CASC-24

CASC-24

CASC-24

CASC-24

Proceedings of the CADE-24 ATP System Competition CASC-24

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 specified time limits on solution attempts. The CADE-24 ATP System Competition (CASC-24) was held on 12th June 2013. The design of the competition and its rules, and information regarding the competing systems, are provided in this report.

1 Introduction

The CADE 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 conference. CASC-24 was held on 12th June 2013, as part of the 24th International Conference on Automated Deduction (CADE-24), in Lake Placid, USA. It is the eighteenth competition in the CASC series [123, 128, 126, 91, 93, 122, 120, 121, 98, 100, 102, 104, 107, 110, 112, 114, 116].

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 [108], and
- specified time limits on solution attempts.

Twenty-nine ATP system versions, listed in Table 1 entered into the various competition and demonstration divisions. The winners of the CASC-J6 (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). [1]

The design and procedures of this CASC evolved from those of previous CASCs [123, 124, 119, 125, 89, 90, 92, 94, 95, 96, 97, 99, 101, 103, 106, 109, 111, 113, 115]. Important changes for this CASC were:

[1] The CASC-J6 LTB winner, Vampire 2.6, was unable to run in the CASC-24 LTB division, due to changes in the batch specification files' format

ATP System	Divisions	Entrants (Associates)	Entrant's Affiliation
5alarm 0.1	EPR	Mark Lemay	-
Beagle 0.4	TFA	Peter Baumgartner (Joshua Bax, Andreas Bauer, Tim Cosgrove)	NICTA
cocATP 0.1.8	THF	Cristobal Camarero	University of Cantabria
CVC4 1.2	FOF FNT	Andrew Reynolds (Clark Barrett, Cesare Tinelli, Morgan Deters)	University of Iowa
E 1.8	FOF FNT EPR LTB	Stephan Schulz	Technische Universität München
E-KRHyper 1.4	FOF FNT EPR LTB	Björn Pelzer	University Koblenz-Landau
E-MaLeS 1.2	FOF	Daniel Kuehlwein (Stephan Schulz, Josef Urban)	Radboud University Nijmegen
iProver 0.9	EPR	CASC	CASC-J6 EPR winner
iProver 1.0	FOF FNT EPR LTB	Konstantin Korovin	University of Manchester
iProver-Eq 0.85	FOF FNT EPR	Christoph Sticksel (Konstantin Korovin)	University of Manchester
iProverModulo 0.7-0.2	FOF	Guillaume Burel	ENSIIE/Cedric
Isabelle 2012	THF	CASC	CASC-J6 THF winner
Isabelle 2013	THF	Jasmin Blanchette (Lawrence Paulson, Tobias Nipkow, Makarius Wenzel)	Technische Universität München
LEO-II 1.6.0	THF FOF	Christoph Benzmüller	Freie Universität Berlin
MaLARea 0.5	LTB	Josef Urban Stephan Schulz, Jiri Vyskocil)	Radboud University Nijmegen
Muscadet 4.3	FOF	Dominique Pastre	University Paris Descartes
Nitrox 2013	FNT	Jasmin Blanchette (Emina Torlak)	Technische Universität München
Paradox 3.0	FNT	CASC	CASC-J6 FNT winner
PEPR 0.0ps	EPR	Tianyi Liang (Cesare Tinelli)	University of Iowa
Princess 2012-06-04	TFA	CASC	CASC-J6 TFA winner
Prover9 2009-11A	FOF	CASC (William McCune, Bob Veroff)	CASC fixed point
Satallax 2.7	THF	Chad Brown	Saarland University
Satallax-MaLeS 1.2	THF	Daniel Kuehlwein (Chad Brown, Josef Urban)	Radboud University Nijmegen
SPASS+T 2.2.19	TFA	Uwe Waldmann	Max-Planck-Institut für Informatik
TEMPLAR::leanCoP 0.8	LTB	Mario Frank (Jens Otten)	University of Potsdam
TPS 3.120601S1b	THF	Chad E. Brown (Peter Andrews)	Saarland University
Vampire 2.6	FOF	CASC	CASC-J6 FOF winner
Vampire 3.0	FOF FNT EPR LTB	Andrei Voronkov (Laura Kovacs)	University of Manchester
Zipperposition 0.2	FOF	Guillaume Burel (Simon Cruanes)	ENSIIE/Cedric

Table 1: The ATP systems and entrants

- CNF problems were used in only the EPR division (because CNF is now the assembly language of ATP).
- The FOF, FNT, and LTB divisions no longer had an assurance ranking class. They had only proof/model ranking classes.
- The LTB division's problem categories were accompanied by sets of training problems and their solutions (taken from the same exports as the competition problems), that could be used for tuning and training during (typically at the start of) the competition.
- There were some minor changes to the Batch Specification Files.
- Systems were expected to use the SZS ontology and standards for reporting their results.
- Systems were expected to use the proposed TPTP ATP System Building Conventions for their installation.
- Execution was monitored by Oliver Roussel's RunSolver.

The competition organizer was Geoff Sutcliffe. The competition was overseen by a panel of knowledgeable researchers who were not participating in the event; the CASC-24 panel members were Armin Biere, Maria Paola Bonacina, and Christoph Weidenbach. 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 the Department of Computer Science, University of Manchester, United Kingdom. The CASC-24 web site provides access to resources used before, during, and after the event: `http://www.tptp.org/CASC/24`

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

2 Divisions

CASC is run in 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 formally ranked. Some divisions are further divided into problem categories, which make 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 competition 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 non-propositional theorems (axioms with a provable conjecture), using the THF0 syntax. 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, using the TFF0 syntax. The TFA division has two problem categories:

- The **TFI** category: TFA with only Integer arithmetic
- The **TFR** category: TFA with only Rational and only Real arithmetic (no mixed rational and real arithmetic)

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

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

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

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

The **EPR** division: Effectively PRopositional clause normal form theorems and non-theorems (clause sets). *Effectively propositional* means non-propositional with a finite Herbrand Universe. 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 non-propositional theorems (axioms with a provable conjecture) from Large Theories, presented in Batches. 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 **HOL** category: Problems exported from HOL Light.
- The **ISA** category: Problems exported from Isabelle.
- The **MZR** category: Problems exported from the Mizar Mathematical Library (MML).

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

2.2 The Demonstration Division

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

3 Infrastructure

3.1 Computers

The computers had

- Four Intel(R) Xeon(R) L5410, 2.333GHz CPUs
- 12GB memory
- Linux 2.6.29.4-167.fc11.x86_64 operating system

Each ATP system ran one job on one computer at a time.

3.2 Problems

3.2.1 Problem Selection

The problems were taken from the TPTP problem library, version v5.5.0. Additionally, for the LTB division problems were taken from publicly available problem sets: the HOL problem category used the HH7150 problem set[2]; the ISA problem category used the SH-CASC-14 problem set[3]; the MZR problem category used the the MPTP2078 problem set[4]. The TPTP version used for CASC is released after the competition has started, so that new problems have not seen by the entrants. Access to and use of the non-TPTP problem sets was controlled to ensure that the system complied with the CASC tuning restrictions, described in Section 6.1.

The problems have to meet certain criteria to be eligible for selection:

- The TPTP uses system performance data to compute problem difficulty ratings [127], and from the ratings classifies problems as one of:
  - Easy: Solvable by all state-of-the-art ATP systems
  - Difficult: Solvable by some state-of-the-art ATP systems
  - Unsolved: Solvable by no ATP systems
  - Open: Theoremhood unknown

  Difficult problems with a rating in the range 0.21 to 0.99 are eligible. Problems of lesser and greater ratings might also be eligible in some divisions (especially batch divisions, because the TPTP problem ratings are computed from non-batch mode results). 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 TPTP distinguishes versions of problems as one of standard, incomplete, augmented, especial, or biased. All except biased problems are eligible.
- In batch divisions, the problems are selected so that there is consistent symbol usage between problems in each category, but there may not be consistent axiom naming between problems.

The problems used are randomly selected from the eligible problems at the start of the competition, based on a seed supplied by the competition panel.

- The selection is constrained so that no division or category contains an excessive number of very similar problems.

[2]http://mizar.cs.ualberta.ca/~mptp/hh1/HH7150.tar.gz
[3]http://www21.in.tum.de/~blanchet/SH-CASC-24.tgz
[4]http://wiki.mizar.org/twiki/bin/view/Mizar/MpTP2078

- 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.

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 [43] (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 than can be used in the categories, 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 (included files are found relative to the `TPTP` environment variable). The problems in each non-batch division, and each LTB batch, are given in increasing order of TPTP difficulty rating. This is aesthetic in non-batch divisions, but practically important in the batches where it is possible to learn from proofs found earlier in the batch.

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

In order to prevent systems from recognizing problems from their file names, symbolic links are made to the selected problems, using names of the form `CCCNNN-1.p` for the symbolic links. `CCC` is the division or problem category name, and `NNN` runs from `001` to the number of problems in the respective division or problem category. The problems are specified to the ATP systems using the symbolic link names.

In the demonstration division the same problems are used as for the competition divisions, with the same preprocessing applied. However, the original fille names can be retained for systems running on computers provided by the entrant.

3.2.4 Batch Specification Files

The problems for each batch division and category 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 wre `HOL`, `ISA`, `MZR`.
- 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 `Axioms` directory in the training data contains all the axiom files that can be used in the competition problems.

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 value of zero indicates no per-problem limit.
- 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-Batch 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 of 10GB was imposed. The time limits 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 batch divisions, 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 competitions division, from the performance data. The THF, TFA, and EPR, divisions have an *assurance* ranking class, ranked according to the number of problems solved, but not necessarily accompanied by a proof or model (thus giving only an assurance of the existence of a proof/model). The FOF, FNT, and LTB divisions have a *proof/model* ranking class, ranked according to the number of problems solved with an acceptable proof/model output. Ties are broken according to the average time over problems solved (CPU time for non-batch divisions, wall clock time for batch divisions). In the competition divisions, class winners were announced and prizes are awarded.

The competition panel decides whether or not the systems' proofs and models are acceptable for the proof/model ranking classes. 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 the assurance ranking classes the ATP systems are not required to output solutions (proofs or models). However, systems that do output solutions are highlighted in the presentation of results.

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 *efficiency measure* balances the number of problems solved with the CPU time taken. It is the average of the inverses of the times for problems solved (CPU times for non-batch divisions, wall clock times for batch divisions, with times less than the timing granularity rounded up to the granularity, to avoid skewing caused by very low times) 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.
- In divisions that use a wall clock time limit 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.

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 the proof/model ranking classes 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.

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 system is described 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, along with information regarding the competition design and procedures, form the proceedings for the competition.

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 for the proof/model ranking classes.

Proof samples for the FOF proof class must include a proof for `SEU140+2`. Model samples for the FNT model class must include models for `NLP042+1` and `SWV017+1`. The sample solutions must illustrate the use of all inference rules. 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 a competition-like environment, and have the following properties. Entrants are advised to check these properties well in advance of the system delivery deadline. This gives the competition organizers time to help resolve any difficulties encountered. Entrants do not have access to the competition computers.

6.1.1 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 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. The soundness testing eliminates the possibility of a system simply delaying for some amount of time and then claiming to have found a solution. 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 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 training problems are not used in the competition. There are at least twice as many training problems as competition problems in each problem category. The training problems and solutions may be used for producing generally useful strategies that extend to "unseen"" 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.
  - 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.

6.1.2 Execution

- Systems must run on a single locally provided standard UNIX computer (the *competition computers* - see Section 3.1). ATP systems that cannot run on the competition computers can be entered into the demonstration division.

- Systems must be executable by a single command line, using an absolute path name for the executable, which might not be in the current directory. In non-batch divisions the command line arguments are the absolute path name of a symbolic link as the problem file name, the time limit (if required by the entrant), and entrant specified system switches. In batch divisions the command line arguments are the absolute path name of the batch specification file, the overall category time limit (if required by the entrant), and entrant specified system switches. No shell features, such as input or output redirection, may be used in the command line. No assumptions may be made about the format of file names.
- Systems must be fully automatic, i.e., all command line switches have to be the same for all problems in each division.

6.1.3 Output

- In non-batch divisions all solution output must be to `stdout`. In batch divisions all solution output must be to the named output file for each problem.
- In batch divisions 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). It is recommended that the result for the problem be output as the last thing before the ending notification line (note, the result must also be output to the solution file anyway). 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 systems must output a distinguished string (specified by the entrant), indicating what solution has been found or that no conclusion has been reached. Systems are expected to use the SZS ontology and standards [105] for this. For example

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

  or

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

  The distinguished strings must be different for:
  - Proved theorems of FOF problems (SZS status `Theorem`)
  - Disproved conjectures of FNT problems (SZS status `CounterSatisfiable`)
  - Unsatisfiable sets of formulae (FOF problems without conjectures) and unsatisfiable set of clauses (CNF problems) (SZS status `Unsatisfiable`)
  - Satisfiable sets of formulae (FNT problems without conjectures) (SZS status `Satisfiable`)

  The first distinguished string output is accepted as the system's result.
- When outputting proofs/models, the start and end of the proof/model must be delimited by distinguished strings (specified by the entrant). Systems are expected to 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 distinguished strings must be different for:

- Proofs (SZS output forms `Proof`, `Refutation`, `CNFRefutation`)
- Models (SZS output forms `Model`, `FiniteModel`, `InfiniteModel`, `Saturation`)

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 [118].

6.1.4 Resource Usage

- The 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 an ATP system terminates of its own accord, it may not leave any temporary or intermediate output files. If an ATP system is terminated by a `SIGXCPU` or `SIGALRM`, it may not leave any temporary or intermediate files anywhere other than in `/tmp`. Multiple copies of the ATP systems must be executable concurrently, in the same (NFS cross mounted) directory. It is therefore necessary that temporary files have unique names.
- 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. The limit is at least 10MB per system.

6.2 System Delivery

For systems running on the competition computers, entrants must email an installation package to the competition organizers by the system delivery deadline. The installation package must be a `.tgz` file containing the system source code, any other files required for installation, and a `ReadMe` file. The `ReadMe` file must contain:

- Instructions for installation Systems are expected to use the proposed TPTP ATP System Building Conventions[5] for their installation.
- Instructions for executing the system, using `%s` and `%d` to indicate where the problem file name and time limit must appear in the command line.
- The distinguished strings indicating what solution has been found, and delimiting proofs/models.

For systems that do not use the proposed TPTP ATP System Building Conventions, the installation procedure may require changing path variables, invoking `make` or something similar, etc, but nothing unreasonably complicated. All system binaries must be created in the installation process; they cannot be delivered as part of the installation package. If the ATP system requires any special software, libraries, etc, which is not part of a standard installation, the competition organizers must be told in the system registration. The system is installed onto the competition computers by the competition organizers, following the instructions in the `ReadMe` file. Installation failures before the system delivery deadline are passed back to the entrant. (i.e., delivery of the installation package before the system delivery deadline provides

[5] `http://www.tptp.org/TPTP/Proposals/SystemBuild.html`

an opportunity to fix things if the installation fails!). After the system delivery deadline no further changes or late systems are accepted.

For systems running on entrant supplied computers in the demonstration division, entrants must deliver a source code package to the competition organizers by the start of the competition. The source code package must be a `.tgz` file containing the system source code.

After the competition all competition division systems' source code is made publically 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 on the competition computers is controlled by a `perl` script, provided by the competition organizers. The jobs are queued onto the computers so that each computer is running one job at a time. In non-batch divisions, all attempts at the Nth problems in all the divisions and categories are started before any attempts at the (N+1)th problems. In batch divisions all attempts in each category in the division are started before any attempts at the next category.

During the competition a `perl` script parses the systems' outputs. If any of an ATP system's distinguished strings are found then the time used to that point is noted. 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 0.4

Peter Baumgartner
NICTA, Australia

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

A first-order prover, beagle accepts arbitrarily quantified input formulas and converts them to clause normal form. The core reasoning component implements the Hierarchic Superposition Calculus with Weak Abstraction [14]. That calculus generalizes the well-known superposition calculus by integrating black-box reasoning for specific theories. For the theories mentioned above, beagle combines quantifier elimination procedures and other solvers to discard proof obligations over these theories.

Strategies
Beagles uses the well-know Discount loop for saturating a clause set under the calculus' inference rules. Simplification techniques include standard ones, such as subsumption deletion, rewriting by ordered unit equations, 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.

Beagle features a splitting rule for clauses that can be divided into variable disjoint parts. Splitting is particularly useful in combination with the quantifier elimination procedure mentioned above, as the theory reasoner then need to deal with conjunctions of quantifier-free unit clauses only.

Beagle uses a single, uniform strategy for every problem.

Implementation
Beagle has been implemented in Scala. It is a full implementation of the calculus mentioned above, albeit a slow one. Fairness is achieved through a combination of measuring clause lengths, depths and their derivation-age.

Beagle's web site is

```
http://users.cecs.anu.edu.au/~baumgart/systems/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 cocATP 0.1.8

Cristobal Camarero
University of Cantabria, Spain

Architecture
cocATP is a Coq-inspired [1] automated theorem prover made on the free time of the author. It implements (the non-inductive) part of Coq's logic (calculus of constructions) and syntax. The proof terms it creates are accepted as proofs by Coq by the addition of a few definitions (substituting the respective inductive definitions of Coq). As in Coq, equality and logical connectives other than implication (which is a dependent product of the logic) are defined adding the proper axioms. The reasoning is tried to be done the more general possible, avoiding to give any special treatment to both equality and logical connectives.

cocATP is currently very underdeveloped. Recently the axiom of choice and excluded middle have been added, but the latter is not actually exploited. The functional extensionality axiom have some records of appropriate usage. The propositional extensionality axiom is not included yet.

In addition to axioms, cocATP includes a little library of basic lemmas, most of which with the proof that cocATP has constructed.

Strategies
Makes a search tree, doing steps trying to be some similar to human reasoning (or more closely a human guessing at Coq's interface). Some pattern-matching is done to to apply hypothesis. Implements existential variables (similar to Coq's tactics evar and eapply).

Implementation
cocATP is fully implemented in Python-2.7. It uses the Ply-3.4 library to build the parsers for both the Coq and TPTP syntaxes. Includes a type-verifier of Calculus of Constructions

without induction constructions, which must be defined with axioms. That is, a buggy partial clone of Coq. There is support for most of the TPTP syntax, problems are translated to a set of calculus of constructions terms. cocATP has been specially prepared for THF, but all the other TPTP formulae without numbers should be accepted. However cocATP does NOT include a SAT solver, thus it will probably not solve your trivial cnf problems.

Expected Competition Performance
Belong to the THF division. Based on local tests with the 2012 battery it will perform worse than TPS.

7.3 CVC4 1.2

Andrew Reynolds
University of Iowa, USA

Architecture
CVC4 [8] is an SMT solver based on the DPLL(T) architecture [64] that includes built-in support for many theories including linear arithmetic, arrays, bit vectors and datatypes. Additionally, it supports universally quantified formulas. When quantified formulas are present, CVC4 typically uses E-matching for answering unsatisfiable, and a finite model finding method for answering satisfiable.

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.

The finite model finding has been developed to target problems containing background theories, whose quantification is limited to finite and uninterpreted sorts. When finite model finding is turned on, CVC4 uses a ground theory of finite cardinality constraints that minimizes the number of ground equivalence classes, as described in [81]. When the problem is satisfiable at the ground level, a candidate model is constructed that contains complete interpretations for all predicate and function symbols. Quantifier instantiation strategies are then invoked to add instances of quantified formulas that are in conflict with the candidate model, as described in [82]. If no instances are added, then the solver reports "satisfiable".

Strategies
For handling theorems, CVC4 uses a variation of E-matching. Quantifier instantiation based on E-matching is performed lazily in CVC4, after the ground solver reports satisfiable.

For handling non-theorems, the finite model finding feature of CVC4 will use a small number of orthogonal quantifier instantiation strategies. Since CVC4 with finite model finding is also capable of answering unsatisfiable, it will be used as a strategy for theorems as well.

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

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

Expected Competition Performance
This is the first year CVC4 has entered the FOF division, where it will likely finish around the middle. The finite model finding feature in CVC4 has undergone a lot of development, so it should perform better than last year.

7.4 E 1.8

Stephan Schulz
Technische Universität München, Germany

Architecture
E 1.8pre [86] is a purely equational theorem prover for full 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.

Strategies
Proof search in E is primarily controlled by a literal selection strategy, a clause evaluation heuristic, and a simplification ordering. The prover supports a large number of pre-programmed literal selection strategies. Clause evaluation 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-24, E implements a new strategy-scheduling automatic mode. The total CPU time available is broken into 5 (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 170 different strategies have been evaluated on all untyped first-order problems from TPTP 5.4.0, and about 140 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 [87]. Superposition and backward rewriting use fingerprint indexing [88], 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 [58, 57]. The prover and additional information are available from

```
http://www.eprover.org
```

Expected Competition Performance
E 1.8 has slightly better strategies than previous versions, and now can produce proof objects internally and much more efficiently. The system is expected to perform well in most proof classes, but will at best complement top systems in the disproof classes.

7.5 E-KRHyper 1.4

Björn Pelzer
University Koblenz-Landau, Germany

Architecture
E-KRHyper [78] is a theorem proving and model generation system for first-order logic with equality. It is an implementation of the E-hyper tableau calculus [11], which integrates a superposition-based handling of equality [6] into the hyper tableau calculus [10]. The system is an extension of the KRHyper theorem prover [134], which implements the original hyper tableau calculus.

An E-hyper tableau is a tree whose nodes are labeled with clauses and which is built up by the application of the inference rules of the E-hyper tableau calculus. The calculus rules are designed such that most of the reasoning is performed using positive unit clauses. Splitting is done without rigid variables. Instead, variables which would be shared between branches are prevented by ground substitutions, which are guessed from the Herbrand universe and constrained by rewrite rules. Redundancy rules allow the detection and removal of clauses that are redundant with respect to a branch. The hyper extension inference from the original hyper tableau calculus is equivalent to a series of E-hyper tableau calculus inference applications. Therefore the implementation of the hyper extension in KRHyper by a variant of semi-naive evaluation [130] is retained in E-KRHyper, where it serves as a shortcut inference for the resolution of non-equational literals.

Strategies
E-KRHyper has traditionally used a uniform search strategy for all problems, based on the aforementioned semi-naive evaluation. As of version 1.4 the prover also contains a prototypical implementation of the common Given-Clause-Algorithm (“Otter-loop”) [62], which may be used as an alternative on problems of a certain size. The E-hyper tableau is generated depth-first, with E-KRHyper always working on a single branch. Refutational completeness and a fair search control are ensured by an iterative deepening strategy with a limit on the maximum term weight of generated clauses. In the LTB division E-KRHyper sequentially tries three axiom selection strategies: an implementation of Krystof Hoder’s SInE algorithm, another incomplete selection based on the CNF representations of the axioms, and finally the complete axiom set.

Implementation

E-KRHyper is implemented in the functional/imperative language OCaml. The system runs on Unix and MS-Windows platforms. the GNU Public License from the E-KRHyper website at: The system accepts input in the TPTP-format and in the TPTP-supported Protein-format. The calculus implemented by E-KRHyper works on clauses, so first order formula input is converted into CNF by an algorithm similar to the one used by Otter [62], with some additional connector literals to prevent explosive clause growth when dealing with DNF-like structures. E-KRHyper operates on an E-hyper tableau which is represented by linked node records. Several layered discrimination-tree based indexes (both perfect and non-perfect) provide access to the clauses in the tableau and support backtracking.

E-KRHyper is available from

```
http://www.uni-koblenz.de/~bpelzer/ekrhyper
```

Expected Competition Performance

There has been some streamlining regarding memory consumption, but overall we expect no big change over last year's performance.

7.6 E-MaLeS 1.2

Daniel Kühlwein[1], Josef Urban[1], Stephan Schulz[2]
[1]Radboud University Nijmegen, The Netherlands
[2]Technische Universität München, Germany

Architecture

E-MaLeS 1.2 is the result of applying the MaLeS 1.2 framework to E prover (version 1.7) [86].

MaLeS (Machine Learning of Strategies) is a framework for automatic tuning of ATPs. It combines strategy finding (similar to ParamILS [48] and BliStr [132]) with strategy scheduling (similar to SatZilla [135]). A description of E can be found above.

Strategies

Given a set of problems, MaLeS 1.2 finds good parameter settings by using a local random search algorithm. The found settings are stored as strategies. For each strategy, MaLeS learns a function that given a new problem predicts the time the ATP needs to solve this problem when using this particular strategy. When trying to solve a new problem, MaLeS uses these prediction functions to create a strategy schedule.

The learning algorithm is described in [55]. Further information can be found on the project website.

Implementation

MaLeS is implemented in python, using the numpy and scipy libraries. MaLeS is open source and available from

```
https://code.google.com/p/males/
```

E prover is available from

```
http://www.eprover.org
```

Expected Competition Performance
E-MaLeS 1.2 should perform better than E 1.7.

7.7 iProver 0.9

Konstantin Korovin
University of Manchester, United Kingdom

Architecture
iProver is an automated theorem prover based on an instantiation calculus Inst-Gen [40, 52] which is complete for first-order logic. One of the distinctive features of iProver is a modular combination of first-order reasoning with ground reasoning. In particular, iProver currently integrates MiniSat [38] for reasoning with ground abstractions of first-order clauses. In addition to instantiation, iProver implements ordered resolution calculus and a combination of instantiation and ordered resolution; see [51] for the implementation details. The saturation process is implemented as a modification of a 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 [41, 51]; global subsumption [51]; 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 with an option of using Brand's transformation. In the LTB division, iProver-SInE uses axiom selection based on the SInE algorithm [47] as implemented in Vampire [45], i.e., axiom selection is done by Vampire and proof attempts are done by iProver.
Major additions in the current version are:

- answer computation,
- several modes for model output using first-order definitions in term algebra,
- Brand's transformation.

Strategies
iProver has around 40 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.

Implementation
iProver is implemented in OCaml and for the ground reasoning uses MiniSat. iProver accepts FOF and CNF formats, where Vampire [45] is used for clausification of FOF problems.
iProver is available from

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

Expected Competition Performance
iProver 0.9 is the CASC-23 EPR division winner.

7.8 iProver 1.0

Konstantin Korovin, Christoph Sticksel
University of Manchester, United Kingdom

Architecture
iProver is an automated theorem prover based on an instantiation calculus Inst-Gen [40, 53] which is complete for first-order logic. iProver combines first-order reasoning with ground reasoning for which it uses MiniSat [38] and was recently extended with PicoSAT [20] and Lingeling [21] (only MiniSat will be used at this CASC). iProver also combines instantiation with ordered resolution; see [51] 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 [41, 51]; global subsumption [51]; 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 with an option of using Brand's transformation. In the LTB division, iProver uses axiom selection based on the SInE algorithm [47] as implemented in Vampire [46], 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,
- model representation, using first-order definitions in term algebra,
- answer substitutions,
- semantic filtering,
- type inference, monotonic [34] and non-cyclic types,
- Brand's transformation.

Type 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. The strategy for satisfiable problems (FNT division) includes finite model finding.

Implementation
iProver is implemented in OCaml and for the ground reasoning uses MiniSat [38]. iProver accepts FOF and CNF formats. Vampire [46, 44] is used for proof-producing clausification of FOF problems as well as for axiom selection [47] in the LTB division.

iProver is available from

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

Expected Competition Performance

Compared to the last year, iProver had a number of changes in internal datastructures affecting mainly extensibility of the code. We expect similar performance in the EPR division and good performance on satisfiable problems (FNT division).

7.9 iProver-Eq 0.85

Christoph Sticksel, Konstantin Korovin
University of Iowa, USA

Architecture

iProver-Eq [54] extends the iProver system [51] with built-in equational reasoning, along the lines of [41]. As in the iProver system, first-order reasoning is combined with ground satisfiability checking where the latter is delegated to an off-the-shelf ground solver.

iProver-Eq consists of three core components: i) ground reasoning by an SMT solver, ii) first-order equational reasoning on literals in a candidate model by a labelled unit superposition calculus [54, 54] and iii) instantiation of clauses with substitutions obtained by ii).

Given a set of first-order clauses, iProver-Eq first abstracts it to a set of ground clauses which are then passed to the ground solver. If the ground abstraction is unsatisfiable, then the set of first-order clauses is also unsatisfiable. Otherwise, literals are selected from the first-order clauses based on the model of the ground solver. The labelled unit superposition calculus checks whether selected literals are conflicting. If they are conflicting, then clauses are instantiated such that the ground solver has to refine its model in order to resolve the conflict. Otherwise, satisfiability of the initial first-order clause set is shown.

Clause selection and literal selection in the unit superposition calculus are implemented in separate given clause algorithms. Relevant substitutions are accumulated in labels during unit superposition inferences and then used to instantiate clauses. For redundancy elimination iProver-Eq uses demodulation, dismatching constraints and global subsumption. In order to efficiently propagate redundancy elimination from instantiation into unit superposition, we implemented different representations of labels based on sets, AND/OR-trees and OBDDs. Non-equational resolution and equational superposition inferences provide further simplifications.

Strategies

Proof search options in iProver-Eq control clause and literal selection in the respective given clause algorithms. Equally important is the global distribution of time between the inference engines and the ground solver. At CASC, iProver-Eq will execute a fixed schedule of selected options.

If no equational literals occur in the input, iProver-Eq falls back to the inference rules of iProver, otherwise the latter are disabled and only unit superposition is used. If all clauses are unit equations, no instances need to be generated and the calculus is run without the otherwise necessary bookkeeping.

Implementation

iProver-Eq is implemented in OCaml and the CVC4 SMT solver [8] for the ground reasoning in the equational case and MiniSat [38] in the non-equational case. iProver-Eq accepts FOF and CNF formats, where Vampire [46, 44] is used for clausification and preprocessing of FOF problems.

iProver-Eq is available from

```
http://www.divms.uiowa.edu/~csticksel/iprover-eq
```

Expected Competition Performance
iProver-Eq has seen some optimisations from the version in the previous CASCs, in particular the integration of CVC4. We expect reasonably good performance in all divisions, including the EPR divisions where instantiation-based methods are particularly strong.

7.10 iProverModulo 0.7-0.2

Guillaume Burel
ENSIIE/Cedric, France

Architecture
iProverModulo [33] is a patch to iProver [51] to integrate polarized resolution modulo [37]. Polarized resolution modulo consists in presenting the theory in which the problem has to be solved by means of polarized rewriting rules instead of axioms. It can also be seen as a combination of the set-of-support strategy and selection of literals.

The integration of polarized resolution modulo in iProver only affects its ordered resolution calculus, so that the instantiation calculus is untouched.

To be able to use polarized resolution modulo, the theory has to be presented as a rewriting system. A theory preprocessor is therefore used to convert the axioms of the problem into such a rewriting system.

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

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

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

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

iProverModulo is available from

```
http://www.ensiie.fr/~guillaume.burel/blackandwhite_iProverModulo.html.en
```

The theory preprocessor is available independently from iProverModulo from

```
http://www.ensiie.fr/~guillaume.burel/blackandwhite_autotheo.html.en
```

Both of them are written in OCaml.

Expected Competition Performance
Although iProver Modulo is based on iProver, we do not expect to perform better than the original iProver. This can be justified by the following arguments: First, iProver Modulo is based on a somewhat old version of iProver (v0.7). Second, the theory preprocessor may produce rewriting systems for which polarized resolution modulo is incomplete. Third, we had to switch off scheduling to cope with the constraints of polarized resolution modulo. Nevertheless, we expect to solve some problems that are difficult for other provers.

7.11 Isabelle 2012

Jasmin C. Blanchette[1], Lawrence C. Paulson[2], Tobias Nipkow[1], Makarius Wenzel[3]
[1]Technische Universität München, Germany, [2]University of Cambridge, United Kingdom, [3]Université; Paris Sud, France

Architecture
Isabelle/HOL 2012 [66] is the higher-order logic incarnation of the generic proof assistant Isabelle. Isabelle/HOL provides several automatic proof tactics, notably an equational reasoner [65], a classical reasoner [77], and a tableau prover [75]. It also integrates external first- and higher-order provers via its subsystem Sledgehammer [76, 24].

Previous versions of Isabelle relied on the TPTP2X tool to translate TPTP files to Isabelle theory files. Starting this year, Isabelle includes a parser for the TPTP syntaxes CNF, FOF, TFF0, and THF0 as well as TPTP versions of its popular tools, invokable on the command line as `isabelle tptp_`*`tool max_secs file.p`*. For example:

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

Two versions of Isabelle participate this year. The *demo* version includes its competitors LEO-II [17] and Satallax [30] 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 2.4 [30] (*demo only*);
  - `leo2` – LEO-II 1.3.2 [17] (*demo only*);
  - `spass` – SPASS 3.8ds [25];
  - `vampire` – Vampire 1.8 (revision 1435) [83];
  - `e` – E 1.4 [87];
  - `z3_tptp` – Z3 3.2 with TPTP syntax [36].
- `nitpick` – For problems involving only the type `$o` of Booleans, checks whether a finite model exists using Nitpick [27].

- `simp` – Performs equational reasoning using rewrite rules [65].
- `blast` – Searches for a proof using a fast untyped tableau prover and then attempts to reconstruct the proof using Isabelle tactics [75].
- `auto+spass` – Combines simplification and classical reasoning [77] under one roof; then invoke Sledgehammer with SPASS on any subgoals that emerge.
- `z3` – Invokes the SMT solver Z3 3.2 [36].
- `cvc3` – Invokes the SMT solver CVC3 2.2 [9].
- `fast` – Searches for a proof using sequent-style reasoning, performing a depth-first search [77]. 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 [59]. 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
Isabelle 2012 is the CASC-J6 THF division winner.

7.12 Isabelle 2013

Jasmin C. Blanchette[1], Lawrence C. Paulson[2], Tobias Nipkow[1], Makarius Wenzel[3]
[1]Technische Universität München, Germany, [2]University of Cambridge, United Kingdom, [3]Université Paris Sud, France

Architecture
Isabelle/HOL 2013 [66] is the higher-order logic incarnation of the generic proof assistant Isabelle. Isabelle/HOL provides several automatic proof tactics, notably an equational reasoner [65], a classical reasoner [77], and a tableau prover [75]. It also integrates external first- and higher-order provers via its subsystem Sledgehammer [76, 24].

Previous versions of Isabelle relied on the TPTP2X tool to translate TPTP files to Isabelle theory files. Isabelle includes a parser for the TPTP syntaxes CNF, FOF, TFF0, and THF0 as well as TPTP versions of its popular tools, invokable on the command line as `isabelle tptp_`*`tool max_secs file.p`*. For example:

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

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:
  - `spass` – SPASS 3.8ds [25];
  - `vampire` – Vampire 1.8 (revision 1435) [83];
  - `e` – E 1.4 [87];
  - `z3_tptp` – Z3 3.2 with TPTP syntax [36].
- `nitpick` – For problems involving only the type `$o` of Booleans, checks whether a finite model exists using Nitpick [27].
- `simp` – Performs equational reasoning using rewrite rules [65].
- `blast` – Searches for a proof using a fast untyped tableau prover and then attempts to reconstruct the proof using Isabelle tactics [75].
- `auto+spass` – Combines simplification and classical reasoning [77] under one roof; then invoke Sledgehammer with SPASS on any subgoals that emerge.
- `z3` – Invokes the SMT solver Z3 3.2 [36].
- `cvc3` – Invokes the SMT solver CVC3 2.2 [9].
- `fast` – Searches for a proof using sequent-style reasoning, performing a depth-first search [77]. 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 [59]. 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
Isabelle won last year by a thin margin. This year: first or second place.

7.13 LEO-II 1.6.0

Christoph Benzmüller
Freie Universität Berlin

Architecture
LEO-II [17], the successor of LEO [16], 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 [15]. 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 [86]. 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 [18]. In fact, the development of LEO-II has largely paralleled the development of the TPTP THF language and related infrastructure [117]. 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 [19] 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
Recent improvements of LEO-II include a revised ATP interface, new translations into first-order logic, rule support for the axiom of choice, detection of defined equality, modified extensional unification, and more flexible strategy scheduling. LEO-II can now also classify some Satisfiable problems and detect some CounterSatisfiable problems. However, these improvements are very likely not strong enough to attack Satallax or Isabelle in CASC.

7.14 MaLARea 0.5

Josef Urban[1], Stephan Schulz[2], Jiri Vyskocil[3]
[1]Radboud University Nijmegen, The Netherlands
[2]Technische Universität München, Germany
[3]Czech Technical University, Czech Republic

Architecture
MaLARea 0.5 [131, 133] is a metasystem for ATP in large theories where symbol and formula names are used consistently. It uses several deductive systems (now E, SPASS, Vampire, Paradox, Mace), as well as complementary AI techniques like machine learning (the SNoW system) based on symbol-based similarity, model-based similarity, term-based similarity, and obviously previous successful proofs. The version for CASC 2013 will mainly use E prover with the BliStr [132] large-theory strategies, possibly also Prover9, Mace and Paradox. The premise selection methods will likely also use the distance-weighted k-nearest neighbor [50] and E's implementation of SInE.

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

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

MaLARea is available from

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

The metasystem's Perl code is released under GPL2

Expected Competition Performance
Thanks to machine learning, MaLARea is strongest on batches of many related problems with many redundant axioms where some of the problems are easy to solve and can be used for learning the axiom relevance. MaLARea is not very good when all problems are too difficult (nothing to learn from), or the problems (are few and) have nothing in common. Some of its techniques (selection by symbol and term-based similarity, model-based reasoning) could however make it even there slightly stronger than standard ATPs. MaLARea has a very good performance on the MPTP Challenge, which is a predecessor of the LTB division, and it is the winner of the 2008 MZR LTB category. MaLARea 0.4 came second in the 2012 MZR@Turing competition and solved most problems in the Assurance class.

7.15 Muscadet 4.3

Dominique Pastre
University Paris Descartes, France

Architecture
The Muscadet theorem prover is a knowledge-based system. It is based on Natural Deduction, following the terminology of [26] and [69], and uses methods which resembles those used by humans. It is composed of an inference engine, which interprets and executes rules, and of one or several bases of facts, which are the internal representation of "theorems to be proved". Rules are either universal and put into the system, or built by the system itself by metarules from data (definitions and lemmas). Rules may add new hypotheses, modify the conclusion, create objects, split theorems into two or more subtheorems or build new rules which are local for a (sub-)theorem.

Strategies
There are specific strategies for existential, universal, conjonctive or disjunctive hypotheses and conclusions, and equalities. Functional symbols may be used, but an automatic creation of intermediate objects allows deep subformulae to be flattened and treated as if the concepts were defined by predicate symbols. The successive steps of a proof may be forward deduction (deduce new hypotheses from old ones), backward deduction (replace the conclusion by a new one), refutation (only if the conclusion is a negation), search for objects satisfying the conclusion or dynamic building of new rules.

The system is also able to work with second order statements. It may also receive knowledge and know-how for a specific domain from a human user; see [70] and [71]. These two possibilities are not used while working with the TPTP Library.

Implementation
Muscadet [72] is implemented in SWI-Prolog. Rules are written as more or less declarative Prolog clauses. Metarules are written as sets of Prolog clauses. The inference engine includes the Prolog interpreter and some procedural Prolog clauses. A theorem may be split into several subtheorems, structured as a tree with "and" and "or" nodes. All the proof search steps are memorized as facts including all the elements which will be necessary to extract later the useful steps (the name of the executed action or applied rule, the new facts added or rule dynamically built, the antecedents and a brief explanation).

Muscadet is available from

```
http://www.math-info.univ-paris5.fr/~pastre/muscadet/muscadet.html
```

Expected Competition Performance
The best performances of Muscadet will be for problems manipulating many concepts in which all statements (conjectures, definitions, axioms) are expressed in a manner similar to the practice of humans, especially of mathematicians [73, 74]. It will have poor performances for problems using few concepts but large and deep formulas leading to many splittings. Its best results will be in set theory, especially for functions and relations. Its originality is that proofs are given in natural style. Changes since last year are that some bugs have been fixed, also some imperfections in the writing of the proofs, and some heuristics have been improved which shorten a number of proofs.

7.16 Nitrox 2013

Jasmin C. Blanchette[1], Emina Torlak[2]
[1]Technische Universität München, Germany, [2]University of California, USA

Architecture
Nitrox is the first-order version of Nitpick [27], an open source counterexample generator for Isabelle/HOL [66]. It builds on Kodkod [129], a highly optimized first-order relational model finder based on SAT. The name Nitrox is a portmanteau of ***Nit***pick and Pa***r***ad***ox*** (clever, eh?).

Strategies
Nitrox employs Kodkod to find a finite model of the negated conjecture. It performs a few transformations on the input, such as pushing quantifiers inside, but 99solver.

The translation from HOL to Kodkod's first-order relational logic (FORL) is parameterized by the cardinalities of the atomic types occurring in it. Nitrox enumerates the possible cardinalities for the universe. If a formula has a finite counterexample, the tool eventually finds it, unless it runs out of resources.

Nitpick is optimized to work with higher-order logic (HOL) and its definitional principles (e.g., (co)inductive predicates, (co)inductive datatypes, (co)recursive functions). When invoked on untyped first-order problem, few of its optimizations come into play, and the problem handed to Kodkod is essentially a first-order relational logic (FORL) rendering of the TPTP FOF problem. There are two main exceptions:

- Nested quantifiers are moved as far inside the formula as possible before Kodkod gets a chance to look at them [27].
- Definitions invoked with fixed arguments are specialized.

Implementation
Nitrox, like most of Isabelle/HOL, is written in Standard ML. Unlike Isabelle itself, which adheres to the LCF small-kernel discipline, Nitrox does not certify its results and must be trusted. Kodkod is written in Java. MiniSat 1.14 is used as the SAT solver.

Expected Competition Performance
Since Nitpick was designed for HOL, it doesn't have any type inference àla Paradox. It also doesn't use the SAT solver incrementally, which penalizes it a bit (but not as much as the missing type inference). Kodkod itself is known to perform less well on FOF than Paradox, because it is designed and optimized for a somewhat different logic, FORL. On the other hand, Kodkod's symmetry breaking seems better calibrated than Paradox's. Hence, we expect Nitrox to end up in second or third place in the FNT division.

7.17 Paradox 3.0

Koen Claessen, Niklas Sörensson
Chalmers University of Technology, Sweden

Architecture
Paradox [35] is a finite-domain model generator. It is based on a MACE-style [61] flattening

and instantiating of the first-order clauses into propositional clauses, and then the use of a SAT solver to solve the resulting problem.

Paradox incorporates the following features: Polynomial-time *clause splitting heuristics*, the use of *incremental SAT*, *static symmetry reduction* techniques, and the use of *sort inference*.

Strategies
There is only one strategy in Paradox:

1. Analyze the problem, finding an upper bound N on the domain size of models, where N is possibly infinite. A finite such upper bound can be found, for example, for EPR problems.
2. Flatten the problem, and split clauses and simplify as much as possible.
3. Instantiate the problem for domain sizes 1 up to N, applying the SAT solver incrementally for each size. Report "SATISFIABLE" when a model is found.
4. When no model of sizes smaller or equal to N is found, report "CONTRADICTION".

In this way, Paradox can be used both as a model finder and as an EPR solver.

Implementation
The main part of Paradox is implemented in Haskell using the GHC compiler. Paradox also has a built-in incremental SAT solver which is written in C++. The two parts are linked together on the object level using Haskell's Foreign Function Interface.

Expected Competition Performance
Paradox 3.0 is the CASC-J6 FNT division winner.

7.18 PEPR 0.0ps

Tianyi Liang, Cesare Tinelli
University of Iowa, USA

Architecture
PEPR is a hybrid parallel automated theorem prover for the effectively propositional fragment of first-order logic (EPR). Its core is based on a variant of the Model Evolution calculus [12, 13], which is a decision procedure for the satisfiability of EPR formulas. During reasoning, the prover maintains a set of possibly non-ground literals, the context, to represent a candidate Herbrand model of the input clause set; it uses resolution-like operations between literals in the context and input clauses to identify instances of the latter that are falsified by the candidate model. These instances are then used to modify ("repair") the context non-deterministically in an attempt to satisfy them. If a context is not repairable, the input set has been proven to be unsatisfiable. A model is found when the context is saturated.

PEPR combines synergistically a portfolio approach, with multiple sub-provers that differ on the strategies they use to repair their context, and MapReduce-style clause-level parallelism for the computation of input clause instances [56]. The parallel architecture of PEPR is based on the Actor model with asynchronous message passing. To minimize communication all literals/clauses, including the input ones, are created locally in each sub-prover, which allows PEPR to run also in a distributed environment.

Because CASC measures performance in terms of total CPU time, the version of PEPR submitted to CASC-24 is actually sequential.

Strategies
In addition to the strategies implemented in Darwin, a theorem prover for the full ME calculus, PEPR implements conjecture distance, candidate coverage and predicate credits as literal selection heuristics, as well as thread scheduling for the parallel part.

Implementation
The sequential core of PEPR is fully implemented in C++, and it features new functionalities from the new C++11 standards. It requires either GCC 4.7 (or above) or MSVC 2012 to compile. To parse the TPTP syntax, it also requires the ANTLR 3.4 library. PEPR accepts the TPTP CNF format for EPR formula natively, and requires an external clausifier for the FOF format. The parallel part is built strictly upon the sequential core, and requires the Charm++ library to compile.

Although PEPR shows some promising results in its ability to exploit multiple cores, it still a prototype. Many configurations inside PEPR need to be further tuned.

Expected Competition Performance
PEPR will enter only the EPR division. Based on local tests with the CASC-23 benchmarks we expect it to rank in the middle range.

7.19 Princess 2012-06-04

Philipp Rümmer, Aleksandar Zeljic
Uppsala University, Sweden

Architecture
Princess [84, 85] is a theorem prover for first-order logic modulo linear integer arithmetic. The prover has been under development since 2007, and represents 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.

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
Princess supports a number of different proof expansion strategies (e.g., depth-first, breadth-first), which are chosen based on syntactic properties of a problem (in particular, the kind of quantifiers occurring in the problem). Further options exist to control, for instance, the selection of triggers in quantified formulae, clausification, and the handling of functions.

For CASC, Princess will run a schedule with a small number of configurations for each problem (portfolio method). The schedule is determined either statically, or dynamically using syntactic attributes of problems (such as number and kind of quantifiers, etc), based on training using 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 employed. Princess is available from

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

Expected Competition Performance
Princess 2012-06-04 is the CASC-J6 TFA division winner.

7.20 Prover9 2009-11A

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

Architecture
Prover9, Version 2009-11A, is a resolution/paramodulation prover for first-order logic with equality. Its overall architecture is very similar to that of Otter-3.3 [62]. It uses the "given clause algorithm", in which not-yet-given clauses are available for rewriting and for other inference operations (sometimes called the "Otter loop").

Prover9 has available positive ordered (and nonordered) resolution and paramodulation, negative ordered (and nonordered) resolution, factoring, positive and negative hyperresolution, UR-resolution, and demodulation (term rewriting). Terms can be ordered with LPO, RPO, or KBO. Selection of the "given clause" is by an age-weight ratio.

Proofs can be given at two levels of detail: (1) standard, in which each line of the proof is a stored clause with detailed justification, and (2) expanded, with a separate line for each operation. When FOF problems are input, proof of transformation to clauses is not given.

Completeness is not guaranteed, so termination does not indicate satisfiability.

Strategies
Like Otter, Prover9 has available many strategies; the following statements apply to CASC-2012.

Given a problem, Prover9 adjusts its inference rules and strategy according to syntactic properties of the input clauses such as the presence of equality and non-Horn clauses. Prover9 also does some preprocessing, for example, to eliminate predicates.

In previous CASC competitions, Prover9 has used LPO to order terms for demodulation and for the inference rules, with a simple rule for determining symbol precedence. For CASC 2012, we are going to use KBO instead.

For the FOF problems, a preprocessing step attempts to reduce the problem to independent subproblems by a miniscope transformation; if the problem reduction succeeds, each subproblem is clausified and given to the ordinary search procedure; if the problem reduction fails, the original problem is clausified and given to the search procedure.

Implementation
Prover9 is coded in C, and it uses the LADR libraries. Some of the code descended from EQP [60]. (LADR has some AC functions, but Prover9 does not use them). Term data structures are not shared (as they are in Otter). Term indexing is used extensively, with discrimination tree indexing for finding rewrite rules and subsuming units, FPA/Path indexing for finding

subsumed units, rewritable terms, and resolvable literals. Feature vector indexing [87] is used for forward and backward nonunit subsumption. Prover9 is available from

```
http://www.cs.unm.edu/~mccune/prover9/
```

Expected Competition Performance

Some of the strategy development for CASC was done by experimentation with the CASC-2004 competition "selected" problems. (Prover9 has not yet been run on other TPTP problems.) Prover9 is unlikely to challenge the CASC leaders, because (1) extensive testing and tuning over TPTP problems has not been done, (2) theories (e.g., ring, combinatory logic, set theory) are not recognized, (3) term orderings and symbol precedences are not fine-tuned, and (4) multiple searches with differing strategies are not run.

Finishes in the middle of the pack are anticipated in all categories in which Prover9 competes.

7.21 Satallax 2.7

Chad E. Brown
Saarland University, Germany

Architecture

Satallax 2.7 [30] 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 [38] 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 [32] with a choice operator [7]. 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 [31]. 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 2.7 is implemented in OCaml. A foreign function interface is used to in teract with MiniSat 2.2.0. Satallax is available from

`http://satallax.com`

Expected Competition Performance
Satallax 2.1 won the THF division at CASC in 2011 and Satallax 2.4 came in second among the competing systems in 2012. Satallax has traditionally been weak on problems involving equational reasoning. Satallax 2.7 should be much better on such problems since most of these problems (in practice) are first-order and can be solved by E, so Satallax 2.7 should perform better than Satallax 2.4 would in the competition.

7.22 Satallax-MaLeS 1.2

Daniel Kühlwein[1], Josef Urban[1], Chad Brown[2]
[1]Radboud University Nijmegen, The Netherlands
[2]Saarland University, Germany

Architecture
Satallax-MaLeS 1.2 is the result of applying the MaLeS 1.2 framework to Satallax (version 2.7) [30]. MaLeS (Machine Learning of Strategies) is a framework for automatic tuning of ATPs. It combines strategy finding (similar to ParamILS [48] and BliStr [132]) with strategy scheduling (similar to SatZilla [135]). A description of Satallax can be found above.

Strategies
Given a set of problems, MaLeS 1.2 finds good parameter settings by using a local random search algorithm. The found settings are stored as strategies. For each strategy, MaLeS learns a function that given a new problem predicts the time the ATP needs to solve this problem when using this particular strategy. When trying to solve a new problem, MaLeS uses these prediction functions to create a strategy schedule.

The learning algorithm is described in [55]. Further information can be found on the project website.

Implementation
MaLeS is implemented in python, using the numpy and scipy libraries. MaLeS is open source and available from

`https://code.google.com/p/males/`

Satallax is available from

`http://satallax.com`

Expected Competition Performance
Satallax-MaLeS 1.2 should perform similar to Satallax 2.7.

7.23 SPASS+T 2.2.19

Uwe Waldmann
Max-Planck-Insitut für Informatik, Germany

Architecture
SPASS+T is an extension of the superposition-based theorem prover SPASS that integrates algebraic knowledge into SPASS in three complementary ways: by passing derived formulas to an external SMT procedure (currently Yices or CVC3), by adding standard axioms, and by built-in arithmetic simplification and inference rules. A first version of the system has been described in [80]; later a much more sophisticated coupling of the SMT procedure has been added [136]. The latest version provides improved support for isint/1, israt/1, floor/1 and ceiling/1 and adds partial input abstraction and history-dependent weights for numerical constants.

Strategies
Standard axioms and built-in arithmetic simplification and inference rules are integrated into the standard main loop of SPASS. Inferences between standard axioms are excluded, so the user-supplied formulas are taken as set of support. The external SMT procedure runs in parallel in a separate process, leading occasionally to non-deterministic behaviour.

Implementation
SPASS+T is implemented in C. SPASS+T is available from

```
http://www.mpi-inf.mpg.de/~uwe/software/#TSPASS
```

Expected Competition Performance
SPASS+T 2.2.16 came second in the TFA division of last CASC; we expect a similar performance in CASC 2013.

7.24 TEMPLAR::leanCoP 0.8

Mario Frank, Jens Otten
University of Potsdam, Germany

Architecture
TEMPLAR is a formula set pruner which has the scope to reduce the set of given axioms to a size which can be handled by current state-of-the-art automated theorem provers. It reads and analyses the given axiom set and searches for a set of formulae which seem to be relevant for the proof of a given formula. The selected formulae can be given to an automated theorem prover with additional information and the theorem prover can attempt a proof. Multiple instances of a theorem prover can be started in parallel while TEMPLAR calculates other formula sets.

The input formulae are transformed into negation normal form (but not into clause normal form) and multiple indexing techniques are applied to speed up the formula selection. The selection itself is done by a dynamic module which uses two different algorithms and triggers them depending on the structure of the current formula set.

TEMPLAR conceptually supports every theorem prover for which an executor (a module) is implemented. Currently, one executor is existent, which is the leanCoP-Executor. leanCop [68, 67] 2.2, a compact connection calculus based theorem prover for classical first order logic is used as inference engine.

The CASC version of TEMPLAR is a beta stadium version of the project and lacks of some features and performance optimizations in order to reduce the memory consumption.

Strategies
TEMPLAR analyses all axioms and gathers information about included predicate symbols, their arity and additional information like their polarities and for instance the complete formula graph.

TEMPLAR uses - depending on the metrics of the formula set (e.g. size of the set) - two different search approaches which are chained, if appropriate. The one approach which is tailored for smaller sets is an unification based graph search with multiple search graph pruning techniques. This is an reconception and extension of the concept of ARDE which was described in [39] and used in the CASC-J6. For this approach, the negation normal forms of all formulae are used. The other approach is an implementation of a frequency based concept commonly used for natural language processing (e.g. automated text summarization) and is tailored for bigger formula sets. This concept aims to recognise the topics of a set of documents and can recognise common symbols and thus should have similar effects like SInE [47] but uses another weighting approach.

Implementation
TEMPLAR is fully implemented in C++ conforming to the C++ standard of 2011. The TPTP and batch file parsing is done by a parser which is implemented in a descriptive meta-language (a grammar) conforming to the C++ syntax. This grammar is transformed into C++ instructions by the boost library (boost spirit Qi). ¡MP¿ The selected formulae can be output in the original form or the Skolem normal form, conforming to the TPTP syntax.

TEMPLAR can be compiled with the GNU Compiler Collection (GCC) with the minimum version 4.6.3 and the Boost library, minimum version 1.42 (1.46 is better). leanCop can be run with SWI Prolog or Eclipse Prolog (5.x) for example.

Currently, TEMPLAR is not available since it is a diploma thesis project but will be published afterwards in the final version.

Expected Competition Performance
TEMPLAR extends a theorem prover by the described pruning techniques but gives the prover the chance to attempt a proof on the original file, too. Thus, the performance is expected to be at least the performance of the underlying prover. Due to the restrictions, some proofs are expected to be found faster and some normally not (in the given time) provable problems are expected to become provable.

7.25 TPS 3.120601S1b

Chad E. Brown[1], Peter Andrews[2]
[1]Saarland University, Germany, [2]Carnegie Melon University, USA

Architecture
TPS is a higher-order theorem proving system that has been developed over several decades under the supervision of Peter B. Andrews with substantial work by Eve Longini Cohen, Dale A. Miller, Frank Pfenning, Sunil Issar, Carl Klapper, Dan Nesmith, Hongwei Xi, Matthew Bishop, Chad E. Brown, Mark Kaminski, Rémy Chrétien and Cris Perdue. TPS can be used to prove theorems of Church's type theory automatically, interactively, or semi-automatically [4, 5]. When searching for a proof, TPS first searches for an expansion proof [63] or an extensional expansion proof [29] of the theorem. Part of this process involves searching for acceptable matings [2]. Using higher-order unification, a pair of occurrences of subformulae (which are

usually literals) is mated appropriately on each vertical path through an expanded form of the theorem to be proved. The expansion proof thus obtained is then translated [79] without further search into a proof of the theorem in natural deduction style.

Strategies
Strategies used by TPS in the search process include:

- Re-ordering conjunctions and disjunctions to alter the way paths through the formula are enumerated.
- The use of primitive substitutions and gensubs [3].
- Path-focused duplication [49].
- Dual instantiation of definitions, and generating substitutions for higher-order variables which contain abbreviations already present in the theorem to be proved [23].
- Component search [22].
- Generating and solving set constraints [28].
- Generating connections using extensional and equational reasoning [29].

Implementation
TPS has been developed as a research tool for developing, investigating, and refining a variety of methods of searching for expansion proofs, and variations of these methods. Its behavior is controlled by hundreds of flags. A set of flags, with values for them, is called a mode. The strategy of the current version of TPS consists of 71 modes. When searching for a proof in automatic mode, TPS tries each of these modes in turn for a specified amount of time (at least 1 second and at most 41 seconds). If TPS succeeds in finding an expansion proof, it translates the expansion proof to a natural deduction proof. This final step ensures that TPS will not incorrectly report that a formula has been proven. TPS is implemented in Common Lisp, and is available from

```
http://gtps.math.cmu.edu/tps.html
```

Expected Competition Performance
TPS was the CASC-22 THF division winner in 2009. In the CASC competitions since 2009 TPS has come in last behind Isabelle, LEO-II and Satallax. There have been no changes to TPS since last year, so there is no reason to believe TPS will win this year.

7.26 Vampire 2.6

Krystof Hoder, Andrei Voronkov
University of Manchester, England

Architecture
Vampire 2.6 is an automatic theorem prover for first-order classical logic. It consists of a shell and a kernel. The kernel implements the calculi of ordered binary resolution and superposition for handling equality. It also implements the Inst-gen calculus. The splitting rule in kernel adds propositional parts to clauses, which are being manipulated using binary decision diagrams and a SAT solver. 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. Although the kernel of the system works only with clausal normal form, the shell accepts a problem in the full first-order logic syntax, clausifies it and performs a number of useful transformations before passing the result to the kernel. Also the axiom selection algorithm Sine [47] can be enabled as part of the preprocessing.

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

Strategies
The Vampire 2.6 kernel provides a fairly large number of options for strategy se lection. The most important ones are:

- Choice of the main procedure:
  - Limited Resource Strategy
  - DISCOUNT loop
  - Otter loop
  - Goal oriented mode based on tabulation
  - Instantiation using the Inst-gen calculus
- 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.

Implementation
Vampire 2.6 is implemented in C++.

Expected Competition Performance
Vampire 2.6 is the CASC-J6 FOF and LTB division winner.

7.27 Zipperposition 0.2

Guillaume Burel, Simon Cruanes
ENSIIE/Cedric, France

Architecture
Zipperposition 0.2 is a superposition prover for first order logic with equality. It is written in OCaml, and largely inspired from E [86]. It features lazy transformation to CNF [42], the Superposition, Equality Resolution and Equality Factoring inference rules, and many simplification rules including unit rewriting, condensation, subsumption and contextual literal cutting.

The prover also uses a so-called ¡I¿knowledge base¡/I¿ that contains description of algebraic theories, of individual axioms (such as associativity and commutativity), lemmas and theory-specific rewrite systems. This base currently contains few definitions and lemmas.

Strategies
Zipperposition allows one to choose a term ordering family among KBO or RPO (using RPO by default). Otherwise, it is totally automated and tries to select a symbol precedence based on its input (e.g., if it detect a symbol definition in the input, it will try to orient it so that demodulation will eliminate the symbol).

Implementation
The prover is written in OCaml. For indexing, it uses perfect discrimination trees for demodulation, feature vector indexes for subsumption, and fingerprint indexing for inferences. It depends on a few OCaml libraries to serialize/deserialize its ¡I¿knowledge base¡/I¿ and to provide a Datalog reasoner at the meta-level. Since Zipperposition is designed to be a prototype for experimenting with superposition, its performance is average at best.

Zipperposition is available from

```
http://www.rocq.inria.fr/deducteam/Zipperposition/
```

Expected Competition Performance
We expect Zipperposition to have decent performance on FOF problems with equality, and poor performance on large CNF problems without equality. The system is not tuned for CASC in any way.

8 Conclusion

The CADE-24 ATP System Competition was the eighteenth 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] The Coq Proof Assistant. http://coq.inria.fr.

[2] P.B. Andrews. Theorem Proving via General Matings. *Journal of the ACM*, 28(2):193–214, 1981.

[3] P.B. Andrews. On Connections and Higher-Order Logic. *Journal of Automated Reasoning*, 5(3):257–291, 1989.

[4] P.B. Andrews, M. Bishop, S. Issar, Nesmith. D., F. Pfenning, and H. Xi. TPS: A Theorem-Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.

[5] P.B. Andrews and C.E. Brown. TPS: A Hybrid Automatic-Interactive System for Developing Proofs. *Journal of Applied Logic*, 4(4):367–395, 2006.

[6] L. Bachmair and H. Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction, A Basis for Applications*, volume I Foundations - Calculi and Methods of *Applied Logic Series*, pages 352–397. Kluwer Academic Publishers, 1998.

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

[8] 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.

[9] 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.

[10] P. Baumgartner, U. Furbach, and I. Niemelä. Hyper Tableaux. In J. Alferes, L. Pereira, and E. Orlowska, editors, *Proceedings of JELIA'96: European Workshop on Logic in Artificial Intelligence*, number 1126 in Lecture Notes in Artificial Intelligence, pages 1–17. Springer-Verlag, 1996.

[11] P. Baumgartner, U. Furbach, and B. Pelzer. Hyper Tableaux with Equality. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction*, number 4603 in Lecture Notes in Artificial Intelligence, pages 492–507. Springer-Verlag, 2007.

[12] P. Baumgartner and C. Tinelli. The Model Evolution Calculus. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction*, number 2741 in Lecture Notes in Artificial Intelligence, pages 350–364. Springer-Verlag, 2003.

[13] P. Baumgartner and C. Tinelli. The Model Evolution Calculus with Equality. In R. Nieuwenhuis, editor, *Proceedings of the 20th International Conference on Automated Deduction*, number 3632 in Lecture Notes in Artificial Intelligence, pages 392–408. Springer-Verlag, 2005.

[14] P. Baumgartner and U. Waldmann. Hierarchic Superposition With Weak Abstraction. In M.P. Bonacina, editor, *Proceedings of the 24t International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2013.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] A. Biere. Lingeling and Friends Entering the SAT Challenge 2012. In A. Balint, A. Belov, A. Diepold, S. Gerber, M. Järvisalo, and C. Sinz, editors, *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, number B-2012-2 in Department of Computer Science Series of Publications B, pages 15–16. University of Helsinki, 2012.

[22] M. Bishop. A Breadth-First Strategy for Mating Search. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 359–373. Springer-Verlag, 1999.

[23] M. Bishop and P.B. Andrews. Selectively Instantiating Definitions. 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 365–380. Springer-Verlag, 1998.

[24] 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.

[25] 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*, Lecture Notes in Artificial Intelligence, 2012.

[26] W.W. Bledsoe. Splitting and Reduction Heuristics in Automatic Theorem Proving. *Artificial Intelligence*, 2:55–77, 1971.

[27] 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.

[28] C.E. Brown. Solving for Set Variables in Higher-Order Theorem Proving. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 408–422. Springer-Verlag, 2002.

[29] C.E. Brown. *Automated Reasoning in Higher-Order Logic: Set Comprehension and Extensionality in Church's Type Theory.* Number 10 in Studies in Logic: Logic and Cognitive Systems. College Publications, 2007.

[30] 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.

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

[32] 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.

[33] G. Burel. Experimenting with Deduction Modulo. 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 162–176. Springer-Verlag, 2011.

[34] 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.

[35] K. Claessen and N. Sörensson. New Techniques that Improve MACE-style Finite Model Finding. In P. Baumgartner and C. Fermueller, editors, *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[36] 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.

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

[38] 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.

[39] M. Frank. Relevanzbasiertes Preprocessing für Automatische Theorembeweiser. In *Leipziger Beiträge zur Informatik - SKIL 2012 - Dritte Studentenkonferenz Informatik Leipzig 2012*, number XXXIV, pages 87–98. Leipziger Informatik-Verbund, 2012.

[40] 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.

[41] 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.

[42] H. Ganzinger and J. Stuber. Superposition with Equivalence Reasoning and Delayed Clause Normal Form Transformation. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction*, number 2741 in Lecture Notes in Artificial Intelligence, pages 335–349. Springer-Verlag, 2003.

[43] 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.

[44] 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.

[45] K. Hoder, L. Kovacs, and A. Voronkov. Interpolation and Symbol Elimination in Vampire. 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 188–195, 2010.

[46] K. Hoder, L. Kovacs, and A. Voronkov. Invariant Generation in Vampire. In P. Abdulla and R. Leino, editors, *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 6605 in Lecture Notes in Computer Science, pages 60–64. Springer-Verlag, 2011.

[47] 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.

[48] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[49] S. Issar. Path-Focused Duplication: A Search Procedure for General Matings. In Swartout W. Dietterich T., editor, *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 221–226. American Association for Artificial Intelligence / MIT Press, 1990.

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

[51] 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.

[52] K. Korovin. Instantiation-Based Reasoning: From Theory to Practice. In R. Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction*, number 5663 in Lecture Notes in Computer Science, pages 163–166. Springer-Verlag, 2009.

[53] 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.

[54] K. Korovin and C. Sticksel. iProver-Eq - An Instantiation-Based Theorem Prover with Equality. 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 196–202, 2010.

[55] D. Kuehlwein, J. Urban, and S. Schulz. E-MaLeS 1.1. In M.P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction*, number 6803 in Lecture Notes in Artificial Intelligence, page To appear. Springer-Verlag, 2013.

[56] T. Liang and C. Tinelli. Exploiting Parallelism in the ME Calculus. In P. Fontaine, R. Schmidt, and S. Schulz, editors, *Proceedings of the 3rd Workshop on Practical Aspects of Automated Reasoning, 6th International Joint Conference on Automated Reasoning*, 2012.

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

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

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

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

[61] W.W. McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA, 2003.

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

[63] D. Miller. A Compact Representation of Proofs. *Studia Logica*, 46(4):347–370, 1987.

[64] 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.

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

[66] T. Nipkow, L. Paulson, and M. Wenzel. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/tutorial.pdf.

[67] 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.

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

[69] D. Pastre. Automatic Theorem Proving in Set Theory. *Artificial Intelligence*, 10:1–27, 1978.

[70] D. Pastre. Muscadet : An Automatic Theorem Proving System using Knowledge and Metaknowledge in Mathematics. *Artificial Intelligence*, 38:257–318, 1989.

[71] D. Pastre. Automated Theorem Proving in Mathematics. *Annals of Mathematics and Artificial Intelligence*, 8:425–447, 1993.

[72] D. Pastre. Muscadet version 2.3 : User's Manual. http://www.math-info.univ-paris5.fr/ pastre/muscadet/manual-en.ps, 2001.

[73] D. Pastre. Strong and Weak Points of the Muscadet Theorem Prover. *AI Communications*, 15(2-3):147–160, 2002.

[74] D. Pastre. Complementarity of a Natural Deduction Knowledge-based Prover and Resolution-based Provers in Automated Theorem Proving. http://www.math-info.univ-paris5.fr/ pastre/compl-NDKB-RB.pdf, 2007.

[75] L. Paulson. A Generic Tableau Prover and its Integration with Isabelle. *Artificial Intelligence*,

5(3):73–87, 1999.

[76] 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*, page To appear, 2010.

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

[78] B. Pelzer and C. Wernhard. System Description: E-KRHyper. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction*, number 4603 in Lecture Notes in Artificial Intelligence, pages 508–513. Springer-Verlag, 2007.

[79] F. Pfenning. *Proof Transformations in Higher-Order Logic.* PhD thesis, Carnegie-Mellon University, Pittsburg, USA, 1987.

[80] V. Prevosto and U. Waldmann. SPASS+T. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning*, number 192 in CEUR Workshop Proceedings, pages 19–33, 2006.

[81] 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*, Lecture Notes in Computer Science, page To appear. Springer-Verlag, 2013.

[82] 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 6803 in Lecture Notes in Artificial Intelligence, page To appear. Springer-Verlag, 2013.

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

[84] 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.

[85] P. Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. 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 274–289. Springer-Verlag, 2012.

[86] 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.

[87] 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, 2004.

[88] 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, 2012.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[105] G. Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 38–49, 2008.

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

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

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

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

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

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

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

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

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

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

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

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

[118] 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.

[119] 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.

[120] G. Sutcliffe and C. Suttner. The CADE-18 ATP System Competition. *Journal of Automated*

Reasoning, 31(1):23–32, 2003.

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

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

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

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

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

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

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

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

[129] 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.

[130] J. Ullman. *Principles of Database and Knowledge-Base Bystems*. Computer Science Press, Inc., 1989.

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

[132] J. Urban. BliStr: The Blind Strategymaker. 2013.

[133] J. Urban, G. Sutcliffe, P. Pudlak, and J. Vyskocil. MaLARea SG1: Machine Learner for Automated Reasoning with Semantic Guidance. In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 441–456. Springer-Verlag, 2008.

[134] C. Wernhard. System Description: KRHyper. Technical Report Fachberichte Informatik 14–2003, Universität Koblenz-Landau, Koblenz, Germany, 2003.

[135] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[136] S. Zimmer. Intelligent Combination of a First Order Theorem Prover and SMT Procedures. Master's thesis, Saarland University, Saarbruecken, Germany, 2007.