Towards Efficient Subsumption

Tanel Tammet

Department of Computing Science, University of Göteborg,
Göteborg, Sweden
tammet@cs.chalmers.se

Abstract. We propose several methods for writing efficient subsumption procedures for non-unit clauses, tested in practice as parts incorporated into the Gandalf family of theorem provers. Versions of Gandalf exist for classical logic, first order intuitionistic logic and type theory.

Subsumption is one of the most important techniques for cutting down search space in resolution theorem proving. However, for many problem categories most of the proof search time is spent on subsumption. While acceptable efficiency has been achieved for subsuming unit clauses (see [7], [2]), the nonunit subsumption tends to slow provers down prohibitively.

We propose several methods for writing efficient subsumption procedures for non-unit clauses, succesfully tested in practice as parts built into the Gandalf family of theorem provers:

- ordering literals according to a certain subsumption measure
- indexing first two literals of each nonunit clause
- pre-computed properties of terms, literals and clauses
- a hierarchy of fast filters for clause-to-clause subsumption
- combining subsumption with clause simplification
- linear search among the strongly reduced number of candidates for back subsumption

The presented methods for substitution were among the key techniques enabling the classical version of Gandalf to win the MIX division of the CASC-14 prover contest in 1997. The approach of the paper is purely empirical, presenting the methods and bringing some statistical evidence.

1 Gandalf Family of Provers

Before continuing with the details of the subsumption methods we will present an overview of the Gandalf family of provers. We use the name Gandalf for the interdependent, code-sharing, resolution-based automated theorem provers we are developing: a resolution prover for first-order intuitionistic logic Tammet [9], for a fragment of Martin-Löf's type theory Tammet [10] and for first-order classical logic (earlier version in Tammet [11]).

C. Kirchner and H. Kirchner (Eds.): Automated Deduction, CADE-15
LNAI 1421, pp. 427–441, 1998. © Springer–Verlag Berlin Heidelberg 1998

1.1 Gandalf for Intuitionistic Logic and Type Theory

The motivation for the intuitionistic version of Gandalf was to build the very first resolution prover for the logic. The type theory version is developed as an assistant for human users of the type theory system ALF, see Magnusson and Nordström [4]. One of our goals was to experiment with the optimised translation methods from type theory to classical logic, presented in Tammet [10]

1.2 Code Sharing and Comparisons

The intuitionistic version presented in Tammet [9] uses the general scheme of building resolution calculi (also called the inverse method) originating from Maslov and Mints [5], augmented with a number of novel search strategies.

Most of the code of the intuitionistic prover is shared with the classical prover. This makes it easy to implement universal low-level resolution strategies, heuristics and engineering solutions, so that they will work both for classical and intuitionistic logic.

Since the resolution proof search in intuitionistic logic typically generates large amounts of non-unit clauses, the efficiency of the subsumption procedure for the non-unit clauses is highly important for the intuitionistic version of Gandalf. Since the same procedure is shared by the intuitionistic and and classical components, the subsumption component is currently one of the most sophisticated parts of the classical version of Gandalf.

However, we note that the top-level search strategies for these logics are still substantially different.

In [9] we observe that for the most of the first-order intuitionistic formulas we have found in the literature, our resolution-based prover compares favourably with the earlier tableaux-based provers.

The similarity of the search paradigm — resolution — and the code for intuitionistic and classical provers also simplifies comparison between the time it takes to prove a formula intuitionistically and classically. Roughly said, in almost all of our experiments the intuitionistic proof search is much harder than the classical proof search.

The current version of Gandalf for the (restricted) type theory uses a modification of the classical prover, with the latter limited to Horn clauses, where classical and intuitionistic provability coincide. The modified classical prover is used as an inference engine in a larger system doing conversions between type theory and classical logic as well as creating proof subtasks by automating structural induction on types. The system always performs the conversion from classical proofs to type theory proofs, which is a relatively complicated part of the system. In particular, we use a separate limited proof search for the conversion only.

1.3 Gandalf for Classical Logic

Although some of our motivations stem from nonclassical logics, we are strongly interested in the classical version in its own right. In particular, Gandalf contains

special strategies for program synthesis using classical logic, see Tammet [8] as well as decision strategies based on orderings, see Fermüller et al [1]. The classical version of Gandalf won the prover competition CASC-14 during the CADE-14 in 1997.

Gandalf implements a number of different basic strategies: binary ordered resolution for several orderings, versions of set-of-support resolution, binary unit resolution, hyperresolution. Enhancements are used for cutting off literals and combining different strategies, in particular forward and backward reasoning, into one run. Equality is handled by ordered paramodulation and demodulation.

1.4 Time Slicing

It is universally recognised that there cannot exist any simple search strategies which are feasible for most or all problems. Typically, different strategies have dramatic differences of behaviour on different problems. It is rather common that a proof which would take years to find with one particular search strategy can be found in a few seconds with another.

Selecting possibly suitable strategies and running them either in parallel or one after another is a regular pattern of practical use of provers by humans. We think that it is worthwhile to automatise at least parts of this Las Vegas type strategy selection algorithm in order to obtain better cooperation between runs with different strategies and to assist a human user — in particular, an unexperienced human user — with a powerful expert system also for the meta-level of theorem proving.

The basic idea of the automatic mode in Gandalf — also used during the CASC-14 competition — is time-slicing: Gandalf selects a set of different search strategies, allocates time to these and finally runs the strategies one after another. The motivation is the following: since it is very hard to determine a *single* suitable strategy by some heuristic procedure, it pays off to try a number of possibly suitable strategies.

The selection of strategies and the percentage of time they receive is in a somewhat ad hoc way based on the characteristics of the problem: percentage of horn and almost-horn clauses, the number of clauses, percentage of clauses where literals can be ordered using term and variable depths. For example, in case of small horn problems hyperresolution and binary unit resolution would get most of the time, whereas in case of large non-horn problems several versions of set-of-support would get most of the time.

The set of candidate strategies contains both pure, complete strategies, incomplete combinations (for example, hyperresolution with set-of-support) and strategies with limitations on term depth and clause length. For pure unit equality problems a fixed set of five different strategies was used.

Once the strategies are selected, they are run one after another. There is some cooperation between different strategies - in case enough memory is available, unit clauses derived during running the previous strategies are kept, in order to cut off literals from newly derived clauses. However, for the CASC-14 competition examples this cooperation was very rarely of any use.

1.5 Implementation and Availability

Gandalf is implemented in Scheme. We are using the *scm* interpreter developed by A. Jaffer and the Scheme-to-C compiler *Hobbit* developed by T. Tammet for the scm system.

The source, binaries, manual and other materials are available at `http://www.cs.chalmers.se/~tammet/gandalf/`.

2 Subsumption: Preliminaries

We use the standard notions of term, literal, unifier, clause and subsumption, see for example Fermüller et al: [1].

A *unit clause* is a clause consisting of a single literal. A term, literal or a clause is *ground* iff it does not contain variables. The *length* of a clause is the number of literals in the clause. The *size* of a term, literal and clause is the number of subterms and literals in it. The *depth* of a literal and a clause is the depth of the deepest term occurring in it.

A literal A *subsumes* a literal B iff there exists a substitution σ such that $A\sigma = B$. A clause C *subsumes* a clause D iff there exists a substitution σ such that $C\sigma \subseteq D$.

The main kinds of subsumption application in a typical resolution prover are *forward* and *backward* subsumption, ordinarily called while processing a newly derived clause.

By *forward subsumption* we mean the process of checking whether any of the input or already derived clauses subsumes the newly derived clause. If yes, then the newly derived clause is eliminated.

By *backward subsumption* we mean the process of eliminating these input and already derived clauses which are subsumed by the newly derived clause.

By *unit* subsumption we mean a special case of subsumption where the subsuming clause is unit.

The Gandalf *given-clause main loop* for inferring and processing clauses keeps two main lists of clauses — *sos* and *usable* — and is similar to most of the other resolution provers (cite from McCune [6]):

```
While (sos is not empty and no refutation has been found)
    1. Let given_clause be the 'lightest' clause in sos;
    2. Move given_clause from sos to usable;
    3. Infer and process new clauses using the inference rules in
           effect; each new clause must have the given_clause as
           one of its parents and members of usable as its other
           parents;  new clauses that pass the retention tests
           are appended to sos;
End of while loop.
```

During the 'process' phase above Gandalf will attempt to forward subsume the clause and if that does not succeed, back subsumption is called. Several clause simplification methods are combined into the forward subsumption process.

Gandalf always generates all the factors of each derived clause. Therefore we prohibit subsumption checks between clauses A and B such that A is longer than B. Should subsumption hold, there will be a shorter factor of A subsuming B.

3 Challenges of Subsumption

The two main reasons why subsumption often takes a large percentage of search time are:

1. Subsumption check has to be called for each newly derived clause C. Each such check must test C against all the existing clauses. Thus the number of clause-to-clause subsumption tests is quadratic to the number of derived clauses. When the latter is large - tens and hundreds of thousands — the square of this number becomes prohibitively large.
2. Checking whether a non-unit clause C of length n subsumes another clause D of length m is a backtracking algorithm which requires m^n literal-to-literal subsumption tests in the worst case. Thus, for long clauses even the time of a single clause subsumption may become prohibitively long.

It is important to consider separately the forward and backward subsumption, as well as unit and nonunit subsumption: the methods for handling these tasks efficiently differ a lot.

The standard way to alleviate problems stemming from the first reason above is the use of special indexing techniques, like discrimination tree indexing and path indexing, see McCune [7].

These indexing methods work very well with forward subsumption, particularly the unit forward subsumption. They are not as good for backward subsumption and they do not give much improvement over the naive linear algorithm in case the average clause length is high.

The standard way to alleviate the problems stemming from the second reason above is to analyse the variable-sharing properties of literals and order the literals accordingly: a literal A which contains a superset of variables in a literal B should be tested before B. This method was proposed and analysed by Leitsch and Gottlob in [3].

4 Importance of Subsumption for Large Non-Horn Problems

When compared to other provers — for example, Otter — Gandalf performs best on large non-Horn clause sets. Most of the industrial verification problems create such sets when converted to the clause form. In mathematics, set theory problems have a similar effect.

For proving non-Horn clause sets it is typically necessary to generate non-unit clauses. The longer they are, the harder the task of subsumption.

Some search strategies, like forward-reasoning hyperresolution, tend to produce relatively short clauses, while others, like backward-reasoning set of support resolution, tend to produce long clauses.

Because most of the existing provers slow down considerably — due to subsumption — when large amounts of long clauses are produced, these provers are preferably used with strategies like hyperresolution, and not with strategies like set of support.

However, for problems which translate into large clause sets — on the order of hundreds of clauses — the forward-reasoning hyperresolution-like strategies are ordinarily a bad choice, since they do not concentrate search on the goal clause(s), as set of support does. However, without fast subsumption, set of support becomes prohibitively slow for large non-Horn clause sets.

Because backward-reasoning tableaux provers do not rely on subsumption to such an extent as the resolution provers do, tableaux systems like SETHEO have been a good choice for finding proofs for large non-Horn clause sets. However, our experience with Gandalf shows that versions of set of support combined with an efficient subsumption procedure are a feasible alternative to the tableaux systems when it comes to proving beforementioned types of problems.

5 Forward Subsumption

5.1 Unit Forward Subsumption

According to McCune [7] the most efficient forward subsumption procedure for unit forward subsumption is obtained by using the full variable-containing discrimination tree. Hence Gandalf keeps all the unit clauses indexed in such a tree and uses the corresponding method for forward subsumption: a newly derived clause $\{L_1, \ldots, L_n\}$ is processed by attempting to forward subsume the literals $L_1, \ldots, L_n$ one after another, with the discrimination tree. No direct clause-to-clause subsumption checks are performed.

Unit Deletion Gandalf combines the following clause simplification method into unit forward subsumption. The discrimination tree leaves contain unit clauses for both negative and positive literals, and in case a literal negative to the checked literal L_i is found at a leaf, the literal L_i is removed from the clause C.

5.2 Nonunit Forward Subsumption

Observe that if the clause D subsumes C, then each literal in D subsumes at least one literal in C.

Several provers combine the discrimination tree with the linear test with a subset of derived clauses. One literal in each nonunit clause is put into the indexed tree. When a new clause C is checked for nonunit subsumption, at first the set S of clauses is retrieved from the tree so that at least one literal in each

clause in S subsumes at least one literal in C. The clauses in S are then checked linearly.

Gandalf takes this idea one step further, indexing on **two** literals from each clause. In principle it is possible to extend the indexing to any n literals, in which case the time spent on linear search will diminish, but the time required for searching the tree will grow very fast as n increases. Our choice of indexing on two literals stems mainly from the following empirical considerations:

- It is usually advantageous to concentrate search on shorter clauses. Hence the number of shorter clauses is likely to be bigger than the number of long clauses. Although this varies a lot, two-literal clauses appear to be fairly common during proof searches.
- The balance between the tree search time and the linear test time appears to be acceptable for most problems when two-literal indexing is used.

Each clause D is ordered according to a certain measure $\succ_s$ described later (the idea is that the $\succ_s$-bigger elements of the clause are less likely to subsume a randomly chosen literal) and the $\succ_s$-first two literals L_1 and L_2 are combined into one *pseudo-literal* $P(L_1, L_2)$ which is then added to the discrimination tree in a standard way.

Hence all the two-literal clauses $\{R_1, R_2\}$ are non-unit forward-subsumed by checking first $P(R_1, R_2)$ and then $P(R_2, R_1)$ against the discrimination tree. No direct clause-to-clause subsumption checks are performed.

For longer clauses $\{R_1, R_2, \ldots, R_n\}$ we *do not* carry on subsumption checking by first checking all clauses of length two, then of length three, etc. Instead we use the previously described indexing on two literals for forming a set of *candidate subsuming clauses* which are later used for clause-to-clause subsumption.

In order to create the list of candidate subsuming clauses we first form pseudo-literals

$$P(R_1, R_2), P(R_1, R_3), \ldots, P(R_1, R_n), \ldots, P(R_n, R_{n-1})$$

representing all ordered pairs of literals in the clause. These pseudo-literals are then checked incrementally for finding clauses which contain a pair of literals subsuming both components of the pseudo-literal. A clause $G : \{G_1, G_2, \ldots, G_m\}$ is a subsumption candidate iff for some substitution σ, some literals G_i and G_j and some pseudo-literal $P(R_u, R_v)$ holds $G_i\sigma = R_u$ and $G_j\sigma = R_v$.

The search for subsumption candidates is organised incrementally, by re-using a path in the discrimination tree for the first component literal. For example, once the path for $P(R_1)$ has been found in the discrimination tree, the found path is used for all of

$$P(R_1, R_2), P(R_1, R_3), \ldots, P(R_1, R_n)$$

instead of re-finding the path for $P(R_1)$ for each pseudo-literal.

Unit Deletion Gandalf combines the following clause simplification method into two-literal forward subsumption. The discrimination tree leaves contain two-literal clauses for both negative and positive literals. In case such a two-literal clause $\{\neg L, R\}$ is found at the leaf that the newly derived clause contains literals L' and R' such that $\{L, R\}$ subsumes $\{L', R'\}$, the literal L' is deleted from the newly derived clause.

Linear Test The set of candidate subsuming clauses contains clauses of length three or more. Once the set of candidate subsuming clauses has been built, they are all tested for subsuming the newly derived clause, using the clause-to-clause subsumption check. The optimised algorithm for this test is presented in the next section.

6 Clause-to-Clause Subsumption

We will first consider the importance of ordering the literals in a suitable way.

Consider the task of checking whether the clause $\{L_1, \ldots, L_n\}$ subsumes the clause $\{R_1, \ldots, R_m\}$. In the general case we need to test all the permutations of literals $L_1, \ldots, L_n$ against the clause $\{R_1, \ldots, R_m\}$. A natural way to do this is to use a backtracking algorithm. First a literal L_1 is matched with $R_1, R_2, \ldots$ until such an R_i is found which is subsumed by L_1, giving a certain substitution σ_1. After that we repeat the same search for $L_2\sigma_1$, etc, until each literal L subsumes a literal R. In case some $L_j\sigma_j$ does not subsume any R_k, the search backtraces to L_{j-1}, attempting to find another R_l subsumed by L_{j-1}, giving a different substitution σ'_j.

The crucial issue here is minimising backtracking. Gottlob and Leitsch [3] suggests the following:

1. In case the literal $L_{j-1}\sigma$ does not contain variables, there is no need to attempt subsuming a different R_l, since the new substitution is empty in any case.
2. Strengthening the previous idea: in case the literal $L_{j-1}\sigma$ does not contain variables *which occur in literals to the right:* $L_j, \ldots, L_n$, there is also no need to attempt subsuming a different R_l.
3. Splitting the clause $\{L_1, \ldots, L_n\}$ into subsets which do not share variables enables analysing these components separately.
4. Ordering the literals in $\{L_1, \ldots, L_n\}$ in such a way that the previous considerations would have maximal effect: literals containing more variables should be tested before literals containing fewer variables. For example, if a literal L_i contains all the variables in $\{L_1, \ldots, L_n\}$, and we test it first, then we only need to retry L_i, never any other literal in the clause.
5. Before full test with backtracking, test each literal in $\{L_1, \ldots, L_n\}$ separately: for each L_i there should be at least one literal R_j which is subsumed by L_i.

The suggestions from Gottlob and Leitsch [3] are implemented in Gandalf with certain pragmatical modifications. The most important of these is an ordering of literals which reflects the probability of a literal subsuming another, randomly picked literal.

6.1 Ordering Methods in Gandalf

Observe that almost all the clause-to-clause subsumption tests during the proof search fail. Hence it is useful to look for the failure first. Hence we first try these literals in $\{L_1, \ldots, L_n\}$ which are less likely to subsume randomly picked literals.

The function *ground*(A) returns 1 if A is ground, 0 otherwise. Functions *size*(A) and *depth*(A) return the size and the depth of the literal, respectively. Function *cnum*(A) returns the number of occurrences of constants in A.

The ordering $A \succ_s B$ is defined in the following way: A and B are compared according to *ground*, *depth*, *size*, *cnum*, in that order. In case any comparison gives a bigger value for A than B, then $A \succ_s B$. In case any comparison gives a bigger value for B than A, then $B \succ_s A$. If the values are equal, the next comparison function is taken.

The proof of the following lemma is easy:

Lemma 1. *If $A \succ_s B$, then A cannot subsume B.*

The main issue is having an ordering which contains many independent comparison functions and satisfies the lemma. We do not claim that this particular order $\succ_s$ is statistically much better than other, similar orders.

The order $\succ_s$ is used in Gandalf in several ways. First, it is used to determine which two literals in the clause should be indexed. Second, it is used for ordering literals in a clause before the clause-to-clause subsumption checks.

We have not implemented splitting the clause into components, as suggested by Gottlob and Leitsch [3], for the reason that most clauses derived during the search typically cannot be split into several non-ground components. The ordering realises the splitting effects for the ground and ground/non-ground components. We have not implemented ordering by variable occurrences either. Instead we prefer larger and deeper literals, which statistically tend to contain more variables than smaller and shallower. We can say that the variable-occurrence ordering suggested in Gottlob and Leitsch [3] is *approximated* by our choice of $\succ_s$, with the latter taking additional considerations into account too.

6.2 Pre-computing the Values

We avoid all costly computations during subsumption. The values of functions *ground*, *depth*, *size* are *cnum* are pre-computed when a clause is stored and saved as consecutive bit fields into a special 4-byte integer in the representation of a literal, containing also the name of the leading predicate. Clauses are presorted according to the ordering.

In order to keep track of whether a literal contains variables occurring also in the following literals, each variable occurrence in a literal is decorated with a special data bit, indicating whether there are any later occurrences.

6.3 Hierarchical Filters

Before a full clause-to-clause subsumption test of B with A is performed, a number of fast checks are performed. In most cases these fast checks establish immediately that A cannot subsume B.

As said before, each literal is decorated with the values of *ground*, *depth*, *size* and *cnum*. Similarly, the whole clause and each term is also decorated with these values for the corresponding object.

The set of predicate names occurring in a literal is encoded as a bit string in an integer. This enables very fast checking (using bitwise machine operations) of whether the set of predicate names in A is a subset of predicate names in B.

The hiearchy of tests performed while checking subsumption of B by A is the following (failure of any test causes failure of the whole test):

1. Is A shorter or of equal length to B?
2. Is it the case that $A \succ_s B$ does not hold?
3. Is it the case that $depth(A) \leq depth(B)$, $size(A) \leq size(B)$ and $const(A) \leq const(B)$?
4. Bitwise check: is the set of predicate names in A a subset of predicate names in B?
5. Is it the case that each literal in A subsumes at least one literal in B?
6. Full test: does A subsume B?

The analogues of steps 2 and 3 are used not only before the full subsumption check of clauses, but also before the full subsumption check of literals and all terms. For example, it is always very quickly determined that a ground term cannot subsume a non-ground term, a deep term cannot subsume a shallow term, etc.

7 Back Subsumption

Simple discrimination trees cannot be used for performing the back-subsumption operation efficiently. Thus several alternative indexing methods have been proposed in the literature, see Graf [2]. However, these methods are significantly less efficient than full variable-containing decision trees for forward subsumption. Back subsumption has degraded the performance of many otherwise highly efficient provers.

Gandalf does not use any indexing methods for back subsumption: it uses simple linear search, but tests only a very small percent of existing clauses for back subsumption. Our experiments show that the Gandalf back-subsumption has excellent efficiency. It it is unclear whether indexing methods are at all superior to the Gandalf-style simple linear back subsumption.

Gandalf keeps the list of existing clauses for back subsumption sorted under clause length and the ordering $\succ_s$. Only these clauses are considered for back subsumption for which none of the parameters *length*, *ground*, *depth* and *size* is less than the corresponding parameters of the newly kept clause A.

When back subsumption with A reaches a clause C such that C is shorter than A, none of the following clauses can be back subsumed and the whole back subsumption process is stopped. Similarly, when checking clauses with a certain length l and a clause C such that $A \succ_s C$ is reached, none of the following clauses with length l can be back subsumed, thus all the following clauses with length l are skipped.

Another important restriction is that **only the clauses in the usable list** are back subsumed. Indeed, since clauses in the sos list do not participate in ordinary resolution steps (they may participate in simplification, demodulation and subsumption steps) not much is gained by eliminating some of them with back subsumption. Clauses in the usable list, on the contrary, participate in ordinary resolution and paramodulation steps, hence eliminating some of them may give a noticeable gain in efficiency.

Because of this restriction we separately check each selected clause in sos for forward subsumption before it is moved to usable: it may be subsumed by a clause derived after this selected clause was derived. Since the operation of selecting a new clause is rare, the extra overhead of a forward subsumption check is neglible.

The motivation for the used scheme of back subsumption is the following. Since the problem of deriving an empty clause is undecidable, statistically the average size of the derived clause is growing during the derivation process. However, a newly kept clause can only subsume these of the existing clauses which are not bigger than the newly kept clause. Since it is likely that most of the older clauses are smaller than the newly kept clause, only a small fraction of the old clauses has to be checked. This motivation is further strengthened by the fact that the sos list is normally much larger than the usable list and we only need to back subsume the usable list.

8 Statistics

In order to give some evidence of the efficiency of the proposed methods, we have chosen to present statistics for the problems posed in the no-equality, non-Horn category of the prover competition CASC-14.

The no-equality, non-Horn category is selected since except for back subsumption, the methods presented in the paper are suitable for non-unit subsumption. The problems in the selected category produce large amounts of non-unit clauses and the efficiency of proving them does not rely on factors like efficient paramodulation and demodulation.

First we bring the table with the competition results for the mentioned category. While the speed of nonunit subsumption is certainly not the only important factor for the overall result — notably, SPASS was successful since it derived very few clauses to start with — together with time slicing it was certainly one of the main factors for the success of Gandalf.

Non-Horn with No Equality Category

Problem	Allpaths	Gandalf	I-THOP	Otter	SCOTT	SETHEO	SPASS	TGTP
SET014-2	76.2	15.5	4.2	TO	TO	1.1	0.2	1.0
SET015-2	TO	60.5	TO	TO	TO	TO	148.2	TO
SET013-1	3.0	TO	15.3	TO	TO	TO	81.0	64.1
SET015-1	1.1	30.5	4.7	TO	TO	1.1	64.1	85.1
SET007-1	TO	TO	25.5	TO	TO	57.9	0.7	3.6
SET012-2	9.6	46.0	TO	TO	TO	TO	20.2	11.8
SET011-1	5.6	6.6	4.5	129.1	250.2	1.3	0.0	1.4
SET055-6	3.0	0.1	13.8	0.4	8.7	5.2	0.5	1.8
SET013-2	TO	50.9	TO	TO	TO	TO	TO	TO
ANA002-2	TO	121.4	TO	TO	TO	36.4	TO	TO
SET005-1	78.1	30.3	3.9	281.9	6.7	1.8	0.1	1.3
SET012-1	0.5	30.4	14.0	TO	TO	4.3	2.1	3.8
Attempted	12	12	12	12	12	12	12	12
Solved	8	10	8	3	3	8	10	9
Time	177.1	392.2	85.9	411.4	265.6	109.1	317.1	173.9
Average	22.1	39.2	10.7	137.1	88.5	13.6	31.7	19.3

The abbreviation `TO` used in the table stands for "timeout".

In the following tables we bring subsumption statistics for the problems in the selected category. The timings and statistics are obtained in a later run than the competition table above. In particular, Gandalf proved successfully (in 18 seconds) the problem SET013-1.

While searching for proofs, Gandalf uses several different search strategies, for example hyperresolution and set of support resolution. Thus the statistics do not depend on one specific strategy, but rather a combination of strategies.

We'd like to turn attention to the ratio of F. full (the number of full, backtracking clause-to-clause subsumption checks performed during forward subsumption) and F. fail (the number of full checks which fail). This indicates that the combination of subsumption candidate selection using the discrimination tree and the fast filtration steps performed during clause-to-clause subsumption is quite precise: approximately five percent of full clause-to-clause subsumption checks succeed.

Also, the number B. full (full backtracking clause-to-clause subsumption checks during back subsumption) is significantly smaller than the corresponding number F. full for forward subsumption.

Explanation of the fields:

- Given: number of given clauses
- Derived: number of derived clauses
- Kept: number of kept clauses
- F. unit: number of forward subsumed unit clauses
- F. double: number of forward subsumed two-literal clauses
- F. long: number of forward subsumed clauses of length three and more
- F. tried: number of clause-to-clause subsumption checks during forward subsumption.
- F. full: number of remaining clause-to-clause forward subsumption checks after fast filters have been passed
- F. fail: number of these remaining clause-to-clause forward subsumption checks (after fast filters have been passed) which failed
- B. full: number of remaining clause-to-clause back subsumption checks after fast filters have been passed

SET014-2

Given	143	Derived	24734	Kept	10154
F. unit	2863	F. double	8309	F. long	2175
F. tried	24828	F. full	12471	F. fail	10208
B. full	2761				

SET015-2

Given	2958	Derived	75368	Kept	20510
F. unit	28790	F. double	14866	F. long	2723
F. tried	65742	F. full	30876	F. fail	28126
B. full	7934				

SET013-1

Given	1366	Derived	14466	Kept	2646
F. unit	1111	F. double	1396	F. long	7377
F. tried	826818	F. full	168118	F. fail	160530
B. full	10011				

SET015-1

Given	5268	Derived	170938	Kept	38451
F. unit	51876	F. double	23190	F. long	24783
F. tried	558302	F. full	412754	F. fail	386724
B. full	27006				

SET007-1

Given	9360	Derived	169975	Kept	30028
F. unit	17578	F. double	49904	F. long	41520
F. tried	3312621	F. full	847768	F. fail	807156
B. full	11027				

SET012-2

Given	2311	Derived	63072	Kept	14775
F. unit	28542	F. double	8984	F. long	1258
F. tried	26489	F. full	12773	F. fail	11471
B. full	7105				

SET011-1

Given	91	Derived	5393	Kept	2695
F. unit	19	F. double	759	F. long	986
F. tried	25610	F. full	9726	F. fail	8541
B. full	2433				

SET055-6

Given	3	Derived	7	Kept	5
F. unit	1	F. double	0	F. long	0
F. tried	0	F. full	0	F. fail	0
B. full	0				

SET013-2

Given	2970	Derived	76086	Kept	20835
F. unit	30332	F. double	15601	F. long	1492
F. tried	43519	F. full	18268	F. fail	16719
B. full	8180				

ANA002-2

Given	4737	Derived	79270	Kept	32779
F. unit	15915	F. double	18260	F. long	10355
F. tried	876717	F. full	246665	F. fail	237277
B. full	7417				

SET005-1

Given	1374	Derived	8152	Kept	3280
F. unit	274	F. double	2654	F. long	1726
F. tried	275975	F. full	117433	F. fail	115824
B. full	10670				

SET012-1

Given	1716	Derived	18885	Kept	4958
F. unit	3169	F. double	4325	F. long	5382
F. tried	293373	F. full	81529	F. fail	75858
B. full	35926				

9 Acknowledgement

This work is supported by the Swedish TFR grant Dnr 96-536.

References

1. C. Fermüller, A. Leitsch, T. Tammet, N. Zamov. Resolution methods for decision problems. *Lecture Notes in Artificial Intelligence vol. 679, Springer Verlag*, 1993.
2. P. Graf. Term Indexing. *Lecture Notes in Computer Science. 1053, Springer Verlag*, 1996.
3. G. Gottlob, A. Leitsch. On the efficiency of subsumption algorithms, *Journa of ACM* **32**(2):280-295, April 1985.
4. L. Magnusson, B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, pages 213-237, *Lecture Notes in Computer Science vol. 806, Springer Verlag*, 1994.
5. G.Mints. Resolution Calculus for The First Order Linear Logic. *Journal of Logic, Language and Information*, **2**, 58-93 (1993).
6. W.McCune. OTTER 3.0 Reference Manual and Users Guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, IL, January 1994.
7. W. McCune. Experiments with discrimination tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, **9**(2):147–167, 1992.
8. T. Tammet. Completeness of Resolution for Definite Answers. *Journal of Logic and Computation*, (1995), vol 4 nr 5, 449-471.
9. T. Tammet. A Resolution Theorem Prover for Intuitionistic Logic. In *CADE-13*, pages 2-16, *Lecture Notes in Computer Science vol. 1104, Springer Verlag*, 1996.
10. T. Tammet, J. Smith. Optimised Encodings of Fragments of Type Theory in First Order Logic. In *Types for Proofs and Programs*, pages 265-287, *Lecture Notes in Computer Science vol. 1158, Springer Verlag*,1996.
11. T. Tammet. Gandalf. *Journal of Automated Reasoning*, **18**(2): 199-204, 1997.