Beagle – A Hierarchic Superposition Theorem Prover

Peter Baumgartner[1], Joshua Bax[1], and Uwe Waldmann[2]

[1] NICTA* and Australian National University, Canberra, Australia
[2] MPI für Informatik, Saarbrücken, Germany

Abstract. *Beagle* is an automated theorem prover for first-order logic modulo built-in theories. It implements a refined version of the hierarchic superposition calculus. This system description focuses on *Beagle*'s proof procedure, background reasoning facilities, implementation, and experimental results.

1 Introduction

This paper describes the automated theorem prover *Beagle*. *Beagle* implements hierarchic superposition [2,7], a calculus for automated reasoning in a hierarchic combination of first-order logic and some background theory. Currently implemented background theories are linear integer and linear rational arithmetics. *Beagle* features new simplification rules for theory reasoning, and well-known ones used for non-theory reasoning. *Beagle* also implements calculus improvements like *weak abstraction* [7] and determining (un)satisfiability w. r. t. quantification over finite integer domains [6].

Beagle is written in Scala, including its implementation of the background reasoners from scratch. Existing SMT solvers can be coupled as background reasoners as well via a textual SMT-LIB interface. *Beagle* accepts problem specifications written in the TFF format (the typed version of the TPTP problem specification language) and in the SMT-LIB format [4,16].

In this paper we describe the above features in more detail and report on *Beagle*'s performance on the TPTP problem library [17] and SMT-LIB benchmarks [16].

2 Hierarchic Theorem Proving

Hierarchic superposition [2,7] is a calculus for automated reasoning in a hierarchic combination of first-order logic and some background theory.[3] We assume that we have a *background ("BG")* prover that accepts as input a set of clauses over a *BG signature* $\Sigma_{\mathrm{B}} = (\Xi_{\mathrm{B}}, \Omega_{\mathrm{B}})$, where Ξ_{B} is a set of *BG sorts* and Ω_{B} is a set of *BG operators*. Terms/clauses over Σ_{B} and BG-sorted variables are called *BG terms/clauses*. The BG prover decides the satisfiability of Σ_{B}-clause sets w. r. t. a *BG specification*, that is, a class of term-generated Σ_{B}-interpretations (called *BG models*) that is closed under isomorphisms. The BG specification is usually some kind of arithmetic, so Ξ_{B} could for

* NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

[3] Due to a lack of space, we can only give a brief overview of the calculus and of the semantics of hierarchic specifications. We refer to [7] for the details.

instance be $\{int\}$ and Ω_B could contain the BG operators $0, 1, -1, 2, -2, \ldots, +, -, <, \leq$. We assume that Ω_B also contains infinitely many *parameters* $\alpha, \beta, \ldots$, that is, additional constants that may be interpreted freely by arbitrary elements of the appropriate domain in different models.

The *foreground ("FG")* theorem prover accepts as input a set of clauses over an extended signature $\Sigma = (\Xi, \Omega)$, where $\Xi_B \subseteq \Xi$ and $\Omega_B \subseteq \Omega$. The sorts in $\Xi_F = \Xi \setminus \Xi_B$ and the operator symbols in $\Omega_F = \Omega \setminus \Omega_B$ are called *FG sorts* and *FG operators*. For instance, Ξ_F might be $\{list\}$ and Ω_F could then contain the operators $\mathsf{empty} :\rightarrow list$, $\mathsf{cons} : int\ list \rightarrow list$, and $\mathsf{length} : list \rightarrow int$. We use sans-serif letters to denote FG operators. A Σ-term is an *FG term* if it is not a BG term, that is, if it contains at least one FG operator or FG-sorted variable. We emphasize that for an FG operator $\mathsf{f} : \xi_1 \ldots \xi_n \rightarrow \xi_0$ in Ω_F any of the ξ_i may be a BG sort. Consequently, a FG term like $\mathsf{length}(\mathsf{cons}(5, \mathsf{empty}))$ may have a BG sort. Every FG operator f with a BG range sort $\xi_0 \in \Xi_B$ is called a *free BG-sorted (FG) operator*.

The intended semantics is that of *conservative extensions of the BG specification*, i. e., Σ-interpretations whose restriction to Σ_B is a model of the BG specification. In the concrete example above, that means that we are only interested in models of the FG clause set whose interpretation of the BG sort *int* is the same as in the given BG models; the models may neither identify different elements of the interpretation of *int*, say 5 and 7, nor interpret BG-sorted FG term like $\mathsf{length}(\mathsf{cons}(5, \mathsf{empty}))$ by some new element that was not present before. We refer to satisfiability in this sense as *$\mathcal{B}$-satisfiability*.

Hierarchic theorem proving requires "abstracting out" terms in preparation for inference rule applications.[4] *Weak abstraction* introduced in [7] abstracts out BG terms other than number constants and variables that occur as subterms of FG terms, so for instance the clause $\mathsf{cons}(\alpha, \mathsf{cons}(x, \mathsf{empty})) \approx \mathsf{cons}(3, \mathsf{cons}(5 + 2, y))$ is converted into

$$z_1 \not\approx \alpha \vee z_2 \not\approx 5 + 2 \vee \mathsf{cons}(z_1, \mathsf{cons}(x, \mathsf{empty})) \approx \mathsf{cons}(3, \mathsf{cons}(z_2, y))$$

whereas $\mathsf{length}(\mathsf{cons}(x, y)) \approx \mathsf{length}(y) + 1$ is left unchanged. See [7] for a discussion of the benefits of weak abstraction.

The FG prover saturates the set of Σ-clauses using the inference rules of hierarchic superposition, such as, e. g.,

$$\text{Negative superposition}\ \frac{l \approx r \vee C \qquad s[u] \not\approx t \vee D}{\mathrm{abstr}((s[r] \not\approx t \vee C \vee D)\sigma)}$$

where σ is an mgu of l and u, These inference rules inherit the ordering and selection restrictions of the standard superposition inference rules [1]; in addition they have the new restriction that only the FG parts of clauses are overlapped. Since the standard inferences can destroy weak abstraction, it is furthermore necessary to apply an explicit weak abstraction to the conclusion. The term ordering $\succ$ needs to satisfy certain properties specific to the hierarchic case, e. g., any concrete number must be smaller than any other ground term. The calculus includes the generic, semantically defined notion of redundancy well-known from standard superposition.

[4] *Abstracting out* a term t that occurs in a clause $C[t]$ means replacing $C[t]$ by $x \not\approx t \vee C[x]$ for a new variable x.

Since the standard superposition inference rules are modified in such a way that only the FG parts of clauses are overlapped, that means in particular that they are never applied to BG clauses derived during the saturation. Such clauses are instead passed to the BG prover. The BG prover implements an inference rule

$$\textsf{Close}\ \frac{C_1 \quad \cdots \quad C_n}{\square} \qquad \text{if } C_1, \ldots, C_n \text{ are BG clauses and } \{C_1, \ldots, C_n\} \text{ is } \mathcal{B}\text{-unsatisfiable.}$$

As soon as one of the two provers derives the empty clause, the input clause set has been shown to be $\mathcal{B}$-unsatisfiable.

The Define *rule.* One of the requirements for the refutational completeness of hierarchic superposition is *sufficient completeness*, i. e., the property that every ground BG-sorted FG term is equal to some BG term. Sufficient completeness of a set of Σ-clauses is a property that is not even recursively enumerable. For certain classes of Σ-clause sets, however, it is possible to establish a variant of sufficient completeness automatically [11,7]: If all BG-sorted FG terms in the input are ground, it suffices to show that each BG-sorted FG term *in the input* is equal to some BG term. This can be achieved by adding a *definition* $\alpha_t \approx t$ for every BG-sorted FG term t occurring in a clause $C[t]$, where α_t is a new parameter (BG constant); afterwards $C[t]$ can be replaced by $C[\alpha_t]$. See [7] for the corresponding *Define* inference rule.

3 Background Reasoning

BG reasoning is represented in *Beagle* as theory specific modules, *"solvers"*, that implement a specific interface. Every solver needs to provide a decision procedure for $\mathcal{B}$-satisfiability of sets of BG clauses. The syntactic fragment of these BG clauses depends on whether free BG-sorted constants are declared as FG constants or as parameters. The former case leads to the A-fragment, the latter case to the EA-fragment. Moreover, if the solver also supports quantifier elimination (QE), the decision procedure receives sets of ground clauses only.

In all examples we use linear integer arithmetic as the background theory.

Quantifier elimination. The solver interface supports specifying a quantifier elimination procedure on BG *formulas*. It is used for eliminating variables that only occur in BG literals. This way, e. g., the clause $\mathsf{P}(x) \vee \neg(x < y) \vee \neg(y < 3)$ becomes $\mathsf{P}(x) \vee \neg(x < 2)$ by QE of y from $\neg(x < y) \vee \neg(y < 3)$. Applying QE this way for clause simplification may destroy refutational completeness, since in general the simplification result is possibly larger (under the clause ordering) than the clause being simplified. To avoid this problem, *Beagle* uses QE for clause simplification only during preprocessing. It also stores with each BG clause its ground version, which is sent to the decision procedure.

Splitting. Beagle optionally splits (in particular) BG clauses into variable disjoint subclauses. If QE is available and *Beagle* is instructed to, a ground version of each BG clause is added to the current clause set, which is split exhaustively into unit clauses by *Beagle*'s splitting rule. As a consequence, the decision procedure receives sets of unit clauses only, akin to SMT solvers.

Simplification. *Beagle* removes disequations of certain forms from clauses by *unabstraction*. For example, if cautious simplification is chosen, literals of the form $x \not\approx d$ are removed by unabstraction only if d is a concrete number.

Aggressive simplification enables the unabstraction of any term, including FG terms. It can possibly break completeness, since there is no guarantee that the unabstracted clause $C[t]$ is smaller than all possible instances of $C[x] \vee x \not\approx t$. The simplification level of FG clauses is controlled by *Beagle*. Typically only the results of cautious unabstraction are kept; aggressive unabstraction is used to derive unit clauses which may demodulate other clauses, but the unit clauses resulting from unabstraction are not kept.

Beyond that, simplification of arithmetic terms is realized through an internal data structure for simplification rules. The current simplification rules are hard-coded in *Beagle*'s implementation language Scala. Hence they are not user-modifiable, but we might change this in a future version. For each solver there are two sets of simplification rules: *cautious* simplification rules, which are known to preserve both sufficient completeness and refutational completeness, and *aggressive* simplification rules, which in general do not preserve these properties. See below for examples.

Solvers. *Beagle* implements solvers for linear integer arithmetic (LIA) and linear rational arithmetic (LRA). It also accepts linear real arithmetic but the differences are merely syntactic. Alternatively to the built-in LIA solver, existing SMT solvers can be coupled via a textual SMT-LIB interface.

3.1 Linear Integer Arithmetic

Quantifier elimination. The built-in LIA solver is based on Cooper's quantifier elimination algorithm and its improvements as introduced in [8]. It accepts arbitrary BG formulas, in particular conjunctions of clauses. The code structure follows roughly the algorithm described in [10]. The LIA solver is used for both deciding satisfiability of sets of BG clauses and for the elimination of variables as described above.

We have integrated several improvements into Cooper's algorithm to make it more practical. For example, in conjunctions that contain the atomic formulas $\alpha < 5$ and $\alpha < 3$ the former can be removed; a limited form of subsumption. Other simple and cheap techniques include elimination of variables that admit unbounded solutions and elimination of equations $\alpha \approx t$ where α does not occur in t. Furthermore, if a conjunction contains the atomic formulas $s_1 < \alpha, \ldots, s_m < \alpha$ and $\alpha < t_1, \ldots, \alpha < t_n$, given that α does not occur elsewhere, then α can be removed by exhaustive resolution. (Resolution of $s < \alpha$ and $\alpha < t$ yields $s + 1 < t$.) If α does occur somewhere else, then this form of resolution can still be used to prove unsatisfiability when $s + 1 < t$ is false. This is similar to the first step of the Omega test for deciding Presburger arithmetic [14].

The improvements mentioned above often help to solve problems much faster.[5] However, some of them are effective only on conjunctions of literals. In support of this, our algorithm deviates from the standard Cooper algorithm by multiplying out disjunctions that arise from quantifier instantiation. This often avoids deeply structured

[5] E.g., the GEG-problems in the TPTP problem library.

"or-and" formulas. As a special case, disjunctive normal form is preserved by solving and multiplying out the conjunctions separately.

The final step of Cooper's algorithm involves instantiation over representatives of congruence classes of solutions for the target variable which quite often lead to prohibitively large formulas. Using an improvement suggested in [10] Beagle occasionally defers this instantiation (based on the expected number of instances) until a later round of quantifier elimination. This is done by substituting a fresh variable and terms that describe the solution classes as occasionally a shorter proof of satisfiability/unsatisfiability can be found using a different variable.

When the Close rule applies *Beagle* determines a minimal unsatisfiable subset of the BG clauses passed to the decision procedure. This is advantageous for the main loop's dependency-directed backtracking since cases which only produce BG clauses that are irrelevant for unsatisfiability do not need to be backtracked to. Currently, minimal unsatisfiable subsets are determined by binary search on the whole clause set passed to the (built-in) LIA solver, or by unsatisfiable cores returned by Z3 [12] as a solver.

Simplification and arithmetic terms normalization. The cautious simplification rules for LIA comprise evaluation of arithmetic terms, e. g. $3 \cdot 5$, $3 < 5$, $\alpha + 1 < \alpha + 1$ (equal lhs and rhs terms in inequations), and rules for TPTP-operators, e. g., $\$to\_rat(5)$, $\$is\_int(3.5)$. For aggressive simplification, integer sorted subterms are brought into a polynomial-like form and are evaluated as much as possible. For example, the term $5 \cdot \alpha + \mathsf{f}(3+6, \alpha \cdot 4) - \alpha \cdot 3$ becomes $2 \cdot \alpha + \mathsf{f}(9, 4 \cdot \alpha)$. These conversions exploit associativity and commutativity of $+$ and $\cdot$. Pure BG formulas always produce proper polynomials, which can be used directly by the QE procedure without further conversions.

Aggressive simplification does not always preserve sufficient completeness. For example, in the clause set $N = \{\mathsf{P}(1+(2+\mathsf{f}(x))),\ \neg\mathsf{P}(1+(x+\mathsf{f}(x)))\}$ the first clause is aggressively simplified, giving $N' = \{\mathsf{P}(3+\mathsf{f}(x)),\ \neg\mathsf{P}(1+(x+\mathsf{f}(x)))\}$. Notice that both N and N' are LIA-unsatisfiable, $\mathrm{sgi}(N) \cup \mathrm{GndTh}(\mathrm{LIA})$ is unsatisfiable, but $\mathrm{sgi}(N') \cup \mathrm{GndTh}(\mathrm{LIA})$ is satisfiable. Thus, N is (trivially) sufficiently complete while N' is not.

We have also implemented heuristics for normalizing equations and inequations for aggressive simplification. Inequations are normalized by first eliminating the operators $>$, $\geq$ and $\leq$ in terms of $<$. The QE procedure treats $<$ as a primitive, so this is a natural choice. Then, the monomials of the lhs and rhs polynomials are moved around so that only positive signs and only addition of monomials (not subtraction) results. The rationale is to normalize terms by removing unnecessary operators. Similar heuristics apply for equations, which also attempt to arrive at orientable equations. Normalizing (in)equations may remove or install sufficient completeness and destroy refutational completeness. Yet, in our experiments we found that aggressive simplification is far superior to cautious simplification in practice, hence it is enabled by default.

3.2 Other Arithmetic Features

Linear Rational Arithmetics. The solver for LRA comprises a Fourier-Motzkin style quantifier elimination procedure for eliminating BG variables as described in Section 3. The decision procedure implements the Simplex algorithm extended to strict inequalities [9]. The cautious simplification rules evaluate arithmetic subterms, and the ag-

gressive simplification rules rewrite sub-terms towards a flat structure by exploiting AC-properties of the operators. Syntactic differences between concrete numbers aside, linear real arithmetics is treated by additional lemmas that are valid in real arithmetics. The LRA solver is not as far developed as the LIA solver.

Nonlinear Arithmetic. *Beagle* features a simplistic treatment of non-linear arithmetics. During preprocessing, every occurrence of a non-linear multiplication subterm $s \cdot t$ is replaced by $\mathsf{nlpp}(s, t)$, where nlpp is a dedicated foreground function symbol of the proper arity. As soon as s or t in $\mathsf{nlpp}(s, t)$ is replaced by a concrete number, the resulting nlpp is turned into a multiplication term again. In the LIA case, axioms for nlpp are added that express multiplication in terms of repeated addition.

4 Proof Procedure

This section provides a summary of *Beagle*'s proof procedure. The proof procedure follows by and large standard techniques, but treats BG formulas in a specific way on some occasions.

Preprocessing. *Beagle* accepts its input formulas in two alternative syntaxes, TPTP-TFF [19] and SMT-LIB (version 2) [4]. The SMT-LIB language is richer than the TPTP-TFF language due to its support for polymorphic sorts and functions. The SMT-LIB also features predefined theories such as arrays. *Beagle* automatically monomorphizes sorts and function symbols, and it generates array axioms as needed.

Both TPTP-TFF and SMT-LIB provide syntax for full first-order logic (not just clausal logic). *Beagle* has two translators into clause normal form (CNF), a standard one and a Tseitin-style translator which introduces definitions for "complex" subformulas. The default is the standard CNF translator, because it gave the better results overall over the problems in the TPTP.

CNF transformation includes Skolemization of existentially quantified variables. *Beagle*'s CNF transformation treats existentially quantified integer variables in a special way, by removing them with QE instead of Skolemization, if possible. For example, the input formula $\forall x : \mathbb{Z}\, \mathsf{P}(x) \vee \exists y : \mathbb{Z}\, y \not\approx x+1$ becomes $\forall x : \mathbb{Z}\, \mathsf{P}(x)$, whereas Skolemization would have given $\forall x : \mathbb{Z}\, \mathsf{P}(x) \vee \mathsf{f}(x) \not\approx x + 1$. In particular, if the input formulas are all BG formulas over the integers, no Skolem functions are introduced, and so *Beagle* is a decision procedure for that class.

Main loop and simplification. Beagle's main loop is the well-known "Discount loop". It maintains two clause sets, *Old* and *New*, where *Old* is initially empty and new is initialized with the input clauses. On each round, a *selected* clause is removed from *New* and simplified by the clauses from *Old* and *New*. The simplified selected clause then is put into *Old* and all inferences between it and the clauses in *Old* are carried out. The resulting clauses are simplified by the *Old* clauses and go into *New* again, this way closing the loop. If a BG clause results, the solver is called with the thus extended current set of all BG clauses.

Implemented simplification techniques include standard ones, like demodulation by unit clauses, proper subsumption deletion, and removing a positive literal L from a

clause in presence of a unit clause that instantiates to the complement of L. All clauses in *Old* are mutually simplified. Backward simplification is optional.

By default, a split rule is enabled that breaks clauses into variable-disjoint subclauses and branches out correspondingly. Dependency-directed backtracking is used to avoid exploring irrelevant cases.

The default term ordering is LPO if BG theories are present, otherwise it is KBO. See [7] for properties of the LPO specific to hierarchic superposition.

Fairness. Fairness is achieved by a combination of clause weights and their derivation age. This is controlled by a parameter "weight-age-ratio", a non-negative number saying how many lightest clauses are selected before an oldest clause is selected. Clause weights are computed in such a way that selection based on weights only would be a fair strategy. In our experiments we used a weight-age-ratio of five.

Auto mode. *Beagle* includes a simple auto mode. When on, *Beagle* first tries the default flag setting. If there is no conclusive result within half of the given time limit, *Beagle* starts again using a setting where BG variables in the input may be instantiated by BG-sorted FG terms, rather than only by BG terms.

5 Implementation

Beagle implements support for both the TPTP-TFF and SMT-LIB input languages using Scala's parser combinator library. *Beagle*'s internal formula representation follows TFF, so to support the SMT-LIB standard it must perform sort monomorphization and adding axioms for predefined theories like arrays. This is done with the help of the separately developed SMTtoTPTP library [5].

Beagle uses a simple term-indexing scheme which is essentially top symbol hashing. This is used to retrieve term positions eligible for superposition or demodulation within clauses.

Scala specific features. *Beagle* makes heavy use of many built in Scala datastructures, primarily `List`, `Vector` and `Map`. Not only are the implementations well optimised but they also provide powerful abstractions allowing for simple and maintainable code.

Scala's declarative style encourages the use of immutable values, which minimizes data duplication. Scala also provides a lazy evaluation feature, which we have found extremely useful for caching data: e. g. the computation of maximal literals in a clause can be deferred until the clause becomes eligible for an inference and it may never be computed if the clause is simplified first. We found that the Scala REPL interpreter is an invaluable tool for debugging: for example, one could take the (usually large) result of an invalid derivation and programmatically investigate it using functional operators like map or filter.

The simple structure of logic formulas is a good fit for property based testing libraries such as `scalacheck`[6] which use grammars to generate random test data. These data are used as input for properties given as universally quantified predicates.

[6] `http://scalacheck.org/`

6 Performance

TPTP. We tried *Beagle* on the first-order problems from the TPTP–v6.1.0 problem library [17] that involve some form of arithmetic, including non-linear, rational and real arithmetics. The experiments were carried out on a MacBook Pro with a 2.3 GHz Intel Core i7 processor and 16 GB of main memory. The CPU time limit was 180 seconds.

Although *Beagle* detected countersatisfiabilty of some of the (73) non-theorem problems, we discuss in the following the performance on the problems with a "theorem" or "unsatisfiable" status only. Of these 972 problems in total *Beagle* was able to prove 781 using automatic strategy selection. The backup strategy was attempted a total of 21 times and was successful in 15 cases, thereof 13 times in the TPTP DAT category.

Table 1 summarizes the results. Broken down by the TPTP problem category we see that *Beagle*'s best performance was on ARI, DAT and NUM. These are characterized by smaller problem sizes with an arithmetic reasoning component. On the other hand performance was much worse on those problems which involve large problem sizes such as SWW and SWV (translations of model-checking problems). *Beagle* failed to solve any HWV problems (large EPR encodings of bounded model-checking) due to the size of the formulas and emphasis on boolean reasoning. The remaining easy (rated < 0.1) problems that *Beagle* failed to solve were all non-theorems, most involving multiplication operators. The two solvable problems with a rating of 1.0 are ARI536=2.p and DAT086=1.p.

Category	ARI	DAT	GEG	HWV	MSC	NUM	PUZ	SEV	SWV	SWW	SYN	SYO
Total	539	103	5	88	2	43	1	6	2	177	1	3
Solved	531	98	5	0	2	41	1	2	2	97	0	2

Rating	≥ 0.0	≥ 0.1	≥ 0.2	≥ 0.3	≥ 0.4	≥ 0.5	≥ 0.6	≥ 0.7	≥ 0.8	≥ 0.9	1.0
Total	972	853	771	527	391	343	253	180	129	97	97
Solved	781	666	584	340	210	162	85	29	12	2	2

Table 1: *Beagle* performance on the TPTP "theorem" or "unsatisfiable" problems. The first table breaks down the number of solved problems by category. The second table filters by problem rating. The column ≥ 0.6, for instance, means "all problems with a rating 0.6 or higher."

We have also coupled the SMT solver Z3 [12] as an alternative to the built-in LIA solver. In our experiments we also tried a modified split rule that leaves BG subclauses unsplit. In particular, BG clauses are never split then. The rationale is that letting the SMT solver deal with (non-unit) BG clauses might be better than the default FG splitting into sets of unit clauses. As an alternative to the built-in LIA solver and using the modified split rule or not hence gives four base configurations.

We ran *Beagle* in all four base configurations and several additional flag settings. But, surprisingly, Z3 does *not* give better results than the built-in solver. We found that the default split rule is superior to the modified one, both in conjunction with Z3 and the built-in solver. Over all settings, however, almost exactly the same problems are

solvable with any of the two solvers and in roughly the same time. This finding might not carry over to problems that require more complex BG reasoning than those in the TPTP.

SMT-LIB. We tested *Beagle* on the 2014 release of SMT-LIB [16] focusing on the logics with an arithmetic component. Specifically these were ALIA, AUFLIA, UFLIA, UF_IDL (integer difference logic) and the corresponding quantifier free problem sets, including QF_LIA. (The LIA category was ignored as it contains only problems from the TPTP). We selected only those problems indicated as unsatisfiable in the problem description and *Beagle* was run with automatic strategy selection (as described above). We found a mix of results: *Beagle* was able to solve a few problems unsolved by SMT solvers[7] yet there were also quite a few problems that were marked as 'trivial' (all SMT solvers in the SMT-Eval 2013 can solve them in under five seconds), which *Beagle* could not solve. Overall *Beagle* solved the following problems by category (QF refers to the quantifier free fragment of the logic to the left):

Logic	ALIA	QF	AUFLIA	QF	UFLIA	QF	UFIDL	QF	QF_IDL	QF_LIA
Total	41	72	4	516	6602	195	62	335	694	2610
Solved	31	40	4	205	1736	155	42	29	24	28

In total *Beagle* solved 89 problems not solved by SMT solvers and these were divided among the following subcategories of 'UFLIA/sledgehammer':

Category	Arrow_Order	FFT	FTA	Hoare	StrongNorm	TwoSquares
Solved	17	2	34	20	2	14

There were many problems which *Beagle* could not parse, as it is not optimized for large problem sets. In total there were 1,391 trivial problems not solved by *Beagle*.

It was not possible to draw broad conclusions about which categories *Beagle* is best suited to. For example, all of the hardest problems *Beagle* solved were among the UFLIA benchmarks, but there were also at least 200 trivial problems from that category were unsolved (in the 'simplify' and 'simplify2' subcategories). Also it was hypothesised that *Beagle* would perform much worse in the quantifier free fragment, and that was the case for QF_IDL and QF_LIA, but not so for QF_UFLIA and QF_AUFLIA.

CASC-J7. Most recently *Beagle* participated in the CASC-J7 competition [18]. in the TFA division (Typed First-order Arithmetic theorems). For this division the problem set consists of typed first-order problems with an arithmetic component over integers, rationals, or reals, of which roughly half were previously unseen by competitors.

Other solvers entered in the TFA category were *CVC4* [3], *SPASS+T* [13], *Zipperposition* (see [18]), and *Princess* [15]. In terms of overall problems solved *Beagle* placed third equal with 173/200 solutions, only three fewer than the winning solver *CVC4*. *Beagle* performed quite well in terms of mean efficiency (solutions per second multiplied by number of solutions); it was outperformed by only *CVC4* [8].

[7] For this we used the difficulty ratings given for SMT-Comp 2014.

[8] For an explanation of how mean efficiency is computed see the CASC-J7 proceedings [18].

7 Availability

Beagle is available at `https://bitbucket.org/peba123/beagle` under a GNU General Public license. The distribution includes the Scala source code and a ready-to-run Java jar-file. A more experimental version of Beagle is maintained at `https://bitbucket.org/joshbax189/beagle`.

References

1. L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
2. L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierarchic first-order theories. *Appl. Algebra Eng. Commun. Comput*, 5:193–212, 1994.
3. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS 6806*, pp. 171–177. Springer, 2011.
4. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, eds., *SMT Workshop*, 2010.
5. P. Baumgartner. SMTtoTPTP – A Converter for Theorem Proving Formats. In A. Felty and A. Middeldorp, eds., *CADE-25*, LNAI. Springer, 2015.
6. P. Baumgartner, J. Bax, and U. Waldmann. Finite Quantification in Hierarchic Theorem Proving. In S. Demri, D. Kapur, and C. Weidenbach, eds., *IJCAR 2014*, *LNAI 8562*, pp. 152–167. Springer, 2014.
7. P. Baumgartner and U. Waldmann. Hierarchic Superposition With Weak Abstraction. In M. P. Bonacina, editor, *CADE-24*, *LNCS 7898*, pp. 39–57. Springer, 2013.
8. D. C. Cooper. Theorem Proving in Arithmetic Without Multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier.
9. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, *LNCS 4144*, pp. 81–94, Springer, 2006.
10. J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
11. E. Kruglov and C. Weidenbach. Superposition Decides the First-Order Logic Fragment Over Ground Theories. *Mathematics in Computer Science*, pp. 1–30, 2012.
12. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, eds., *TACAS*, *LNCS 4963*, pp 337–340. Springer, 2008.
13. V. Prevosto and U. Waldmann. SPASS+T. In G. Sutcliffe, R. Schmidt, and S. Schulz, eds., *ESCoR: Empirically Successful Computerized Reasoning*, CEUR Workshop Proceedings, pp. 18–33, Seattle, WA, USA, 2006.
14. W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *ACM/IEEE conference on Supercomputing*, pp. 4–13. ACM, 1991.
15. P. Rümmer. A Constraint Sequent Calculus for First-Order Logic With Linear Integer Arithmetic. In I. Cervesato, H. Veith, and A. Voronkov, eds., *LPAR*, *LNCS 5330*, pp. 274–289. Springer, 2008.
16. SMT-LIB, The Satisfiability Modulo Theories Library. `http://smt-lib.org/`.
17. 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.
18. G. Sutcliffe. The 7th IJCAR Automated Theorem Proving System Competition - CASC-J7. *AI Communications*, 28:To appear, 2015.
19. G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP Typed First-order Form with Arithmetic. In N. Bjørner and A. Voronkov, eds., *LPAR-18*, *LNCS 7180*. Springer, 2012.