finds a proof, the system stops and returns the result. Leo+III is Leo-III 'plus' Satallax as a subprover.

Strategies

Leo-III runs multiple search strategies in parallel. These strategies containing some incomplete versions, that are outperforming the complete versions for some problem inputs. The search also differs in the employed relevance filters, preprocessing techniques and hence the considered formula set.

Ultimately, Leo-III is in its first version an enhancement of LEO-II. The main improvement in comparison to its predecessor is the better equational handling with the new calculus, and the multi-search of the agent architecture.

Implementation

Leo-III is implemented in Scala. Its natural problem representation is the TPTP THF language [9], but it can process every language of the TPTP including TFF and FOF. Leo-III is available from:

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

Expected Competition Performance

This version is only for demonstrative purposes. For the use of Satallax a different set of preprocessing techniques is used, but this version should at least be as competitive as Leo-III in the competition.

7.12 Nitpick 2015

Jasmin Blanchette
Inria Nancy, France

Architecture

Nitpick [17] is an open source counterexample generator for Isabelle/HOL [47]. It builds on Kodkod [117], a highly optimized first-order relational model finder based on SAT. The name Nitpick is appropriated from a now retired Alloy precursor. In a case study, it was applied successfully to a formalization of the C++ memory model [16].

Strategies

Nitpick employs Kodkod to find a finite model of the negated conjecture. The translation from HOL to Kodkod's first-order relational logic (FORL) is parameterized by the cardinalities of the atomic types occurring in it. Nitpick enumerates the possible cardinalities for each atomic type, exploiting monotonicity to prune the search space [14]. If a formula has a finite counterexample, the tool eventually finds it, unless it runs out of resources.

SAT solvers are particularly sensitive to the encoding of problems, so special care is needed when translating HOL formulas. As a rule, HOL scalars are mapped to FORL singletons and functions are mapped to FORL relations accompanied by a constraint. More specifically, an n-ary first-order function (curried or not) can be coded as an $(n + 1)$-ary relation accompanied by a constraint. However, if the return type is the type of Booleans, the function is more efficiently coded as an unconstrained n-ary relation. Higher-order quantification and functions bring complications of their own. A function from *sigma* to *tau* cannot be directly passed as an argument in FORL; Nitpick's workaround is to pass —*sigma*— arguments of type *tau* that encode a function table.

Implementation

Nitpick, like most of Isabelle/HOL, is written in Standard ML. Unlike Isabelle itself, which adheres to the LCF small-kernel discipline, Nitpick does not certify its results and must be trusted.

Nitpick is available as part of Isabelle/HOL for all major platforms under a BSD-style license from

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

Expected Competition Performance

Thanks to Kodkod's amazing power, we expect that Nitpick will beat both Satallax and Refute with its hands tied behind its back.

7.13 Princess 160606

Philipp Rümmer
Uppsala University, Sweden

Architecture

Princess [64, 65] is a theorem prover for first-order logic modulo linear integer arithmetic. The prover uses a combination of techniques from the areas of first-order reasoning and SMT solving. The main underlying calculus is a free-variable tableau calculus, which is extended with constraints to enable backtracking-free proof expansion, and positive unit hyper-resolution for lightweight instantiation of quantified formulae. Linear integer arithmetic is handled using a set of built-in proof rules resembling the Omega test, which altogether yields a calculus that is complete for full Presburger arithmetic, for first-order logic, and for a number of further fragments. In addition, some built-in procedures for nonlinear integer arithmetic are available.

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

Strategies

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

Implementation

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

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

Expected Competition Performance

Princess should perform roughly as in 2015. Compared to last year, initial support for outputting proofs was added, though not for all relevant configurations yet.

7.14 Refute 2015

Jasmin Blanchette
Inria Nancy, France

Architecture

Refute [119] is an open source counterexample generator for Isabelle/HOL [47] based on a SAT solver, and Nitpick's [17] precursor.

Strategies

Refute employs a SAT solver to find a finite model of the negated conjecture. The translation from HOL to propositional logic is parameterized by the cardinalities of the atomic types occurring in the conjecture. Refute enumerates the possible cardinalities for each atomic type. If a formula has a finite counterexample, the tool eventually finds it, unless it runs out of resources.

Implementation

Refute, like most of Isabelle/HOL, is written in Standard ML. Unlike Isabelle itself, which adheres to the LCF small-kernel discipline, Refute does not certify its results and must be trusted.

Refute is available as part of Isabelle/HOL for all major platforms under a BSD-style license from

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

Expected Competition Performance

We expect Refute to beat Satallax but also to be beaten by Nitpick.

7.15 Satallax 2.8

Nik Sultana
Cambridge University, United Kingdom

Architecture

Satallax 2.8 [18] 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 [25] is responsible for much of the search for a proof. The theoretical basis of search is a complete ground tableau calculus for higher-order logic [20] with a choice operator [1]. A problem is given in the THF format. A branch is formed from the axioms of the problem and the negation of the conjecture (if any is given). From this point on, Satallax tries to determine unsatisfiability or satisfiability of this branch. Satallax progressively generates higher-order formulae and corresponding propositional clauses [19]. These formulae and propositional clauses correspond to instances of the tableau rules. Satallax uses the SAT solver MiniSat as an engine to test the current set of propositional clauses for unsatisfiability. If the clauses are unsatisfiable, then the original branch is unsatisfiable. Additionally, Satallax may optionally generate first-order formulas in addition to the propositional clauses. If this option is used, then Satallax peroidically calls the first-order theorem prover E to test for first-order unsatisfiability. If the set of first-order formulas is unsatisfiable, then the original branch is unsatisfiable.

Strategies

There are about a hundred flags that control the order in which formulas and instantiation terms are considered and propositional clauses are generated. Other flags activate some optional extensions to the basic proof procedure (such as whether or not to call the theorem prover E). A collection of flag settings is called a mode. Approximately 500 modes have been defined and tested so far. A strategy schedule is an ordered collection of modes with information about how much time the mode should be allotted. Satallax tries each of the modes for a certain amount of time sequentially. Satallax 2.7 has strategy schedule consisting of 68 modes. Each mode is tried for time limits ranging from 0.1 seconds to 54.9 seconds. The strategy schedule was determined through experimentation using the THF problems in version 5.4.0 of the TPTP library.

Implementation

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

```
http://mathgate.info/cebrown/satallax/
```

Expected Competition Performance

Satallax 2.8 is the CASC-25 THF division winner.

7.16 Satallax 3.0

Michael Färber
Universität Innsbruck, Austria

Architecture

Satallax 3.0 [18] 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 [25] is responsible for much of the proof search. The theoretical basis of search is a complete ground tableau calculus for higher-order logic [20] with a choice operator [1]. Problems are given in the THF format.

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

Strategies

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

Implementation

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

```
http://satallaxprover.com
```

Expected Competition Performance

Since 2015, systems are required to return TSTP proofs. Previous versions of Satallax could only construct such proofs if E was not used in the search. Satallax 3.0 can construct a proof when using E. Since some problems are (effectively) only solvable when using E, this should improve performance over last year. In addition, some support for guiding the search using interpretations has been implemented. This is also expected to improve performance.

7.17 Vampire 4.0

Giles Reger
University of Manchester, United Kingdom

Architecture

Vampire 4.0 is an automatic theorem prover for first-order logic. Vampire implements the calculi of ordered binary resolution and superposition for handling equality. It also implements the Inst-gen calculus and a MACE-style finite model builder. Splitting in resolution-based proof search is controlled by the AVATAR architecture, which uses a SAT solver to make splitting decisions. Both resolution and instantiation based proof search make use of global subsumption.

A number of standard redundancy criteria and simplification techniques are used for pruning the search space: subsumption, tautology deletion, subsumption resolution and rewriting by ordered unit equalities. The reduction ordering is the Knuth-Bendix Ordering. Substitution tree and code tree indexes are used to implement all major operations on sets of terms, literals and clauses. Internally, Vampire works only with clausal normal form. Problems in the full first-order logic syntax are clausified during preprocessing. Vampire implements many useful preprocessing transformations including the Sine axiom selection algorithm.

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

Strategies

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

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

Implementation

Vampire 4.0 is implemented in C++.

Expected Competition Performance

Vampire 4.0 is the CASC-25 FOF, FNT, EPR, and LTB division winner.

7.18 Vampire 4.1

Giles Reger
University of Manchester, United Kingdom

Architecture

Vampire [39] 4.1 is an automatic theorem prover for first-order logic. Vampire implements the calculi of ordered binary resolution and superposition for handling equality. It also implements the Inst-gen calculus [36] and a MACE-style finite model builder [56]. Splitting in resolution-based proof search is controlled by the AVATAR architecture [118] which uses a SAT or SMT solver to make splitting decisions. Both resolution and instantiation based proof search make use of global subsumption [36].

A number of standard redundancy criteria and simplification techniques are used for pruning the search space: subsumption, tautology deletion, subsumption resolution and rewriting by ordered unit equalities. The reduction ordering is the Knuth-Bendix Ordering. Substitution tree and code tree indexes are used to implement all major operations on sets of terms, literals and clauses. Internally, Vampire works only with clausal normal form. Problems in the full first-order logic syntax are clausified during preprocessing. Vampire implements many useful preprocessing transformations including the SinE axiom selection algorithm.

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

Strategies

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

- Choices of saturation algorithm:
  - Limited Resource Strategy [63].
  - DISCOUNT loop
  - Otter loop
  - Instantiation using the Inst-Gen calculus
  - MACE-style finite model building with sort inference
- Splitting via AVATAR
- A variety of optional simplifications.
- Parameterized reduction orderings.
- A number of built-in literal selection functions and different modes of comparing literals.
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection.
- Set-of-support strategy.
- Ground equational reasoning via congruence closure.

- Addition of theory axioms and evaluation of interpreted functions.
- Use of Z3 [22] with AVATAR to restrict search to ground-theory-consistent splitting branches.
- Extensionality resolution [29] with detection of extensionality axioms.

Implementation

Vampire 4.1 is implemented in C++.

Expected Competition Performance

Vampire 4.1 should be an improvement on Vampire 4.0 and VampireZ3 1.0, which won 5 divisions between them last year. Note that this year there is not a seperate VampireZ3 entry as Vampire 4.1 includes Z3.

7.19 VampireZ3 1.0

Giles Reger
University of Manchester, United Kingdom

Architecture

VampireZ3 version 1.0 is a combination of Vampire 4.0 and Z3 4.3.1. Vampire 4.0 uses the AVATAR architecture to make splitting decisions. Briefly, the first-order search space is represented in the SAT solver with propositional symbols consistently naming variable-disjoint components. A SAT solver is then used to (iteratively) select a subset of components to search. In VampireZ3 the Z3 SMT solver is used in place of a SAT solver and *ground* components are translated into Z3 terms. This means Z3's efficient methods for ground reasoning with equality and theories are exposed by AVATAR, as the SMT solver only produces theory-consistent models.

Strategies

All strategies of Vampire 4.0 are available. Z3 is only used when splitting is selected.

Implementation

Vampire and Z3 are both implemented in C++.

Expected Competition Performance

VampireZ3 1.0 is the CASC-25 TFA division winner.

8 Conclusion

The CADE-J8 ATP System Competition was the twenty-first large scale competition for classical logic ATP systems. The organizer believes that CASC fulfills its main motivations: stimulation of research, motivation for improving implementations, evaluation of relative capabilities of ATP systems, and providing an exciting event. Through the continuity of the event and consistency in the the reporting of the results, performance comparisons with previous and future years are easily possible. The competition provides exposure for system builders both within and outside of the community, and provides an overview of the implementation state of running, fully automatic, classical logic, ATP systems.

References

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

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

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

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

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

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

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

[14] J. Blanchette and A. Kraus. Monotonicity Inference for Higher-Order Formulas. *Journal of Automated Reasoning*, page To appear, 2011.

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

[16] J. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ Concurrency. In M. Hanus, editor, *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 113–124. ACM Press, 2011.

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

[18] 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.

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

[20] 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.

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

[22] L. de Moura and N. Bjorner. 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.

[23] H. de Nivelle. Theorem Proving for Classical Logic with Partial Functions by Reduction to Kleene Logic. *Journal of Logic and Computation*, page To appear, 2014.

[24] H. de Nivelle. Subsumption Algorithms for Three-Valued Geometric Resolution. In N. Olivetti and A. Tiwari, editors, *Proceedings of the 8th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, page To appear, 2016.

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

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

[27] H. Ganzinger and K. Korovin. Integrating Equational Reasoning into Instantiation-Based Theorem Proving. In J. Marcinkowski and A. Tarlecki, editors, *Proceedings of the 18th International Workshop on Computer Science Logic, 13th Annual Conference of the EACSL*, number 3210 in Lecture Notes in Computer Science, pages 71–84. Springer-Verlag, 2004.

[28] 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.

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

[30] T. Hales. A Revision of the Proof of the Kepler Conjecture. *Discrete and Computational Geometry*, 44(1):1–34, 2010.

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

[32] 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.

[33] Z. Khasidashvili and K. Korovin. Predicate Elimination for Preprocessing in First-order Theorem Proving. In N. Creignou and D. Le Berre, editors, *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science. Springer-Verlag, 2016.

[34] M. Kinyon, R. Veroff, and P. Vojtechovsky. Loops with Abelian Inner Mapping Groups: an Application of Automated Deduction. 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 151–164. Springer-Verlag, 2013.

[35] K. Korovin. iProver - An Instantiation-Based Theorem Prover for First-order Logic (System Description). In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 292–298, 2008.

[36] 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.

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

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

[39] 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.

[40] R. Kumar, M. Myreen, M. Norrish, and S. Owens. CakeML: A Verified Implementation of ML. In P Sewell, editor, *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191. ACM Press, 2014.

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

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

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

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

[45] 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.

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

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

[48] J. Otten. leanCoP 2.0 and ileancop 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic. In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 283–291, 2008.

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

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

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

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

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

[54] 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, pages 1–11, 2010.

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

[56] 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*, Lecture Notes in Computer Science. Springer-Verlag, 2016.

[57] A. Reynolds. *Finite Model Finding in Satisfiability Modulo Theories*. PhD thesis, The University of Iowa, Iowa City, USA, 2013.

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

[59] 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.

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

[61] 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.

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

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

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

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

[66] 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.

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

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

[69] 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.

[70] 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.

[71] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[88] 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.

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

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

[91] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

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

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

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

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

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

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

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

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

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

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

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

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

[104] 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.

[105] 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.

[106] 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.

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

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

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

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

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

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

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

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

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

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

[117] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 4424 in Lecture Notes in Computer Science, pages 632–647. Springer-Verlag, 2007.

[118] 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.

[119] T. Weber. *SAT-based Finite Model Generation for Higher-Order Logic*. PhD thesis, Technische Universität München, Munich, Germany, 2008.

[120] M. Wisniewski, A. Steen, and C. Benzmüller. The Leo-III Project. In A. Bolotov and M. Kerber, editors, *Proceedings of the Joint Automated Reasoning Workshop and Deduktionstreffen*, page 38, 2014.