Integrating Gandalf and HOL

Joe Hurd*

Computer Laboratory
University of Cambridge
joe.hurd@cl.cam.ac.uk

Abstract. Gandalf is a first-order resolution theorem-prover, optimized for speed and specializing in manipulations of large clauses. In this paper I describe `GANDALF_TAC`, a HOL tactic that proves goals by calling Gandalf and mirroring the resulting proofs in HOL. This call can occur over a network, and a Gandalf server may be set up servicing multiple HOL clients. In addition, the translation of the Gandalf proof into HOL fits in with the LCF model and guarantees logical consistency.

1 Introduction

Gandalf [8] [9] [10] is a resolution theorem-prover for first-order classical logic with equality. It was written in 1994 by Tanel Tammet (`tammet@cs.chalmers.se`) and won the annual CASC competitions in 1997 and 1998, beating off competition from Spass, Setheo and Otter. Gandalf is optimized for speed, and specialises in manipulations of large clauses.

HOL [3] [7] is a theorem-prover for higher-order logic, with a small logical core to ensure consistency and a highly general meta-language in which to write proof procedures.

In this paper I describe `GANDALF_TAC`, a HOL tactic that sits between these two provers, enabling first-order HOL goals to be proved by Gandalf. Using a first-order prover within a higher-order logic is not new, and many ideas have been explored here before (e.g., FAUST and HOL [6], SEDUCT and LAMBDA [2], 3TAP and KIV [1]). However, there are two significant novelties in the work presented here:

- The use of a completely seperate 'off-the-shelf' theorem-prover, treating it as a black box.
- The systematic use of a generic plug-in interface.

There is an increasing trend for HOL tactics to perform proof-search as much as possible outside the logic, for reasons of speed. When a proof is found, only then is the complete verification executed in the logical core, producing the official theorem (and incidentally validifying the correctness of the proof search). Perhaps the first instance of a first-order prover regenerating its proof in HOL

* Supported by an EPSRC studentship

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 311–321, 1999.

was the FAUST prover developed at Karlsruhe[6], and more recently John Harrison wrote `MESON_TAC` [4]; a model-elimination prover for HOL that performs the search in ML. `GANDALF_TAC` extends this idea, completely seperating the proof search from the logical core by sending it to an external program (which was probably not designed with this application in mind).

`GANDALF_TAC` is a Prosper plug-in, and as such does not purport to be a universal proof procedure, but rather a component of an underlying proof infrastructure. The Prosper[1] project aims to deliver the benefits of mechanised formal analysis to system designers in industry. Essential to this goal is an open proof architecture, allowing formal methods technology to be combined in a modular fashion. To this end the Prosper plug-in interface[2] was written by Michael Norrish, enabling developers to add specialised verification tools (like Gandalf) to the core proof engine in a relatively uniform way. `GANDALF_TAC` was the first plug-in to be written, and in a small way provided a test of concept of the Prosper frame-work.

Although it is a digression from the main point of the paper, since nothing has been published on Prosper to date it might be appropriate to provide an short description of the system. Fig. 1 shows an overview of the open proof architecture, in which client applications submit requests to a Core Proof Engine (CPE) server, which in turn might farm out subproblems to plug-in servers. The Plug-In Interface (PII) exists on both the CPE side as an ML API, and on the plug-in side as an internet server that spawns the desired plug-in on request. `GANDALF_TAC` is implemented in ML and communicates with a Gandalf wrapper

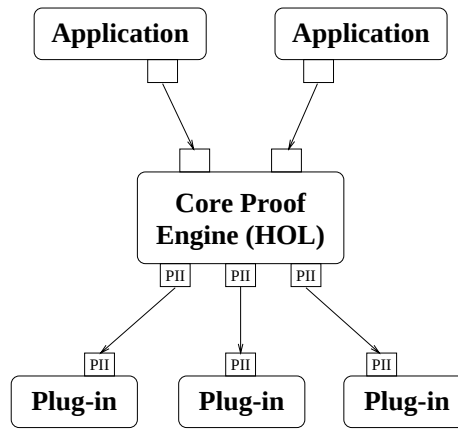

Fig. 1. Overview of the Prosper architecture.

[1] The Prosper homepage is at `http://www.dcs.gla.ac.uk/prosper/`.
[2] The Prosper plug-in interface homepage is at `http://www.cl.cam.ac.uk/users/-mn200/prosper`.

script by passing strings to and fro. The wrapper script takes the input string, saves it to a file, and invokes Gandalf with the filename as a command-line parameter, passing back all output. This is illustrated in Fig. 2.

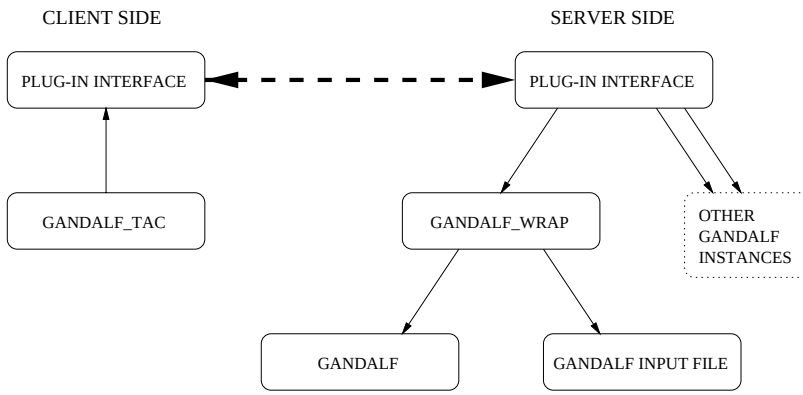

Fig. 2. The Gandalf internet server.

2 How It Works

Briefly, `GANDALF_TAC` takes the input goal, converts it to a normal form, writes it in an acceptable format, sends the string to Gandalf, parses the Gandalf proof, translates it to a HOL proof, and proves the original goal. Fig. 3 shows the procedure in pictorial form.

We will run through each stage in turn, tracking the metamorphosis of the goal

$$\forall ab.\ \exists x.\ Pa \vee Pb \Rightarrow Px$$

2.1 Initial Primitive Steps

In the first stage of processing we assume the negation of the goal:

$$\{\neg(\forall ab.\ \exists x.\ Pa \vee Pb \Rightarrow Px)\} \vdash \neg(\forall ab.\ \exists x.\ Pa \vee Pb \Rightarrow Px)$$

2.2 Conversions

In this phase we convert the conclusion to Conjunctive Normal Form (CNF), and for this we build on a standard set of HOL conversions, originally written by John Harrison in HOL-Light [5] and ported to HOL98 by Donald Syme. In order, the conversions we perform are:

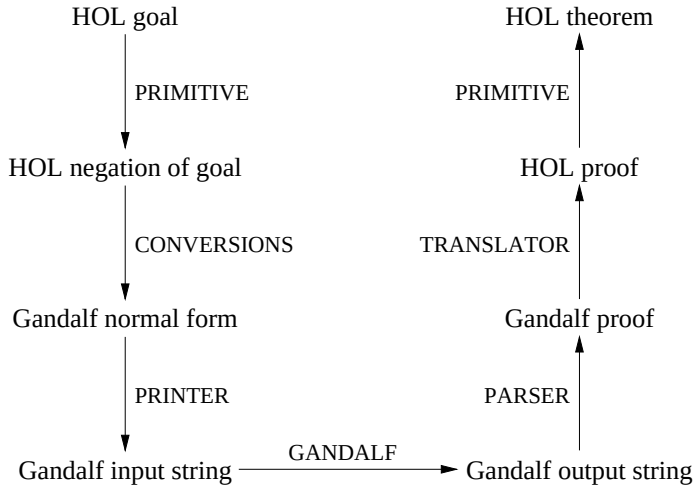

Fig. 3. Overview of `GANDALF_TAC`.

1. Negation normal form.
2. Anti-prenex normal form.
3. Move existential quantifers to the outside (partial Skolemization).
4. Conjunctive normal form.

After these conversions we extract the conjuncts to give to Gandalf.

Our example becomes:

$$\{\neg(\forall ab.\ \exists x.\ Pa \vee Pb \Rightarrow Px)\} \vdash \exists ab.\ \big(\forall x'.\ \neg(Px')\big) \wedge \big(Pa \vee Pb\big)$$

and the relevant terms we will pass to Gandalf are:

$$\neg(Px')$$
$$Pa \vee Pb$$

2.3 Printing

We now have our goal in a form acceptable to Gandalf, and we can write it to a string and send it to the program. We must add a header and a footer of control information, and we must also take care to rename all the HOL constants/variables, so that constants and existentially quantified variables have names of the form $\mathtt{c}n$, and universally quantified variables have names of the form $\mathtt{x}n$. Another wrinkle is that Gandalf will not accept propositional variables directly, so we shoehorn them in by pretending that they're 1 place predicates on a constant symbol not mentioned anywhere else.

This is the input string that we send to Gandalf in our example:

```
%-----------------------%
% hol -> gandalf formula %
%-----------------------%
set(auto).
assign(max_seconds,300).
assign(print_level,30).

list(sos).

-c10(x1).

c10(c5) |
c10(c6).
end_of_list.
```

2.4 Calling Gandalf

We are now ready to call Gandalf with the input string, and here we use the Prosper plug-in interface library.

By default, when `GANDALF_TAC` is loaded a Gandalf internet server is automatically started on the local machine and registered with the system. Communicating with it is simply a matter of calling routines in the plug-in interface; in particular we can spawn a new Gandalf process, send it an input string, and receive its output as a string.

However, the generic system is as detailed in Fig. 2. The Gandalf internet server can be run on any machine on the network as long as its location is registered with the plug-in interface when the client initialises. In this general setup we can have HOL sessions on several machines all making use of the same Gandalf server. The default is for ease of installation only.

Here is what the Gandalf server returns in our example:

```
Gandalf v. c-1.0c starting to prove: gandalf.26884

strategies selected:
((binary 30 #t) (binary-unit 90 #f) (hyper 30 #f)
(binary-order 15 #f) (binary-nameorder 60 #f 1 3)
(binary-nameorder 75 #f))

********* EMPTY CLAUSE DERIVED *********

timer checkpoints: c(2,0,28,2,30,28)

1 [] c10(c5) | c10(c6).
2 [] -c10(x).
3 [binary,1,2,binary_s,2] contradiction.
```

Note that the variable `x1` we passed in to Gandalf has disappeared, and a new variable `x` has appeared. In general we can assume nothing about the names of variables before and after the call.

2.5 Parsing

Our task is now to parse the output string into a Gandalf proof structure, ready to be translated into the corresponding HOL proof. We first check that the string "EMPTY CLAUSE DERIVED" occurs in the output string (or else the tactic fails), and then cut out the proof part of the output string. The use of ML parser combinators enabled a parser to be quickly constructed, and we put the result into special purpose datatypes for storing proof steps, simplifications and Gandalf terms.

Here is the result of the parse on our running example:

```
[(1, (Axiom(), []),
  [(true, Branch(Leaf "c10", Leaf "c5")),
   (true, Branch(Leaf "c10", Leaf "c6"))]),
 (2, (Axiom(), []),
  [(false, Branch(Leaf "c10", Leaf "x"))]),
 (3, (Binary((1, 1), (2, 1)), [Binary_s(2, 1)]),
  [])]
: (int * (Proofstep * Clausesimp list) * (bool * Tree) list) list
```

2.6 Translating

The proof translator is by far the most complicated part of `GANDALF_TAC`. Gandalf has four basic proof steps (binary resolution, hyper-resolution, factoring and paramodulation) and four basic simplification steps (binary resolution, hyper-resolution, factoring and demodulation). Each Gandalf proof line contains exactly one proof step followed by an arbitrary number of simplification steps to obtain a new clause (which is numbered and can be referred to in later proof lines). The proof is logged in detail and in addition after each proof line the desired clause is printed, allowing a check that the line has been correctly followed.

The problem is that even though the proofs are logged in detail, they are occasionally not logged in enough detail to make them unambiguous. The situations in which they are ambiguous are rare, usually involving large clauses when more than one disjunct might match a particular operation, but they occur often enough to make it necessary to tackle the issue.

To illustrate the problem, there may be several pairs of disjuncts that it is possible to factor, or simplifying with binary resolution may be applicable to more than one disjunct in the clause. Gandalf also freely reorders the disjuncts in the axioms with which it has been supplied, requiring some work to even discover which of our own axioms it is using![3]

[3] In addition, my version of Gandalf had a small bug in the proof logging routine requiring some guesswork to determine the exact literals used in the hyper-resolution

At this point it would have been easy to contact the author of Gandalf and ask him to put enough detail into the proofs to completely disambiguate them. However, we decided to remain faithful to the spirit of the project by treating Gandalf as a black-box, and looked instead for a solution within `GANDALF_TAC`.

We implement a Prolog-style depth-first search with backtracking to follow each line, if necessary trying all possible choices to match the Gandalf clause with a HOL theorem. If there were many ambiguities combined with long proof lines, this solution would be completely impractical, fortunately however ambiguous situations occur rarely and there are usually not many possible choices, so efficiency is not a key question. The only ambiguity that often needs to be resolved is the matching of axioms and this is performed in ML, but all other proof steps are performed as HOL inferences on theorems.

The final line in the Gandalf proof is a contradiction, so the corresponding line in the HOL world is too:

$$\{\neg(\forall ab.\ \exists x.\ Pa \vee Pb \Rightarrow Px)\} \vdash \bot$$

2.7 Final Primitive Steps

After translation, we need only use the contradiction axiom in order to establish our original goal:

$$\vdash \forall ab.\ \exists x.\ Pa \vee Pb \Rightarrow Px$$

3 Results

3.1 Performance

In Table 1 we compare the performance of `GANDALF_TAC` with `MESON_TAC`, using a set of test theorems taken mostly from the set that John Harrison used to test `MESON_TAC`, most of which in turn are taken from the TPTP (Thousands of Problems for Theorem Provers) archive[4]. In each line we give the name of the test theorem, followed by the (real) time in seconds to prove the theorem and the number of HOL primitive inference steps performed, for both the tactics. In addition, after the `GANDALF_TAC` primitive inferences, we include in brackets the number of these inferences that were wasted due to backtracking. As can be seen the number of wasted inferences is generally zero, but occasionally an ambiguity turns up that requires some backtracking.

The other thing to note from Table 1 is that `MESON_TAC` beats `GANDALF_TAC` in almost every case, the only exceptions lying in the hard end of both sections. Why is this? Table 2 examines how the time is spent within both tactics. We divide up the proof time into 3 phases:

steps. Of course this can be easily fixed by the author, but it is a good illustration of the type of problem encountered when using an off-the shelf prover.

[4] The TPTP homepage is `http://wwwjessen.informatik.tu-muenchen.de/~tptp/`.

Table 1. Performance comparison of GANDALF_TAC with MESON_TAC.

Goal	MESON_TAC		GANDALF_TAC		
Non-equality					
T	0.027	31	0.034	32	(–)
$P \vee \neg P$	0.075	72	2.003	108	(0)
MN_bug	0.122	166	2.062	286	(0)
JH_test	0.137	176	2.762	312	(0)
P50	0.159	243	1.638	441	(0)
Agatha	0.438	872	3.916	1891	(0)
ERIC	0.989	490	2.750	1268	(0)
PRV006_1	1.064	1501	41.044	8097	(109)
GRP031_2	1.267	713	8.083	3699	(251)
NUM001_1	2.090	1138	40.190	4019	(0)
COL001_2	2.150	847	39.240	2620	(0)
LOS	5.705	917	5.110	2565	(0)
GRP037_3	7.149	2151	79.370	11988	(0)
NUM021_1	7.535	1246	17.210	4352	(0)
CAT018_1	12.226	2630	61.585	13477	(0)
CAT005_1	63.849	2609	66.200	13371	(0)
Equality					
$x = x$	0.090	54	0.041	35	(–)
P48	0.394	636	2.707	495	(0)
PRV006_1	0.648	1053	13.558	4015	(0)
NUM001_1	0.768	876	7.032	3012	(0)
P52	1.157	1122	–	–	(–)
P51	1.294	1079	–	–	(–)
GRP031_2	1.377	757	7.946	3699	(251)
GRP037_3	3.402	1466	26.844	8242	(0)
CAT018_1	7.646	1809	28.560	8611	(0)
NUM021_1	7.737	1026	10.765	3423	(0)
CAT005_1	30.514	1784	29.490	8505	(0)
COL001_2	56.948	700	4.930	1273	(0)
Agatha	–	–	12.626	3409	(0)

– Conv: Conversion into the required input format.
– Proof: Proof-search using native datatypes.
– Trans: Translation of the proof into HOL.

All the entries in this table are geometric means, so the first line represents the geometric means of times for phases of GANDALF_TAC to prove all the non-equality problems in Table 1. Geometric means were chosen here so that ratios are meaningful, and the large difference between GANDALF_TAC and MESON_TAC is in the translation phase; GANDALF_TAC struggles to translate proofs in time comparable to finding them, but for MESON_TAC they are completely insignificant.

Another interesting difference is in the proofs of equality formulae; MESON_TAC has a sharp peak in this entry, but there is no anologue of this for GANDALF_TAC.

Table 2. Breakdown of time spent within GANDALF_TAC and MESON_TAC.

	Conv.	Proof	Trans.	Total
GANDALF_TAC				
Non-equality	1.67	5.55	2.14	11.57
Equality	4.20	5.27	7.19	17.82
Combined	2.51	5.42	3.64	13.99
MESON_TAC				
Non-equality	0.26	0.36	0.06	0.90
Equality	0.43	1.27	0.09	2.61
Combined	0.33	0.66	0.07	1.50

It seems likely that this is because Gandalf's has built-in equality reasoning, whereas equality has to be axiomatised in formulae sent to MESON_TAC.

3.2 Gandalf the Plug-In

Putting aside performance issues for the moment, the project has also contributed to the development of the plug-in concept. GANDALF_TAC provides evidence that plug-ins can coexist with the idea of an LCF logical core, and that efficient proving does not have to mean accepting theorems on faith from an oracle.

GANDALF_TAC has also tested the plug-in interface code, which is simple to use, enabling one to concentrate on proof issues without having to think about system details. Once the interface becomes part of the standard HOL distribution then any Gandalf user will be able to download the GANDALF_TAC ML source and wrapper shell script, and use Gandalf in their HOL proofs. Hopefully we will see many more plug-ins appearing in the future.

4 Conclusion

The most important thing to draw from this project is the need for good standards. If Gandalf used a good standard for describing proofs that retained the human-readable aspect but was completely unambiguous, then the project would have been easier to complete and the result would be more streamlined. On the positive side, Gandalf has a good input format that is easy to produce, and the Plug-in interface is an example of a good development standard, taking care of system issues and allowing the programmer to think on a more abstract level. If either of these had been absent, then the project would have been stalled considerably. A program can only be useful as a component of a larger system if the interface is easy for machines as well as people, i.e., simple and unambiguous as well as short and readable.

What is the future for GANDALF_TAC? There are several ways in which the performance could be improved, perhaps the most effective of which would be to enter into correspondence with the author of Gandalf, so that completely explicit

(perhaps even HOL-style) proofs could be emitted. Another approach would be to reduce the number of primitive inferences performed, both the initial conversion to CNF and proof translation could be improved in this way. Perhaps they could even be performed outside the logic, minimizing the primitive inferences to a fast verification stage once the right path had been found. There is much scope for optimization of `GANDALF_TAC` on the ML/HOL side, and the results obtained suggest that such an effort might be sufficient for Gandalf's natural advantages of built-in equality reasoning and coding in C to allow `GANDALF_TAC` to overtake `MESON_TAC` in some domains (e.g., hard problems involving equality reasoning).

To look at another angle, `GANDALF_TAC` is a step towards distributed theorem-proving. It is easy to imagine several proof servers (perhaps several Gandalf servers each with different strategies selected), and a client interface designed to take the first proof that returned and throw the rest away. Distributing such a CPU-intensive, one time activity as theorem-proving makes economic sense, although there are many problems to be solved here, such as how to divide up a problem into pieces that can be seperately solved and joined together at the end.

Finally, one thing that was obvious while developing `GANDALF_TAC` is that a tool like Gandalf does not really fit in with the interactive proof style popular at present. If Gandalf is used in a proof and takes 3 minutes to prove a subgoal, then every time the proof is run there will be a 3 minute wait. What would perhaps be useful is to save proofs, so that only the speedy verification is run in the future, not the extensive search. Maybe then we would begin to see more tools like Gandalf applied in proofs, as well as the fast tactics that currently dominate.

5 Acknowledgements

I would like to thank Konrad Slind for helping me with all aspects of this work; when I began I was but a HOL novice. Mike Gordon, my supervisor, also gave me valuable advice on how to attack the problem and make the most of the opportunity. Michael Norrish made some useful additions to the Prosper plug-in interface to solve some of the problems I was having, and Myra VanInwegen, Marieke Huisman and Daryl Stewart sympathetically listened to my frequent status reports in the tea-room. Larry Paulson kindly postponed my scheduled group talk when it was obvious that I needed another week. Finally, the comments from the anonymous referees greatly improved the final version.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integration of automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume II, chapter 4, pages 97–116. Kluwer, 1998.

[2] H. Busch. First-order automation for higher-order-logic theorem proving. In *Higher Order Logic Theorem Proving and Its Applications*, volume Lecture Notes in Computer Science volume 859. Springer-Verlag, 1994.

[3] M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.

[4] J. Harrison. Optimizing proof search in model elimination. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 313–327, New Brunswick, NJ, 1996. Springer-Verlag.

[5] J. Harrison. *The HOL-Light Manual (1.0)*, May 1998. Available from http://-www.cl.cam.ac.uk/users/jrh/hol-light/.

[6] R. Kumar, Th. Kropf, and Schneider K. Integrating a first-order automatic prover in the hol environment. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications (HOL91)*, pages 170–176, Davis, California, August 1991. IEEE Computer Society Press.

[7] K. Slind. *HOL98 Draft User's Manual, Athabasca Release, Version 2*, January 1999. Available from http://www.cl.cam.ac.uk/users/kxs/.

[8] T. Tammet. A resolution theorem prover for intuitionistic logic. In *Cade 13*, volume Lecture Notes in Computer Science volume 1104. Springer Verlag, 1996.

[9] T. Tammet. *Gandalf version c-1.0c Reference Manual*, October 1997. Available from http://www.cs.chalmers.se/~tammet/.

[10] T. Tammet. Towards efficient subsumption. In *Automated Deduction: Cade 15*, volume Lecture Notes in Artificial Intelligence volume 1421. Springer, 1998.