THORN 10

08/06/2023 © Mark Tarver

1.0 Introduction

THORN 10 is a theorem prover for first-order logic and is a major simplification and improvement on the predecessor programs FTP (Fast Theorem Prover) and THORN 1-9. As a result of rationalising the proof process THORN 10 requires fewer primitive inferences rules than its predecessors and weighs in at only 550 lines of code.

THORN is short for **T**heorem prover derived from **HORN** clause logic since it uses Prolog compilation techniques for speed. THORN is a **near-Prolog theorem prover** similar to Stickel's PTTP.

2.0 Downloading and Installing

THORN 10 runs under Shen with the standard library installed. You can download THORN 10 and the sample programs from here. If you have Windows, there is an image of Shen-Scheme in that directory for Windows. Click on shen.bat and the standard library and THORN 10 will be installed.

If you do not have Windows you need to create a working image of Shen with a standard library installed and then type (load "THORN 10.shen").

There are sample files in the folder **Problems**; to run the problems simply type (load <filename>). Look in prf.txt where each proof is written from the last problem entered.

pelletier.shen These are propositional calculus problems from Pelletier.

schubert.shen (Schubert's Steamroller) This contains Schubert's Steamroller.

L.shen A formalisation of propositional calculus from Mendelson in first-order logic.

set.shen (basic set theory) This contains some basic set theory problems.

ec.shen (equivalential calculus) This file contains equivalential calculus problems.

group.shen (**group theory**) This file contains group theory problems requiring reasoning over identity.

3.0 How THORN works in brief

THORN contains two main top level functions. **kb->** (enter knowledge base) which enters a list of first-order propositions (props) which are then complied into a **near-Prolog program**. These props constitute the **background theory**. **<-kb** interrogates this theory; given a prop it returns either **true** (the prop is derivable) or **false** or may fail to terminate. If **true** is returned a file **prf.txt** is created in the home directory with the proof enclosed.

kb-> overwrites existing information so it is not possible to increment a knowledge base nor remove any information from it. The types of **kb**-> and <-**kb** are (list prop) $\rightarrow$ symbol and prop $\rightarrow$ boolean respectively.

3.1 First Order Syntax in THORN

1. Logical constants are =>, v, &, ~, <=>, all, exists.
2. Symbols other than logical constants (proper symbols) are propositional atoms.
3. Predicates and functions are represented by proper symbols.
4. A term is either a proper symbol, string, boolean or number or a function heading n (n >= 0) terms.
5. A first-order atom is a predicate followed by n terms all enclosed in […].
6. Any atom is a prop (proposition).
7. If P and Q are props, so is [~ P], [P => Q], [P <=> Q], [P v Q], [P & Q]. Also [all X P] and [exists X P] where X is a proper symbol.
8. The prefix eq stands for =; (e.g. [eq a b] is [a = b]).

prop is a type and hence inputs can be tested for syntax correctness by the type checker. See the problem files for examples of props.

3.2 Flags

Incrementally bounded depth first search is used up to a predetermined finite bound set by the user. THORN 10 is thus incomplete. The global **thorn.*depth*** takes a natural number n and sets this bound at n. The default is 20. **thorn.*=l?*** is a global variable that takes a boolean and if **true** compiles the background theory for reasoning with equality. The default is **false**. **thorn.defaults** is a zero place function that resets these variables to their defaults.

4.0 How THORN works in more detail

4.1 Derivation Rules

There are 6 derivation rules in THORN.

reverse Skolemisation (*revsk*) where P_{sk} is the result of reverse Skolemising P $\frac{P_{sk}}{P}$	&*right* (&r) $\frac{P \quad Q}{(P \,\&\, Q)}$	$\exists$*right* (er) where $P_{Y/X}$ is the result of replacing all free Xs in P by a fresh variable Y $\frac{P_{Y/X}}{(\exists\, X\, P)}$
hypothesis (*hyp*) where P and Q unify $\frac{}{P \mid\!- Q}$	*hypothetical disjunction* (*hypdisj*) where Δ is the set of all complements of Γ where $\Gamma = \{P_1 \ldots P_n\} - \{P_i\}$ $\frac{\Delta \mid\!- P_i}{(P_1 \ldots \text{v}\ P_n)}$	*indirect chaining* where P and R_1 unify with MGU σ $\frac{(\sim R),\ P <\!\!-\!\!- Q \mid\!- \sigma\,(Q \,\&\, \ldots\, R_n)}{P <\!\!-\!\!- Q \mid\!- (R_1 \,\&\, \ldots\, R_n)}$

4.1.1 Comments

Reverse Skolemisation proceeds as does ordinary Skolemisation except that the roles of the quantifiers are transposed; that is to say, universal quantifiers are eliminated in favour of Skolem functions and constants, and existential quantifiers remain. This process converts to prenex form in which the matrix is in CNF.

Reverse Skolemisation links with the ∃*right* rule; the remaining existential quantifiers are eliminated and their bound variables replaced by free variables.

Example: from (f a), (g a) prove (∃ x ((f x) & (g x)))

(f a), (g a) |- (∃ x ((f x) & (g x)))

(f a), (g a) |- ((f X) & (g X)) by ∃*right*

(f a), (g a) |- (f X) by *hyp* {X |-> a}

(f a), (g a) |- (f a) by *hyp*

Indirect chaining combines *backward chaining* and *indirect proof*; that is, in Prolog style given a rule P <-- Q (P if Q) and a conjunctive goal (R_1 & … R_n) if P and R_1 unify with MGU σ then R_1 is replaced by the body Q and (Q & … R_n) are dereferenced by σ. In indirect chaining (~ R) is added as an assumption since it is legitimate to assume the negation of what one is trying to prove.

The rule of *hypothetical disjunction* allows us to deal with a case where the conclusion is a *clause*; that is a disjunction of literals. Given (P_1 … v P_n) we can nominate any P_i as the goal and assume the complements of the remainder. This is really a generalisation of the rule of *disjunctive syllogism* that says from (P v Q) and (~ P) we can conclude Q. Note that *hypothetical disjunction* is non-deterministic since it is not determined which P_i is nominated.

4.2 Ordering the Rules into a Proof Strategy

1. Reverse Skolemisation is applied first.
2. &*right* and ∃*right* are applied to exhaustion.
3. *Hypothetical disjunction* is applied to the clauses generated. This is a choice point.
4. *Proof by hypothesis* and *indirect chaining* are applied to solve the problem. The latter particularly is non-deterministic since different rules may be used to solve the problem.
5. Prolog style chronological backtracking is used to deal with choice points.

4.3 Implementation Notes

In THORN, the Prolog convention is followed that (P <-- (Q & R)) is written (P <-- Q R), much as in Prolog where the conjunction sign is assumed implicitly. The indirect chaining rule looks like this

indirect chaining

where P and R_1 unify with MGU σ

$$\frac{(\sim R),\ P \leftarrow Q \mid\!-_{\sigma}\ \text{append}(Q, (\ldots R_n))}{P \leftarrow Q \mid\!- (R_1 \ldots R_n)}$$

In Prolog there is the limiting case that Q might be empty in which case P <-- Q is termed a **fact**. Hence &right is not needed in stage 4. We implicitly assume the rule

finish

____

()

to terminate a proof when all goals have been solved.

4.3.1 Implementation in Prolog

These rules are implemented in Shen Prolog; the &*right* rule exists in two forms. If in (P & Q), P is ground then the rule is applied such that P is solved without any dependency with Q. In Edinburgh Prolog this can be written using cuts.

```
&right([P & Q]) :- ground?(P), !, solve(P), !, solve(Q).
&right([P & Q]) :- solve(P), solve(Q).
```

Full backtracking is used over the choice points in the proof procedure.

4.4 Completeness for Propositional Calculus

THORN is sound, complete and terminating for propositional calculus at all bounds. It is not complete for first-order logic.

4.5 Compilation of Background Theories

THORN is designed to be used wrt axiom sets which constitute background theories. That is, Prolog compilation techniques are used to compile these theories to efficient code and this code is used to administer the *hyp* and *indirect chaining* rules. Full unification with an occurs check is used.

The compilation of these theories is enabled by entering a list of props to the function kb-> which augments the knowledge base by the props in question. If the compilation succeeds kb-> returns **compiled**. The type of kb-> is (list prop) $\rightarrow$ symbol.

The *modus operandus* of the compilation process is

1. Reduce the props to a list of clauses,
2. For each clause (P_1 … v P_n) generate n *contrapositives* where a contrapositive is created by nominating some Pi as the *head* and forming the complements of the remainder into the *body*. A contrapositive is thus similar in form to a Horn clause except that the head and the elements of the body are literals and can therefore use negation.
3. Compile the contrapositives into code that reflects the indirect chaining rule mentioned.
4. Define contrapositive procedure as similar to a Horn clause procedure i.e. a set composed of all and only those contrapositives whose heads share the same predicate. At the head of each such procedure place code that enacts the *hyp* rule.

A compiler optimisation is to check the background theory to see if it is a Horn clause theory. If so, then the code is compiled without using indirect chaining but instead using ordinary Prolog backward chaining.

4.6 Performance

All figures were gained from a HP Envy workstation using Shen under Chez Scheme.

4.6.1 Propositional Pelletier Problems

The propositional Pelletier problems were solved in times too small to measure in an individual case and were returned with 0.0s. The entire set of these problems, 16 in all, were processed as a batch by loading a file. The cumulative time was 0.015625s to solve all of them and generate proofs. The longest proof was the proof generated by (<-kb [[[p <=> q] <=> r] <=> [p <=> [q <=> r]]]) which took up 4,405 lines. This problem overflows Shen-SBCL.

4.6.2 Set Theory

This background theory was compiled into THORN.

[all x [~ [m x e]]]
[all x [all y [[[sub x y] & [sub y x]] <=> [=s x y]]]]
[all x [all y [[sub x y] <=> [all z [[m z x] => [m z y]]]]]]
[all x [all y [[pow x y] <=> [all z [[m z x] <=> [sub z y]]]]]]
[all x [all y [[com x y] <=> [all z [[m z x] <=> [~ [m z y]]]]]]]
[all x [all y [all z [[prod x y z] <=> [all w [[m w x] <=> [[pair w] & [[m [fst w] y] & [m [snd w] z]]]]]]]]]
[all x [all y [all z [[int x y z] <=> [all w [[m w x] <=> [[m w y] & [m w z]]]]]]]]
[all x [all y [all z [[un x y z] <=> [all w [[m w x] <=> [[m w y] v [m w z]]]]]]]]

The following theorems were posed wrt this background theory.

problem	time (sec)	inferences	steps in proof
(<-kb [un a a a])	0.0	1788	11
(<-kb [int a a a])	0.0	1918	11
(<-kb [all x [all y [all z [[un x y z] <=> [un x z y]]]]])	0.09375	610,778	42
(<-kb [all x [all y [[pow x y] => [m y x]]]])	0.0	1931	7
(<-kb [all x [all y [[prod x y e] => [=s x e]]]])	0.0	1,000	9
(<-kb [all x [all y [[sub x y] => [un y x y]]]])	0.0	16,662	18

4.6.3 Equivalential Calculus

There is one function e where (e x y) holds if x is equivalent to y and one predicate p meaning is provable. THORN could do three of the 13 equivalential calculus problems; the others were terminated after 3 minutes.

problem	time (sec)	inferences	steps in proof
YQL	0.34375	2,974,612	75
YQF	0.0	21,301	85
YQJ	0.109375	1,255,142	69

4.6.4 Schubert's Steamroller

problem	time (sec)	inferences	steps in proof
Schubert's Steamroller	0.171875	3,433,030	78

4.6.5 Mendelson's System L

A Formalisation of Mendelson's axioms for propositional calculus with some theorems therefrom.

[prv [imp P [imp Q P]]]
[prv [imp [imp P [imp Q R]] [imp [imp P Q] [imp P R]]]]
[prv [imp [imp [neg P] [neg Q]] [imp [imp [neg P] Q] P]]]
[[[prv [imp P Q]] & [prv P]] => [prv Q]]

[[prv [or P Q]] <=> [prv [imp [neg P] Q]]]
[[prv [and P Q]] <=> [prv [neg [or [neg P] [neg Q]]]]]
[[prv [equiv P Q]] <=> [prv [and [imp P Q] [imp Q P]]]]

problem	time (sec)	inferences	steps in proof
(<-kb [prv [imp p p]])	0.0	273	7
(<-kb [prv [or p [neg p]]])	0.0	630	8
(<-kb [prv [imp [neg [neg p]] p]])	0.046875	264,489	19

4.7 Equality

Extending reasoning wrt equality within an extended Prolog program requires adding the following rules to the knowledge base.

=left	=right
(Xi = Yi) (F X1 …Yi …Xn) _____ (F X1 … Xn)	_____ (X = X)

In THORN, eq is used as the identity predicate; [all x [eq x x]] is added automatically when equality is required.

For each contrapositive procedure headed by a predicate F the following procedure is appended (**assertz** in Prolog terms).

(F X1 … Xn) <-- (=l-terms [X1 … Xn] [Y1 … Yn]) (F Y1 … Yn)

=l-terms is a Prolog procedure. It allows the replacement of any term Xi by Yi if Xi = Yi can be proven. In Edinburgh Prolog it would be written as:

```
=l-terms([X | Y], [W | Y]) :- eq(X,W).
=l-terms([X | Y], [W | Y]) : - =l-terms(X, W).
=l-terms([X | Y], [X | Z]) :- =l-terms(Y, Z).
```

4.7.1 Group Theory

Equality adds greatly to the branching of search space. Only simple problems can be solved with this addition. The following group theory axioms were used.

[all x [all y [all z [eq [+ x [+ y z]] [+ [+ x y] z]]]]]
[all x [eq [+ e x] x]]
[all x [eq [+ x e] x]]
[all x [eq [+ x [inv x]] e]]

problem	time (sec)	inferences	steps in proof
(<-kb [all a [all x [[eq [+ a x] e] => [eq a [inv x]]]]])	3.593	39,651,779	13
(<-kb [all x [eq [inv [inv x]] x]])	0.765625	16,573,203	13