Technical Report TR-ARP-3-94

Automated Reasoning Project
Research School of Information Sciences and Engineering
and Centre for Information Science Research
Australian National University

March 26, 1994

SCOTT: SEMANTICALLY CONSTRAINED OTTER SYSTEM DESCRIPTION

John Slaney, Ewing Lusk and William McCune

Abstract This is the announcement of SCOTT 1.0 from the proceedings of CADE-12. SCOTT combines OTTER and FINDER into a new theorem prover.

SCOTT: Semantically Constrained Otter System Description

John Slaney[1], Ewing Lusk[2] and William McCune[2]

[1] Australian National University, Canberra 0200, Australia
[2] Argonne National Laboratory, Argonne, Illinois 60439-4801, USA

The theorem prover SCOTT, early work on which was reported in [3], is the result of tying together the existing prover OTTER [1] and the existing model generator FINDER [4] to make a new system of significantly greater power than either of its parents. The functionality of SCOTT is broadly similar to that of OTTER, but its behaviour is sufficiently different that we regard it as a separate system.

1 OTTER

We briefly review the algorithm of OTTER, a first order theorem prover embodying the set of support strategy. It chains forward from a set of input clauses until either the search space is exhausted or the empty clause is deduced, showing that a goal has been matched. The clauses are divided into two lists: the *usable* clauses and the *set of support*. The main cycle of its proof search is as follows.

1. **Select** a given clause g from the set of support. Move g into the usable list.
2. **Generate** immediate consequences of g in combination with the usable clauses. If the empty clause is found, stop.
3. **Rewrite** the consequences if appropriate rewrite rules are in force.
4. **Filter** out unwanted consequences, such as subsumed ones or those which are too long to be worth keeping.
5. **Update** the clause database by adding the surviving consequences to the set of support.

The rules of inference available to define 'immediate consequence' for phase 2 of this loop include several varieties of resolution and hyper-resolution as well as paramodulation and demodulation (rewriting) for equational reasoning. It is clear that heuristics and other search-direction techniques may be applied at several points in the process. SCOTT gives a heuristic for use in the 'Select' phase and a restriction strategy applicable either in the 'Generate' phase or as a filter.

2 FINDER

FINDER is not a theorem prover in the ordinary sense, but searches for small models of first order theories presented to it as sets of clauses in a simple many-sorted language. The domain of interpretation being fixed first, as containing

just a few objects, FINDER's problem is to seek functions defined on those objects to interpret the function symbols of the language in such a way that all of the clauses come out true. Thus it treats its search as a constraint satisfaction problem in which the ground instances of the clauses, restricted to the chosen domain, give rise to sets of constraints.

To illustrate with a trivial example, the input

```
sort     element  cardinality = 3
function *: element,element -> element.
clause   a * (b * c) = (a * b) * c.
end
```

will cause FINDER to enumerate the semigroups of order 3. Settings may be invoked, for example to make it stop after finding one model or to make it return a null result after searching unsuccessfully for two seconds or after a thousand backtracks or the like. When called as a procedure from another program such as SCOTT, FINDER may be instructed to remember a model it has generated and subsequently to test arbitrary clauses for truth or falsehood in that model. Note that a clause is regarded as having all its variables bound by implicit universal quantifiers, so it has a definite truth value in any interpretation.

3 The combined system

SCOTT itself is best seen as OTTER with some additional capabilities. It can appeal to an interpretation called the *guiding model* in which, of course, some of the clauses which occur in the proof search are true and others are false. The guiding model is discovered by a FINDER module and may be changed from time to time during the proof search in order to make more of the clauses true. Details of the guiding model are not known to the OTTER module, but the latter may send a clause to an oracle called `is_good` which returns the (boolean) truth value of the clause in the model.

There are two ways in which the guiding model can be used. One is the *false preference strategy*. In the selection phase, when the next given clause is chosen, OTTER normally applies a weighting function to the clauses in the set of support and chooses one of the lightest. By default, the weight is just the number of symbols in the clause, though interesting behaviour can result from more elaborate functions.[3] The false preference strategy arises from the thought that the goal is a consequence of clauses which imply it rather than of clauses which do not. Naturally, OTTER does not have access to information as to whether clauses really imply the goal (or there would be no need for a proof search) but it can tell whether they imply it in the guiding model, and this should be some approximation to real implication. That is, the guiding model is chosen so as to make the goal false (and a lot of the kept clauses true) and then some preference attaches to choosing given clauses which are false in the guiding model. This is achieved by testing each kept clause in the 'Update' phase and

[3] A relatively simple example is the 'deletion strategy' discussed in [2].

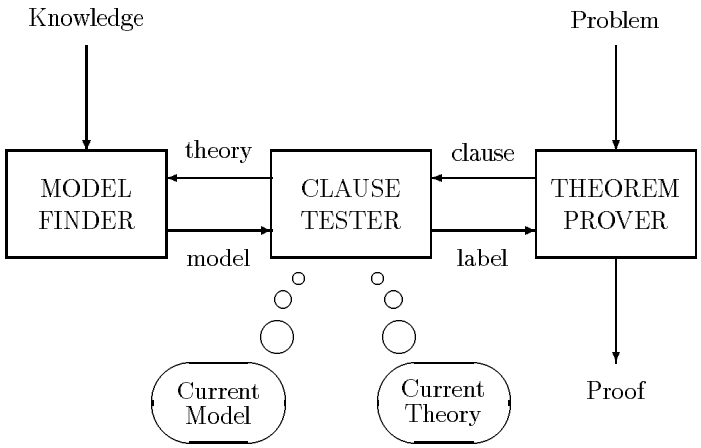

Fig. 1. Semantics in Theorem Proving

adding a constant to its weight if it is true in the guiding model. The constant is determined by an assignment in the OTTER input file.

The second use of the guiding model is for a form of rule restriction which we have dubbed *dynamic semantic resolution*. At present, we have implemented only the simple form sometimes called model resolution: in each inference, at least one of the parent clauses must be false in the guiding model. This restriction is an extension of the set of support idea: at each step, after the given clause is moved into the usable list, we can think of the clauses as divided afresh into axioms and set of support, the axioms being those usable clauses true in the guiding model (and hence a consistent set). There is never any need to resolve axioms with each other, which warrants the model resolution restriction. What makes SCOTT's implementation 'dynamic' is that the guiding model need not be given in advance but can be discovered and repeatedly changed in response to the clauses occurring in the search. In the present implementation, the restriction is applied in the 'Filter' phase, after the consequences have been deduced. It would clearly be more efficient to apply it earlier, to prevent the deductions, but the present version is sufficient for experimental purposes.

Figure 1 shows the basic structure of SCOTT. Two input files are needed. One contains the problem just as for OTTER, and the other is a FINDER file containing the language definitions, the clauses in the initial usable list (which have to be true in the guiding model for integrity in the context of OTTER's set of support algorithm) and optionally any other domain-specific knowledge which may help direct FINDER to good models. There must also be some condition such as a time limit to cause FINDER to stop the search for a model and return a null result in cases where the search fails. The two modules—prover and modeller—do not communicate directly, but exchange messages with the 'clause tester'. At any given time during the proof search, the clause tester has

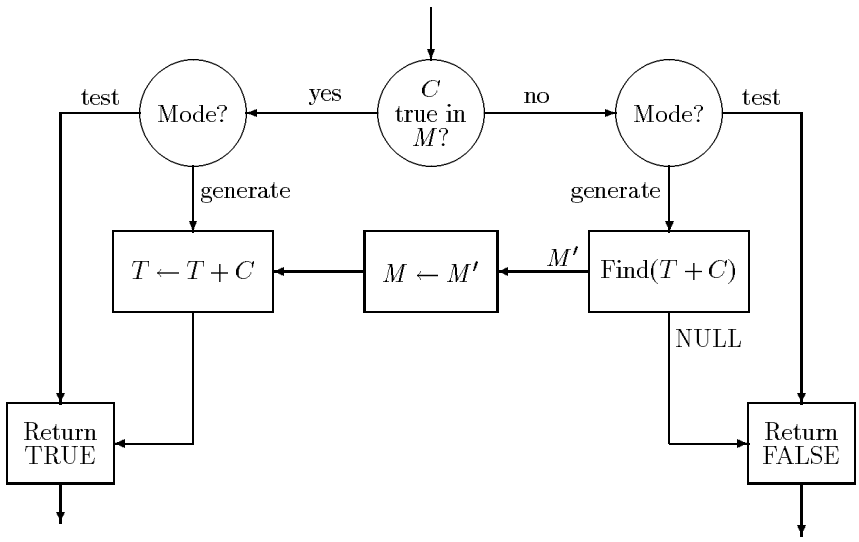

Fig. 2. Clause Tester Flowchart

in its memory a 'current theory', which is a set of clauses, and a 'current model' in which the current theory is true. After FINDER has been called initially with its input file in order to set these up, the clause tester runs for a while in 'generating' mode and then switches to 'testing' mode. The switch is governed by a setting in the FINDER file. The logic of the clause tester is shown in Figure 2. The next clause is C, the current model is M and the current theory is T. In test mode, the returned value is simply the truth value of C in M. In generate mode, if C is false in M an attempt is made to find a better M' in which all of T is true and C is also true. Then if C is true in the guiding model it is added to T. Finally, in any case, its truth value is returned.

4 Comment

SCOTT brings semantic information gleaned from the proof attempt into the service of the syntax-based theorem prover. We find it appealing that the guiding model is thus automatically adapted to the specific problem and to the particular proof search method being applied to it. There are many ways in which such information could help to guide a proof search. We have implemented two of them. The results of our experiments to date have been encouraging if somewhat mixed. We looked at some of the hard condensed detachment problems of [2] on which SCOTT is reasonably successful. As we report in [3] it generally improves on OTTER by a factor of two or so. In extreme cases, it can be over 1000 times faster than OTTER, though there are also cases where model resolution can actually cause inefficiency. The false preference strategy is generally useful, though the optimum weight to be added to true clauses has to be guessed

or determined by experiment. With the correct setting, it improves OTTER's performance on a version of Luka-5, one of the hardest problems in [2], by almost two orders of magnitude.

With binary resolution as the rule of inference, its restriction by means of a guiding model obviously retains completeness. Where other rules such as hyper-resolution or paramodulation are used, the model strategy is in general incomplete. Hence it must be applied with care. The false preference strategy does not introduce incompleteness. In any case, SCOTT has OTTER as a sub-program and is capable of running exactly as OTTER, with all semantic features disabled, so in a sense nothing is lost even where incompleteness occurs.

Thus we offer SCOTT not as the solution to all known problems but as an interesting way of adding power to an already powerful prover by making some new heuristics available to it. We welcome further experimentation with the ideas it incorporates.

5 Availability

SCOTT is available by anonymous ftp from `arp.anu.edu.au` where it is in file `pub/scott/scott-1.0.tar.Z`. The sources include both OTTER and FINDER, each of which may be separately compiled and installed if desired. Upgrades and new releases of both OTTER and FINDER should be compatible with SCOTT.

References

1. W. McCune, *OTTER 2.0 Users Guide*, Technical report ANL-90/9, Argonne National Laboratory, Argonne, IL, 1990.
2. W. McCune & L. Wos, *Experiments in Automated Deduction with Condensed Detachment*, **Proc. 11th International Conference on Automated Deduction**, 1992, pp. 209–223.
3. J. Slaney, *SCOTT: A Model-Guided Theorem Prover*, **Proc. 13th International Joint Conference on Artificial Intelligence**, 1993, pp. 109–114.
4. J. Slaney, *FINDER, Finite Domain Enumerator: Version 2.0 Notes and Guide*, Technical report TR-ARP-1/92, Automated Reasoning Project, Australian National University, Canberra, 1992.
5. J. Slaney, *FINDER, Finite Domain Enumerator: Version 3.0 Notes and Guide*, Document with program sources, anonymous ftp, arp.anu.edu.au, file ARP/FINDER/finder-3.0.1.tar.Z.