Reference Manual

GANDALF

version c-1.0c

by

Tanel Tammet

October 1997

Department of Computing Science
University of Göteborg / Chalmers Univ. of Technology
S-412 96 Göteborg, Sweden

This work was supported by TFR Dnr 96-536

Contents

Gandalf Reference Manual

version c-1.0c

by

Tanel Tammet

tammet@cs.chalmers.se

Abstract

Gandalf is a resolution theorem prover for first order classical logic with equality. Gandalf is also used as a name for a family of theorem provers, currently including versions for classical 1-st order logic, intuitionistic 1-st order logic, propositional linear logic and a subset of Martin Löf's type theory.

The current manual concers the version of Gandalf for classical first order logic, which won the yearly ATP competition CASC in 1997.

Gandalf has a powerful automatic mode and, differently from most of the other existing provers, it is well optimised for handling problems where big amounts of long clauses are derived.

The WWW home page of Gandalf is
`www.cs.chalmers.se/~tammet/gandalf/`

1 Introduction

Gandalf is used as a name for a family of theorem provers, currently including versions for classical 1-st order logic, intuitionistic 1-st order logic, propositional linear logic and a subset of Martin Löf's type theory. These provers share large parts of their code.

The current manual concers the version of Gandalf for classical first order logic.

Gandalf implements a large number of various search strategies. The usage of these strategies can be either controlled by the human user or by the powerful automatic mode of Gandalf. The automatic mode first selects a set of different strategies which are likely to be useful for a given problem and then tries all these strategies one after another. Gandalf is also well optimised for handling problems where big amounts of *long* clauses are derived.

Gandalf is not interactive - it reads in a file containing the problem, outputs information about the progress of proof search and eventually outputs a detailed proof.

Gandalf is written in Scheme and compiled to C by the Scheme-to-C compiler Hobbit developed by the author of Gandalf. Gandalf has been ported to a number of UNIX, MS-DOS and Windows platforms.

The executable of a compiled Gandalf contains parts of the Scheme interpeter scm developed by Aubrey Jaffer, which are covered by the Gnu Public Licence (GPL). The compiled executables of Gandalf are currently thus also covered by the GPL. The copyright of the Gandalf source belongs to Tanel Tammet, but the source is otherwise completely free for use, modification and distribution.

New versions of Gandalf, manual, papers, example input files, related pointers etc can be found on the Gandalf home page `www.cs.chalmers.se/~tammet/gandalf/`.

1.1 Implemented strategies

The following basic strategies are implemented in Gandalf:

- set-of-support resolution
- binary resolution
- unit resolution
- hyperresolution
- paramodulation for equalities
- forward- and back demodulation for equalities
- methods for ordering literals in a clause, yielding several ordered resolution strategies
- methods for directing equalities
- tautology elimination, forward and backward subsumption
- selectable limits on clause length, term depth, etc.
- special methods for 'endgame search' applied when the time limit is nearing an end – f. ex. backward search in the hyperresolution case
- autonomous mode, selecting and trying a number of strategies one after another

Most of the strategies above can be combined, possibly resulting in an incomplete strategy.

Gandalf has a powerful autonomous mode. In this mode Gandalf first checks whether a clause set has certain properties, then selects a set of different strategies which are likely to be useful for a given problem and then tries all these strategies one after another, allocating less time for highly specialised and incomplete strategies and more time for general and complete strategies. For the autonomous mode to be efficient it is crucial that the user indicates how much time is given to Gandalf to search for the proof.

Despite the autonomous mode, a lot of work with Gandalf involves interaction with the user. After the user has encoded a problem into clauses, the user can choose inference rules, set options to control the processing of inferred clauses, and decide which input clauses are to be in the initial set of support.

1.2 Other versions of Gandalf

There exist versions of Gandalf for type theory [5], intuitionistic logic [4] and propositional linear logic [6]. Check the home page of Gandalf for these versions. Not all of these versions are necessarily available on the net.

In particular, the constructive type theory version of Gandalf contains a module for automated induction. It is likely that the future versions of Gandalf for classical logic will also contain modules for automated induction.

Gandalf is not directly targeted toward synthesizing or verifying formal hardware or software systems. However, work on versions of Gandalf directed toward verification and synthesis is going on (see [3]). Check the Gandalf home page or contact tammet@cs.chalmers.se

1.3 Why "Gandalf"

There is a number of converging reasons for using the name 'Gandalf':

- The hacker's dictionary mentions the term 'automagical', which clearly covers what automated theorem provers do. Gandalf is originally used as a name of the powerful wizard in the famous Tolkien books "Hobbit" and "The Lord of the Rings".
- Author compiles the Scheme source of Gandalf with his compiler "Hobbit" (one of the first Scheme compilers was called "Rabbit").
- The earliest version of Gandalf was used for Martin-Löf's type theory. The systems devloped at Chalmers for handling the latter are called ALF (Another Logical Framework), HALF (Haskell ALF), etc.

1.4 Performance

Due to the large number of various strategies implemented and the autonomous mode, Gandalf performs reasonably well on most of the first order problem categories.

Compared to other provers, it performs best in cases where a large number of long clauses are derived, ie. non-horn clause sets. It is not specially optimised for Horn clauses or pure equational problems. For these two categories the provers like Otter and Waldmeister are likely to outperform Gandalf, although not by a wide margin.

Gandalf is not optimised for purely propositional problems.

An earlier version of Gandalf participated in CASC-13 (1996) and lost in the mixed category both to Otter and Spass.

The current version of Gandalf participated in CASC-14 (1997) and won the general (mixed) division, proving 56 of the presented 72 problems. The second and third runner SPASS and Otter proved 51 and 50 problems, respectively. Gandalf also participated in the pure equality division, where it ranked third (47 of 50 problems) after Waldmeister (49 of 50 problems) and Otter (48 of 50 problems).

The main difference between the versions used in CASC-13 and CASC-14 is the new autonomous mode.

1.5 History

The author has earlier implemented an inverse method prover (in Standard Lisp) and a resolution prover 'Resolve' (in muLisp), which are not supported any more and which have no direct relations with Gandalf.

The work on Gandalf has been going on from 1994. It has been heavily influenced by the intutionistic resolution version of Gandalf [4] which shares a large part of code with the classical version of Gandalf.

The input-output syntax is essentially (though not exactly) the same as that of the simplest and most widely used form in Otter [7]. This choice is motivated by the popularity and wide use of Otter. Several design aspects of Gandalf (eg the given-clause algorithm, indexing for forward subsumption and demodulation) are also inspired by Otter.

The only erlier official release of a prover from the Gandalf family is a prover for propositonal linear logic, incorporating both the tableau and the resolution prover. However, this prover shares very little code with the other versions of Gandalf.

There have been no earlier official public releases of classical or intuitionistic Gandalf. Executables and the scheme source of the classical version have been made available only during the prover competitions CASC-13 and CASC-14. The executable and source c-1.0b which was used and made available in CASC-14 is the same as the current version c-1.0c, with the single difference: by default Gandalf now stops running once the allocated time has come to an end. There is a new flag `set(run_forever)` which makes Gandalf run forever, behaving exactly like the version c-1.0b.

1.6 Future improvements

Work on improving Gandalf is proceeding in four main directions:

- Improved user interface.
- Incorporating reduction strategies, insipired by some of the strategies used in SPASS and the research of the author in ordering strategies [1].
- Devising special strategies for synthesis (see [3]) and verification.
- Incorporating modules for automatic induction, some of them already present in the type theory version [5].

2 Outline of Gandalf's Inference Process

Once a strategy has been decided upon, either by the aunomous mode or by the user, Gandalf starts making inferences, trying to derive an empty clause.

The basic inference mechanism of Gandalf is the widely used *given-clause algorithm*, essentially the same as used in Otter [7], for example.

Gandalf maintains four main lists of clauses:

`usable.` This list contains clauses that are available to make inferences.

`sos.` Clauses in list *sos* (set of support) are not available to make inferences; they are waiting to participate in the search.

`demodulators`. These are oriented equalities that are used as rules to rewrite newly inferred clauses.

`cutters`. Unit clauses which are used for simplifying clauses by cutting off literals from a clause. This list can contain clauses from previous runs in the autonomous mode of Gandalf.

The *main loop* for inferring and processing clauses is exactly the same as that of Otter (cite from [7]):

```
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.
```

Step 1 details: the 'lightest' clause is selected five times, then the first clause in the `sos` queue is selected once, then the 'lightest' is selected five times again, etc. This implements a combination of best-first and breadth-first search.

The set of support strategy requires the user to partition the input clauses into two sets: those with support and those without. For each inference, at least one of the parents must have support. Retained inferences receive support. In other words, no inferences are made in which all parents are nonsupported input clauses.

3 Obtaining Gandalf

You can get Gandalf and the related materials from the Gandalf home page on the web. Gandalf comes in several different forms:

- Ready-built executables for Sun Sparc, DEC Alpha, PC under NetBSD, PC under MS-DOS/Windows etc
- C source for UNIX
- C source for MS-DOS/Windows
- Scheme source together with a (C source) distribution of the Scheme interpreter scm4e2 by Aubrey Jaffer and the (Scheme source) Scheme-to-C compiler Hobbit by Tanel Tammet.

There is a useful README file included in each distribution.

In case an executable is provided for your architecture, we suggest fetching the executable and separate documentation and example archives, avoiding the source.

In case there is no executable for your system (or you want to hack the C source), fetch the C source. The C source for MS-DOS/Windows is best compiled with the DJGPP version 2 or later (the port of gcc).

In case you are unable to compile the C source, try modifying the Makefile – it includes several options (good for DEC cc on Alpha, gcc on NetBSD, new Sun

cc compiler) which are commented out by default. If that does not work, try to build the Scheme interpreter scmlit first (see the following paragraph) - Gandalf uses several .o files of the interpreter, and the Gandalf C source proper is likely to compile easily.

If you have major trouble compiling the C source (or you want to hack the actual Scheme source), fetch the Scheme source together with the interpreter and a Scheme-to-C compiler. You can also get the new versions of the interpreter separately from `www-swiss.ai.mit.edu/~jaffer/`. However, for compiling Gandalf we suggest using the older version scm4e2 distributed on the Gandalf home page. The Hobbit compiler produces one large C file as output, and unless you split it up yourself, it is unlikely that you can compile it with any other compiler than gcc. You can also run Gandalf under the interpreter, what makes it ca 50 times slower, or compile it with a compiler different from Hobbit, which is likely to produce slower code, since Hobbit and Gandalf are optimised for each other.

4 Running Gandalf

The name of the gandalf executable has to be seven characters long. In addition to the executable Gandalf also requires an initialisation file `gandalf.ini` to reside in the same directory as gandalf.

Each time gandalf tries to prove a problem, it creates a temporary file `gandalf.tmp` in the directory where the gandalf executable resides. It is possible to change the name and location of this temporary file by changing the file `gandalf.ini` (look for `gandalf.tmp` in `gandalf.ini`).

Gandalf takes one argument – the name of the file to be proved For example: `gandalf myprobl.in`

Gandalf can be killed by pressing *ctrl-c* or (except under DOS/Windows) issuing the signal SIGXCPU (signal 30): `kill -30 ...`

If (and only if) Gandalf finds a proof, then its output contains a substring `EMPTY CLAUSE FOUND`

Gandalf is not interactive. It takes an input file name as an argument and writes to the standard output:

`gandalf` *input-file* `>` *output-file*

No command-line options are accepted; all options are given in the input file.

5 Syntax

Gandalf requires that the input problems are in the clause form.

Gandalf does not recognise formula syntax and it does not recognize propositional variables.

Gandalf uses the simplest form of Otter syntax for input and proof output (see earlier section "History") with minor modifications. Therefore we use a few selected parts of the Otter manual in the current section. See
`www.mcs.anl.gov/home/mccune/ar/otter/` for the Otter manual.

The problem has to be in the clause form, terms and atoms written in the prefix form. There are no infix operators. The only special and reserved symbols are `equal`, `ans`, `=`, `!=`, `if`, `case`.

5.1 Preparing TPTP problem files

If you are using TPTP (for TPTP see `www.cs.jcu.edu.au/~tptp/`), use the following command for converting the problems to the suitable form for gandalf:

```
tptp2X -t rm_equality:stfp -f otter:hypothesis:'set(auto),assign(max_seconds,180)'
```

where instead of 180 use the actual amount of seconds given for one problem. However, neither the `assign(max_seconds,...)` nor `set(auto)` parts are obligatory. Neither is `hypothesis:`, although it is strongly recommended.

5.2 Incorrect syntax

There is only minimal control for the correctness of syntax in Gandalf c-1.0c.

In case you use wrong syntax, Gandalf will probably crash with an uninformative error message. Careful syntax checks will be incorporated in the later versions.

5.3 Comments

Comments can be placed in the input file by using the symbol `%`. All characters from the first `%` on a line to the end of the line are ignored.

5.4 Names for Symbols

Names for Predicates and Functions

The name of a predicate or a function symbol is a string of alphanumeric characters and `_` which starts with either a character or `_`. Upper- and lower case characters are distinguished.

Names for Variables and Constants

The name of a variable or a constant is a string of alphanumeric characters and `_`. Upper- and lower case characters are distinguished.

If the first symbol of a name is a digit, the rest has to contain digits only. For example, `235` is OK, `1ab` is not).

The first symbol determines whether a name represents a variable or a constant.

By default a name represents a variable if it starts with `u`, `v`, `w`, `x`, `y`, or `z`. Otherwise a name represents a constant. For example, `x1` is a variable, `a1` is a constant.

If the flag `prolog_style_variables` is set, a name represents a variable if and only if it starts with an upper-case letter. For example, `X1` is a variable, `x1` is a constant.

Reserved and Built-in Names

The name `equal` in the prefix position is recognized as an equality predicate.

The names `ans`, `if`, `case`, `=` and `!=` are reserved and *should not be used!*. Some of them are used in the experimental, yet unfinished search techniques built into Gandalf.

Overloaded Symbols

The user can use a name for more than one purpose: the same name can be used as a function symbol, predicate symbol, variable or a constant. It is not allowed to use the same name for an n-ary function symbol and also for an m-ary function symbol (or predicate).

5.5 Term and Clause Syntax

Terms and Atoms

Constants and variables are terms. An n-ary function symbol applied to n terms is also a term. Zero-ary function and predicate syntax (eg `f()`) is not allowed. An atom is a term which does not occur inside another term. Propositional atoms like `p` are not allowed.

The way to write complex terms and atoms is the following: the function or predicate symbol, opening parenthesis, arguments separated by commas, then closing parenthesis, for example, `f(a,g(b),c)` and `equal(f(x,e),x)`.

Literals and Clauses

A literal is either an atom or the negation of an atom. A clause is a disjunction of literals. The built-in symbols for negation and disjunction are `-` and `|`, respectively.

Clauses are written in infix form and terminated with a period. Two example clauses:

```
-p(a) | r(x,y) | equal(c,d).
p(f(a)).
```

6 Commands and the Input File

Input to Gandalf consists of a small set of commands, some of which indicate that a list of clauses follows the command.

All lists of objects are terminated with `end_of_list`.

The commands are given in the following table:

Table 1: Commands

`set(`*flag_name*`).`	`% set a flag`
`clear(`*flag_name*`).`	`% clear a flag`
`assign(`*parameter_name*`,`*integer*`).`	`% assign an integer to a parameter`
`list(`*list_name*`).`	`% read a list of clauses`

6.1 Input of Options

Gandalf recognizes flags and parameters. Flags are Boolean-valued options; they are changed with the `set` and the `clear` commands, which take the name of the flag as the argument.

Parameters are integer-valued options; they are changed with the `assign` command, which takes the name of the parameter as the first argument and an integer as the second. Examples are

```
set(binary_res).              % enable binary resolution
clear(demodulate).            % do not use demodulation
assign(max_seconds, 300).     % stop after about 300 CPU seconds
```

The options are described later.

6.2 Input of Lists of Clauses

A list of clauses is specified with one of the following and is terminated with `end_of_list`. Each clause is terminated with a period.

```
list(usable).
list(sos).
```

Example:

```
list(usable).

p(a).
equal(g(x),x) | s(x,y) | r(f(y)).
-p(g(a)) | r(f(y)).
-s(a,b).

end_of_list.

list(sos).

-r(f(b)).

end_of_list.
```

The sos (set of support) and usable lists should be used for splitting the input clauses between sos and the rest. It is assumed that the usable list of clauses is satisfiable.

Typically either the proved theorem or the theorem and hypothesis are put into `sos`, while the axioms of the theory are put into `usable`.

In case the input clauses are not split in such a way, the automatic mode will split the clauses itself while using sos-based strategies: fully negative clauses will be in `sos` and the rest will be in `usable`.

Despite the splitting into `sos` and `usable`, the non-sos-based strategies of the automatic mode will not use this splitting.

In a non-automatic mode Gandalf relies on the splitting provided in the input file and will not attempt to split the clause set itself. However, in case only the usable list is present in the input file and the sos list is lacking, Gandalf will treat the usable list as the sos list.

7 Options

Options are used for selecting strategies, input/output formats, limits, etc. Despite using the Otter syntax for options, most of the options Gandalf uses are different from Otter.

Flags are Boolean-valued options, and parameters are integer-valued options. When the user changes an option, Gandalf sometimes automatically changes other options.

In case Gandalf c-1.0c does not recognise an option, it will simply ignore it. The future versions should issue warning messages in such a case.

A subset of options is usable both in the autonomous mode and the user-controlled mode, while the rest are usable only in the user-controlled mode.

7.1 Generally Usable Options

`set(auto).` – automatic mode. Default: on. Redundant. Cleared by selecting a user-controlled strategy, see following sections.

7.1.1 Time options

`assign(max_seconds, <nrofseconds>).` – enables Gandalf to optimize for this time limit – both in the autonomous mode and for the user selected strategies. In the latter case Gandalf uses special 'endgame search' when time starts running out. Gandalf will also stop after that limit, unless `run_forever` is set. By default processor time, not wall-clock time is used. Default: 180 seconds

`set(run_forever).` – causes Gandalf not to stop after the assigned time is over. However, using `assign(max_seconds,N)` is still useful, since it causes optimisation for the assigned time. Default: off.

`set(wall_time).` – causes Gandalf to use wall-clock time, not processor time. Default: off.

`assign(checkpoints, <listofcheckpoints>).` – use the checkpoint list output by the prover. Overrides all timing choices and simulates the previous search with the given timer checkpoints. See the section "Outline of the Autonomous Mode". Default: none.

7.1.2 Memory control options

`assign(max_kept, <nr>).` – maximal nr of kept clauses. Default: 20000.

7.1.3 I/O options

`assign(print_level, <nr>).` – a level of output (between 0 and 100). Default: 40. Meaning:

- 20 – derivation found/not
- 30 – derivation shown
- 35 – statistics shown
- 38 – strategy changes shown
- 40 – input clauses in final form
- 50 – given clauses
- 60 – kept clauses
- 63 – back subsumption
- 65 – cutoff literals and clauses
- 70 – raw clauses
- 80 – removal reasons
- 90 – usable clauses

`set(print_tracechar).` – prints a special indication char upon various operations. Default: off.

`set(prolog_style_variables).` – use prolog style variables (names starting with an upper-case letter), in contrast to otter-style vars (names starting with x,y,z,u,v,w) Default: off.

`set(print_debug).` – display debugging information useful for the developer only. Default: off.

7.2 Options for User Controlled Strategies

7.2.1 Strategy options

When these options are used, Gandalf will not be in the autonomous mode. The strategy options are mutually exclusive: do not use more than one of these.

`set(hyper_res).` – hyperresolution. Default: off.

`set(hyper_order_res).` – hyperresolution with ordered positive clauses. Default: off.

`set(binary_res).` unrestricted binary resolution. Default: off.

`set(binary_unit_res).` – unit binary resolution. Default: off.

`set(binary_order_res).` – ordered binary resolution: order based on variable occurrence depths. Default: off.

`set(binary_nameorder_res).` – ordered binary resolution: order based on predicate names, whereas positive literals are preferred (ie backward reasoning). Default: off.

`set(binary_weightorder_res).` – ordered binary resolution: order based on literal sizes. Default: off.

7.2.2 Demodulating and orderings

Use these options with the user controlled srategies only.

`clear(demodulate).` – disables demodulation. Default: demodulation on.

`set(back_demodulate).` – enables both forward and backward demodulation. Default: off.

`set(eq_order_depth).` – order equalities using variable and term depth. Default: on.

`set(eq_order_lex).` – order equalities using a strange and probably bad version of lexicographic tree ordering. Default: off.

7.2.3 Clause size limits

Use these options with the user controlled strategies only.

`assign(max_literals, <nr>).` – maximal nr of literals in a kept clause. Default: no limit.

`assign(max_depth, <nr>).` – maximal term depth of a kept clause. Default: no limit.

`assign(max_weight, <nr>).` – maximal size (amount of subterms, incl. constants and variables) of a kept clause. Default: no limit.

8 Output and the Proof

The desired amount of output can be selected with the `assign(print_level, <nr>)` option, described in the previous section. In general, output levels higher than 50 produce huge amounts of hard-to-understand data, and are thus not recommended for normal use. Output levels under 20 do not indicate whether a proof was found and are thus also not recommended (in practice they are reserved for specially modified versions and running under the interpreter).

The proof format loosely follows the Otter proof format. Each line in a proof contains three parts: the clause number, the list indicating how the clause was derived (derivation list), the actual clause.

In case the derivation list is empty, the clause is an axiom. Otherwise the first element of the derivation list is the type of the derivation rule used. There are four possibilities: `binary, hyper, para, factor`. The following numbers indicate the parent clauses.

After the list of parent clause numbers in the derivation list there can occur symbols indicating which simplifications were applied to the initially derived clause.

There are four possibilities: `binary_s, hyper_s, factor_s, demod`. `binary_s` indicates that a literal was cut off using the clause with the number following `binary_s`. The cutting clause can be either a unit clause or a binary clause. `hyper_s` and `factor_s` indicate byperresolution and factoring simplifications, respectively. `demod` indicates that demodulation was used with the clauses with numbers following `demod`.

A number of the clause in the derivation list often contains additional information, separated from the number proper with the period symbol. The first number after period indicates which literal in the clause was used. If that number is lacking, the first literal was used.

For paramodulation it is also indicated which subterm was paramodulated into. First the path to the unified term in equality is presented by period-separated numbers, indicating which element of the currently selected subterm has to be taken next. Symbol `#t` indicates that the left argument of the equality was used for paramodulation, while `#f` indicates that the right argument of the equality was used for paramodulation. After that comes the path to the subterm which was paramodulated into, presented as a space-separated list in parentheses.

9 Correctness, Completeness, Bugs

Gandalf has been carefully tested for correctness, hence it is likely that it won't give a 'derivation' where none really exists.

However, we **give no guarantees whatsoever** considering the correctness. Gandalf *might* contain bugs making it incorrect for certain input files. In case you observe incorrectness (Gandalf 'refuting' an unsatisfiable clause set), please report to the author: `tammet@cs.chalmers.se`.

Completeness (in the sense: would Gandalf find a proof, given an unlimited amount of time and memory) has not been checked as carefully.

Certain special strategies of Gandalf are incomplete on purpose. For example, in the general case all combinations of the set of support strategy and some strategy different from unrestricted binary resolution are incomplete.

Considering pure unrestricted strategies it is likely (although we give no guarantees) that the implementations of binary resolution and hyper-resolution are complete, as well as the set of support strategy and unrestricted binary resolution.

It is also likely that the autonomous mode is complete.

Considering combinations with paramodulation and demodulation (Gandalf uses paramodulation automatically when it detects that equality is present), it is likely that the combination with an unrestricted binary resolution is complete.

There is a known bug: the combination of hyperresolution with paramodulation is incomplete, due to the fact that in the hyperresolution context paramodulation is only applied to positive clauses. However, this incompleteness is not likely to surface very often.

Gandalf might contain all kinds of bugs. In case you observe a serious bug, except crashes etc caused by the incorrect syntax of the input file, please report to the author: `tammet@cs.chalmers.se`.

10 Outline of the Autonomous Mode

The autonomous mode contains the following steps:

1. The input clause set is analysed.
2. A set of strategies along with the percentage of time allocated is selected, based on the previous step.
3. The selected strategies are run one after another until either the proof is found, memory is filled up or time runs out.

Before Gandalf starts proof search it outputs the list of selected strategies. Each element of the list has the following format: *(base-strategy seconds sos-used-flag term-depth-limit clause-length-limit)*. The latter two may be lacking, meaning that no limits are imposed.

There is some cooperation between the separate runs: unit clauses produced during the old runs are used for simplifying clauses by cutting off literals. However, this feature has so far been rarely useful in practice.

The clause numbering is global in the sense that a new run does not start from the number one again.

The user-imposed limit on the number of kept clauses (default 20000) is local in the sense that a single run may not keep more than the limited amount of clauses. When a new run is started, the old clauses (except the unit clauses) are discarded. The old unit clauses are discarded only in case memory starts running out.

In case Gandalf runs out of memory, it stops. In the autonomous mode it means that the following passes won't be run, and a possibly simple proof might never be found. Hence it is important that the user selects such a limit on the number of kept clauses (using `assign(max_kept, <nr>)`) that Gandalf does not run out memory. Too low a limit will, of course, also prohibit Gandalf from finding a proof.

When Gandalf observes that it will soon reach the limit of the number of kept clauses, it starts to keep clauses selectively, preferring small clauses and discarding large clauses.

Since Gandalf splits the time given by `assign(max_seconds, <nr>)` (default 180 seconds) up between the selected strategies, it is important that the user gives the right time limit, ie the actual amount of time Gandalf will have for searching the proof. In case you want Gandalf to keep looking for an unlimited amount of time, use `set(run_forever)`: this causes the *last selected strategy*, which is typically a sensible complete strategy, run forever.

When Gandalf finds a proof, it outputs a line indicating *timer checkpoints*, for example:

```
timer checkpoints: c(4,0,9,7,30,10)
```

The timer checkpoints are used for the faithful reproduction of the proof search (for example, on a slower or faster architecture). In case you give a line assigning timer checkpoints, for example

```
assign(checkpoints,c(4,0,9,7,30,10)).
```

in the input file, Gandalf does not use time limits for the various strategy selections in the given run, but uses the given checpoints to faithfully reproduce the old search instead.

11 Selected Algorithms

We give a brief presentation of some of the algorithms and data structures used in Gandalf.

11.1 Subsumption

Forward Subsumption

The subsumption component is currently the most sophisticated single part of the classical version of Gandalf.

All the unit clauses are kept in the discrimination tree with variables. Forward unit subsumption is performed using this tree.

Our non-unit forward subsumption procedure uses the following steps:

1. A special double-literal discrimination tree with variables is used for full subsumption with two-literal clauses and for collecting potential longer subsumers: *two* largest literals of each clause are indexed, with "largeness" preferring ground over non-ground, bigger term size over smaller, more variables over less variables.

2. Potential subsumers retrieved by the previous step have length three or more. We use the following multi-layered subsumption algorithm for long clauses:

   First we check clause length, size, number of variables, maximal term depth - all this data is explicitly attached to each clause. Then we check predicate symbols, encoded into a bit field. After that subsumability of each literal is checked separately. If that passes, we do the full check, where the crucial feature is a suitable ordering, with big ground literals and literals containing most variables leading. Each variable, literal and subterm is decorated with flags and characterising data in order to minimize backtracking.

Back Subsumption

Back subsumption is performed by comparing the kept clause to a list of existing clauses one after another: no special indexing methods are used.

However, the list of existing clauses is kept for back subsumption in a specially sorted way: clauses are sorted by length, size and maximal term depth. Only these clauses are considered for back subsumption for which none of these three parameters is under the corresponding parameters of the newly kept clause.

In addition, **only the clauses in the usable list** are back subsumed. Because of this we separately check each selected clause in sos for forward subsumption before it is moved to usable.

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.

11.2 Demodulation

We use a full discrimination tree with variables for forward demodulation. We do not use any indexing methods for back demodulation.

11.3 Clause Simplification by Unit Deletion

The same discrimination tree as is used for unit subsumption is also used for simplifying clauses by deleting literals, using existing unit clauses.

We also use the same double-literal variables-containing discrimination tree for unit deletion. Here is the algorithm for deleting literals in the derived clause:

1. In case we have a unit clause ¬`L` such that `L` subsumes a literal `L'` in the derived clause, this `L'` is deleted. Ordinary discrimination tree is used for the check.

2. In case we have a two-literal clause ¬`L | R` such that the derived clause contains such literals `L'` and `R'` that `L | R` subsumes `L' | R'`, the literal `L'` is deleted. The special double-literal discrimination tree is used.

3. In case we have a unit clause ¬`L` such that `L` unifies with a literal `L'` in the derived clause and only these variables in `L'` which do not occur elsewhere in the clause are instantiated, then `L'` is deleted. Simple check-with-all-units and a special unification algorithm are used.

11.4 Data structures

We use a full variable-containing discrimination tree for demodulation, forward unit subsumption and unit deletion, double-literal clause subsumption and deletion (see earlier sections).

The search for unifiable literals uses a simple clause list augmented by indexing on the predicate name only. Paramodulation uses a simple clause list. Back subsumption uses lists of clauses *ordered* by length, size and maximal depth.

Terms, literals and clauses are represented in an ordinary lisp style as lists.

All the variables, constants, function and predicate symbols are represented as integers and treated as bit fields - certain bits in the integer give extra information about the term.

Concretely, each atom and term is explicitly decorated with the following data, encoded into certain bits in the first element of the corresponding list: number of constant occurrences, term size, term depth, ground/nonground flag, contains-repeated-variables-flag, the 'biggest' variable name occurrence.

Variables are also encoded as integers. During unification/matching the value of a variable is kept in an array indexed by certain bits in the number representing the variable. Each variable occurrence is explicitly decorated with the following data, encoded into certain bits of the integer: single-occurrence-in-a-literal flag, no-occurrences-in-following-literals flag, no-occurrences-in-other-literals flag.

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] 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.

[3] T. Tammet. Completeness of Resolution for Definite Answers. *Journal of Logic and Computation*, (1995), vol 4 nr 5, 449-471.

[4] 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.

[5] 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.

[6] T. Tammet. Proof strategies in linear logic. *Journal of Automated Reasoning*, 12:273–304, 1994.

[7] W.McCune. OTTER 3.0 Reference Manual and Users Guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, IL, January 1994.