TRVTH : A Theorem Prover Which Uses Sorcerer or AI without LISP

Randall A. Helzerman
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
phone: (317) 494-3430
helzecn.purdue.edu

July 15, 1994

Abstract

TRVTH is a set of C++ classes which implement a first-order logic theorem prover. TRVTH was written for two purposes: First, to fill a ecological niche : there is a great need for a C++- based theorem prover. Second, to explore the possibility that, because of new tools like sorcerer, LISP may no longer be the language of choice for some AI applications.

1 Introduction

Artificial Intelligence (AI) strives to duplicate some of the reasoning and problem-solving abilities of the human mind. Theorem provers are one of the most useful results of research in this area. They are used for many applications, from mathematics research to expert systems.

The programming language of choice for AI has traditionally been LISP; so much so that once someone seriously proposed to define AI as "any program written in LISP." Indeed, LISP has much to offer to the AI researcher. One of its the main strengths is its ability to conveniently represent and manipulate trees, which are ubiquitous in AI programming. Until recently, this fact alone was reason enough to prefer LISP. However, because of sorcerer's tree recognition and manipulation abilities, the C programmer needs no longer to envy they LISP programmer. Indeed, it is reasonable to predict that the availablity of such a tool will cause a few of the more adventurous LISP veterans to make the switch.

This paper describes one such AI project: TRVTH, a theorem prover. Although there are many excellent theorem provers written in LISP, their power is unavailable to the programmer who needs or wishes to code in C++. Therefore TRVTH meets a growing need in the AI community.

In the rest of the paper we briefly motivate the use of theorem provers by giving a toy example of how they are used, and then we discuss exactly what kinds of tree manipulations are necessary in the course of theorem proving, and how sorcerer helped us. Finally, we wrap up with some general comments on LISP vs. C++ for AI, and how tools like Sorcerer and PCCTS affect the programming language balance of power.

2 Theorem Proving

Theorem provers take as input facts expressed in logical formulas, and uses them to deduce answers to questions. As an example how how theorem provers could be used to represent parts of the world, consider the "blocks world" scene in figure 1. There are four stacked blocks resting on a table. This scene can be described by formulas of first order logic, e.g. "(on block-a block-b)" means that block A is on top of block B. Here, "on" is called a *relation* because it shows how its two arguments, block-a and block-b, are related. A declarative English sentence can usually be directly translated into a logical formula by making the verb of the sentence be the relation and the subject and predicate its arguments. In this way, AI practitioners can store and retrieve facts about the parts of the world in which they want to model.

However, the real power of theorem proving systems is that because they are able to make correct inferences, more facts can be retrieved out of a database than what are explicitly entered. For example, suppose we augmented the formulas of figure 1 with the following two formulas (which are meant to capture the intuitive meaning of the word "above"):

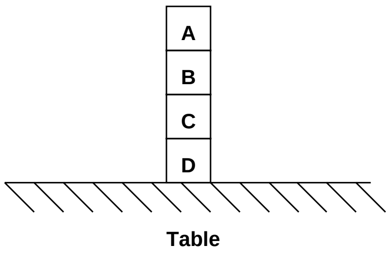
```
;; If you're on me, you're above of me.
(forall (X) (forall (Y)
   (if (on X Y)
       (above X Y))))

;; If you're on someone who is
;; above me, then you're above me.
(forall (X) (forall (Y) (forall (Z)
   (if (and (on X Y) (above Y Z))
       (above X Z)))))
```

If the theorem prover was asked the query

(above A table)

(which in English would be "Is block A about the table?") it could answer "yes" even though the fact "(above A table)" had not been explicitly entered. It is this power of being able to pull more out of the model than what is put in which makes



(on block-a block-b)	;; Block A is on top of Block B
(on block-b block-c)	;; Block B is on top of Block C
(on block-c block-d)	;; Block C is on top of Block D
(on block-d table)	;; Block D is on the table

Figure 1: An example how a scene composed of blocks would be represented in first order logic.

3 Sorcerer used to get to Clause Form

Because a full explanation of the inference process would be beyond the scope of this paper, we will concentrate on the operation which most immediately utilizes Sorcerer: The conversion of formulas to clause form. This occurs in several steps described in the sequel.

Step 1. Eliminate Implications

Everywhere that $X \rightarrow Y$ ("If X is true, then Y is also true) appears in a formula, it can be replaced by $\neg X \vee Y$.

```
// Eliminate Implications.

lf_ei
        :
#(r:TRVTH_IMP arg1:. arg2:.)
<< r->token=TRVTH_OR; arg1->negate();>>
|.
;
```

Step 2. Move negations to atoms

This consists of doing several things. First, we must eliminate double negations, i.e. subformulas of the form $\neg\neg X$ are equivalent to X. Then we use the distributive laws to move all of the remaining negations to the atoms. The following is the Sorcerer source code to do this. It is kind of interesting in that it actually modifies the tree it is working on as it is walking down it–yet totally without munging the tree or confusing Sorcerer. It does this by explicitly recursively calling itself on nodes. A good mental image of the process would be to consider SORCERER to be a supersonic plane, the tree which it is modifying to be the air, and the modifications to the tree to be the shock-wave which proceeds the supersonic plane and flows around its wings and fuselage. Just as the shockwave has enough power to rip the plane to shreds, but never does because careful engineering keeps the winds flowing around the plane, so also SORCERER could get "blasted to bits" by getting the DOWN and RIGHT pointers all messed up. However, like a wavefront which starts at the root of the tree and moves to the leaves, the modifications to the tree always keep one step ahead of SORCERER, and so and SORCERER always is parsing nodes which have already been modified, hence it is never confused.

```
// Move negations to atoms

lf_mna  :
(#(TRVTH_NOT #(TRVTH_NOT .)))?
#(r:TRVTH_NOT #(u:TRVTH_NOT s:.))
<< r->copy_to_this(s);
   u->shallow_delete();
   trvth_lf_mna(trvth_tree_parser, &s); >>
|
(#(TRVTH_NOT #(TRVTH_OR .)))?
#(r:TRVTH_NOT #(s:TRVTH_OR arg1:. arg2:.))
<< r->copy_to_this(s);
   r->token=TRVTH_AND;
   arg1->negate(); arg2->negate();
   s->shallow_delete();
   trvth_lf_mna(trvth_tree_parser, &arg1);
   trvth_lf_mna(trvth_tree_parser, &arg2);
>>
|
(#(TRVTH_NOT #(TRVTH_AND .)))?
#(r:TRVTH_NOT #(s:TRVTH_AND arg1:. arg2:.))
<< r->copy_to_this(s);
   r->token=TRVTH_OR;
   arg1->negate(); arg2->negate();
   s->shallow_delete();
   trvth_lf_mna(trvth_tree_parser, &arg1);
   trvth_lf_mna(trvth_tree_parser, &arg2);
>>
|
(#(TRVTH_NOT #(TRVTH_EXISTS . .)))?
#(r:TRVTH_NOT #(e:TRVTH_EXISTS v:. vp:.))
<< r->copy_to_this(e);
   r->token=TRVTH_FORALL;
   vp->negate();
   trvth_lf_mna(trvth_tree_parser, &vp);
   e->shallow_delete(); >>
|
(#(TRVTH_NOT #(TRVTH_FORALL . .)))?
#(r:TRVTH_NOT #(e:TRVTH_EXISTS v:. vp:.))
<< r->copy_to_this(e);
   r->token=TRVTH_EXISTS;
   vp->negate();
   trvth_lf_mna(trvth_tree_parser, &vp);
   e->shallow_delete(); >>
|.
;
```

4 LISP vs. C++ for AI Programming

Until now, LISP (and to a smaller degree Prolog) have reigned supreme in AI programming circles, almost unchallenged by other programming languages. LISP does indeed offer many advantages, not the least of which are an interpreted mode (which encourages modular adjustment) and garbage collection.

However, C++ brings to the table many advantages as well, including faster execution time, and programmer familiarity. AI practitioners and researchers, who normally would shun the use of C or C++ should, in the light of new tools like Sorcerer and PCCTS, take a second look.

References

[1] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufman, 1987.

[2] Matt Ginsberg. *Essentials of Artificial Intelligence.* Morgan Kaufmann Publishers, Inc., San Mateo, California, USA, 1993.

[3] Nils Nilsson. *Principles of Artificial Intellegence.* Tioga Publishing, Palo Alto, CA, 1980.

[4] A. Tarski. Logic, semantics, metamathematics: Papers from 1923 to 1938. Oxford, England, 1956.