An Exploration of Semantic Resolution

Louise Dennis

MSc Information Technology: Knowledge Based Systems

Department of Artificial Intelligence

University of Edinburgh

1994

Abstract

Semantic resolution is a suggested refinement of Robinson's resolution method for automatic theorem proving [Robinson 65]. It involves the use of models or interpretations to guide the search for a refutation. Several variants of semantic resolution exist, some of which are only complete for Horn sets. The implementation and testing of two variants, Semantic Lush and Semantic Clash resolution, is described. The performance of both variants with Horn sets was disappointing and some possible reasons for this are suggested. These reasons implied that the performance might well improve as harder problems were tried. The performance of Semantic Clash resolution on non-Horn sets was much more promising than it had been on Horn sets and did indeed appear to be an improvement over the non-semantic methods examined.

Acknowledgements

I would first of all like to thank my supervisors, Prof. Alan Bundy and Dr. Ian Green for their help and guidance during the course of the research reported here.

I would also like to thank Dr. Dave Robertson for the breadth first search algorithm used by SemCheck.

Paf Turner helped to debug the program and wrote an awk script for counting resolvants.

Richard Reeve proof read the whole thesis.

And finally, I thank the SERC who provided finanacial support during my year of study.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is Resolution?

Resolution is a theorem proving strategy first proposed by Robinson [Robinson 65]. It is a method of proof by refutation. That is, it works by proving that the negation of a theorem implies a contradiction. Resolution is a method of deducing consequences of first order predicate logic sentences. The end result of a refutation will be the deduction of the "empty clause" which is equivalent to the deduction of $false$. This means the method has found a contradiction. In theory, at least, resolution is *sound* and *complete*. This means that it can prove any theorem that is true and won't prove any theorem that is false. In practice, time constraints limit the number of true theorems it can actually prove.

1.1.1 Clausal Form

Resolution applies only to first order formulae in "clausal form". It so happens that all mathematical theorems can be translated into clausal form, so this is not a problematic restriction.

A statement is in clausal form if it is a disjunction of "literals", $L_1 \vee ... \vee L_n$, where each literal is a proposition, $P(\bar{x})$, or a negated proposition, $\neg P(\bar{x})$. See example 1.1 below.

Example 1.1 *The following is a clause consisting of 3 literals, $odd(X)$, $even(X)$ and $\neg integer(X)$*

$$odd(X) \vee even(X) \vee \neg integer(X)$$

In clausal form $C \vee false$ is the same as C. Hence the empty clause can be considered as representing $false \vee false \vee false \vee \ldots$ which conflates into $false$.

The empty clause is generally written as $\Box$.

An alternative representation for clauses is Kowalski form. In this the disjunction is represented as an implication. The propositions that were negated are now conjoined before the arrow. The rest of the literals are disjoined after the arrow. See example 1.2 below.

Example 1.2 *The clause in example 1.1 is represented as*

$$integer(X) \rightarrow odd(X) \vee even(X)$$

in Kowalski form

The empty clause is represented by $\rightarrow$.

The former representation is generally more useful in proofs, and as a representation for a computer. The latter, however, is more intuitive and seems to convey more information about what a clause "means" to a human. Both representations are used throughout, as appropriate. It should be fairly clear, however, which is being used at any one point. For full details of clausal and Kowalski forms see [Bundy 83].

1.1.2 Binary Resolution

Binary resolution is a simplified form of resolution. It takes two clauses and removes one literal from each whilst forming a new clause out of the remaining literals. Definition 1.1 formally sets out this process.

Definition 1.1 (Binary Resolution) *Two clauses, $C' \vee P'$ and $C'' \vee \neg P''$ have a binary resolvant if P' and P'' have a most general unifier, ϕ. The resolvant is $(C' \vee C'')\phi$.*

The unifier of two propositions P' and P'' is a set of substitutions of variables, constants and functions for the variables in each clause such that $P'\phi \equiv P''\phi$. The "most general unifier" is the unifier that involves the smallest possible set of such substitutions. $C' \vee P'$ and $C'' \vee \neg P''$ are referred to as the parents of the resolvant, $(C' \vee C'')\phi$.

The following is an example of binary resolution.

Example 1.3 *The binary resolvant of*
$parallel(A, B, C, D) \rightarrow eq(C, B, A, B, C, D)$[1] *and*
$eq(a, c, b, c, a, d) \rightarrow$
is

$parallel(b, c, a, d) \rightarrow$

The most general unifier of $eq(a, c, b, c, a, d)$ *and* $eq(C, B, A, B, C, D)$ *is* $\{C/a, B/c, A/b, D/d\}$[2].

Since the resolution strategy involves unifying unnegated propositions with negated propositions. It is important to distinguish apart those literals that represent unnegated and those which represent negated propositions. The shorthand terminology defined in definition 1.2 is used for this purpose.

Definition 1.2 (Positive and Negative Literals)

- *A positive literal is an unnegated proposition.*
- *A negative literal is a negated proposition.*

1.1.3 Full Resolution

In many cases binary resolution is sufficient to produce a refutation. However sometimes a stronger form of resolution, called full resolution is required. See definition 1.3 below.

Definition 1.3 (Full Resolution) $(C' \vee C'')\phi$ *is a full resolvant of the clauses* $C' \vee P_1' \vee \ldots \vee P_m'$ *and* $C'' \vee \neg P'' \vee \ldots \vee \neg P_n''$ *if* ϕ *is the most general unifier of all the* $P's$ *and* $P''s$

[1]As a convention, throughout the text variables are represented by upper-case letters and constants by lower-case letters

[2]C/a means that a is substituted for C everywhere C occurs in the clause

The following is an example of full resolution.

Example 1.4 *A full resolvant of*
$\rightarrow p(f(X)) \vee q(X)$ *and*
$p(Y) \wedge p(f(a)) \rightarrow$
is

$\rightarrow q(a)$

The most general unifier of $p(Y)$, $p(f(a))$ *and* $p(f(X))$ *is* $\{Y/f(a), X/a\}$.

1.2 What is Semantic Resolution?

As is mentioned above, although resolution is a powerful technique, it is still unable to prove even slightly complex theorems on its own, within a reasonable time. As a result many refinements have been developed to limit the search space involved in a resolution proof. These include Lush resolution [Bundy 83], unit resolution [Henschen & Wos 74], Selected Literal resolution and Semantic resolution, which can be specialised to Set-of-Support resolution [Chang & Lee 73]. Some of these are complete for all sets of input clauses, some are only complete for certain types of clauses, e.g. Horn clauses.

Definition 1.4 (Horn Clauses and Sets)

- *A Horn clause is a clause that contains no more than one positive literal.*
- *A Horn set contains only Horn clauses.*

The idea of using semantic information to guide theorem proving predates resolution. Gelernter's geometry machine used diagrams of the constructions under consideration to guide the search [Gelernter 63]. Gilmore [Gilmore 70] later showed how his method could be applied to resolution. The diagrams were used to order the clauses for consideration and to remove some branches of the search tree from consideration entirely. The geometrical theorems considered by Gelernter and Gilmore were all represented by Horn sets.

From this the general term “Semantic resolution” grew up. It is used to cover several different varieties of resolution theorem proving. Semantic resolution uses the evaluation of the clauses within some model or interpretation to guide the search. Most commonly it requires that one parent in each resolution be falsified by the interpretation. It also frequently requires the resolvant to be false in the interpretation. The reason Semantic resolution is interested in clauses that are false in the interpretation, is that such clauses are more likely to be involved in the proof of a contradiction.

All the various refinements have produced savings in the amount of search undertaken by the theorem prover. Hence there is considerable interest in combining the refinements to produce even more efficient methods. Lush resolution, for instance, is such a combination of Linear and Input resolution.

Unfortunately such combinations are not always complete, so care has to be taken when proposing a new refinement. In particular Lush resolution is incomplete for non-Horn sets unless the Input restriction is weakened to allow ancestor resolution. Even with ancestor resolution it remains incomplete once combined with Semantic resolution.

Issues that need to be considered for any refinement involving Semantic resolution are the efficiency of the strategy in terms of pruning the search space, the clause sets for which the refinement is complete, the types of interpretation that may be used and the efficiency of evaluating clauses within the interpretation.

1.3 An Exploration of Semantic Resolution

On paper Semantic resolution appears a useful refinement of the basic resolution method. The aim of this project was to explore its effectiveness by implementing two versions of Semantic resolution and comparing their performances, both with each other and with those of their semantics-free equivalents. The end view was to see whether it was worthwhile trying to implement Semantic resolution in an Automated Theorem Prover, and, if so, to discover if there were any guidelines that could be provided on subjects such as the choice of interpretation. The two variants chosen were Semantic Lush resolution and Semantic Clash resolution. Semantic Lush resolution is only complete for Horn sets.

Unfortunately, it was only possible to consider a small problem set in the time available and that set consisted primarily of fairly simple problems. The performance of Semantic resolution was disappointing, particularly for Horn sets, suggesting that there may be no value in Semantic Lush resolution. It is certainly possible, however, that this was a side effect of the size of the problems that were being considered. Semantic Clash resolution performed well for non-Horn sets, particularly when applied to the more complex problems.

1.4 Project Outline

The various forms of Semantic resolution are overviewed in chapter 2.

Chapter 3 gives formal definitions of Lush, Semantic Lush, Clash and Semantic Clash resolution. A new variant of Semantic Clash resolution, called Multiple Model Semantic Clash resolution is defined and its completeness is proved for Horn sets.

Chapter 4 discusses the implementations of the four resolution methods in the SemCheck theorem prover.

Chapter 5 discusses issues surrounding the actual testing of the resolution methods, such as the choice of problem sets.

Chapter 6 presents the results of the testing and discusses the issues raised by them. Further work is suggested based both on the results and on other issues raised elsewhere in the project.

1.5 Summary

In this chapter the resolution method was defined and the basic idea behind Semantic resolution was sketched out. There was also a brief discussion of this project and its results.

Chapter 2

Approaches to Semantic Resolution

Broadly speaking an interpretation can function in two ways during Semantic resolution. It can be used as a guide for choosing the parent clauses from which a resolvant is generated, and it can be used to veto certain resolvants from being placed in the search space.

2.1 Some Definitions

Before beginning to describe the various forms of semantic resolution it is important to establish the definitions of model and interpretation.

Definition 2.1 *An interpretation, I, for a set, S, of clauses consists of a nonempty domain, D, of objects and an assignment of the predicate, function, and constant symbols occurring in S such that each* n*-ary predicate symbol is assigned an* n*-ary relation on D, each* n*-ary function symbol is assigned an* n*-ary function of D into D, and each constant symbol is assigned an element of D.[Henschen 76]*

Definition 2.2 *A clause is false in the interpretation I if it has a ground instance which is false in I. Otherwise it is true in the interpretation.*

Definition 2.3 *An interpretation is a model for a set of clauses, S, if all the clauses in S are true in the interpretation.*

2.2 Semantic Clash Resolution

Slagle was the first to propose a form of Semantic resolution [Slagle 67]. It is a refinement of Clash resolution (see chapter 3). Clash resolution builds up chains of resolvants from a "nucleus" clause and a set of "electron" clauses. Semantic Clash resolution requires that all the clash resolvants generated should be false in the interpretation supplied and that all the electrons involved in a clash should be false. As Slagle proposed it, this is complete for renamable sets. A renamable set is one that can be converted into a Horn Set by "renaming" some (or all) of the literals.

Example 2.1 *For instance, the set* $\{A_1 \vee A_3, \neg A_2 \vee A_3\}$ *can be* renamed *as the Horn Set* $\{A_1 \vee \neg A_4, \neg A_2 \vee \neg A_4\}$ *where* $A_4 \equiv \neg A_3$.

Slagle calls his version *Maximal* Semantic Clash resolution. In the ground case this requires that every literal in the nucleus of the clash that is true in the Model be resolved upon. However, it doesn't appear that he intends this condition to be transferred to non-ground situations.

Chang and Lee [Chang & Lee 73] extended Slagle's method with relative ease to non-Horn sets. They use a slightly different definition of a clash, no longer stipulating that the electrons should resolve away literals in the nucleus. It isn't clear whether or not this is necessary in order to preserve completeness.

A version of Semantic Clash resolution is used by Slaney in his SCOTT program [Slaney 94] and proved to be more efficient than the resolution method already implemented in the OTTER Theorem Prover. The SCOTT algorithm is similar, but not identical to either the versions of Slagle or Chang and Lee. In particular it places no conditions on the actual resolvants of a clash, though Slaney mentions that these could be implemented.

Semantic Clash resolution is complete when only one interpretation is used. If the statement of a theorem contains only Horn clauses, then more than one model can be used to choose electrons etc. More on this in chapter 3.

2.3 Semantic Lush Resolution

Bundy [Bundy 83] combines Semantic resolution with Lush resolution. This is based on the work of Gilmore and Gelernter on the Geometry Machine [Gelernter 63] [Gilmore 70]. Semantic Lush resolution is only complete for Horn sets. The semantic component places no explicit restriction on the parents of a resolution. It requires only that all resolvants be falsified by an interpretation[1] in which all the hypotheses of the theorem (i.e. all the input clauses except the goal clause) are true[2]. However, in terms of evaluation it is very powerful allowing multiple small models of the input clauses to be used each of which may veto the development of a branch.

2.4 Semantic Positive Unit Resolution

Definition 2.4 (Unit Clauses) *A unit clause contains only one literal. If that literal is positive then the clause may also be referred to as a positive unit. If the literal is negative, then the clause is a negative unit.*

Positive Unit resolution requires that one parent of the resolution is a positive unit. It is complete for Horn sets. Henschen [Henschen 76] attempts to combine Semantic resolution with Positive Unit resolution. This requires some relaxation of one of the strategies in order to guarantee completeness. He does this by requiring that either one parent is a false positive unit in the interpretation or both one parent and the resolvant are false. Again this is only known to be complete for Horn sets. Henschen regards his method not only as a way to prune the search space but also as a way to guide the search by giving a preference to certain

---

[1] In reality this means that one parent of every resolution will also be false in this interpretation

[2] This type of interpretation has several interesting properties when used with Horn sets. It is discussed and defined more fully in chapter 3

axioms. He does this by considering using an interpretation that falsifies one or more of the axioms. This being the case the process clearly no longer allows the arbitrary addition of further interpretations since it would be possible to have a set of interpretations such that every one of the input clauses were true in at least one. If the interpretations were to be restricted to those used by Bundy [Bundy 83] then further interpretations could be added but the technique would essentially become a form of standard Semantic Lush resolution. The only input clause false in such model is the "goal clause", which contains no positive literals. No resolvant of a Horn set will contain a positive literal. Hence there would be no positive units in the refutation that could be falsified by the interpretation.

2.5 SL$\mathcal{M}_n$, Linear Selected Literal Resolution with Models

A fourth approach combines Semantic resolution with Selected Literal resolution [Brown 73]. It is called SL$\mathcal{M}_n$ (Selected Literal, Linear, with Models $\mathcal{M}_1, ..., \mathcal{M}_n$). For every resolvant produced, one parent must be an input clause and the other must be false in all the Herbrand models $\mathcal{M}_1, ..., \mathcal{M}_n$. Furthermore every clause has an ordering that dictates which literals may be resolved upon. Herbrand models are sets of ground literals. A literal (clause) is regarded as true in the model if it is contained in the model. This creates evaluation problems for some combinations of non ground clauses with models, as is illustrated below.

Example 2.2 *Say clause* $C \equiv P(x)$ *and we are looking at the model* $\mathcal{M} = \{P(a), \neg P(b)\}$. *C is neither contained in the model nor is it out of the model.*

In this case Brown *decomposes* C into two separate clauses, $P(a)$ and $P(b)$. For his method to be complete, the clause set must first be *fully decomposed* with respect to the models employed, i.e. there are no clauses that present evaluation problems, such as that in example 2.2, in any of the models. Brown's method is complete for renamable sets.

2.6 Hereditary Lock Resolution

Sandford [Sandford 80] combines Semantic resolution with Boyer's Lock-resolution [Boyer 71] and Luckham's Model Strategy [Luckham 70] into a system he calls Hereditary Lock resolution (HLR). This is complete for both Horn and non-Horn sets. The strategy builds up False Substitution Lists (FSLs) for each clause.

A FSL is a list of literals that appear in the set. These literals may have some or all of their free variables replaced by constants. A FSL is said to be feasible if there is some substitution which converts every literal in the FSL into a ground literal that is false in the model. Input clauses have empty FSLs.

Each clause has a truth value designator (t.v.d.). This can be T, F or T/F. The t.v.d. depends upon the relationship between the clause and its FSL in the model. Consider all the assignments of model domain elements to free variables occurring in the clause. If the clause is true under all the assignments that make the FSL false (i.e. all the literals in the FSL are false) then the t.v.d. of the clause is T. If the clause is false under all the assignments in which the FSL is false then the t.v.d. is F. Otherwise the t.v.d. is T/F.

Each resolvant must have one false parent (i.e. one whose t.v.d. is F or T/F) and one true parent (whose t.v.d. is T or T/F). All resolvants must have feasible FSLs. The selected literal part of the strategy is achieved by using a double lock numbering to decide which literals are to be resolved upon.

Sandford proves that HLR is complete for any sound semantic model. A sound semantic model has an associated Herbrand interpretation such that any sentence that is false in the Herbrand interpretation is also false in the model. Unfortunately it was never established whether HLR was complete under subsumption checking and elimination of tautologies (standard Lock resolution is incomplete under elimination of tautologies, but Sandford has hopes that the double lock system might avoid this problem). One advantage of Sandford's method is the inclusion of the third "truth value" which may be regarded as either true or false. This allows for the use of infinite models in which only some of the possible substitutions are checked for falsity, undecided cases being marked T/F. The method is, however, very complex and hard to understand intuitively.

2.7 Summary

This chapter looked briefly at several forms of Semantic resolution, namely, Semantic Clash resolution, Semantic Lush resolution, Semantic Unit resolution, SL$\mathcal{M}_n$ and HLR. It also gave definitions of the important concepts of interpretations, models, and truth in an interpretation.

Chapter 3

Lush Resolution, Clash Resolution and their Semantic Equivalents

In the time available it was impossible to investigate all the various forms of Semantic resolution. In the end two were chosen for closer consideration, Semantic Lush and Semantic Clash resolution. This chapter defines them in detail. It also defines an extension of Semantic Clash resolution named "Multiple Model Semantic Clash Resolution" and proves its completeness for Horn sets.

3.1 Model-Like Interpretations

Semantic Lush resolution as described in [Bundy 83], does not use arbitrary interpretations of the input clauses but only interpretations of a particular sort. In the end interpretations of this type were the only ones used in testing the two methods.

These interpretations are essentially models of the theory and theorem to be proved. That is they are models of the sentences of the theorem *before* they are placed in the clausal form used by the resolution procedure. Sentences are placed in this clausal form by a process of negating the conjecture to be proved, converting to a disjunctive form, replacing variables bound by universal quantifiers with free variables and those bound by existential quantifiers by skolem constants and functions. Full details of this can be found in [Bundy 83].

I've called these Model-like interpretations and defined them below. Where the word "model" is used in the text it should be understood as meaning "Model-ikeinterpretation" unless specified otherwise.

Definition 3.1 *A Model-like interpretation, I, with respect to a set of clauses S is an interpretation such that:*

1. *There is a goal clause, G, which contains no positive literals, in S. G is false in I*

2. *All other clauses are true in I.*

3.2 Lush Resolution

Lush resolution produces a proof tree, with the branches representing possible paths to the empty clause and the nodes representing goal clauses. In its basic format, described below, it is complete for Horn sets but not for non-Horn sets.

Definition 3.2 (Lush Resolution) *Let S be a set of clauses which contains a goal clause, G, and has a subset of input clauses, I. A resolvant, R, is a Lush resolvant if*

1. *one parent is the goal clause, G.*

2. *the other parent is in I.*

Definition 3.3 (Lush Refutation) *A Lush refutation of a set of clauses, S, is a chain of resolvants, $R_1, ..., R_n(= \Box)$ such that R_i is a Lush resolvant of the set $\{S, R_1, ..., R_{i-1}\}$ taking S as the input clauses and R_{i-1} as the current goal.*

Essentially each chain $R_1, ..., R_n$ is a branch on the proof tree. Figure 3–1 is an example of a Lush refutation search tree.

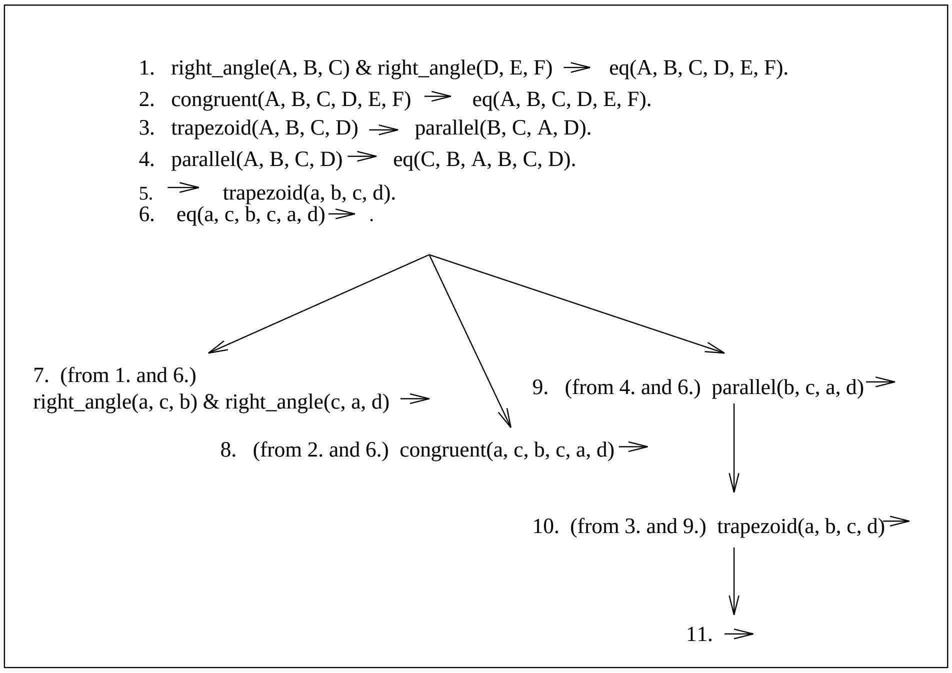

Figure 3–1: A Lush Refutation Search Tree

3.2.1 Adapting Lush Resolution to Non-Horn Sets

Two methods have been suggested for expanding Lush resolution to non-Horn sets [Bundy 83]. These are case analysis and ancestor resolution.

Case Analysis

Case analysis suggests taking each non-Horn clause and breaking it down into Horn clauses. This is done by taking each positive literal of the clause and disjoining it with the clause's negative literals to form a new (Horn) clause. If the theorem can be proved with Lush resolution for each of these new Horn clauses, so the method asserts, then it is true.

Example 3.1 *Say, for instance, you wished to deduce the empty clause from the following clauses (from [Bundy 83]).*

1. $\neg is\_prime(X) \vee is\_prod\_primes(X)$.

2. $\neg is\_power\_3(X) \vee is\_prod\_primes(X)$.

3. $is\_prime(a) \vee is\_power\_3(a)$.

4. $is\_prod\_primes(a)$.

This contains one non-Horn clause, namely, $is\_prime(a) \vee is\_power\_3(a)$. This is split into $is\_prime(a)$ and $is\_power\_3(a)$. Case analysis then asserts that if a Lush refutation can be found of

1. $\neg is\_prime(X) \vee is\_prod\_primes(X)$.

2. $\neg is\_power\_3(X) \vee is\_prod\_primes(X)$.

3. $is\_prime(a)$.

4. $is\_prod\_primes(a)$.

and

1. $\neg is\_prime(X) \vee is\_prod\_primes(X)$.

2. $\neg is\_power\_3(X) \vee is\_prod\_primes(X)$.

3. $is\_power\_3(a)$.

4. $is\_prod\_primes(a)$.

Then the theorem is true.

This appears to be a plausible contention. However, what is being claimed is that

$$A \equiv \forall \bar{x}(\Phi(\bar{x}) \wedge (P(\bar{x}) \vee Q(\bar{x})))$$

(where Φ is a conjunction of clauses) is unsatisfiable iff

$$B \equiv \forall \bar{x}(\Phi(\bar{x}) \wedge P(\bar{x})) \wedge \forall \bar{x}(\Phi(\bar{x}) \wedge Q(\bar{x}))$$

is unsatisfiable. But, although

$$\not\models A \Rightarrow \not\models B,$$

It is not the case that

$$\not\models B \Rightarrow \not\models A.$$

Example 3.2 *Consider the set,* $S = \{P(a), \neg P(b), Q(b), \neg Q(a), \forall x(P(x) \vee Q(x)\}$*. This set is clearly satisfiable. However the sets*
$S'_1 = \{P(a), \neg P(b), Q(b), \neg Q(a), \forall x P(x)\}$ *and*
$S'_2 = \{P(a), \neg P(b), Q(b), \neg Q(a), \forall x Q(x)\}$ *are both unsatisfiable.*

This means there are Lush refutations of both S'_1 and S'_2 when S is in fact satisfiable. Hence case analysis is unsound. Henschen suggests in [Henschen 76] that it may be possible to get round this by insisting that "consistent" substitutions be used in the individual cases. If so the above example would fail since b will be substituted for x in the refutation of S'_1 while a will be substituted in the refutation of S'_2.

Ancestor Resolution

The alternative to case analysis is ancestor resolution which relaxes the Input restriction. Instead of one parent of the resolution being in the Input set, it requires that one parent of the resolution be in the branch for which the goal clause is the leaf node. Although this will mean an increase in the size of the search space, it is certainly sound, making it preferable to case analysis. It should be noted that when ancestor resolution is used, goal clauses will no longer necessarily contain only negative literals. The initial goal clause in a refutation will not contain any positive literals but subsequent resolvants, which are used as goal clauses, may contain some.

Lush resolution with ancestor resolution is called SL resolution.

3.2.2 Semantic Lush Resolution

Semantic Lush resolution is Lush resolution (either with or without ancestor resolution) with the added restriction that the resolvant must be false in all the models (Model-like interpretations) of the input set provided. A Semantic Lush refutation is one in which all the resolvants are Semantic Lush resolvants.

It is important to note that, when applied to Horn sets, Semantic Lush resolution can only veto branches that don't lead to a solution. Any branch that could terminate in □ at some point will be left in the search space. This follows from the proof of its completeness for Horn sets [Bundy 83].

Semantic Lush resolution is only complete for Horn sets since even one model can prove too powerful for non-Horn sets (see figure 3–2). This tree is produced using SL resolution. However, the model *num3* in which a takes the value 3, and the various predicates take their obvious values, vetos both the clause marked $$ and the clause marked ++. More on this can be found in [Bundy 83]

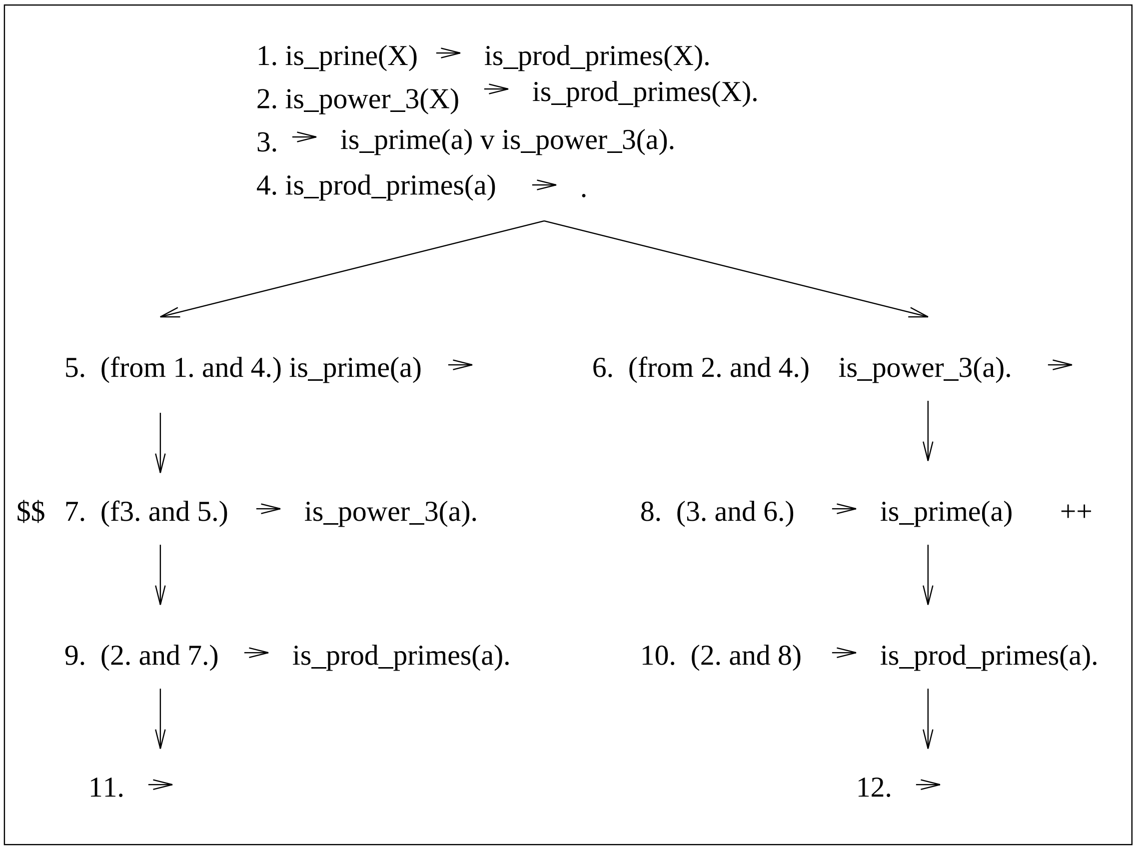

Figure 3–2: A non-Horn Clause Search Tree [Bundy 83]

3.3 Clash Resolution

Definition 3.4 (Clash Resolution) *Let P be an ordering of predicate symbols. A finite set of clauses $\{E_1, ..., E_q, N\}, q \geq 1$, is called a* clash *with respect to P (or P-*clash *for short), if and only if $E_1, ..., E_q$ (called* electrons*) and N (called the* nucleus*) satisfy the following conditions:*

1. *Let $R_1 = N$. For each $i = 1, ..., q$, there is a resolvant R_{i+1} of R_i and E_i.*

2. *The literal in E_i which is resolved upon contains one of the largest predicate symbols in E_i, for each $i = 1, ..., q$ according to the ordering P.*

*R_{q+1} is called a PI-*resolvant *(of the PI-clash $\{E_1, ..., E_q, N\}$.*

Definition 3.5 (Clash Refutation) *A Clash refutation of a set of clauses, S, is a tree of clauses. The empty clause, $\Box$, is the root and the input clauses S are the leaves. Each node is a Clash resolvant of some of the nodes above it in the tree. i.e. the nodes on the first level are Clash resolvants of the input clauses, those on the second level, Clash resolvants of the input clauses and the first level resolvants, and so on. For any node in the tree the clauses in the branch above it are said to be a Clash deduction of that node. A Clash refutation of a set of clauses, S, is also a Clash deduction of $\Box$ from S.*

3.3.1 Semantic Clash Resolution

Definition 3.6 *Let I be an interpretation. Let P be an ordering of predicate symbols. A finite set of clauses $\{E_1, ..., E_q, N\}, q \geq 1$, is called a* semantic clash *with respect to P and I (or PI-*clash *for short), if and only if $E_1, ..., E_q$ (called* electrons*) and N (called the* nucleus*) satisfy the following conditions:*

1. *$E_1, ..., E_q$ are false in I.*

2. *Let $R_1 = N$. For each $i = 1, ..., q$, there is a resolvant R_{i+1} of R_i and E_i.*

3. *The literal in E_i which is resolved upon contains one of the largest predicate symbols in E_i, for each $i = 1, ..., q$.*

4. *R_{q+1} is false in I.*

*R_{q+1} is called a PI-*resolvant *of the PI-clash $\{E_1, ..., E_q, N\}$ [Chang & Lee 73].*

A Semantic Clash refutation is the same as a Clash refutation except with Semantic Clash resolvants. The above is the version of Semantic Clash resolution given by Chang and Lee. As stated in chapter 2, Slagle defines it slightly differently [Slagle 67]. In particular the electrons have to resolve away literals occurring in the nucleus. He gives a ground version of the method, as well, in which all the literals in the nucleus that are true in the interpretation are resolved away by the clash.

3.3.2 Multiple Interpretation Semantic Clash Resolution Defined

As it is defined Semantic Clash resolution only allows the use of one interpretation for choosing electrons and vetoing resolvants. Semantic Lush resolution, on the other hand, can use several interpretations to veto branches in a proof. It is conceivable that in some cases a large model would appear the most effective in vetoing resolvants. However, any saving made might well be lost due to the time taken in evaluating the clauses within the model. In many cases large models can be broken down into a cross product of smaller models. Any statement that is false in the large model will be false in at least one of the smaller models. What is more, there may be sentences that are false in the large model that are true in one of the small models.

Example 3.3 *Let I_1 have domain $\{a\}$ and I_2 have domain $\{b\}$. Each interpretation has one predicate symbol in the language, $p1$ and $p2$ respectively. $p1(a)$ is true whilst $p2(b)$ is false. Consider the clause, $C \equiv p(X)$. Assume that p is interpreted as $p1$ in I_1 and $p2$ in I_2. C is true in I_1 and false in I_2. In the cross product $I_1 \times I_2$, $p(X)$ translates as $(p1(Y), p2(Z)), Y \in I_1, Z \in I_2$. This is false in $I_1 \times I_2$, since $p2(Z)$ is false in I_2. Hence C is false in $I_1 \times I_2$, but true in I_1.*

It takes considerably less time to evaluate a statement in a number of small models than it does to evaluate it in one large model. Hence the use of multiple interpretations is a potentially powerful tool for increasing the efficiency of Semantic Lush resolution especially since several small models, combined, may well have *more* power of veto than their cross product. It seemed worthwhile investigating the possibility of applying the same idea to Semantic Clash resolution. It turned out that this was possible for Horn sets, provided all the interpretations used were Model-like.

As an aside, Henschen considers using several small interpretations to represent one large one [Henschen 76]. Unfortunately, his Semantic Positive Unit resolution is not complete under the arbitrary addition of extra interpretations. Hence, he has a problem with the fact that a clause may be vetoed by one of the smaller interpretations when it is false in the larger one. As a result he insists that a resolvant be false in *all* the smaller interpretations before allowing it to be vetoed from the clause space.

The multiple interpretation version of Semantic Clash resolution is as follows.

Definition 3.7 *Let $I_1, ..., I_n$ be Model-like interpretations. Let P be an ordering of predicate symbols. A finite set of clauses $\{E_1, ..., E_q, N\}, q \geq 1$, is called a semantic clash with respect to P and $I_1, ..., I_n$ (or $P[I_1, ..., I_n]$-clash for short), iff $E_1, ..., E_q$ (called electrons) and N (called the Nucleus) satisfy the following conditions:*

1. *$E_1, ..., E_q$ are false in each of $I_1, ..., I_n$.*
2. *Let $R_1 = N$. For each $i = 1, ..., q$, there is a resolvant R_{i+1} of R_i and E_i.*
3. *The literal in E_i which is resolved upon contains one of the largest predicate symbols in E_i,for each $i = 1, ..., q$.*
4. *R_{q+1} is false in each of $I_1, ..., I_n$.*

R_{q+1} is called a $P[I_1, ..., I_n]$-resolvant.

Definition 3.8 *A Multiple Model Semantic Clash refutation, or $P[I_1, ..., I_n]$-clash refutation is a Clash refutation in which all the nodes are $P[I_1, ..., I_n]$-clash re-*

solvants of their daughters, as opposed to Clash resolvants. Similarly a $P[I_1, ..., I_n]$*-clash deduction of a Clause is a Clash deduction in which all the nodes are* $P[I_1, ..., I_n]$*-clash resolvants of their daughters.*

The Completeness of Multiple Model Semantic Clash Resolution For Horn Sets

The proof of completeness begins by proving that if a minimally unsatisfiable ground Horn set is true in some Model-like interpretation then all the atoms (propositional letters) are true in that interpretation. Using this fact it is then possible to prove ground completeness and from that (with the aid of a lifting lemma) general completeness.

Lemma 3.1 *If S is a minimally unsatisfiable set of ground Horn clauses and I is a Model-like interpretation of S then all the atoms of S are true in I.*

Proof. The proof proceeds by induction on the size of the atom set, A, of S. In the case of ground clauses the atoms can be regarded as propositional letters. A literal will be either an atom or a negated atom.

Base Case

$A = 1, \{Q\}$, say. S contains the clauses Q and $\neg Q$. $\neg Q$ must be the goal clause which is false in I, therefore Q must be true in I.

Inductive Hypothesis

For all $0 < i \leq n$, if S is a minimally unsatisfiable Horn set containing i atoms, and I is a Model-like interpretation of S then all the atoms of S are true in I.

Inductive Step

Let $A = n + 1$.
There are two cases.

Case 1. All the atoms appear in the goal clause. Since this is false in I all the atoms are true in I.

Case 2. There is an atom that does not appear in the goal clause. For a contradiction suppose there is an atom, Q, not appearing in the goal clause that is false in I.

Q must appear as a literal at least twice in S, once negated and once unnegated otherwise S would not be minimally unsatisfiable. Since S is a Horn set, the clause, C, in which Q appears as a positive literal must be of the form

$$C \equiv \Phi \vee Q$$

where Φ is a disjunction of negative literals. We know C is true in I. Since Q is false in I, Φ must be true in I.

Remove all clauses containing $\neg Q$ and remove Q from all the remaining clauses, to produce the set S'. S' is unsatisfiable (otherwise $I' \cup \neg Q$ would satisfy S (where I' is some model of S' not involving Q[1])). Since Q does not appear in the goal clause of S that clause remains in S' and is false in $I \setminus \{\neg Q\}$. Any sentence in S of the form $\Phi \vee Q$ becomes the sentence Φ in S' and which is true in $I \setminus \{\neg Q\}$. All clauses in S that did not contain Q or $\neg Q$ are unchanged and are true in $I \setminus \{\neg Q\}$ if they were true in I. Hence, since I was a Model-like interpretation of S, $I \setminus \{\neg Q\}$ is a Model-like interpretation of S'. By Induction, all atoms in S' are true in $I \setminus \{\neg Q\}$, *but* Φ is a disjunction of negative literals, therefore Φ is false in $I \setminus \{\neg Q\}$, and we have a contradiction. Therefore all atoms are true in I.

This result has some interesting implications. For instance, it implies that all Model-like interpretations of ground Horn sets are equivalent. All positive literals will be true in the interpretation, all negative literals false. Hence, there is no benefit to be gained in using more than one interpretation in ground cases. The result does not obviously carry over into non-ground sets, since one interpretation may make different literals true under different assignments of variables. However, the results did suggest an interesting side issue would be the examination of the effectiveness of using a multiplicity of models in Semantic resolution.

[1] This is possible since all occurrences of Q have been removed from S

Theorem 3.1 (Ground Completeness of Multiple Model Semantic Clash Resolution For Horn Sets) *Let P be an ordering of predicate symbols in a finite unsatisfiable set S of ground Horn clauses. And let $I_1, ..., I_n$ be Model-like interpretations of S then there is a $P[I_1, ..., I_n]$-clash deduction of $\Box$ from S.*

Proof. As discussed above all interpretations of ground Horn sets are equivalent. Choose one such interpretation, I.

We know there is an PI-clash deduction of $\Box$ from S using I [Chang & Lee 73]. Hence there is a $P[I]$-clash deduction of $\Box$ from S.

There now follows the lifting lemma that will allow ground completeness to be converted into general completeness. It is adapted from the lifting lemma for binary resolution that can be found in [Chang & Lee 73].

Lemma 3.2 (Lifting Lemma) *If $E'_1, ..., E'_n, N'$ are instances of $E_1, ..., E_n, N$ respectively and P is an ordering of the predicate symbols in $\{E_1, ..., E_n, N\}$ such that R' is a clash-resolvant of $\{E'_1, ..., E'_n, N'\}$, then there is a clash-resolvant R of $\{E_1..., E_n, N\}$ s.t. R' is an instance of R.*

Proof. Throughout this proof clauses are treated as sets of literals. That is a clause $L_1 \vee L_2$ is represented as $\{L_1, L_2\}$. This makes it easier to represent the way resolution forms new clauses, since it can be done using established set theory notation.

First standardize apart $E_1, ..., E_n$, and N.

Let L'_i, N'_i be the literals in E'_i and N' resolved upon, hence

$$R' = (E'_1\gamma \setminus \{L'_1\gamma\}) \cup ... \cup (N'\gamma \setminus (\{N'_1, .., N'_n\}\gamma)$$

where γ is the composition of the most general unifiers, γ_i, of L'_i [2] and $\neg N'_i$, i.e $\gamma = \gamma_1 \cup \ldots \cup \gamma_n$.

---

[2] L_i is the "largest" literal in E_i

Since $E'_1, ..., E'_n, N'$ are instances of $E_1, ..., E_n, N$ there is a θ s.t. $E'_i = E_i\theta$ $0 < i \leq n$ and $N' = N\theta$ [3].

Let $L_i^1, ..., L_i^{r_i}$ be the literals in E_i corresponding to L'_i (i.e. $L_i^1\theta = ... = L_i^{r_i}\theta = L'_i$) since unification won't remove predicate letters, $L_i^1, ..., L_i^{r_i}$ contain the largest predicate in E_i .

Similarly let $N_i^1, ..., N_i^{s_i}$ be the literals in N corresponding to N'_i

Obtain a most general unifier λ_i for $\{L_i^1, ..., L_i^{r_i}\}$.
Obtain a most general unifier μ_i for $\{N_i^1, ..., N_i^{s_i}\}$.

Let $L_i = L_i^1\lambda_i$ and $N_i = N_i^1\mu_i$.
Then L_i is a literal in the factor $E_i\lambda_i$ of E_i and N_i is a literal in the factor $N\mu_i$ of N.

Let $\lambda = \lambda_1 \circ ... \circ \lambda_n \circ \mu_1 \circ ... \circ \mu_n$. Note that both θ and λ unify $\{L_i^1, ..., L_i^{r_i}\}$ and $\{N_i^1, ..., N_i^{s_i}\}$ for all $0 < i \leq n$. However, θ also grounds any other literals appearing in $E_1, \ldots, E_n, N$ hence λ is more general than θ.

Clearly L'_i is an instance of L_i and N'_i is an instance of N_i. Since $L'_i\gamma_i = \neg N'_i\gamma_i$ L_i and $\neg N_i$ are unifiable. Let σ_i be a most general unifier of L_i and $\neg N_i$. Let $\sigma = \sigma_1 \cup ... \cup \sigma_n$. Note that since $L'_i\gamma_i = \neg N'_i\gamma_i$ and L'_i is an instance of L_i and $L_i\sigma_i = \neg N_i\sigma_i$, σ_i is more general than γ_i Let

$$R = ((E_1\lambda)\sigma \setminus \{L_1\sigma\}) \cup ... \cup ((N\lambda)\sigma \setminus \{N_1, ..., N_n\}\sigma)$$

$$= ((E_1\lambda)\sigma \setminus (\{L_1^1, ..., L_1^{r_1}\}\lambda)\sigma) \cup ... \cup \ ((N\lambda)\sigma \setminus (\{N_1^1, .., N_1^{s_1}, N_2^1, ..., N_n^{s_n}\}\lambda)\sigma).$$

$$= (E_1(\lambda \circ \sigma) \setminus \{L_1^1, ..., L_1^{r_1}\}(\lambda \circ \sigma)) \cup ... \cup (N(\lambda \circ \sigma) \setminus \{N_1, .., N_1^{s_1}, ..., N_n^{s_n}\}(\lambda \circ \sigma)).$$

R is a resolvant of $E_1, ..., E_n, N$. R' is an instance of R since

$$R' = (E'_1\gamma - L'_1\gamma) \cup ... \cup (N'\gamma - \{N'_1, .., N'_n\}\gamma)$$

$$= (E_1\theta)\gamma \setminus \{(L_1\theta)\gamma\} \cup ...$$

---

[3] Only one θ is needed because the electrons and nucleus have all been standardized apart

$$= (E_1(\theta \circ \gamma) \setminus ...$$

and $\lambda \circ \sigma$ is more general than $\theta \circ \gamma$, since σ is more general than γ and λ is more general than θ.

Theorem 3.2 (Completeness of Multiple Interpretation Semantic Clash resolution for Horn Sets) *If P is an ordering of predicate symbols in a finite and unsatisfiable set S of clauses, and if $I_1, ..., I_n$ are Model-like interpretations of S, then there is a $P[I_1, ..., I_n]$-clash deduction of $\Box$ from S.*

Proof. The proof proceeds by showing how a $P[I_1, ..., I_n]$-clash deduction of $\Box$ from S can be constructed from a $P[I_1, ..., I_n]$-clash deduction of $\Box$ from S', a set of ground instances of the sentences in S. By ground completeness (theorem 3.1) we know that if S' is unsatisfiable there is a $P[I_1, ..., I_n]$-clash deduction of $\Box$ from S'. And by Herbrand's theorem we know that if S is unsatisfiable there is a finite unsatisfiable set S' of ground instances of clauses in S. Hence it follows that if S is unsatisfiable there is a $P[I_1, ..., I_n]$-clash deduction of $\Box$ from S. Hence Multiple Interpretation Semantic Clash resolution is complete for Horn sets.

Let S' be a finite unsatisfiable set of ground instances of clauses in S. And let D' be a $P[I_1, ..., I_n]$-clash deduction of $\Box$ from S'. From D', we can produce a $P[I_1, ..., I_n]$-clash deduction of $\Box$ from S by replacing clashes in the proof as follows:

Replace each ground input clause by a clause in S of which it is an instance. Say the electrons and nucleus of a clash resolvant in D' have all been replaced by clauses in S. These clauses constitute a clash (by lemma 3.2). Replace their resolvant with the clash-resolvant of which it is an instance (again using lemma 3.2). This is a $P[I_1, ..., I_n]$-clash resolvant of the electrons and nucleus since all the electrons and the resolvant have a ground instance that is false in all the interpretations. In this fashion, each ground resolvant in D' is replaced by a clause of which it is an instance. The terminal node is the empty clause, $\Box$, hence it is replaced by $\Box$. It is easy to see that this new deduction tree is a $P[I_1, ..., I_n]$-clash deduction of $\Box$ from S.

3.4 Summary

This chapter gave formal definitions of Lush, Clash, Semantic Lush and Semantic Clash resolution. It also defined several other concepts such as Model-like interpretations, Lush refutation and the like. An extension of Semantic Clash resolution was proposed and its completeness proved for Model-like interpretations and Horn sets. In the process it was noted that there is, essentially, only one Model-like interpretation of a ground Horn set.

Chapter 4

The SemCheck Resolution Theorem Prover

The Program used for testing the various resolution refinements was written in Sicstus Prolog and has been called SemCheck for ease of reference.

4.1 Clause Representation

Each individual literal in a clause was represented separately as

```
cls(Clause_Name, Literal, positive)  or
cls(Clause_Name, Literal, negative)
```

depending on whether the literal was positive or negative.

Example 4.1 *The predicates*

```
cls(trap, trapezoid(x1, x2, x3, x4), negative).
cls(trap, parallel(x2, x3, x1, x4), positive).
```

represent the clause

$$trapezoid(x1, x2, x3, x4) \rightarrow parallel(x2, x3, x1, x4)^1.$$

[1]Semcheck represents variables as atoms that beginning with an x

Resolvants of clauses were inserted into the database as they were formed and labeled `r(Clause1, Clause2)` where `Clause1` and `Clause2` were the parent clauses in the case of Lush resolution. In the case of Clash resolution `Clause1` was a list of the electrons used.

This representation proved to be problematic. The algorithms used for resolution required a considerable amount of backtracking. A breadth first search strategy was employed. At each level of the search all the resolvants for the next level were generated by enforced backtracking through the resolution algorithm calling the `setof` predicate. This algorithm first chose a name for the new clause and then asserted the literals it contained. When this algorithm was backtracked through, the new literals were asserted *before* the name was changed from the previous resolvant.

Eventually the simplest solution found was to maintain a 'literal pool' during the resolution procedure. Initially consisting of all the literals in the parent clauses, members could be removed from the list as resolution took place. Once the final 'pool' was decided upon *then* a name was chosen for the resolvant and the new clause asserted. Clearly, in retrospect, a representation that involved such 'pools' from the beginning would have been better.

4.2 Search Strategy

Lush resolution uses a branch structure in the search space, in which resolvants can only be formed from clauses in the same branch. Clash resolution produces levels of resolvants. The first level is formed from clashes in the input clauses. The second from clashes in the input clauses and resolvants in the first level, and so on. At all times resolvants can be formed from any clauses in the higher levels.

Clearly, talk of "search strategy" in terms of clash resolution is fairly irrelevant. The only course is to produce all the resolvants in each level in turn and then proceed to the next.

With Lush resolution, however, there is a choice between depth first search, exploring each branch until either the empty clause is reached or no further resolvants can be formed, and breadth first search extending all the branches by one resolvant at a time. Depth first search uses up less memory space, but it is unknown how many, and how deep the branches are that it explores before terminating that branch. Hence there is a risk that a depth first search strategy may spend a lot of time exploring a very deep branch when a solution could be found on a much shorter branch. Breadth first search, on the other hand, will find the shortest solution, but is more expensive in terms of computer memory etc. since it has to keep track of all possible branches.

Breadth First Search was chosen for SemCheck. Since Clash resolution produces the resolvants a level at a time, it seemed desirable for reasons of comparability to mimic a similar process for the lush resolution. A Prolog algorithm for breadth first search based on one written by Dr. Dave Robertson was used.

4.3 The Resolution Mechanism

The overall resolution procedures for the Lush and Clash resolution varied substantially since Lush resolution only ever employs two clauses in generating a resolvant whereas clash resolution employs several. However, clash resolution uses intermediate resolvants in building up the final clash, each of which employ two clauses, therefore the basic algorithms for producing a resolvant from two clauses were very similar.

4.3.1 The General Resolution Algorithm

The general form of the Resolution Algorithm used is as follows:

```
RESOLVE <Clause A> <Clause B>
        FORM THE SET (a) OF <pairs of literals with their Unifiers>
                one from Clause A and one from Clause B, which
                unify and comply with any constraints (e.g. in
                clash resolution it is the ``largest'' literal in
                the Electron that is being resolved upon).
        FORM THE SET (b) OF <literals from Clause A and Clause B>
        FORM THE SET (c) OF <sets of literals>
```

```
            where a pair of literals from SET (a) have been
            removed from SET (b) and the unifier composed with
            the remaining literals.
        RETURN these sets of literals.
```

The sets of literals so formed, are the literal pools mentioned above in section 4.1.

Factoring

The above algorithm represents binary resolution since only one literal from each clause is involved in the resolution. In fact binary resolution is incomplete for non-Horn clauses. The approach adopted was to "factor" the clauses before applying binary resolution. $(C \vee P)\phi$ is said to be a factor of the clause $C \vee P_1 \vee \ldots \vee P_m$ if ϕ is the most general unifier of $P_1, \ldots, P_m$. The input clauses were factored at the start of the refutation process whereas resolvants were factored just after they were produced. Factors were stored in the clause set as individual clauses in their own right.

This method of treating factoring caused problems when the goal clause in SL resolution could be factored, since this presented SemCheck with a choice of goal clauses. A patch solution was used in which the goal clause was not factored, but this did create some anomalies in the results between Clash and SL resolution when a factor of the goal clause could be used to find a solution very quickly.

Example 4.2 *Consider the following problem taken from TPTP [Suttner & Sutcliffe 94] and found originally in [Socher-Ambrosious 92].*

1. $\rightarrow p(X) \vee p(f(X))$.

2. $p(X) \wedge p(X) \rightarrow$.

In SL resolution the second clause will be the goal clause. Hence, it won't be factored by the program. Clash resolution, however, factors it into $p(X) \rightarrow$ and from there can find an empty resolvant in the second level of resolution, by resolving it against the first clause twice. The figure 4–1 shows the path to the solution for a SemCheck Clash refutation. Figure 4–2 shows the path taken by a SemCheck

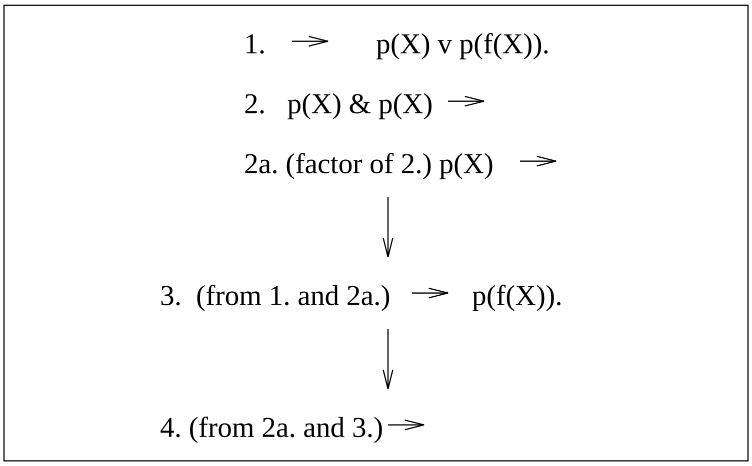

Figure 4–1: A Clash Refutation with Factoring

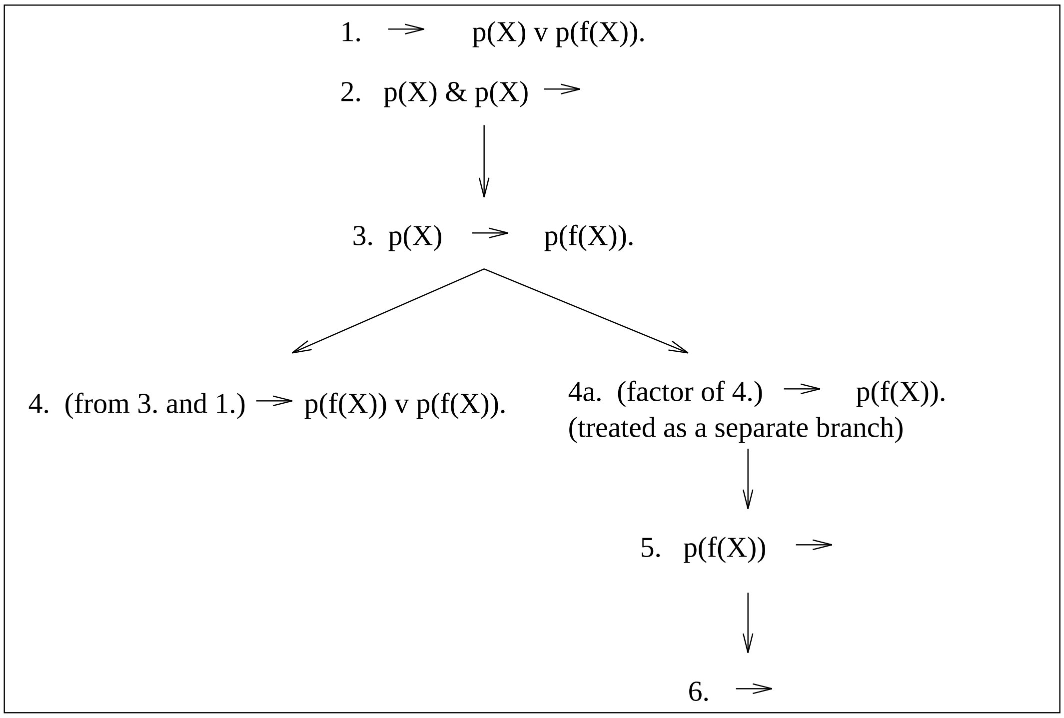

Figure 4–2: An SL Refutation with Factoring

SL refutation which can't factor clause 2. It should be born in mind that other branches are being generated for exploration as well, so the deeper search tree represents considerably more effort in terms of search. When tested on this problem SemCheck took 18760 milliseconds of cpu time to find an SL refutation and only 2570 milliseconds to find a Clash refutation, less than 1/6th of the time.

Clearly if more time were available it would be desirable to integrate factoring better into the resolution algorithm to avoid these problems. Possibly by factoring the goal clause and presenting the factors as separate starting branches.

4.3.2 Lush Resolution

The Lush resolution algorithm is fairly simple once the general algorithm is in place. To extend a branch:

```
EXTEND <Branch> <Goal Clause> <Models (which may be empty)>
        PICK a clause in the Branch
        RESOLVE Clause with the Goal Clause
        FACTOR the Resolvants
        SCREEN OUT the that are subsumed by other clauses appearing
        in the Branch.
        SCREEN OUT the resolvants that are true in the Models.
        RETURN one of the remaining Clauses.
```

All possible extensions are found by backtracking through the algorithm as part of the breadth first search.

4.3.3 Clash Resolution

Clash resolution is more complex. It builds up the clashes as it performs a linear search on the clauses, in a manner similar to Lush resolution except that it is strictly input. SemCheck only allows any one clause to be used once in a clash.

```
EXTEND <Clauses> <Models> <Numbering>
        PICK a Nucleus
        FORM CLASHES for the Nucleus.
        REPEAT for a New Nucleus.
        SCREEN all resulting resolvants for those that subsume each
        other.
        CHECK for empty clause.
```

Various strategies were tried for building up the set of clash resolvants of a nucleus. In the end depth first search of all the possible combinations of electrons was used.

Consider the clauses in figure 4–3. The figure represents a search tree for

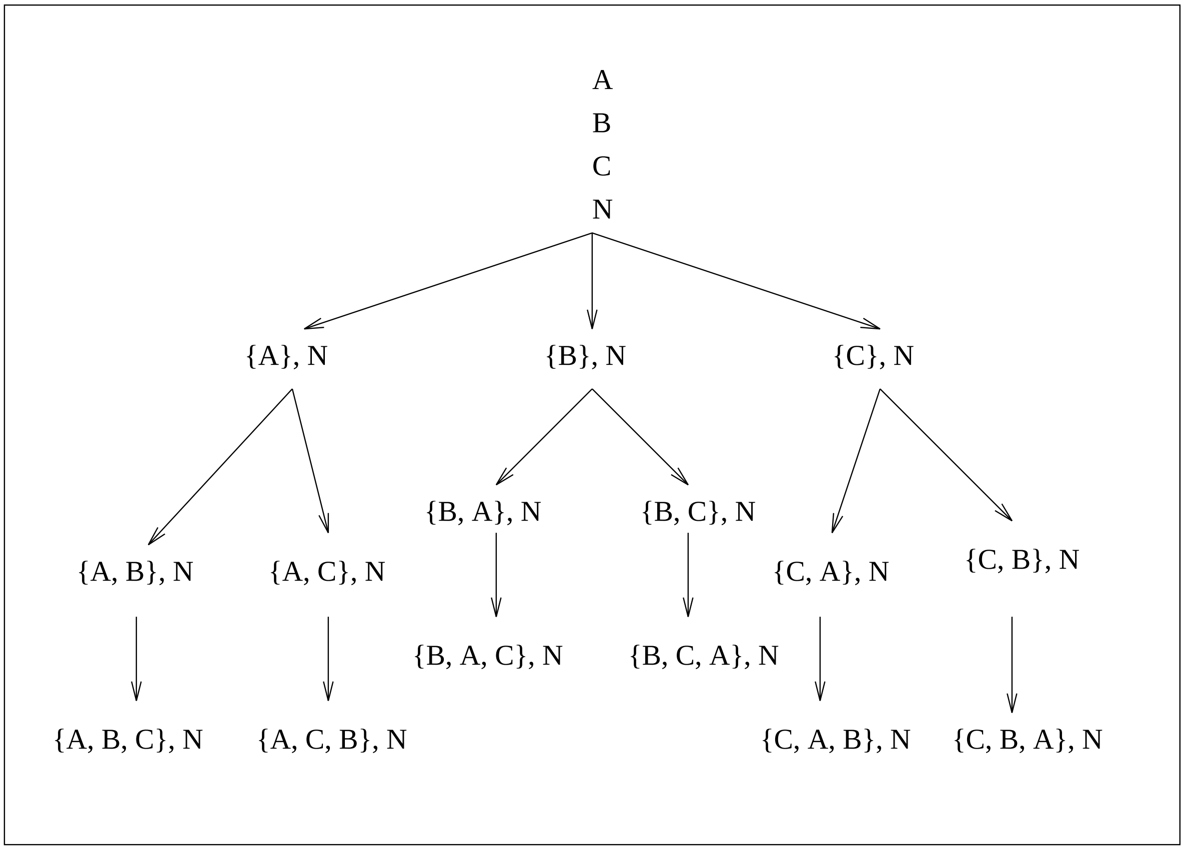

Figure 4–3: A Search Tree for Generating Clashes

building up clashes using N as the nucleus. Every node in the tree represents a potential clash resolvant of the four clauses. It is working under the assumption that an electron may only appear once in a given clash. This is a restricted form of Clash resolution but it helped simplify the search undertaken by SemCheck. Obviously some of the nodes are likely to be duplicates of others. However, it is not a trivial task to represent this in the search strategy since, for instance, it may be possible to generate a resolvant of B and N, and A and the resolvant of B and N, but not A and N. At the same time, it is also fairly clear that a "more intelligent" strategy could be devised than one which searches the whole space and then rejects duplicate resolvants. Since each node in the tree represents a potential intermediate clash resolvant similarities with Lush resolution become clear. The tree is, in fact, very like a Lush resolution search tree with the nucleus acting as the first goal clause. Hence the algorithm for forming clashes was very similar to the Lush resolution algorithm.

```
FORM CLASHES <Clauses> <Models> <Numbering>
                   <Electrons> <Nucleus> <Intermediate Resolvant>
        PICK a clause that is false in the Models AND that is not
        already an Electron, or the Nucleus.
        RESOLVE the clause and the Intermediate Resolvant
        FACTOR the resolvants so produced.
        SCREEN for subsumed Resolvants.
        IF a resolvant is false in the Models
                   THEN return it as a clash
        FORM CLASHES for all the resolvants.
        REPEAT for remaining Clauses.
```

It is clear from this that the Clash resolution algorithm has to do a significantly greater amount of work to produce each level.

The SemCheck algorithm differs from Chang and Lee's version of Semantic Clash resolution [Chang & Lee 73]. Chang and Lee allow an electron to appear more than once in a clash and SemCheck vetos this. Initially this was due to a misinterpretation of the definition. Some thought shows, however, that for many problems (including all those represented by Horn sets) the empty clash would appear in the first level of proof, simply because there is a refutation involving only input clauses. Although, this may appear to be no bad thing, the increase in search space at each level could well be prohibitive. Testing showed that at a certain level of complexity Clash resolution took a prohibitively long time to search the space whilst Semantic Clash resolution (with its restrictions on the choice of electron) found a solution very quickly. This variant on Clash resolution did not appear to affect completeness. At least it successfully proved all the theorems on which it was tested.

Possible improvements might be to implement some kind of maximality condition, i.e. something like Slagle's definition of maximality [Slagle 67], adapted to non-ground clauses. Slagle's version also insists that the electrons resolve away clauses in the nucleus, so it might not be so prohibitive, computationally to allow duplicates of electrons in the clash.

Alternatively, note that the algorithm above adds every possible clash resolvant to the set. Some of these resolvants will share the same nucleus and have electrons sets one of which is a subset of the other. It may be that Semantic Clash resolution is complete when only the resolvant with the larger electron set is returned as a clash. An implementation could act as an experimental test of this.

4.4 Comparability of the Algorithms for Lush and Clash Resolution

In trying to compare the two methods it is important that their implementations should be as similar as possible. This way implementational issues are less likely to bias the results one way or another.

SemCheck uses the same unification, model evaluation, substitution, factoring and set theory algorithms for both methods of resolution. The basic resolution algorithms are virtually the same, except that the clash version has an extra check to select an appropriate literal from the electrons.

The form of the programs surrounding the two methods are also similar. The main differences, therefore are those caused by the subsiduary search trees generated by clash resolution in order to form the next level of resolvants, and these are an inherent part of the method, although in the form used by SemCheck they may be overly inefficient.

4.5 Model Representation

To define a model the program used two predicates,

```
model_domain(Modelname, Element)
```

and

```
model_function(Modelname, [Symbolname,Relation/Function/Constant]).
```

`model_domain/2` defines the domain, D, of the model while `model_function/2` defines the assignment ϕ of predicate, function and constant symbols.

To evaluate a clause, C, in the model it was first translated into $\phi(C)$, a statement in the language of the model. All ground instances of $\phi(C)$ were created by substituting the elements of D in for free variables occurring in $\phi(C)$. The program then called the ground instance of each literal as a Prolog predicate. If the predicate succeeded for any positive literal or failed for any negative literal then the clause was considered true under that assignment of variables to domain elements. If the clause was true under all assignments then it was true in the model.

Example 4.3 *Consider the clause, C,*

```
cls(right_soln, product(x1, right_solution(x1, x2), x2), positive).
```

This is intended to be a statement about group theory. Say, we want to evaluate it in integer arithmetic modulo 2. First we define the domain, D of the model:

```
model_domain(mod2, 0).
model_domain(mod2, 1).
```

And then the assignment ϕ:

```
model_function(mod2, [product, plus_mod2]).
model_function(mod2, [right_solution, right_solution_mod2]).
```

the program then translates the literal

```
product(x1, right_solution(x1, x2), x2)
```

into

$$plus\_mod2(x1, right\_solution\_mod2(x1, x2), x2).$$

This is the only literal in C so the clause has been translated. There are four possible assignments of variables in D to this literal, namely,

$$(plus\_mod2(0, right\_solution\_mod2(0,0), 0)$$

$$(plus\_mod2(0, right\_solution\_mod2(0,1), 1)$$

$$(plus\_mod2(1, right\_solution\_mod2(1,0), 0)$$

$$(plus\_mod2(1, right\_solution\_mod2(1,1), 1)$$

these are then called by the program. Assuming that the predicates `plus_mod2` and `right_solution_mod2` have been defined consistently with integer arithmetic modulo 2 all four should succeed. Hence the clause is determined to be true in the model.

Obviously, for large domains and sentences containing large numbers of free variables, this evaluation process can be extremely time consuming. In order to reduce the time spent, the program was able to mark clauses as true or false in any given model. This meant that the clause need only be evaluated once within the model. This was particularly beneficial for the clash resolution algorithm which had to check for falsity when choosing electrons to resolve upon. It was also possible to mark the input clauses as true in this way, as part of the input to SemCheck, in order to save time.

4.6 The Output of the SemCheck Theorem Prover

As SemCheck works on a proof it writes out the resolvants that are added to the clause set as they are generated in the form `r(Number, Clause/Electrons, Clause)`. This represents the resolvant, `Number`, of the two Clauses/Electrons and the Nucleus. If the resolvant is empty it writes `empty(Clause/Electrons, Clause)` . `Clause` will either be the name of some previous resolvant or the name of a factor of an original clause written `f(Number, Clause)` where `Number` is the number of the factor as it was produced by the factoring algorithm and `Clause` is the original name given to the clause in the input. After it has completed searching for a refutation it writes `provable` or `unprovable` followed by the cpu time in milliseconds taken by the search

Example 4.4 *Consider the following example from TPTP [Suttner & Sutcliffe 94] which comes originally from [Pelletier 86] and [Kalish & Montegue 64]:*

clause_1 $F(x) \wedge G(x) \rightarrow .$

clause_2 $F(x) \wedge H(x) \rightarrow .$

clause_3 $\rightarrow I(a).$

clause_4 $\rightarrow F(a).$

clause_5 $\rightarrow H(x) \vee J(x).$

clause_6 $I(x) \wedge J(x) \rightarrow .$

Using clause_6 as the goal clause SemCheck produced the following output for lush resolution.

```
r(1,clause_6,f(1,clause_3))
r(1,clause_6,f(1,clause_5))
r(1,r(1,clause_6,f(1,clause_3)),f(1,clause_5))
r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_2))
r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_3))
r(1,r(1,r(1,clause_6,f(1,clause_3)),f(1,clause_5)),f(1,clause_2))
r(1,r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_2)),f(1,clause_3))
r(1,r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_2)),f(1,clause_4))
r(1,r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_3)),f(1,clause_2))
empty(r(1,r(1,r(1,clause_6,f(1,clause_3)),f(1,clause_5)),f(1,clause_2)),
                                                         f(1,clause_4))
empty(r(1,r(1,r(1,clause_6,f(1,clause_3)),f(1,clause_5)),f(1,clause_2)),
                                                         f(1,clause_4))
r(1,clause_6,f(1,clause_3))
r(1,clause_6,f(1,clause_5))
r(1,r(1,clause_6,f(1,clause_3)),f(1,clause_5))
r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_2))
r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_3))
r(1,r(1,r(1,clause_6,f(1,clause_3)),f(1,clause_5)),f(1,clause_2))
r(1,r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_2)),f(1,clause_3))
r(1,r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_2)),f(1,clause_4))
r(1,r(1,r(1,clause_6,f(1,clause_5)),f(1,clause_3)),f(1,clause_2))
empty(r(1,r(1,r(1,clause_6,f(1,clause_3)),f(1,clause_5)),f(1,clause_2)),
                                                         f(1,clause_4))
empty(r(1,r(1,r(1,clause_6,f(1,clause_3)),f(1,clause_5)),f(1,clause_2)),
```

```
                                                        f(1,clause_4))
provable
1570
```

and for clash resolution

```
r(1,[f(1,clause_4)],f(1,clause_1))
r(1,[f(1,clause_5)],f(1,clause_2))
r(1,[f(1,clause_6),f(1,clause_5)],f(1,clause_2))
empty([f(1,clause_3),f(1,clause_4),f(1,clause_6),f(1,clause_5)],
                                                        f(1,clause_2))
r(1,[f(1,clause_6)],f(1,clause_3))
r(1,[f(1,clause_5),f(1,clause_6)],f(1,clause_3))
empty([f(1,clause_4),f(1,clause_2),f(1,clause_5),f(1,clause_6)],
                                                        f(1,clause_3))
r(1,[f(1,clause_6)],f(1,clause_5))
r(1,[f(1,clause_3),f(1,clause_6)],f(1,clause_5))
empty([f(1,clause_4),f(1,clause_2),f(1,clause_3),f(1,clause_6)],
                                                        f(1,clause_5))
r(1,[f(1,clause_5)],f(1,clause_6))
r(1,[f(1,clause_3),f(1,clause_5)],f(1,clause_6))
empty([f(1,clause_4),f(1,clause_2),f(1,clause_3),f(1,clause_5)],
                                                        f(1,clause_6))
r(1,[f(1,clause_4)],f(1,clause_1))
r(1,[f(1,clause_5)],f(1,clause_2))
r(1,[f(1,clause_6),f(1,clause_5)],f(1,clause_2))
empty([f(1,clause_3),f(1,clause_4),f(1,clause_6),f(1,clause_5)],
                                                        f(1,clause_2))
r(1,[f(1,clause_6)],f(1,clause_3))
r(1,[f(1,clause_5),f(1,clause_6)],f(1,clause_3))
empty([f(1,clause_4),f(1,clause_2),f(1,clause_5),f(1,clause_6)],
                                                        f(1,clause_3))
r(1,[f(1,clause_6)],f(1,clause_5))
r(1,[f(1,clause_3),f(1,clause_6)],f(1,clause_5))
empty([f(1,clause_4),f(1,clause_2),f(1,clause_3),f(1,clause_6)],
                                                        f(1,clause_5))
r(1,[f(1,clause_5)],f(1,clause_6))
r(1,[f(1,clause_3),f(1,clause_5)],f(1,clause_6))
empty([f(1,clause_4),f(1,clause_2),f(1,clause_3),f(1,clause_5)],
                                                        f(1,clause_6))
provable
3830
```

4.7 Summary

This chapter gave an overview of the SemCheck program that was written for comparing Clash, Lush, Semantic Clash and Semantic Lush resolution. It discussed the representation of clauses and models. It gave pseudo-code versions of the important algorithms and discussed their comparability. It also presented a sample of the program's output.

Chapter 5

Test Procedure

5.1 Test Problems — TPTP

TPTP or 'Thousands of Problems for Theorem Provers' is a library of problems for automated theorem provers available via the internet [Suttner & Sutcliffe 94]. The aim of the TPTP project is to provide an independent, well-documented and wide-ranging set of problems on which theorem provers may be tested. In testing the performance of a theorem prover there is always a worry that the test problems chosen will, intentionally or unintentionally, be those on which it will perform well. The existence of such a library helps avoid such difficulties and so aid the comparability of provers. Hence problems from the TPTP library were used in testing the performance of Semantic resolution.

5.1.1 Choosing Problems for Testing

There was not sufficient time available to test SemCheck on all the problems in TPTP. Furthermore, the program was fairly inefficient and would not have reached a solution for most of the harder problems within a reasonable space of time. It was therefore desirable to find some way of picking easy problems from the database. Although Suttner and Sutcliffe intend to give the problems in the TPTP library a difficulty rating at some point in the future based on a generalised idea of how "hard" theorem provers find the problem, none exist at present. As a result a crude measure was used based on the number of clauses in the problem. The problems were tried out by SemCheck in order of ascending "size". This had the added advantage that in the initial stages of testing and debugging, it was easier to follow what the SemCheck was doing and hence to obtain some degree of confidence in its correctness.

5.1.2 Adapting TPTP Problems for use in SemCheck

Version TPTP-v1.1.1 was used in testing. It was fairly simple to convert the clause representation from TPTP to one usable by the program. The ordering of the literals and clauses were left unchanged for clash resolution. Lush resolution requires the specification of a "goal" clause. In particular, the goal clause should be the negation of the theorem to be proved. Hence SemCheck was given extra information about the clause set and (occasionally) a different clause ordering from that used in TPTP. Conceivably SemCheck could be adapted to pick a goal clause for itself by, for instance, checking for clauses with no positive literals.

5.1.3 Creating Models

For many of the problems, model domains suggested themselves readily from the area from which the problem was taken. Most of the clauses contained a mixture of functions and predicates. Prolog, however, deals almost exclusively with predicates, so it was frequently necessary to write a `val/2` predicate which succeeded when its second argument was instantiated to the value of the function in the first argument.

Example 5.1 *Take the following sentence in group theory.*

$$\forall x(product(inverse(x), x, identity)$$

$product(X, Y, Z)$ is a predicate, but $inverse(X)$ is a function and cannot be represented as such in Prolog. The following is the representation for the group formed by arithmetic modulo 2, that was provided for use with the above sentence.

```
model_domain(1, 0).
model_domain(1, 1).

model_function(1, [product, plus1]).
```

```
model_function(1, [inverse, inv]).
model_function(1, [identity, 0]).
```

These describe the domain of the model and provide a translation for the predicates, functions and constants. It is now necessary to provide functions to evaluate the sentence in the model once it has been translated.

```
plus1(X, Y, Z):-
        val(X, N1),
        val(Y, N2),
        val(Z, N3),
        N4 is N1 + N2,
        N3 is N4 mod 2.

val(X, X):-
        integer(X).
val(inv(X), Z):-
        val(X, N1),
N2 is 2 - N1,
Z is N2 mod 2.
```

Here `val/2` *is being used to evaluate the function* $inverse(X)$*.*

Partial Models

Sometimes it was possible to only partially specify the domain of the model. This was useful when the most intuitive model of the domain was infinite. Care had to be taken when doing this, however, and it was only used in a small number of cases.

Example 5.2 *The following is an example where a partial model was used.*

The problem was to prove that if a is prime and $a = b^2/c^2$ then $a|b$. The clause set was

1. $\rightarrow product(X, X, square(X))$

2. $product(X, Y, Z) \rightarrow product(Y, X, Z)$

3. $product(X, Y, Z) \rightarrow divides(X, Z)$

4. $prime(X) \wedge product(Y, Z, W) \wedge divides(X, W) \rightarrow$
   $divides(X, Z) \vee divides(X, W)$

5. $\rightarrow prime(a)$

6. $\rightarrow product(a, square(c), square(b))$

7. $divides(a, b) \rightarrow$

The existence of the function $square(X)$ immediately suggest an infinite domain (although using some kind of modular arithmetic is also a possibility). In this case however a partial model domain was specified.

```
model_domain(model, a).
model_domain(model, b).
model_domain(model, c).
model_domain(model, square(a)).
model_domain(model, square(b)).
model_domain(model, square(c)).
```

This domain was sufficient for the test since any clause generated by the proof that was true in the model was true under all assignments of variables from the domain and similarly, any clause that was false in the model was false under some assignment of variables from the domain.

The main problem with using partial models as a method of dealing with infinite domains is a risk that a clause could be wrongly classified as true in the model and completeness lost through the vetoing of that branch. To be sure that such an effect would not occur, it is really necessary to know, in advance, the clauses appearing in the refutation which, in the long term, renders the whole process useless. It may be that more sophisticated methods of describing models could be used (since humans can evaluate truth and falsity in infinite models by a process of proofs and counterexamples), or alternatively that it might be possible to develop some theory about what should be contained in a partial domain. There was not sufficient time to explore these issues as part of this project.

5.2 Comparing Results

There are two obvious methods of comparing the performance of theorem provers. These are comparison of cpu time taken by the theorem prover and the space searched by it. On the whole, space searched is preferred since it isn't dependent upon the implementation of the prover. Unfortunately, in the case of Semantic resolution, the time taken is an issue, since a model may prune away a vast proportion of the search space, but the cost of evaluating statements in that model may be prohibitive.

Even comparison of Lush and Clash resolution was complicated by the presence of the additional search for intermediate resolvants in Clash resolution. Many of these intermediate resolvants were not added into the search space and so were not produced as output for the program. Essentially this meant that a good deal more "hidden" work was being done by Clash resolution for each resolvant generated. This would tend to suggest that cpu time should be used as a yardstick for this comparison. However, as is discussed in chapter 4, the algorithm for generating the search was very inefficient and could be seen as slowing down the Clash refutation procedure needlessly.

In the event both methods were used.

5.3 Summary

This chapter discussed the issues involved in actually using the SemCheck theorem prover. In particular providing problems and models on which it should be tested. It also discussed some of the problems involved in trying to compare the methods under consideration.

Chapter 6

Results and Conclusions

Figures 6–1, 6–2, 6–3, 6–4 are graphs of the comparisons of cpu time and space searched by the various methods. They have been divided into Horn clauses and non-Horn clauses. The x-axis is simply an ordering of the problems used in testing. They have been placed in order of increasing cpu time for Lush resolution, and tables have been provided showing which problems correspond to which numbers on the x-axes of the graphs.

6.1 Horn Sets

6.1.1 Cpu Time

In fact, the log of the cpu times has been taken, simply to allow all the times to appear together on the same graph. Where Clash resolution failed to produce a refutation within 24 hours the line has been broken and "FAILED" written. The order of the problems is set out in table 6–1. The graph suggests several interesting points. Firstly there seems to be very little difference between the performances of Lush and Semantic Lush resolution, at least initially, although there is a suggestion towards the end that Semantic Lush may be beginning to gain some advantage for the harder problems.

Secondly, and surprisingly Semantic Clash resolution appears, on the whole, to be performing less well than Clash resolution (though, it should be noted that when it is better than clash resolution it is much better, particularly in the case

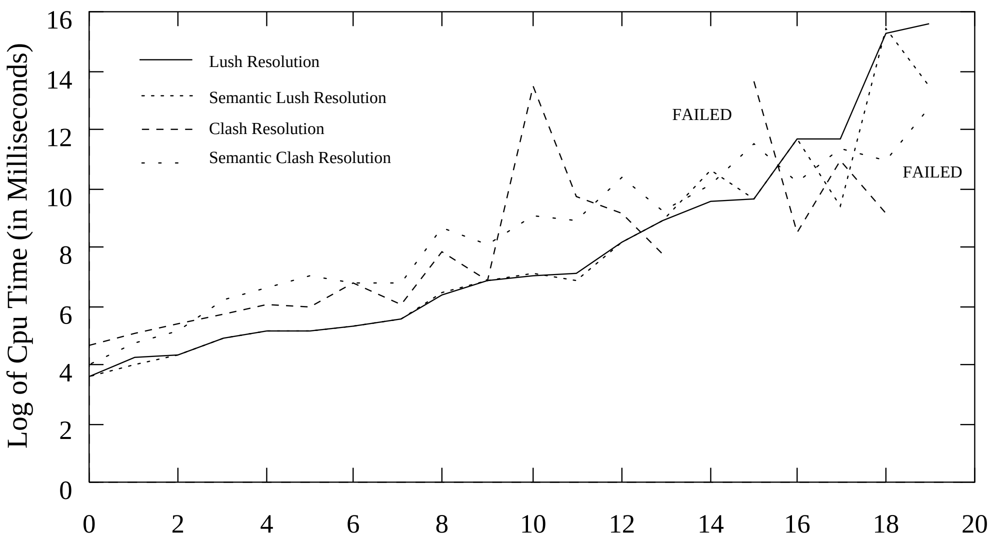

Figure 6–1: Log of Cpu Times taken to find Refutation by Lush, Semantic Lush, Clash and Semantic Clash resolution for Horn Sets

of the last problem where Clash resolution failed to find a solution at all). This appears to be because of the restrictions Semantic Clash resolution places upon the electrons (see example 6.1).

Example 6.1 *Consider the clause set*

topology_generated_40

$$\rightarrow element\_of\_collection(xx, top\_of\_basis(x))$$

$$\vee element\_of\_set(f11(x, xx), xx)$$

set_theory_6 $element\_of\_set(x, empty\_set) \rightarrow$

lemma_1b_2 $element\_of\_collection(empty\_set, top\_of\_basis(f)) \rightarrow$

Using Clash resolution SemCheck produced the following output in which empty clashes were found in the first level of the refutation.

```
empty([f(1,set_theory_6),f(1,lemma_1b_2)],
                                  f(1,topology_generated_40))
empty([f(1,lemma_1b_2),f(1,topology_generated_40)],
```

No.	*Syntactic Name*	*Semantic Name*	*Description*
1	SYN001-1.001	Allways	All Signed combinations of some propositions
2	SYN048-1	FPell18	Pelletier Problem 18
3	SYN064-1	FPell35	Pelletier Problem 35
4	SYN046-1	FPell15	Pelletier Problem 15
5	SYN049-1	FPell19	Pelletier Problem 19
6	MSC007-1.002	Pigeon	Cook pigeon-hole problem
7	SYN040-1	FPell01	Pelletier Problem 1
8	SYN005-1.002	Orl	Disjunctions that form a contradiction
9	SYN050-1	FPell20	Pelletier Problem 20
10	SYN005-1.003	Orl	Disjunctions that form a contradiction
11	COL007-1	WkFxdPtL	Weak Fixed Point for L
12	GEO079-1	AltAngle Eq	The alternate interior angles in a trapezoid are equal
13	SYN079-1	FPell57	Pelletier Problem 57
14	SYN005-1.004	Orl	Disjunctions that form a contradiction
15	COL055-1	MBird06	Happy Birds
16	SYN010-1.002	Letz	Letz Problems
17	SYN005-1.005	Orl	Disjunctions that form a contradiction
18	SYN004-1.002	Implies2	Implications that form a contradiction
19	SYN005-1.006	Orl	Disjunctions that form a contradiction
20	GRP028-1	SemiGRIdEx1	In semigroups, left and right solutions $\Rightarrow$ right id exists

Table 6–1: The Order in which Horn problems appear on the x-axes of figures 6-1 and 6-2

```
                                                    f(1,set_theory_6))
empty([f(1,set_theory_6),f(1,lemma_1b_2)],
                                           f(1,topology_generated_40))
empty([f(1,lemma_1b_2),f(1,topology_generated_40)],
                                                    f(1,set_theory_6))
provable
```

```
550
```

However, when SemCheck was using Semantic Clash resolution the empty clash was delayed to the second level, since, only one of the input clauses was true in the Model-like interpretation being used.

```
r(1,[f(1,lemma_1b_2)],f(1,topology_generated_40))
empty([r(1,[f(1,lemma_1b_2)],f(1,topology_generated_40))],
                                                      f(1,set_theory_6))
r(1,[f(1,lemma_1b_2)],f(1,topology_generated_40))
empty([r(1,[f(1,lemma_1b_2)],f(1,topology_generated_40))],
                                                      f(1,set_theory_6))
provable
750
```

Thirdly, there is a suggestion[1] that there comes a point where Semantic Clash resolution begins to perform faster than Lush resolution, and maybe even Semantic Lush resolution.

6.1.2 Search Space

The following (figure 6–2) is a graph of the search space generated by the various problems. Once again the problems are placed along the x-axis in the same order as before (i.e. in increasing cpu time for Lush resolution) and the log of the search space has been taken in order to fit all the results on one graph.

Of the four methods, Semantic Clash resolution appears on the whole to generate the fewest resolvants.

---

[1]There is simply not enough data available to make it anything more than that, since such problems were taking SemCheck several hours to solve

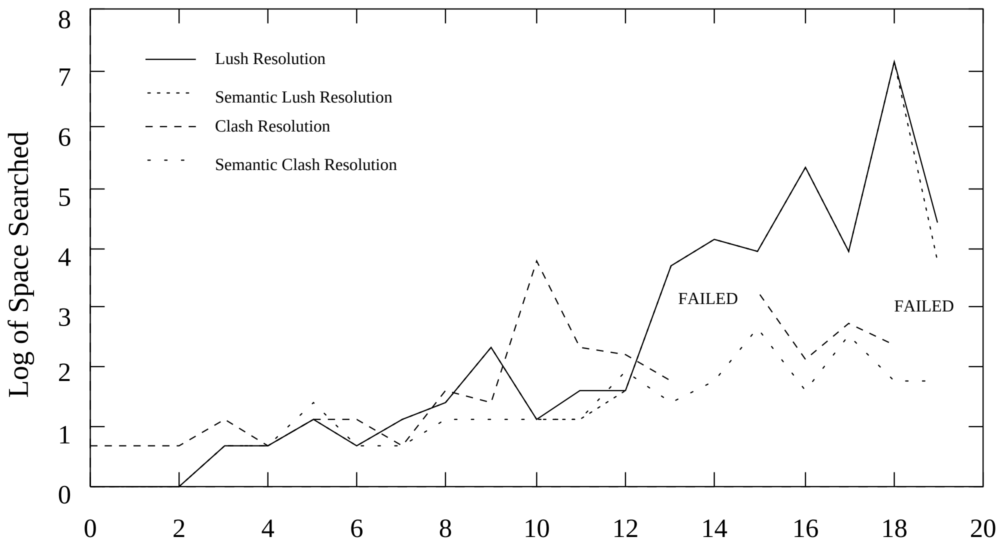

Figure 6–2: Log of Space Searched in Finding a Refutation by Lush, Semantic Lush, Clash and Semantic Clash resolution for Horn Sets

Lush and Semantic Lush resolution have identical search space except in two cases, GEO079-1 and GRP028-1[2]. The Output for GEO079-1 can be found in appendix B. The apparent ineffectiveness of Semantic Lush resolution is a startling result, and needs careful consideration.

Models have no Power of Veto for Ground Problems

Consider a minimally unsatisfiable ground horn set, S. It has already been established in chapter 3 that there is only one Model-like Interpretation, I, of S, and that all the atoms of S, are true in I. The resolvants in a Lush refutation of a Horn set contain no positive literals, i.e. they consist of a disjunction of negated atoms, hence they are all false in I. This means they can't be vetoed by I, so the presence of I has no effect on the proof.

[2]neither of which can really be seen on the graph. Lush resolution had a search space of 5 and Semantic Lush resolution a search space of 4 for GEO079-1. With GRP028-1 the search space for Lush resolution was 85 clauses and for Semantic Lush resolution it was 44

This is a start toward explaining the phenomenon, but it is not satisfactory since relatively few of the problems were ground.

All Paths Lead to a Solution

One possibility is that the relative simplicity of the problems chosen biased the results. Semantic Lush resolution cannot veto a branch that can lead to the empty clause. For simple problem sets it may well be that the only branches that can be formed are those leading to contradictions.

Are these Solutions Supported by the Experimental data?

SemCheck was tested on 20 Horn sets. Semantic Lush resolution failed to reduce the Search Space in 18 cases. Of these, in 6 cases the problems were ground and in 12 all the paths lead to solutions.

This suggests that the explanations for the poor performance of Semantic Lush resolution offered above are correct. It seems likely that these problems are only going to afflict relatively simple problems so there is hope that more extensive testing, on harder problems might well show some improvement in the performance.

6.2 Non-Horn Sets

6.2.1 Cpu Time

Once again the logs of the cpu times are used in figure 6–3, and the problems ordered as in table 6–2 (in order of increasing cpu time taken by SL resolution).

Most strikingly Semantic Clash resolution is here performing marginally better than Clash resolution. The reasons why, exactly, are unclear since Semantic Clash resolution is still delaying the formation of clashes. More on this in section 6.2.2.

Again there comes a point where Semantic Clash resolution appears to start performing better than SL resolution. Though it is early to say that this is will continue to be the case.

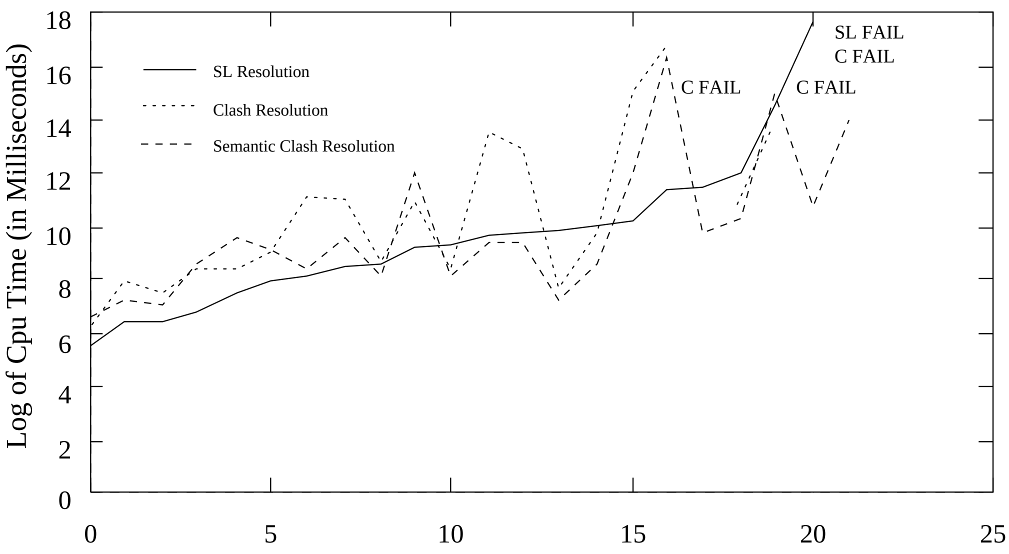

Figure 6–3: Log of Cpu Times taken to find Refutation by Sl, Clash and Semantic Clash resolution for Non-Horn Sets

6.2.2 Search Space

Once again the log of the search space has been taken figure 6–4. Problems are ordered by increasing cpu time for SL resolution. This time, where Clash resolution has failed "C FAIL" is written, where SL resolution has failed "SL FAIL" is written. Comparing figure 6–4 with figure 6–2 shows that there is, on the whole, a greater difference between the space searched by Clash and Semantic Clash resolution, suggesting that the delaying effect may now be less serious.

Various possible reasons suggest themselves for this reduction in search space. For a start the problems afflicting Horn sets are not relevant to non-Horn sets. It is no longer necessarily the case that all the positive literals in a ground clause set will be true in a Model-like interpretation. Nor will resolvants consist only of negative literals, hence interpretations may well be able to veto resolvants even when applied to ground sets. Secondly, in non-Horn sets resolvants in branches leading to a solution may be true in a model, hence those branches may be vetoed by the model. The reason that Semantic Clash resolution can still preserve completeness is that the clashes can "step over" such problematic cases. However it seems likely that Semantic Clash resolution won't return quite so many clashes to the clause

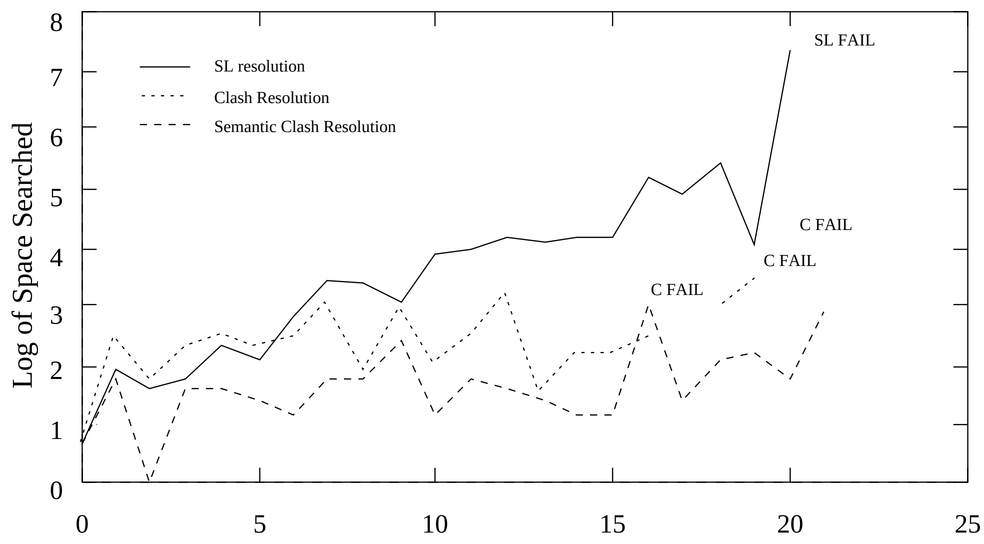

Figure 6–4: Log of Space searched by SL, Clash and Semantic Clash resolution while looking for a refutation for Non-Horn Sets

set as Clash resolution does, even in cases where all the possible branches lead to a solution.

Another possibility is that a non-Horn set naturally has a larger search space at each level since non-Horn clauses tend to have more possible resolvants. So Clash resolution spends a long time exploring each level fully, where Semantic Clash resolution has less space to search because of the restrictions on which clauses may be used as electrons.

It is fairly plain from both comparisons of search space that Lush or SL resolution almost invariably produces more resolvants than Clash resolution. However it should be remembered that much of the work done by Lush resolution is also done by clash resolution as part of the "hidden" search for clashes. In several cases where either Lush or SL resolution displayed a large search space, Clash resolution failed to find a refutation at all within a reasonable amount of time (24 hours).

No.	*Syntactic Name*	*Semantic Name*	*Description*
1	TOP0022-2	BasisTplgLem2	Topology generated by a basis forms a topological space, part 2
2	SYN045-1	FPell13	Pelletier Problem 12
3	SET043-5	Russell	Russell's Paradox
4	SYN008-1	Relev1	A problem to demonstrate the usefulness of relevancy testing
5	SYN061-1	FPell31	Pelletier Problem 31
6	SYN066-1	FPell37	Pelletier Problem 37
7	SYN001-1.002	Allways	All signed combinations of some propositions
8	SYN047-1	FPell17	Pelletier Problem 17
9	SYN081	FPell59	Pelletier Problem 59
10	SYN006-1	Splits	A problem to demonstrate controlling splits
11	SYN051-1	FPell21	Pelletier Problem 21
12	SYN029-1	EW2	EW2
13	SYN028-1	EW1	EW1
14	SYN002-1.000	Even	Even Problems
15	SYN052-1	FPell22	Pelletier Problem 22
16	SYN053-1	FPell23	Pelletier Problem 23
17	SYN002-1.002	Even	Even Problems
18	SET044-5	ARussellSet	Anti-Russell Sets
19	SYN031-1	MQW	MQW
20	NUM014-1	VallsPrm	If a is a prime and $a = b^2/c^2$ then a divides b
21	SYN044-1	FPell10	Pelletier Problem 10
22	SYN054-1	FPell24	Pelletier Problem 24

Table 6–2: The Order in which non-Horn problems appear on the x-axes of figures 6-3 and 6-4

6.3 Disjunctions that form a Contradiction

An interesting class of problems to look at are those called 'Disjunctions that form a Contradiction' by the TPTP database and coming originally from [Plaisted 92]. These are Horn sets, but all possible Lush resolution branches can lead to a contradiction so Semantic Lush resolution was ineffective and its results are not considered here. The syntactic names of the ones tested were SYN005-1.002, SYN005-1.003, ..., SYN005-1.006.

In general SYN005-1.00N is as follows

disjunction $p1(X1, X2) \wedge p2(X2, X3) \wedge \ldots \wedge pN(XN, X1) \rightarrow$

p1 $\rightarrow p1(a, a)$

p2 $\rightarrow p2(a, a)$

...

pN $\rightarrow pN(a, a)$.

Plaisted writes:

> On [these examples] locking resolution (even with a bad choice of indices) and SL-resolution generate search spaces of size polynomial in n, but positive unit resolution, all-negative resolution, set-of-support strategy, ancestry-filter form, and input resolution all generate search spaces of size exponential in n.[Plaisted 92]

This is born out by the comparisons of cpu time and search space in figures 6–5 and 6–6. Looking at the output for SYN005-3 it becomes clearer what is happening.

```
lush(test,[],disjunction)
r(1,disjunction,f(1,p1))
r(1,disjunction,f(1,p2))
r(1,disjunction,f(1,p3))
r(1,r(1,disjunction,f(1,p1)),f(1,p2))
r(1,r(1,disjunction,f(1,p1)),f(1,p3))
```

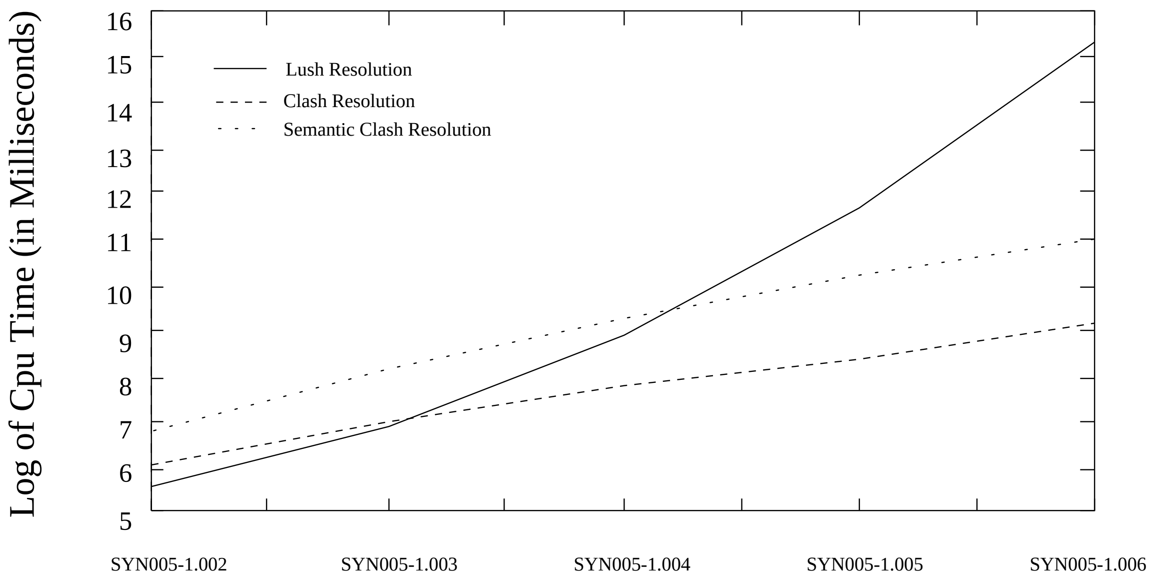

Figure 6–5: Log of Cpu Times taken by Lush, Clash an Semantic Clash resolution to find a refutation for the SYN005 problems

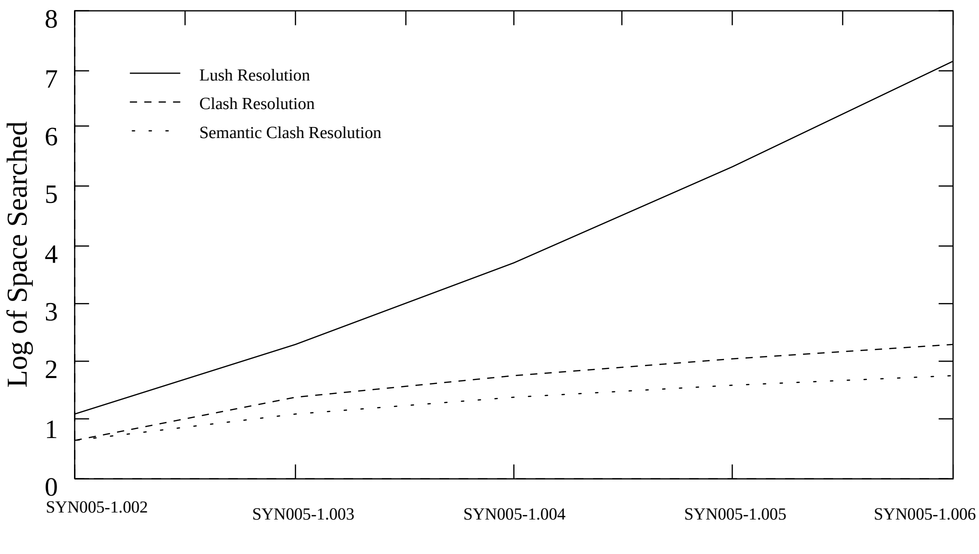

Figure 6–6: Log of Space Searched by Lush, Clash and Semantic Clash resolution while looking for a refutation for the SYN005 problems

```
r(1,r(1,disjunction,f(1,p2)),f(1,p1))
r(1,r(1,disjunction,f(1,p2)),f(1,p3))
r(1,r(1,disjunction,f(1,p3)),f(1,p1))
r(1,r(1,disjunction,f(1,p3)),f(1,p2))
empty(r(1,r(1,disjunction,f(1,p1)),f(1,p2)),f(1,p3))
empty(r(1,r(1,disjunction,f(1,p1)),f(1,p2)),f(1,p3))
r(1,disjunction,f(1,p1))
r(1,disjunction,f(1,p2))
r(1,disjunction,f(1,p3))
r(1,r(1,disjunction,f(1,p1)),f(1,p2))
r(1,r(1,disjunction,f(1,p1)),f(1,p3))
r(1,r(1,disjunction,f(1,p2)),f(1,p1))
r(1,r(1,disjunction,f(1,p2)),f(1,p3))
r(1,r(1,disjunction,f(1,p3)),f(1,p1))
r(1,r(1,disjunction,f(1,p3)),f(1,p2))
empty(r(1,r(1,disjunction,f(1,p1)),f(1,p2)),f(1,p3))
empty(r(1,r(1,disjunction,f(1,p1)),f(1,p2)),f(1,p3))
provable
1040
```

In the Clash resolution output below the predicate ordering makes p3 the "largest" predicate letter and p1, the "smallest".

```
clash(test,[],numb)
r(1,[f(1,p3)],f(1,disjunction))
empty([f(1,p1),f(1,p2),f(1,p3)],f(1,disjunction))
r(1,[f(1,disjunction)],f(1,p3))
empty([f(1,p1),f(1,p2),f(1,disjunction)],f(1,p3))
r(1,[f(1,p3)],f(1,disjunction))
empty([f(1,p1),f(1,p2),f(1,p3)],f(1,disjunction))
r(1,[f(1,disjunction)],f(1,p3))
empty([f(1,p1),f(1,p2),f(1,disjunction)],f(1,p3))
provable
1050
```

The obvious domain for an interpretation of these sentences is the singleton $\{a\}$. In a Model-like interpretation $pi(a, a)$ will be true for all $2 < i \leq N$. This intpretation was used by the Semantic Clash refutation below.

```
clash(test,[model],numb)
r(1,[f(1,disjunction)],f(1,p3))
r(1,[r(1,[f(1,disjunction)],f(1,p3))],f(1,p2))
empty([r(1,[r(1,[f(1,disjunction)],f(1,p3))],f(1,p2))],f(1,p1))
r(1,[f(1,disjunction)],f(1,p3))
r(1,[r(1,[f(1,disjunction)],f(1,p3))],f(1,p2))
empty([r(1,[r(1,[f(1,disjunction)],f(1,p3))],f(1,p2))],f(1,p1))
provable
3420
```

Lush resolution is forced to build up every possible combination of the three positive unit clauses. The selected literal part of Clash resolution however, cuts down on much of this search space by ordering the consideration of the positive units.

Perhaps more interestingly, although Semantic Clash resolution is not as effective as Clash resolution in this case, the search space it has generated is, in fact, one branch of the Lush resolution search tree. With a little thought it is clear why this is happening. Initially only the goal clause is false in the model, hence only the goal clause may appear as an electron in any clash. Thereafter it is the goal clause and Clash resolvants that are false, therefore either the goal clause or a Clash resolvant must appear in any clash.

6.4 Semantic Clash Resolution

In the more complex problems where Clash resolution was failing to find a refutation in a reasonable time, Semantic Clash resolution was outperforming Lush resolution for similar reasons as that discussed in section 6.3. The Semantic component restricting the choice of electrons, which appeared to be handicap for Horn sets, was now beneficial, preventing the algorithm searching through and producing large numbers of clashes in a time consuming fashion and hence reducing the amount of "hidden" work done by the algorithm. At the same time, the selected literal component was reducing the number of resolvants produced, in comparison to a Lush refutation proof, reducing the search space.

A striking example of this was problem SET044-5 in TPTP, originating in [Pelletier 86]. The statement of the theorem is that if an anti-Russell set exists[3] then every set has a complement. The clause set had four clauses.

clause_1 $element(X, a) \rightarrow element(X, X)$

clause_2 $element(X, X) \rightarrow element(X, a)$

clause_3 $element(X, f(Y)) \wedge element(X, Y) \rightarrow$

clause_4 $\rightarrow element(X, Y) \vee element(X, f(Y))$

The predicates were ordered giving literals containing f precedence over literals containing $element$. The following model was used.

```
model_domain(model, a).
model_domain(model, b).

model_function(model, [X, X]).

e(a, b).
e(b, a).
e(b, b).
element(X, Y):-
        val(Y, Z),
        val(X, N),
        e(N, Z).

val(a, a).
val(b, b).
```

[3] A Set that contains exactly those sets that contain themselves

```
val(f(a), b).
val(f(b), a).
```

The Lush resolution search for a refutation contained 136 resolvants whereas Semantic Clash resolution produced the following search

```
r(1,[f(1,clause_3)],f(1,clause_1))
r(2,[f(1,clause_3)],f(1,clause_1))
r(1,[r(1,[f(1,clause_3)],f(1,clause_1))],f(1,clause_2))
empty([r(1,[r(1,[f(1,clause_3)],f(1,clause_1))],f(1,clause_2)),
                r(1,[f(1,clause_3)],f(1,clause_1))],f(1,clause_4))
```

Full details of the clauses, models and program output for this problem can be found in appendix B.

6.5 Multiple Interpretation Semantic Resolution

Considering the generally disappointing performance of Semantic Lush resolution there seemed little reason to suppose that using more than one model would have any effect on the performance, but it was reasonable to look for some effects in Semantic Clash resolution. However, as it turned out there were remarkably few occasions when it was even relevant to consider the use of more than one model.

Since, as was shown in chapter 3, there is only one Model-like interpretation of a ground Horn set there was not much point in considering using more than one model in these cases.

In section 6.3 it was pointed out that there was only one Model-like interpretation for that problem which had a one element domain. It would have been pointless considering larger domains in that problem since they would have had no greater powers of veto and it would have taken longer to evaluate truth or falsity in the domain. Once again talk of multiple models was redundant.

In the final event there were only about 3 cases where it made any sense to talk of multiple models at all, COL007-1, GEO079-1 and GRP028-1. Mulitple models were investigated for COL007-1 and GEO079-1. In neither of these was any advantage gained by the consideration of more than one model. However, with only two[4] problems under consideration it is impossible to draw any conclusions.

6.6 Concluding Remarks

Four resolution refinements have been examined. Two were semantics-free methods, Lush and Clash resolution. The other two were their semantic equivalents. All four methods were tested on a selection of simple problems.

From the data available it appears that there may be very little benefit to be gained, in terms of time, in using models with Horn sets, although Semantic Clash resolution did show a smaller search space than Clash resolution. However, it seems likely that this was a result of simplicity of the problems used. There certainly appears to be no point in applying Semantic Lush resolution to ground Horn Sets since it was shown that this can not perform any better than Lush resolution.

The results for non-Horn sets were more positive and implied that for some complex problems Semantic Clash resolution was the most efficient of the methods examined.

6.7 Further Work

There is clearly a great deal more work do be, simply by way of testing SemCheck out on more problems so that conclusions may be drawn more confidently from the data.

Before this was undertaken it would also be advisable to examine the Clash resolution algorithm and see if this could be improved, several suggestions for doing this are put forward in chapter 4.

[4]Mulitple model weren't investigated for GRP028-1 because of time constraints

There are other improvements,as well, that could be made to SemCheck, e.g in the treatment of factoring for Lush resolution as was discussed in section 4.3.1.

It would also be worth implementing Slagle's Maximal Clash resolution. It is unclear whether or not this is complete for Non-Horn sets, but substantial testing should either produce a counter-example or generate a degree of confidence in the method from experimental testing. There may well be other variants of Clash resolution which restrict the choice of electrons and clashes formed in other ways and it might well be worth investigating and testing these, especially since Clash resolution had such trouble with some of the test problems.

In the light of the performance of Semantic resolution for Horn sets, it is particularly important to examine a larger problem set. The difficulties encountered, such as all branches leading to a solution may prove to be relatively rare once more complex problems are examined, but it is also possible that they affect most Horn sets.

There is room for more work in terms of model definition. In particular methods for dealing with infinite domains. It may be that partial models are a solution. If so, methods for defining them need to be developed that don't rely on knowledge of the resolvants appearing in the proof tree.

6.8 Summary

This chapter examined the results of the SemCheck program used on a relatively small set of simple problems. It compared the performances of Lush, Semantic Lush, Clash and Semantic Clash resolution both in terms of cpu time and search space. It also looked individually at a couple of interesting results.

It came to the surprising conclusion that Semantic resolution seems to be rather ineffective when applied to Horn sets but was unable to say whether this was an implicit problem or a side effect of the simplicity of the problems examined.

On the other hand, in the case of non-Horn sets, there were signs that Semantic Clash resolution was the most effective of the methods tested.

Bibliography

[Boyer 71] R. Boyer. *Locking: A Restriction of Resolution.* Unpublished PhD thesis, University of Texas at Austin, 1971.

[Brown 73] F.M. Brown. The use of several models as a refinement of resolution with sets of Horn clauses. Memo no. 63, Dept. of Computational Logic, University of Edinburgh, 1973.

[Bundy 83] A. Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, Inc., 1983.

[Chang & Lee 73] C. L. Chang and R. T. C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* New York: Academic, 1973.

[Gelernter 63] H. Gelernter. Empirical explorations of the geometry theorem-proving machine, and Realization of a geometry theorem-proving machine. In *Computers and Thought*, pages 153 – 63,134 – 52. McGraw Hill, 1963.

[Gilmore 70] P.C. Gilmore. An examination of the geometry theorem machine. *Artificial Intelligence*, 1:171–87, 1970.

[Henschen & Wos 74] L. Henschen and L. Wos. Unit refutations and Horn sets. *J. ACM*, 21:590–605, October 1974.

[Henschen 76] L. J. Henschen. Semantic resolution for Horn sets. *IEEE Transactions on Computing*, 25(8), August 1976.

[Kalish & Montegue 64] D. Kalish and R. Montegue. *Logic: Techniques of Formal Reasoning*. World, Harcourt and Brace, New York, 1964.

[Luckham 70] D. Luckham. Refinement theorems in resolution theory. In *Proc. IRIA Symp. Automatic Demonstration, Versailles, France, 1968*, pages 163–190. Springer-Verlag, 1970.

[Pelletier 86] F.J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.

[Plaisted 92] D. A. Plaisted. A simplified problem reduction format. *Artificial Intelligence*, 2(10):227–261, 192.

[Robinson 65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Sandford 80] D.M. Sandford. *Using Sophisticated Models in Resolution Theorem Proving*. Springer-Verlag, 1980.

[Slagle 67] J.R. Slagle. Automatic theorem proving with Renamable and Semantic resolution. *J. ACM*, 14(4):687–697, 1967.

[Slaney 94] J. Slaney. Scott: A model-guided theorem prover. In *Proc. CADE-12, Nancy, France*. 1994.

[Socher-Ambrosious 92] R. Socher-Ambrosious. How to avoid the derivation of redundant clauses in reasoning systems. *Journal of Automated Reasoning*, 9(1):77–97, 1992.

[Suttner & Sutcliffe 94] C. B. Suttner and G. Sutcliffe. The TPTP problem library. Technical Report AR-94-03[1] or Technical Report 93/11[2], [1]Institut fur Informatik, TU Munchen, and [2]Department of Computer Science, James Cook University, Australia, 1994.

Appendix A

Sample Program Code

A.1 Lush Resolution

```
%       File:  semantic.pl
%       A Semantic Resolution Theorem Prover
%
%       Author: Louise Dennis
%
%       Date:   June 1994
%
%       Representation: The clauses are stored as sets
%               of literals.  Each literals is marked as either a
%               positive or negative literals as follows:
%               cls(Clause_Name, Literal, positive)  or
%               cls(Clause_Name, Literal, negative).
%               Clauses may belong to test sets
%                       test_set(Test_Set_Name, Clause_Name).
%
%               Models consist of two predicates -
%               model_function(ModelName, [LanguageSymbol, ModelSymbol]).
%               model_domain(ModelName, DomainElement).
%               There should also be functions for evaluating truth
%               within the model.
%
%               Variables are considered to be any atom beginning
%               with an x.
%
%       Functions Provided:
%               lush(ClauseSet, Model, GoalClause)
%               Finds a semantic Lush resolution refutation of the
%               ClauseSet in the cpu time, Time.  Truthvalue is
%               provable,  unprovable, or error.
%
%       Other files:
%               Requires functions provided in the files -
%               unification.pl          functions for unifying
%                                       literals
%               standard.pl             substitution functions
%                                       set theory functions
%                                       standard functions
%               modeleval.pl            functions for determining
%                                       truth in models.
%               ?                       A file of Clause and Models
%               resolution.pl           Functions for resolution.
%
%-----------------------------------------------------------------
% SEMANTIC RESOLUTION THM PROVING
%-----------------------------------------------------------------
% lush(+ClauseSet, +Model, +GoalClause)
% returns provable, unprovable or error on a test set and the
% cpu time taken to find the result.
:- dynamic lush/4.

lush(N, Model, Goal):-
```

```
        % clean up.
        findall((Z, X, Y, L, P), retract(cls(r(Z, X, Y), L, P)),_),
        findall((Z, X, L, P), retract(cls(f(Z, Y), L, P)),_),
        findall((Z, X, Y, Z), retract(false(r(Z, X, Y), Z)),_),
        findall((Z, X, Y, Z), retract(notfalse(r(Z, X, Y), Z)),_),

        statistics(runtime, _),
        bagof(M, (test_set(N, M), \+ M == Goal), Clauses),
        f(Clauses, Clauses1),
        %     factors the input clauses.

        refute([Goal| Clauses1], Model, TV),
        statistics(runtime, [_, Time]),
        write(TV), nl,
        write(Time), nl.

%       refute(+Clauselist, Model,  -TV)
%       Performs semantic LUSH resolution
% exhaustively on the clauses in search of a refutation.  Using
% case analysis for non-horn sets.

refute(Clauses, Model, error):-
        \+ Model == [],
        member(N, Clauses),
        \+hornclause(N).

refute(Clauses, Model, unprovable):-
        \+ agenda_search(Clauses, Model).

refute(Clauses, Model, provable):-
        findall((Z, X, Y, L, P), retract(cls(r(Z, X, Y), L, P)),_),
        agenda_search(Clauses, Model).

%---------------------------------------------------------------------
%              THE SEARCH
%---------------------------------------------------------------------
%       agenda_search is heavily based on a search program written
%       by Dave Robertson
%
%       agenda_search(+Clauselist, +Model)
%       Performs a breadth first search of the Clauselist.  Daughters
%       are created by performing semantic LUSH resolution.  The search
% terminates when a path (branch) headed by empty is found.

agenda_search(Clauselist, Model):-
        a_search([path(Clauselist)], Model).

a_search([path(Path)|Rest], Model):-
        findall(path(NewPath), extend_path(Path, NewPath, Model), NewPaths),
        \+ member(path([empty(_, _)|_]), NewPaths),
        append(Rest, NewPaths, NewAgenda),
        a_search(NewAgenda,  Model).

a_search([path(Path)|_],  Model):-
        findall(path(NewPath), extend_path(Path, NewPath,  Model), NewPaths),
        member(path([empty(_, _)|_]), NewPaths).

%       extend_path(+Branch, -Newbranch, +OriginalClauselist)
%       extend_path has been re-written entirely from the original
%       extend-path used by Dave Robertson which was for finding routes
%       routes around the London Underground.
%       extend_path finds a possible resolution of the most recently
%       added goal clause with one of the previous clauses in the
%       branch. It checks that the new clause so produced is not
%       subsumed by any other clauses in that branch and that it is
%       false in all the models supplied.

extend_path([C1|List], [R|[C1|List]], Models):-
        member(C2, List),
        \+ =(C1, C2),
        resolve(C1, C2, R),
        \+ subsumed_by(R, [C1|List]),
        false_in_models(R, Models),
        write(R), nl.
```

```
%-------------------------------------------------------------------
%               RESOLUTION
%-------------------------------------------------------------------
%       resolves the two clauses N and M together and creates their
%       resolvant and returns its name if they are resovable.  Returns
%       unresolvable if they are not and error if either N, or M is
%       not a number of a clause ordoes not store a prolog term that
%       represents a clause.  On backtracking resolve/3 will produce
%       further resovants until it reaches unresolvable.  If the
%       resolvant is the empty clause resolve returns empty.

resolve(M, N, Resolvant):-
        check_clause(M),
        check_clause(N),
        standardize(M, N), !,
        resolve1(M, N, ResolvantPools),
        factor(ResolvantPools, Rs),
        makeresolvant(M, N, Rs, Resolvant).

resolve(_, N, error):-
        \+ check_clause(N).

resolve(M, _, error):-
        \+ check_clause(M).

%       makeresolvant(+Clause1, +Clasue2, +LiteralPools, -Resolvant)
%       Picks a literal Pool and asserts a clause that is the
%       resolvant represented by that pool.
makeresolvant(X, Y, Pools, empty(X, Y)):-
        member([], Pools), !.
makeresolvant(X, Y, Pools, r(N, X, Y)):-
        member(Pool, Pools),
        version_number(X, Y, N),
        findall((L, P), (member((L, P), Pool),
               assert(cls(r(N, X, Y), L, P))), _).

%       resolve1(+Clause1, +Clause2, -Resolvants)
%       Performs binary resolution on the two clauses and returns
%       list of pools of the literals contained in the possible
%       resolvants.

resolve1(M, N, NewPools):-
        resolve_literals(M, N, Pools),
        makeunifiedpools(Pools, NewPools).

resolve1(_, _, unresolvable).

% resolve_literals(+Clause1, +Clause2, -ResolvantLiteralPool)
% Provides an intermediary step.  Converting clause names into
% literal pools.
resolve_literals(M, N, PoolsOut):-
        findall((L1, P1), cls(M, L1, P1), PoolM),
        findall((L2, P2), cls(N, L2, P2), PoolN),
        append(PoolM, PoolN, PoolIn),
        resolution(PoolM, PoolN, PoolIn, PoolsOut), !.

%       resolution(+ClausePool1, +ClausePool2, LiteralPool, -Pools)
%       generates as many binary resolvants as possible of the Clauses
%       supplied.

resolution(PoolM, PoolN,  PoolIn, Pools):-
        setof(([(L1, P1), (L2, P2)], Unifier),
      ( member((L1, P1), PoolM),
      member((L2, P2), PoolN),
      \+ P2 == P1,
      unified(L1, L2, Unifier)), ElementSet),
        setof((Pool, Unifier),
      X^Y^Poola^(member(([X, Y], Unifier),
      ElementSet),
memberout(X, PoolIn, Poola), memberout(Y, Poola, Pool)), Pools).
```

A.2 Clash Resolution

```
%       File:  clash.pl
%       A Clash Resolution Theorem Prover
%
%       Author: Louise Dennis
%
%       Date:   June 1994
%
%       Representation: The clauses are stored as sets
%               of literals.  Each literals is marked as either a
%               positive or negative literals as follows:
%               cls(Clause_Name, Literal, positive)  or
%               cls(Clause_Name, Literal, negative).
%               Clauses may belong to test sets
%                       test_set(Test_Set_Name, Clause_Name).
%
%               Models consist of two predicates -
%               model_function(ModelName, [LanguageSymbol, ModelSymbol]).
%               model_domain(ModelName, DomainElement).
%               There should also be functions for evaluating
%               truth within the model.
%
%               Numberings are determined by a predicate
%               numb(Numbering, PredicateName, Number)
%               e.g. (numbering1, greater_than, 3).
%
%               Variables are considered to be any atom beginning
%               with an x.
%
%       Functions Provided:
%               clash(ClauseSet, ModelList, Numbering)
%               Finds a clash resolution refutation of the ClauseSet
%               in the cpu time, Time.  Truthvalue is provable,
%               unprovable, or error.
%
%       Other files:
%               Requires functions provided in the files -
%               unification.pl          functions for unifying
%                                       literals
%               standard.pl             substitution functions
%                                       set theory functions
%                                       standard functions
%               modeleval.pl            modelevalution
%               resolution.pl           functions used in resolution
%                                       algorithm
%               ?                       Sets of clauses, models and
%                                       numberings.
%
%-----------------------------------------------------------------
%               CLASH RESOLUTION THM PROVING
%-----------------------------------------------------------------
%       clash(+ClauseSet, +Model, +Numbering)
%       returns provable, unprovable or error on a test set and the
%       cpu time taken to find the result.
:- dynamic clash/5.

clash(TestSet, Model, Numbering):-
        % clean up.
        findall((N, X, Y, L, P), retract(cls(r(N, X, Y), L, P)),_),
        findall((N, X, L, P), retract(cls(f(N, X), L, P)),_),
        findall((N, X, Y, Z), retract(false(r(N, X, Y), Z)),_),
        findall((N, X, Y, Z), retract(notfalse(r(N, X, Y), Z)),_),

        statistics(runtime, _),
        bagof(M, test_set(TestSet, M), Clauses),
        f(Clauses, Clauses1),
        %       Factors input clauses

        refute(Clauses1, Model, Numbering, TruthValue),
        statistics(runtime, [_, Time]),
```

```
        write(TruthValue), nl,
        write(Time), nl.

%       refute(+Clauses, +Model, +Numbering, -TV)
%       performs clash resolution
%       exhaustively on the clauses in search of a refutation.

refute(Clauses, Model, _, error):-
        %       Checking set is eleigable if mulitple model semantic
        %       clash resolution is being used.
        \+ ( Model == [] ; Model = [_]),
        member(N, Clauses),
        \+hornclause(N).

refute(Clauses, Model, Numbering, unprovable):-
        \+ agenda_search(Clauses, Model, Numbering).

refute(Clauses, Model, Numbering, provable):-
        findall((N, X, Y, L, P), retract(cls(r(N, X, Y), L, P)),_),
        agenda_search(Clauses, Model, Numbering).

%----------------------------------------------------------------------
%               THE SEARCH
%----------------------------------------------------------------------
%       agenda_search is heavily based on a search program written
%       by Dave Robertson
%
%       agenda_search(+Clauses, +Model, +Numbering)
%       Performs a breadth first search of the Clauses.  Daughters are
%       created by performing semantic LUSH resolution.  The search
%       terminates when a path (branch) headed by empty is found.

agenda_search(Clauses, Model, Numbering):-
        a_searchC(Clauses, Model, Numbering).

a_searchC(Clauses,_, _):- member(empty(_, _), Clauses).

a_searchC(Clauses, Model, Numbering):-
        bagof(NewC, extend_pathC(Clauses, NewC, Model, Numbering), NewClauses),

        %      NewClauses is a list of lists.  Smooth turns this into
        %      a single list.
        smooth(NewClauses, New),
        redundancy_check(New, NewClauses2, []),
        append(NewClauses2, Clauses,  NewAgenda),
        a_searchC(NewAgenda, Model, Numbering).


%       extend_pathC(+Clauses, -Resolvant, +Model, +Numbering)
%       extend_path has been re-written entirely from the original
%       extend-path used by Dave Robertson which was for finding routes
%       around the London Underground.
% extend_path finds a clash resolvant of some of the clauses.

extend_pathC(Clauses, R, Model, Numb):-
        % pick a nucleus and store the literals in the literal
        %       Pool.
        member(Nucleus, Clauses),

        findall((L, P), cls(Nucleus, L, P), Pool),
        memberout(Nucleus, Clauses, Clauses2),
        selected_lush(Pool, Clauses2, Numb, Model, Pools),
        makeresolvants(Pools, R2, Nucleus),

        %       Check removes any resolvants from the list that are
        %       subsumed by Clauses or are true in the model(s).
        check(R2, Clauses, Model, R),
        \+ R == [].

%---------------------------------------------------------------------
%               RESOLUTION
%---------------------------------------------------------------------
%       selected_lush(+Pool, +Clauses, +Numbering, +ModelList,
%       -Resolvant)
```

```
%       performs input resolution to find clash resolvants.  In a
%       manner similar to Lush Resolution.

selected_lush(Pool, Clauses, Numb, Model, R):-
        a_search_sl([([],Pool)], Clauses, Numb, Model, R),!.

%       All branches have been fully explored.  End search.
a_search_sl([], _, _, _, []).

%       Intermediate clash is passed back and the extended branch is
%       placed in a NewAgenda to be explored further.
a_search_sl([(E, P)|Rest], Clauses, Numb, Model, [(E, P)|R]):-
        \+ E == [],
        fim(P, Model, E),
        \+ sb(P, Clauses),
        %      Checking that the clash is false in the model and
        %      passes the subsumption check before it is passed back
        %      by the algorithm.   Different predicates needed from
        %      false_in_models/2 and subsumed_by/2 since we are
        %      dealing with literal pools, not clause names.

        bagof(NewPath, extend_path_sl((E, P), NewPath, Clauses,
                          Numb, Model), NewPaths),
        smooth(NewPaths, N),
        sorta(N, [], Paths),
        \+ member((_, []), Paths),
        append(Paths, Rest, NewAgenda),
        a_search_sl(NewAgenda, Clauses, Numb, Model, R), !.

%       Intermediate clash is not passed back.  Extended branch is
%       added on to the end of the Agenda.
a_search_sl([(E, P)|Rest], Clauses, Numb, Model, R):-
        (E == [];  \+ fim(P, Model, E); sb(P, Clauses)),
        bagof(NewPath, extend_path_sl((E, P), NewPath, Clauses, Numb, Model),
                                                                NewPaths),
        smooth(NewPaths, N),
        sorta(N, [], Paths),
        \+ member((_, []), Paths),
        append(Paths, Rest, NewAgenda),
        a_search_sl(NewAgenda, Clauses, Numb, Model, R), !.

%       Empty clause has been reached.  End branch.
a_search_sl([(E, P)|_],  Clauses, Numb, Model, [empty(X)]):-
        bagof(NewPath, extend_path_sl((E, P), NewPath, Clauses, Numb, Model),
                                                                NewPaths),
        smooth(NewPaths, N),
        sorta(N, [], Paths),
        member((X,[]), Paths), !.

%       This branch cannot be extended further.  Pass back clash and
%       continue exploring other branches.
a_search_sl([(E, P)|Rest], Clauses, Numb, Model, [(E, P)|R]):-
        \+ E == [],
        fim(P, Model, E),
        \+ sb(P, Clauses),
        \+ bagof(NewPath, extend_path_sl((E, P), NewPath,
                         Clauses, Numb, Model), _),
        a_search_sl(Rest, Clauses, Numb, Model, R), !.

%       This branch cannot be extended further.  Furthermore the
%       resolvant is either true in some model or is subsumed by some
%       of the input clauses.  Forget this branch and continue
%       exploring the others.
a_search_sl([(E, P)|Rest], Clauses, Numb, Model, R):-
        (E == []; \+ fim(P, Model, E); sb(P, Clauses)),
        \+ bagof(NewPath, extend_path_sl((E, P), NewPath,
                     Clauses, Numb, Model), _),
        a_search_sl(Rest, Clauses, Numb, Model, R), !.

%       extend_path_sl(+Branch, -BranchExtensions, +InputClauses,
%               +Numbering +Model)
%       forms new branches from possible binary resolvants of the
%       intermediate resolvant and a member of the input clauses.
extend_path_sl((Electrons, Pool), NewPaths, Input, Numb, Model):-
```

```
        member(E, Input),
        \+ member(E, Electrons),
        false_in_models(E, Model),
        resolve_literalsC(E, Pool, Numb, Pools),
        makeunifiedpools(Pools, NewPools),
        factor(NewPools, NewNewPools),
        makepaths([E|Electrons], NewNewPools, NewPaths).

%       resolve_literals(+Nucleus, -IntermediateResolvantPool,
%                    +Numbering, -Pools)
%       Sorts out the various resolvant pools to be used in binary
%       resolution and then produces binary resolvants.
resolve_literalsC(M, PoolSoFar, Numb, Pools):-
        findall((L, P), (cls(tmp, L, P), retract(cls(tmp, L, P))), _),
        findall((L, P), (member((L, P), PoolSoFar), assert(cls(tmp, L, P))), _),
        standardize(M, tmp),
        findall((L, P), cls(M, L, P), FirstPool),
        findall((L, P), cls(tmp, L, P), SecondPool),
        append(FirstPool, SecondPool, PoolIn), !,
        resolution(M, FirstPool, SecondPool, Numb, PoolIn, Pools).

%       resolution(+Electron, +ElectronPool, +ResolvantPool,
%       +Numbering, LiteralPool, -Pools)
%       generates as many binary resolvants as possible of the Electron
%       and the intermediate resolvant supplied.
resolution(M, PoolM, PoolN,  Numb, PoolIn, Pools):-
        setof(([(L1, P1), (L2, P2)], Unifier),
      ( member((L1, P1), PoolM),
      \+ larger_literal(M, L1, Numb, PoolM),
      member((L2, P2), PoolN),
      \+ P2 == P1,
      unified(L1, L2, Unifier)), ESet),
        setof((Pool, Unifier), X^Y^Poola^(member(([X, Y], Unifier), ESet),
              memberout(X, PoolIn, Poola), memberout(Y, Poola, Pool)),
              Pools).

%       sorta(+Paths, +PathsChecked, -Paths)
%       weeds out redundant clause pools.
sorta([], Paths, Paths):- !.
sorta([(Electrons, Pool)|Paths], PathsSoFar, NewPaths):-
        \+ check_es(Electrons, Pool, Paths),
        \+ check_es(Electrons, Pool, PathsSoFar),
        sorta(Paths, [(Electrons, Pool)| PathsSoFar], NewPaths), !.
sorta([(Electrons, Pool)|Paths], PathsSoFar, NewPaths):-
        check_es(Electrons, Pool, Paths),
        sorta(Paths, PathsSoFar, NewPaths), !.
sorta([(Electrons, Pool)|Paths], PathsSoFar, NewPaths):-
        check_es(Electrons, Pool, PathsSoFar),
        sorta(Paths, PathsSoFar, NewPaths), !.

check_es(Electrons, Pool, Paths):-
        member((E, P), Paths),
        same_set(E, Electrons),
        subsumes(Pool, P), !.

%       makepaths(+Electrons, +Pools, -Clashes).
%       combines the notation for electrons and the possible pools
%       generated  by them and the nucleus into the path notation.
makepaths(_, [], []).
makepaths(Electrons, [H|T], [(Electrons, H)| T1]):-
        makepaths(Electrons, T, T1), !.

%---------------------------------------------------------------------
%       DETERMINING THE "SIZE" OF A LITERAL
%---------------------------------------------------------------------
%       larger_literal
% (+Clause, +Literal, +Numbering, +PoolIn)
% suceeds if there is a "larger" literal than Literal in the
%       Clause which is also part of the Pool available, asscording to
%       the Numbering.
larger_literal(Clause, Literal, Numb, Pool):-
        cls(Clause, L1, P),
        member((L1, P), Pool),
        \+ L1 == Literal,
```

```
        literal_size(L1, Numb, N1),
        literal_size(Literal, Numb, N),
        N1 > N, !.

% literal_size(+Function, +Numbering, -Number)
% Returns the number of the "largest" predicate appearing in the
% function according to Numbering.
literal_size(L, _, 0):-
        atomic(L),!.

literal_size(L, Numb, M1):-
        functor(L, F, N),
        numb(Numb, F, M1),
        llarg(L, N, M2, Numb),
        M1 > M2, !.

literal_size(L, Numb, M2):-
        functor(L, F, N),
        numb(Numb, F, M1),
        llarg(L, N, M2, Numb),
        M1 <= M2, !.

% llarg(+Function, +Arity, -Number, +Numbering)
% Number is the largest Predicate Number appearing in the argument
% list of the Function.
llarg(_, 0, 0, _):-!.

llarg(L, N, M1, Numb):-
        arg(N, L, A),
        literal_size(A, Numb, M1),
        N1 is N - 1,
        llarg(L, N1, M2, Numb),
        M1 > M2, !.

llarg(L, N, M2, Numb):-
        arg(N, L, A),
        literal_size(A, Numb, M1),
        N1 is N - 1,
        llarg(L, N1, M2, Numb),
        M1 < M2, !.

%-----------------------------------------------------------------------
%       NAMING RESOLVANTS
%-----------------------------------------------------------------------
%       checkpool(+LiteralPool, +Electrons, +Nucleus),
%       Fails if the LiteralPool is already represented as a resolvant
%       of the Electrons and the Nucleus.
check_pool(Pool, Electrons, Nucleus):-
        cls(r(N, Electrons, Nucleus),_, _),
        findall((L, P), cls(r(N, Electrons, Nucleus), L, P), Pool2),
        perm(Pool, Pool2), !,
        fail.

check_pool(_, _, _).

%---------------------------------------------------------------------
%       OTHER
%---------------------------------------------------------------------
%       redundancy_check(Clauselist, ClauselistOut, ClauselistIn),
%       Looks at the clauses in Clauselist one at a time and checks
%       that they are not subsumed by any other Clause in that List.
%       Returns a list of the remaining clauses.
redundancy_check([], Clauses, Clauses).
redundancy_check([H| T], Clauses, T1):-
        \+ subsumed_by(H, T),
        \+ subsumed_by(H, T1),
        write(H), nl,
        redundancy_check(T, Clauses, [H| T1]), !.
redundancy_check([_|T], Clauses, T1):-
        redundancy_check(T, Clauses, T1), !.

%       memberi(+Element, +List).
%       checks whether the Element is a member of List.  Does not
%       allow backtracking.
```

```
memberi(X, [X|_]):-!.
memberi(X, [_|T]):-
    memberi(X, T),!.

%       smooth(+Listoflists, -List)
%       Joins all the lists in Listoflists together and returns them
%       as a list.
smooth([], []):-!.
smooth([H | T], T1):-
        smooth(T, T2),
        append(H, T2, T1), !.

%       check(+Resolvants, +Clauses, +Model, -ResolvantList)
%       Suceeds if all the resolvants in the ResolvantList are in
%       Resovlants and are not subsumed by clauses and are false in
%       the Model(s).
check([], _, _, []):-!.
check([R | T], Clauses, Model, [R | T1]):-
        \+ subsumed_by(R, Clauses),
        false_in_models(R, Model),
        check(T, Clauses, Model, T1), !.
check([_ | T], Clauses, Model, T1):-
        check(T, Clauses, Model, T1), !.

%       makeresolvants(+Paths, -ResolvantNames, +Nucleus)
%       names and asserts clauses corresponding to the Paths.
makeresolvants([], [], _):-!.
makeresolvants([empty(Electrons)|_], [empty( Electrons, Nucleus)], Nucleus):-!.
makeresolvants([(Electrons, Pool)|T], [r(N, Electrons, Nucleus)| R], Nucleus):-
        version_number(Electrons, Nucleus, N),
        findall((L, P), (member((L, P), Pool),
                     assert(cls(r(N, Electrons, Nucleus), L, P))), _),
        makeresolvants(T, R, Nucleus),!.

%       fim(+Pool, +Model, +ElectronList)
%       Converts the Pool into a clause (if it is a "new" resolvant)
%       and then calls false_in_models/2.
fim(P, Model, E):-
        \+ E == [],
        findall((L, P1), (cls(f(1, tmp), L, P1),
                  retract(cls(f(1,tmp), L, P1))), _),
        findall((L, P1), (member((L, P1), P),
                  assert(cls(f(1, tmp), L, P1))), _),
        false_in_models(f(1, tmp), Model),
        findall((L, P1), (cls(f(1,tmp), L, P1),
                  retract(cls(f(1, tmp), L, P1))), _),!.

%       sb(+Pool, +Clauses)
%       Converts the Pool into a clause (if it is a "new" resolvant)
%       and then calls subsumes_by/2.
sb(P, Clauses):-
        findall((L, P1), (cls(tmp, L, P1), retract(cls(tmp, L, P1))), _),
        findall((L, P1), (member((L, P1), P), assert(cls(tmp, L, P1))), _),
                 subsumed_by(tmp, Clauses),
        findall((L, P1), (cls(tmp, L, P1), retract(cls(tmp, L, P1))), _),!.
```

A.3 Model Evaluation

```
%       File:  modeleval.pl
%       Model Evaluation Predicates
%
%       Author: Louise Dennis
%
%       Date:   June 1994
%
%       Representation: The clauses are stored in a file test.pl
%               as sets of literals.  Each literals is marked as
%               either a positive or negative literals as follows:
```

```
%               cls(Clause_Name, Literal, positive)  or
%               cls(Clause_Name, Literal, negative).
%               Clauses may belong to test sets
%                       test_set(Test_Set_Name, Clause_Name).
%
%               Models are stored in a file models.pl.  They consist
%               of two predicates -
%               model_function(ModelName, [LanguageSymbol, ModelSymbol]).
%               model_domain(ModelName, DomainElement).
%               There may also be small function for evaluating truth
%               within the model.
%
%               Variables are considered to be any atom beginning
%               with an x.
%
%---------------------------------------------------------------------
% EVALUATING TRUTH IN THE MODEL
%---------------------------------------------------------------------
% false_in_models(+Clause, +Models)
% succeeds if the clause is false in all the models.

false_in_models(_, []):- !.

false_in_models(Clause, [Model | Models]):-
        false_in_model(Clause, Model),
        false_in_models(Clause, Models),!.

% false_in_model(+Clause, +Model)
% succeeds if the clause is false in the model.  That is there
% is a grounding of the clause in the model that is false in the
%       model.

false_in_model(_, nomodel):-!.

false_in_model(f(_, Clause), Model):-
        false(Clause, Model),!.

false_in_model(f(_, Clause), Model):-
        notfalse(Clause, Model),!,
        fail.

false_in_model(f(N, C), Model):-
        \+ false(C, Model),
        \+ notfalse(C, Model),
        findall(Var, L^(cls(f(N,C), L, _), o_check([L], Var), vars(Var)),
                Vars1),
        findall(S,(substitution(Vars1, Model, S),
                false_in(S, f(N, C), Model)), X),
        \+ X == [],
        assert(false(C, Model)),!.

false_in_model(Clause, Model):-
        \+ Clause = f(_, _),
        false(Clause, Model),!.

false_in_model(Clause, Model):-
        \+ Clause = f(_, _),
        notfalse(Clause, Model),!,
        fail.

false_in_model(C, Model):-
        \+ C = f(_, _),
        \+ false(C, Model),
        \+ notfalse(C, Model),
        findall(Var, L^(cls(C, L, _), o_check([L], Var), vars(Var)), Vars1),
        findall(S,(substitution(Vars1, Model, S), false_in(S, C, Model)), X),
        \+ X == [],
        assert(false(C, Model)),!.

% substitution(+Variables, +Model, -Substitution)
% produces a substituion for each variable from the model domain.
substitution([], _, []).
substitution([H|T], Model, [(H, D)|T1]):-
        \+ member(H, T), !,
```

```
        model_domain(Model, D),
        substitution(T, Model, T1).
substitution([H|T], Model, T1):-
        member(H, T), !,
        substitution(T, Model, T1).

%       ground(+Literal, +Substitution, +Model, -GroundLiteral)
%       rewrites the literal using the semantic function defined
%       by the model and grounds it according to the substitutions
%       provided.
ground(G, Sub, G1,_):-
        vars(G),
        sublist([G], Sub, [G1]),!.
ground(G, _, G1, Model):-
        atomic(G),
        \+ vars(G),
        model_function(Model, [G, G1]),!.
ground(G, Sub, G1, Model):-
        \+ atomic(G),
        G =.. [F | A],
        ground1(A, Sub, Model, A1),
        model_function(Model, [F, F1]),
        G1 =.. [F1 | A1],!.

ground1([], _,  _, []).
ground1([H|T], Sub, Model, [H1| T1]):-
        ground(H, Sub, H1, Model),
        ground1(T, Sub, Model, T1).

% false_in(+Substitution, +Clause, +Model)
false_in(Sub, Clause, Model):-
        \+ true_in(Sub, Clause, Model),!.

% true_in(+Substitution, +Clause, +Model)
% succeeds if one literal in the clause is true in the model.
true_in(Sub, Clause, Model):-
        cls(Clause, L, positive),
        ground(L, Sub, G, Model),
        call(G),!.

true_in(Sub, Clause, Model):-
        cls(Clause, L, negative),
        ground(L, Sub, G, Model),
        \+ call(G),!.

% o_check(+List, ?Variable)
% breaks down the "functions" in the list to see if Variable
% occurs in them.
o_check([], _):- fail.

o_check([X|_], X).

o_check([H|_], X):-
        H =.. [_|List],
        o_check(List, X).

o_check([_|T], X):-
        o_check(T, X).
```

A.4 Subsumption Checking etc.

```
%       File:  resolution.pl
%
%       Author: Louise Dennis
%
%       Date:   August 1994
%
% Description: Various Predicates for substitution, set theory etc.
% to be used with program2.pl and unification.pl
```

```
%-----------------------------------------------------------------------
% SUBSUMPTION
%-----------------------------------------------------------------------
%       subsumed_by(+Clause, +List)
%       succeeds if Clause is subsumed by some clause appearing in the
%       List.
subsumed_by(_, []):-
        fail.

subsumed_by(C, [H|_]):-
        standardize(C, H),
        bagof((L1, P), cls(H, L1, P), Literals),
        unisome(Literals, C, _, _),!.

subsumed_by(C, [H|T]):-
        bagof((L1, P), cls(H, L1, P), Literals),
        \+ unisome(Literals, C, _, _),
        subsumed_by(C, T),! .

%       unisome(+Literals, +Clause, -Unifier, Clauselist).
%       Succeeds if the clause is subsumed by the literals.  That is
%       the literals represent some clause C.  Such that Clause <->
%       C.theta v E.

unisome([], _, [], []).

unisome([(L1, P)|T], C, Unifier, [(L2, P)|List]):-
        unisome(T, C, Unifier2, List),
        cls(C, L2, P),
        \+ member((L2, P), List),
        sublist([L1], Unifier2, [L1d]),
        match([(L1d, L2)], [], Unifier1),
        append(Unifier1, Unifier2, Unifier).

%       unified(+Literal1, +Literal2, -Unifier)
%       suceeds if Unifier is the Most General Unifier of the two Literals.
unified(L, L1, Unifier):-
        unify([(L, L1)], [], Unifier), !.



%-------------------------------------------------------------------
%       TESTING FOR HORN CLAUSES
%-------------------------------------------------------------------
%       hornclause(+Clause).
%       succeeds if Clause is a horn clause.  Achieved by checking
%       that there is only one positive literal in the Clause.

hornclause(N):-
        findall(L, cls(N, L, positive), Positive),
        length(Positive, M),
        =<(M, 1),!.
%--------------------------------------------------------------------
%       RESOLUTION
%--------------------------------------------------------------------
%       consistent_with(+Unifier1, +Unifier2, -Unifier3)
%       checks that Unifier1 and Unifier2 are consistent with each
%       other and tidies them up a bit and combines them to produce
%       Unidier3.
consistent_with(Unifier1, Unifier2, Unifier3):-
        append(Unifier1, Unifier2, Unifier),
        unify(Unifier, [], Unifier3),!.

%--------------------------------------------------------------------
%       makeset(+Pool1, +Pool2, -Pool3)
%       forms Pool3 by joining Pool1 and Pool2 and removing any
%       members of Pool1 and Pool2 that "contain" other members. i.e.
%       it checks for subsumption.
makeset(P1, P, Pool):-
        findall(X, (member(X, P1), \+poolmember(X, P)), Poola),
        findall(X, (member(X, P), \+poolmember(X, Poola)), Poolb),
        append(Poola, Poolb, Pool).

%       poolmember(Pool, Pools)
```

```
%       checks that Pool does not contain any mamber of Pools.
poolmember(X, Pool):-
        member(Y, Pool),
        subres(X, Y),!.

%       subres(+Pool1, +Pool2)
%       checks that Poo1's unifier is not conatined in Pool2's and
%       that Pool2's literals are not contained in Pool1.
subres((Pool1, Unifier1), (Pool2, Unifier2)):-
        subbag(Unifier2, Unifier1),
        subbag(Pool1, Pool2),!.

%       subbag(+X, +Y)
%       succeeds if Y is contained in X.
subbag(_, []):-!.
subbag(S, [H1 | T1]):-
        member(H1, S),
        memberout(H1, S, S1),
        subbag(S1, T1),!.
%-----------------------------------------------------------------------
%       makeunifiedpools(+Pools, -NewPools)
%       takes a list of Literal Pools with unifiers and returns a
%       list of those Pools once the unifiers have been substituted
%       into them.
makeunifiedpools([], []):-!.
makeunifiedpools([(Pool, Unifier) | Pools], [NewPool |Newpools]):-
        findall((L, P), (cls(rp, L, P), retract(cls(rp, L, P))), _),
        findall((L, P), (member((L, P), Pool), assert(cls(rp, L, P))), _),
        subinto(rp, Unifier),
        findall((L, P), (cls(rp, L, P), retract(cls(rp, L, P))), NewPool),
        makeunifiedpools(Pools, Newpools),!.

%       version_number(+X, +Y, -Number)
%       Produces the smallest number for labelling a resolvant of the
%       X and Y that hasn't already been used.
version_number(X, Y, 1):-
        \+ cls(r(_, X, Y),_,_), !.

version_number(X, Y, N1):-
        cls(r(N, X, Y), _, _),
        N1 is N + 1,
        \+ cls(r(N1, X, Y), _, _),!.

%       vn(+X,  -Number)
%       Produces the smallest number for labelling a factor of X that
%       hasn't already been used.
vn(H, 1):-
        \+ cls(f(_, H), _, _), !.
vn(H, N1):-
        cls(f(N, H), _, _),
        N1 is N + 1,
        \+ cls(f(N1, H), _, _).

%-----------------------------------------------------------------------
%       FACTORING
%-----------------------------------------------------------------------
%       factor(+X, -Y)
%       Succeeds if Y is a list of the factors  of the literals in X.
factor([], []):-!.
factor([H|T], Out):-
        setof(F, X^factors(H, F, X), Factors),
        sub_check(Factors, [], Fs),
        factor(T, T1),
        append(Fs, T1, Out),!.

%       factors(+X, -Y, -Unifier)
%       Succeeds if Y is a factor of the literals in X, unified by
%       Unifier.
factors([], [], []).
factors([X|M], [X1|K], Unifier):-
        factors(M, K, Unifier),
        s([X], Unifier, [X1]).
factors([(L1, P1)|M], K, Unifier):-
        member((L2, P1), M),
```

```
        unified(L1, L2, Unifier1),
        s(M, Unifier1, M1),
        factors(M1, K, Unifier2),
        append(Unifier1, Unifier2, Unifier).

%       f(+X, -Y)
%       Succeeds if Y is a list of the factors  of the clauses in X.
f([], []):-!.
f([H|T], New):-
        findall((L, P), cls(H, L, P), Pool),
        setof(Pools, X^factors(Pool, Pools, X), P),
        nam(H, P, Names),
        f(T, T1),
        append(Names, T1, New), !.

%       nam(+Clause, +Pools, -Names)
%       Names and asserts the factors of the Clause.
nam(_, [], []):-!.
nam(H, [Pool | Pools], [f(N, H)|Names]):-
        vn(H, N),
        findall((L, P), (member((L, P), Pool),
                assert(cls(f(N, H), L, P))), _),
        nam(H, Pools, Names), !.

%       s(+List, +Unifier, -List)
%       substitution predicate.  Substitutes the Unifer into the
%       Literals in the list.
s([], _, []):-!.
s([(H, P) | T], Unifier, [(H1, P) | T1]):-
        sublist([H], Unifier, [H1]),
        s(T, Unifier, T1), !.

%       sub_check(+Pools, +CheckedPoolsSoFar, -CheckedPools)
%       subsumption checking for the factors.
sub_check([], Factors, Factors):-!.
sub_check([H | T], Fs, Factors):-
        \+ (member(X, T), subsumes(X, H)),
        \+ (member(X, Fs), subsumes(X, H)),
        sub_check(T, [H | Fs], Factors), !.
sub_check([H | T], Fs, Factors):-
        member(X, T),
        subsumes(X, H),
        sub_check(T, Fs, Factors), !.
sub_check([H | T], Fs, Factors):-
        member(X, Fs),
        subsumes(X, H),
        sub_check(T, Fs, Factors), !.

subsumes([], []):-!.
subsumes([H|T], L):-
        member(K, L),
        match([(K, H)], [], Unifier),
        sublist(T, Unifier, T1),
        memberout(K, L, L1),
        subsumes(T1, L1), !.
```

Appendix B

Sample Program Output

There follow some samples of the output of the program applied to various problems mentioned in the disseration.

B.1 The Alternate Interior Angles in a Trapezoid are Equal

This problem appears in Slagle's paper [Slagle 67] but has been taken from the TPTP database of problems [Suttner & Sutcliffe 94] which was used for testing SemCheck, where its syntatctic name is GEO079-1.

The problem is to prove that the alternate interior angles in a trapezoid are equal. This is the problem under consideration in figure 3–1. It is also one of the few problems where Semantic Lush resoution reduced the search space (see chapter 6).

The input file is below. The header is taken from the original TPTP file. The clauses have been placed in the format required by SemCheck.

An ordering of the predicates and two models have also been supplied. The first model represents a square, and the second model a square with a rightangle triangle attached (see figure B–1). NB. Sometimes a predicate/function/constant symbol is the same for both models in which case it is defined only once using an underscore instead of a model name. e.g. `model_function(_, [a, a])`.

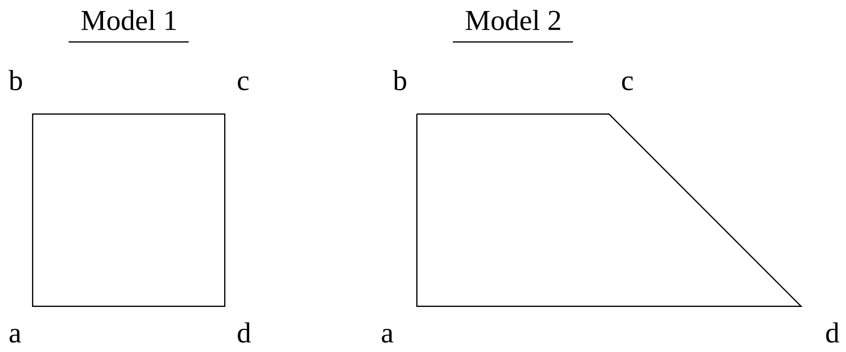

Figure B–1: Two Models for Slagle's Geometry Theorem

In fact, both models behaved identically, but the output is included here for completeness.

B.1.1 The Input File

```
%------------------------------------------------------------------------
% File     : GEO079=AltAngleEq-1 : TPTP v1.1.1. Bugfixed v1.1.0.
% Domain   : Geometry
% Problem  : The alternate interior angles in a trapezoid are equal
% Version  : [Chang, 1970] axioms : Incomplete.
% English  : The alternate interior angles formed by a diagonal of a (not
%            necessarily isosceles) trapezoid are equal.

% Refs     : Slagle J.R. (1967), Automatic Theorem Proving With Renamable
%            Semantic Resolution, Journal of the ACM 14(4), 687-697.
% Source   : [Slagle, 1967]
% Names    : GEOMETRY THEOREM [Slagle, 1967]

% Status   :
% Syntax   : Number of clauses          :    6 (   0 non-Horn)(   2 unit)
%            Number of literals         :   11 (   0 equality)
%            Number of predicate symbols :   5 (   0 propositional)
%            Number of function symbols  :   4 (   4 constant)
%            Number of variables        :   20 (   0 singleton)
%            Maximal clause size        :    3
%            Maximal term depth         :    1

% Comments :
%------------------------------------------------------------------------
:- dynamic cls/3.

cls(right_angles_are_equal,right_angle(xxx,xx,x),negative).
cls(right_angles_are_equal,right_angle(xxxxxx,xxxxx,xxxx),negative).
cls(right_angles_are_equal,eq(xxx,xx,x,xxxxxx,xxxxx,xxxx),positive).
test_set(test,right_angles_are_equal).

cls(corresponding_angles_are_equal,
                 congruent(xxxxxx,xxxxx,xxxx,xxx,xx,x),negative).
cls(corresponding_angles_are_equal,eq(xxxxxx,xxxxx,xxxx,xxx,xx,x),
                 positive).
test_set(test,corresponding_angles_are_equal).

cls(trapezoid_definition,trapezoid(xxxx,xxx,xx,x),negative).
cls(trapezoid_definition,parallel(xxx,xx,xxxx,x),positive).
test_set(test,trapezoid_definition).

cls(interior_angles_are_equal,parallel(xxxx,xxx,xx,x),negative).
```

```
cls(interior_angles_are_equal,eq(xx,xxx,xxxx,xxx,xx,x),positive).
test_set(test,interior_angles_are_equal).

cls(a_trapezoid,trapezoid(a,b,c,d),positive).
test_set(test,a_trapezoid).

cls(prove_angles_equal,eq(a,c,b,c,a,d),negative).
test_set(test,prove_angles_equal).

%---------------------------------------------------------------------
% NUMB
%---------------------------------------------------------------------
numb(1, right_angle, 5).
numb(1, eq, 3).
numb(1, congruent, 4).
numb(1, trapezoid, 2).
numb(1, parallel, 1).

%--------------------------------------------------------------------
% MODELS
%---------------------------------------------------------------------
:-dynamic notfalse/2.

notfalse(right_angles_are_equal, _).
notfalse(corresponding_angles_are_equal, _).
notfalse(trapezoid_definition, _).
notfalse(interior_angles_are_equal, _).
notfalse(a_trapezoid, _).

model_function(_, [right_angle, rightangle]).
model_function(1, [eq, equal1]).
model_function(2, [eq, equal2]).
model_function(_, [congruent, corresponding]).
model_function(_, [trapezoid, trapezoid]).
model_function(1, [parallel, p1]).
model_function(2, [parallel, p2]).
model_function(_, [a, a]).
model_function(_, [b, b]).
model_function(_, [c, c]).
model_function(_, [d, d]).

model_domain(_, a).
model_domain(_, b).
model_domain(_, c).
model_domain(_, d).

equal1(A, B, C, D, E, F):-
        val(a1(A, B, C), N1, 0),
        val(a1(D, E, F), N2, 0),
        N1 == N2.

equal2(A, B, C, D, E, F):-
        val(a2(A, B, C), N1, 0),
        val(a2(D, E, F), N2, 0),
        N1 == N2.

val(a1(a, b, c), 90, _).
val(a1(b, c, d), 90, _).
val(a1(c, d, a), 90, _).
val(a1(d, a, b), 90, _).
val(a1(a, b, d), 45, _).
val(a1(c, b, d), 45, _).
val(a1(c, d, b), 45, _).
val(a1(a, d, b), 45, _).
val(a1(c, a, b), 45, _).
val(a1(c, a, d), 45, _).
val(a1(a, c, d), 45, _).
val(a1(a, c, b), 45, _).
val(a1(X, Y, Z), N, 0):-
        val(a1(Z, Y, X), N, 1).

val(a2(a, b, c), 90, _).
val(a2(b, c, d), 135, _).
val(a2(c, d, a), 45, _).
```

```
val(a2(d, a, b), 90, _).
val(a2(a, b, d), 60, _).
val(a2(c, b, d), 30, _).
val(a2(c, d, b), 15, _).
val(a2(a, d, b), 30, _).
val(a2(c, a, b), 45, _).
val(a2(c, a, d), 45, _).
val(a2(a, c, d), 90, _).
val(a2(a, c, b), 45, _).
val(a2(X, Y, Z), N, 0):-
        val(a2(Z, Y, X), N, 1).

p1(X, Y, Z, W):-
        pll1(X, Y, Z, W, 0).

%       pll1/4 runs through all the possible combinations of the
%       four points to see if one has been recorded as parallel.
pll1(a, b, c, d, _).
pll1(a, d, b, c, _).
pll1(X, Y, Z, W, N):-
        (N == 0; N == 2; N == 4; N == 6),
        N1 is N + 1,
        pll1(Y, X, Z, W, N1).
pll1(X, Y, Z, W, N):-
        (N == 1; N == 3; N == 5; N == 7),
        N1 is N + 1,
        pll1(Z, W, X, Y, N1).


p2(X, Y, Z, W):-
        pll2(X, Y, Z, W, 0).
pll2(a, d, c, b, _).
pll2(X, Y, Z, W, N):-
        (N == 0; N == 2; N == 4; N == 6),
        N1 is N + 1,
        pll2(Y, X, Z, W, N1).
pll2(X, Y, Z, W, N):-
        (N == 1; N == 3; N == 5; N == 7),
        N1 is N + 1,
        pll2(Z, W, X, Y, N1).

rightangle(X, Y, Z):-
        val(a1(X, Y, Z), 90, 0).
rightangle(X, Y, Z):-
        val(a2(X, Y, Z), 90, 0).

%       corresponding/6 is, in fact, irrelevant to proving the
%       theorem, hence it fails all the time, to prevent time
%       being taken in trying to evaluate it.
corresponding(_, _, _, _, _, _):- fail, !.

trapezoid(a, b, c, d).
```

B.1.2 Output

SemCheck outputs the names of the resolvants as they are added to the search space. `f(N, Y)` represents factor N of input clause Y. `r(N, X, Y)` represents resolvant, N, of clauses, X and Y. In the case of clash resolution, X is a list of electrons and Y is the nucleus.

SemCheck searches for unprovability first. So if the theorem is provable it traverses the search space twice. The Lush resolution algorithm generates the empty clause twice. It then writes `provable` or `unprovable` and the cpu time in milliseconds taken by the proof.

To run the program 2 predicates can be called depending on the type of resolution being used, `lush/3` and `clash/3`. To produce a standard lush resolution proof of a theorem

```
|?- lush(TestSet, [], GoalClause).
```

was called. Where `TestSet` is the name of the set of clauses and `GoalClause` is the name of the goal clause to be used by the theorem prover.

Semantic lush resolution proofs were called by.

```
|?- lush(TestSet, ModelList, GoalClause).
```

Where `TestSet` and `GoalClause` are as above and `ModelList` is a list of the names of the models to be used in the proof[1].

Clash resolution proofs were called by

```
|?- clash(TestSet, ModelList, Numbering).
```

Where `TestSet` and `ModelList` are as defined for Lush theorem proving. `Numbering` is the name of the Numbering given to the predicates occuring in the test set.

Lush Resolution with No Models

```
r(1,prove_angles_equal,f(1,right_angles_are_equal))
r(1,prove_angles_equal,f(1,corresponding_angles_are_equal))
r(1,prove_angles_equal,f(1,interior_angles_are_equal))
r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                                    f(1,trapezoid_definition))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                 f(1,trapezoid_definition)),f(1,a_trapezoid))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                 f(1,trapezoid_definition)),f(1,a_trapezoid))
r(1,prove_angles_equal,f(1,right_angles_are_equal))
r(1,prove_angles_equal,f(1,corresponding_angles_are_equal))
r(1,prove_angles_equal,f(1,interior_angles_are_equal))
r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                                    f(1,trapezoid_definition))
```

[1] The program wrote `error` if an attempt to use models with a non-horn set of clauses was made.

```
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
provable
1270
\end{verbatim
\subsubsection{Lush Resolution Using Model 1}
\begin{verbatim}
r(1,prove_angles_equal,f(1,interior_angles_are_equal))
r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                                  f(1,trapezoid_definition))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
r(1,prove_angles_equal,f(1,interior_angles_are_equal))
r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                                  f(1,trapezoid_definition))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
provable
1090
```

Lush Resolution Using Model 2

```
r(1,prove_angles_equal,f(1,interior_angles_are_equal))
r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                                  f(1,trapezoid_definition))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
r(1,prove_angles_equal,f(1,interior_angles_are_equal))
r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                                  f(1,trapezoid_definition))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
empty(r(1,r(1,prove_angles_equal,f(1,interior_angles_are_equal)),
                                f(1,trapezoid_definition)),f(1,a_trapezoid))
provable
1060
```

Clash Resolution Using No Models

```
r(1,[f(1,prove_angles_equal)],f(1,right_angles_are_equal))
r(1,[f(1,prove_angles_equal)],f(1,corresponding_angles_are_equal))
r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid))
r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal))
empty([r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid))],
               r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal)))
empty([r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal))],
                          r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid)))
empty([r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal)),
                                f(1,a_trapezoid)],f(1,trapezoid_definition))
empty([r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid)),
                     f(1,prove_angles_equal)],f(1,interior_angles_are_equal))
empty([r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal)),
                                f(1,trapezoid_definition)],f(1,a_trapezoid))
empty([r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid)),
                     f(1,interior_angles_are_equal)],f(1,prove_angles_equal))
r(1,[f(1,prove_angles_equal)],f(1,right_angles_are_equal))
r(1,[f(1,prove_angles_equal)],f(1,corresponding_angles_are_equal))
r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid))
r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal))
empty([r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid))],
               r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal)))
empty([r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal))],
                          r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid)))
```

```
empty([r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal)),
                                f(1,a_trapezoid)],f(1,trapezoid_definition))
empty([r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid)),
                    f(1,prove_angles_equal)],f(1,interior_angles_are_equal))
empty([r(1,[f(1,interior_angles_are_equal)],f(1,prove_angles_equal)),
                                 f(1,trapezoid_definition)],f(1,a_trapezoid))
empty([r(1,[f(1,trapezoid_definition)],f(1,a_trapezoid)),
                     f(1,interior_angles_are_equal)],f(1,prove_angles_equal))
provable
16780
```

Clash Resolution Using Model 1

```
r(1,[f(1,prove_angles_equal)],f(1,interior_angles_are_equal))
r(1,[r(1,[f(1,prove_angles_equal)],f(1,interior_angles_are_equal))],
                                                  f(1,trapezoid_definition))
empty([r(1,[r(1,[f(1,prove_angles_equal)], f(1,interior_angles_are_equal))],
                              f(1,trapezoid_definition))], f(1,a_trapezoid))
r(1,[f(1,prove_angles_equal)],f(1,interior_angles_are_equal))
r(1,[r(1,[f(1,prove_angles_equal)],f(1,interior_angles_are_equal))],
                                                  f(1,trapezoid_definition))
empty([r(1,[r(1,[f(1,prove_angles_equal)], f(1,interior_angles_are_equal))],
                               f(1,trapezoid_definition))], f(1,a_trapezoid))
provable
9540
```

Clash Resolution Using Model 2

```
r(1,[f(1,prove_angles_equal)],f(1,interior_angles_are_equal))
r(1,[r(1,[f(1,prove_angles_equal)],f(1,interior_angles_are_equal))],
                                                  f(1,trapezoid_definition))
empty([r(1,[r(1,[f(1,prove_angles_equal)], f(1,interior_angles_are_equal))],
                               f(1,trapezoid_definition))], f(1,a_trapezoid))
r(1,[f(1,prove_angles_equal)],f(1,interior_angles_are_equal))
r(1,[r(1,[f(1,prove_angles_equal)],f(1,interior_angles_are_equal))],
                                           f(1,trapezoid_definition))
empty([r(1,[r(1,[f(1,prove_angles_equal)], f(1,interior_angles_are_equal))],
                               f(1,trapezoid_definition))], f(1,a_trapezoid))
provable
7580
```

B.2 Anti-Russell Sets

This is included as an illustration of a case where Semantic Clash resolution vastly reduced the search space. The file presented to SemCheck appears below, followed by the program output. It is a non-Horn set so Semantic Lush resolution wasn't used. It is problem SET044-5 in TPTP.

B.2.1 The Input Files

```
%--------------------------------------------------------------------------
% File     : SET044=ARussellSet-5 : TPTP v1.1.1. Bugfixed v1.0.0.
% Domain   : Set Theory
% Problem  : Anti-Russell Sets
% Version  :
% English  : If there were an anti-Russell set (a set that contains
```

```
%            exactly those sets that are members of themselves), then not
%            every set has a complement.

% Refs     : Pelletier F.J. (1986), Seventy-Five Problems for Testing
%            Automatic Theorem Provers, Journal of Automated Reasoning
%            2(2), 191-216.
%          : Pelletier F.J. (1988), Errata, Journal of Automated
%            Reasoning 4(2), 235-236.
% Source   : [Pelletier, 1986]
% Names    : Pelletier 40 [Pelletier, 1986]
%          : p40.in [ANL]

% Status   :
% Syntax   : Number of clauses          :    4 (   1 non-Horn)(   0 unit)
%            Number of literals         :    8 (   0 equality)
%            Number of predicate symbols :   1 (   0 propositional)
%            Number of function symbols  :   2 (   1 constant)
%            Number of variables        :    6 (   0 singleton)
%            Maximal clause size        :    2
%            Maximal term depth         :    2

% Comments : This problem is incorrect in [Pelletier, 1986] and is
%            corrected in [Pelletier, 1988].
%--------------------------------------------------------------------------
:- dynamic cls/3.

cls(clause_1,element(x,a),negative).
cls(clause_1,element(x,x),positive).
test_set(test,clause_1).

cls(clause_2,element(x,x),negative).
cls(clause_2,element(x,a),positive).
test_set(test,clause_2).

cls(clause_3,element(xx,f(x)),negative).
cls(clause_3,element(xx,x),negative).
test_set(test,clause_3).

cls(clause_4,element(xx,x),positive).
cls(clause_4,element(xx,f(x)),positive).
test_set(test,clause_4).

%-------------------------------------------------------------------
%       NUMB
%-------------------------------------------------------------------
numb(numb, element, 1).
numb(numb, f, 2).

%-------------------------------------------------------------------
%       MODEL
%-------------------------------------------------------------------

model_domain(model, a).
model_domain(model, b).

model_function(model, [X, X]).

e(a, b).
e(b, a).
e(b, b).
element(X, Y):-
        val(Y, Z),
        val(X, N),
        e(N, Z).

val(a, a).
val(b, b).
val(f(a), b).
val(f(b), a).
```

B.2.2 Lush Resolution

Only one repeat is shown.

```
lush(test,[],clause_3)
r(1,clause_3,f(1,clause_1))
r(2,clause_3,f(1,clause_1))
r(3,clause_3,f(1,clause_1))
r(1,clause_3,f(1,clause_2))
r(1,clause_3,f(1,clause_4))
r(2,clause_3,f(1,clause_4))
r(3,clause_3,f(1,clause_4))
r(4,clause_3,f(1,clause_4))
r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4))
r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4))
r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_2))
r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_4))
r(2,r(3,clause_3,f(1,clause_1)),f(1,clause_4))
r(3,r(3,clause_3,f(1,clause_1)),f(1,clause_4))
r(1,r(1,clause_3,f(1,clause_2)),f(1,clause_1))
r(1,r(1,clause_3,f(1,clause_2)),f(1,clause_4))
r(2,r(1,clause_3,f(1,clause_2)),f(1,clause_4))
r(2,r(2,clause_3,f(1,clause_4)),clause_3)
r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_2))
r(2,r(2,clause_3,f(1,clause_4)),f(1,clause_2))
r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_4))
r(1,r(3,clause_3,f(1,clause_4)),clause_3)
r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_1))
r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_2))
r(2,r(3,clause_3,f(1,clause_4)),f(1,clause_4))
r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3)
r(2,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3)
r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2))
r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4))
r(2,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4))
r(2,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3)
r(3,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3)
r(1,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2))
r(2,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2))
r(1,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4))
r(2,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4))
r(4,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4))
r(1,r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_2)),f(1,clause_4))
r(2,r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_2)),f(1,clause_4))
r(2,r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3)
r(1,r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2))
r(2,r(2,r(3,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3)
r(1,r(2,r(3,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2))
r(2,r(3,r(3,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3)
r(1,r(3,r(3,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2))
r(1,r(3,r(3,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4))
r(2,r(3,r(3,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4))
r(1,r(1,r(1,clause_3,f(1,clause_2)),f(1,clause_1)),f(1,clause_4))
r(1,r(1,r(1,clause_3,f(1,clause_2)),f(1,clause_4)),f(1,clause_1))
r(2,r(2,r(1,clause_3,f(1,clause_2)),f(1,clause_4)),clause_3)
r(2,r(2,r(2,clause_3,f(1,clause_4)),clause_3), r(2,clause_3,f(1,clause_4)))
r(1,r(2,r(2,clause_3,f(1,clause_4)),clause_3),f(1,clause_1))
r(2,r(2,r(2,clause_3,f(1,clause_4)),clause_3),f(1,clause_1))
r(1,r(2,r(2,clause_3,f(1,clause_4)),clause_3),f(1,clause_2))
r(2,r(2,r(2,clause_3,f(1,clause_4)),clause_3),f(1,clause_4))
r(3,r(2,r(2,clause_3,f(1,clause_4)),clause_3),f(1,clause_4))
r(4,r(2,r(2,clause_3,f(1,clause_4)),clause_3),f(1,clause_4))
r(1,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),r(2,clause_3,f(1,clause_4)))
r(2,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),r(2,clause_3,f(1,clause_4)))
r(1,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),clause_3)
r(1,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),f(1,clause_4))
r(2,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),f(1,clause_4))
r(1,r(2,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),r(2,clause_3,f(1,clause_4)))
r(2,r(2,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),r(2,clause_3,f(1,clause_4)))
r(1,r(2,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),clause_3)
r(3,r(2,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),clause_3)
r(1,r(2,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),f(1,clause_4))
r(2,r(2,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),f(1,clause_4))
```

```
r(3,r(2,r(2,clause_3,f(1,clause_4)),f(1,clause_2)),f(1,clause_4))
r(1,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_4)),r(2,clause_3,f(1,clause_4)))
r(1,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_4)),clause_3)
r(2,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_4)),clause_3)
r(3,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_4)),clause_3)
r(1,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_4)),f(1,clause_1))
r(1,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_4)),f(1,clause_2))
r(2,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_4)),f(1,clause_2))
r(3,r(1,r(2,clause_3,f(1,clause_4)),f(1,clause_4)),f(1,clause_2))
r(1,r(1,r(3,clause_3,f(1,clause_4)),clause_3),r(3,clause_3,f(1,clause_4)))
r(1,r(1,r(3,clause_3,f(1,clause_4)),clause_3),f(1,clause_1))
r(2,r(1,r(3,clause_3,f(1,clause_4)),clause_3),f(1,clause_1))
r(3,r(1,r(3,clause_3,f(1,clause_4)),clause_3),f(1,clause_1))
r(1,r(1,r(3,clause_3,f(1,clause_4)),clause_3),f(1,clause_2))
r(1,r(1,r(3,clause_3,f(1,clause_4)),clause_3),f(1,clause_4))
r(2,r(1,r(3,clause_3,f(1,clause_4)),clause_3),f(1,clause_4))
r(3,r(1,r(3,clause_3,f(1,clause_4)),clause_3),f(1,clause_4))
r(1,r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_1)),r(3,clause_3,f(1,clause_4)))
r(1,r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_1)),clause_3)
r(2,r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_1)),clause_3)
r(3,r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_1)),clause_3)
r(1,r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_1)),f(1,clause_4))
r(2,r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_1)),f(1,clause_4))
r(1,r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_2)),r(3,clause_3,f(1,clause_4)))
r(1,r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_2)),clause_3)
r(2,r(1,r(3,clause_3,f(1,clause_4)),f(1,clause_2)),f(1,clause_4))
r(2,r(2,r(3,clause_3,f(1,clause_4)),f(1,clause_4)),r(3,clause_3,f(1,clause_4)))
r(2,r(2,r(3,clause_3,f(1,clause_4)),f(1,clause_4)),clause_3)
r(3,r(2,r(3,clause_3,f(1,clause_4)),f(1,clause_4)),clause_3)
r(4,r(2,r(3,clause_3,f(1,clause_4)),f(1,clause_4)),clause_3)
r(1,r(2,r(3,clause_3,f(1,clause_4)),f(1,clause_4)),f(1,clause_1))
r(2,r(2,r(3,clause_3,f(1,clause_4)),f(1,clause_4)),f(1,clause_1))
r(1,r(2,r(3,clause_3,f(1,clause_4)),f(1,clause_4)),f(1,clause_2))
r(2,r(2,r(3,clause_3,f(1,clause_4)),f(1,clause_4)),f(1,clause_2))
r(1,r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_2))
r(2,r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_2))
r(1,r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_4))
r(3,r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_4))
r(1,r(2,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_2))
r(1,r(2,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_4))
r(1,r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2)),clause_3)
r(2,r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2)),
                                                                      f(1,clause_1))
r(2,r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2)),
                                                                      f(1,clause_4))
r(1,r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(2,r(1,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(2,r(2,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),
                                                      r(1,clause_3,f(1,clause_1)))
r(2,r(2,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(3,r(2,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(4,r(2,r(1,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(1,r(2,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_2))
r(1,r(2,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_4))
r(1,r(3,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_4))
r(2,r(3,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),clause_3),f(1,clause_4))
r(1,r(1,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2)),
                                                      r(1,clause_3,f(1,clause_1)))
r(2,r(1,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2)),
                                                      r(1,clause_3,f(1,clause_1)))
r(3,r(1,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2)),
                                                      r(1,clause_3,f(1,clause_1)))
r(4,r(1,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_2)),
                                                      r(1,clause_3,f(1,clause_1)))
r(1,r(1,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(2,r(1,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(1,r(2,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),
                                                      r(1,clause_3,f(1,clause_1)))
r(1,r(2,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(2,r(2,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(3,r(2,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(1,r(4,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(2,r(4,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),clause_3)
r(1,r(4,r(3,r(1,clause_3,f(1,clause_1)),f(1,clause_4)),f(1,clause_4)),
```

```
                                                              f(1,clause_2))
empty(r(1,r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_2)),f(1,clause_4)),
                                                 r(2,clause_3,f(1,clause_1)))
r(1,r(1,r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_2)),f(1,clause_4)),
                                                              f(1,clause_1))
empty(r(1,r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_2)),f(1,clause_4)),
                                                 r(2,clause_3,f(1,clause_1)))
r(2,r(1,r(1,r(2,clause_3,f(1,clause_1)),f(1,clause_2)),f(1,clause_4)),
                                                              f(1,clause_1))
provable
98250
```

B.2.3 Clash Resolution

SemCheck failed to find a refutation for this problems using Clash Resolution in under 24 hours.

B.3 Semantic Clash Resolution

```
clash(test,[model],numb)
r(1,[f(1,clause_3)],f(1,clause_1))
r(2,[f(1,clause_3)],f(1,clause_1))
r(1,[r(1,[f(1,clause_3)],f(1,clause_1))],f(1,clause_2))
empty([r(1,[r(1,[f(1,clause_3)],f(1,clause_1))],f(1,clause_2)),
                           r(1,[f(1,clause_3)],f(1,clause_1))],f(1,clause_4))
r(1,[f(1,clause_3)],f(1,clause_1))
r(2,[f(1,clause_3)],f(1,clause_1))
r(1,[r(1,[f(1,clause_3)],f(1,clause_1))],f(1,clause_2))
empty([r(1,[r(1,[f(1,clause_3)],f(1,clause_1))],f(1,clause_2)),
                           r(1,[f(1,clause_3)],f(1,clause_1))],f(1,clause_4))
provable
17330
```

Appendix C

TPTP Problems used during Testing with Results

Semantic Name		*Description*			
Syntactic Name	*No. Clauses*	*Resolution Method*	*Model*	*Size of Space Searched*	*Cpu Time*
WkFxdPtL		Weak Fixed Point for L			
COL007-1	7	lush	empty	3	1180
			model1	3	1200
			model2	3	1210
			models1+2	3	1210
		clash	empty	44	747520
			model1	3	8400
			model2	3	20760
			models1+2	3	8400
MBird06		Happy Birds			
COL055-1	7	lush	empty	63	13310
			model1	63	40740
			model2	63	42430
			model1+2	63	44340
		clash	empty	faile	failed
			model1	6	24200
			model2	6	42790
			model1+2	6	24510
AltAngleEq		The alternate interior angles in a trapezoid are equal			
GEO079-1	6	lush	empty	5	1270
			model1	3	1090
			model2	3	1060
			model1+2	3	1070
		clash	empty	10	16780
			model1	3	9540
			model2	3	7580
			model1+2	3	7980

Semantic Name		*Description*			
Syntactic Name	*No. Clauses*	*Resolution Method*	*Model*	*Size of Space Searched*	*Cpu Time*
SemiGRIdEx1		In semigroups, left and right solutions $\Rightarrow$ right id exists			
GRP028-1	4	lush	empty	185	6252250
			model	44	743420
		clash	empty	failed	failed
			model	6	319890
Pigeon		Cook pigeon-hole problem			
MSC007-1.002	3	lush	empty	3	190
			model	3	190
		clash	empty	3	380
			model	4	1130
VallsPrm		If a is a prime and $a = b^2/c^2$ then a divides b			
NUM014-1	7	lush	empty	57	2077110
		clash	empty	33	990070
			model	9	3252530
Russell		Russell's Paradox			
SET043-5	2	lush	empty	5	680
		clash	empty	6	1750
			model	1	1120
ARussellSet		Anti-Russell Sets			
SET044-5	4	lush	empty	136	98250
		clash	empty	failed	failed
		clash	model	4	17330
Allways		All signed combinations of some propositions			
SYN001-1.001	2	lush	empty	1	40
			model	1	40
		clash	empty	2	100
			model	1	60
SYN001-1.002	4	lush	empty	16	3410
		clash	empty	12	72290
		clash	model	3	4230
Even		Even Problems			
SYN002-1.000	2	lush	empty	60	18760
		clash	empty	5	2570
			model1	4	2300
			model2	4	2360
SYN002-1.002	2	lush	empty	186	91330
		clash	empty	12	19363750
			model1	20	13492070
			model2	20	13502940
Implies2		Implications that form a contradiction			
SYN004-1.002	5	lush	empty	52	11550
			model	52	12430
		clash	empty	15	52020
		clash	model	13	86280

Semantic Name		*Description*			
Syntactic Name	*No. Clauses*	*Resolution Method*	*Model*	*Size of Space Searched*	*Cpu Time*
Orl		disjunctions that form a contradiction			
SYN005-1.002	3	lush	empty	3	260
			model	3	260
		clash	empty	2	430
			model	2	890
SYN005-1.003	4	lush	empty	10	1040
			model	10	1080
		clash	empty	4	1050
			model	3	3420
SYN005-1.004	5	lush	empty	41	7590
			model	41	7950
		clash	empty	6	2430
			model	4	10810
SYN005-1.005	6	lush	empty	206	114390
			model	206	118970
		clash	empty	8	4710
		clash	model	5	26100
SYN005-1.006	7	lush	empty	1237	4555090
			model	1237	4807850
		clash	empty	10	9630
			model	6	60930
Splits		A problem to demonstrate controlling splits			
SYN006-1	7	lush	empty	22	12270
				22	12030
				22	10040
		clash	empty	20	55910
		clash	model1	11	162640
		clash	model2	13	185440
		clash	model3	14	351440
Relev1		A problem to demonstrate the usefulness of relevancy testing			
SYN008-1	6	lush	empty	6	920
		clash	empty	10	4200
		clash	model	5	5470
Letz		Letz problem			
SYN010-1.002	6	lush	empty	53	15500
			model	53	15810
		clash	empty	24	823080
			model	14	102450
EW1		EW1			
SYN028-1	6	lush	empty	63	17180
		clash	empty	25	398590
			model1	7	18240
			model2	5	12440
			model3	7	14780

Semantic Name		*Description*			
Syntactic Name	*No. Clauses*	*Resolution Method*	*Model*	*Size of Space Searched*	*Cpu Time*
EW2		EW2			
SYN029-1	5	lush	empty	55	15160
		clash	empty	13	770940
		clash	model	6	12350
MQW		MQW			
SYN031-1	5	lush	empty	219	171790
		clash	empty	22	49840
			model	8	29290
FPell01		Pelletier Problem 1			
SYN040-1	4	lush	empty	2	210
			model	2	210
		clash	empty	3	950
			model	2	900
FPell10		Pelletier Problem 10			
SYN044-1	6	lush	empty	1537	47829400
		clash	empty	failed	failed
			model	6	48260
FPell13		Pelletier Problem 13			
SYN045-1	4	lush	empty	7	640
				10	870
		clash	empty	12	2770
		clash	model1	3	1400
		clash	model2	6	8730
FPell15		Pelletier Problem 15			
SYN046-1	3	lush	empty	2	140
			model	2	130
		clash	empty	3	310
			model	2	540
FPell17		Pelletier Problem 17			
SYN047-1	5	lush	empty	31	4880
		clash	empty	22	65690
			model	6	14200
FPell18		Pelletier Problem 18			
SYN048-1	2	lush	empty	1	70
			model	1	60
		clash	empty	2	170
			model	1	120
FPell19		Pelletier Problem 19			
SYN049-1	3	lush	empty	2	190
			model	2	190
		clash	empty	2	430
			model	2	770
FPell20		Pelletier Problem 20			
SYN050-1	5	lush	empty	4	590
			model	4	610
		clash	empty	5	2720
			model	3	5720

Semantic Name		*Description*			
Syntactic Name	*No. Clauses*	*Resolution Method*	*Model*	*Size of Space Searched*	*Cpu Time*
FPell21		Pelletier Problem 21			
SYN051-1	4	lush	empty	48	11450
		clash	empty	8	4460
			model1	3	3370
			model2	3	3350
FPell22		Pelletier Problem 22			
SYN052-1	5	lush	empty	65	23180
		clash	empty	9	19220
			model	3	5070
FPell23		Pelletier Problem 23			
SYN053-1	5	lush	empty	36	25860
		clash	empty	25	3381710
			model	6	147330
FPell24		Pelletier Problem 24			
SYN054-1	6	lush	empty	failed	failed
		clash	empty		
			model	19	1183130
FPell31		Pelletier Problem 31			
SYN061-1	6	lush	empty	10	1770
		clash	empty	13	4280
			model	5	14590
FPell35		Pelletier Problem 35			
SYN064-1	2	lush	empty	1	80
			model	1	80
		clash	empty	2	230
			model	1	190
FPell37		Pelletier Problem 37			
SYN066-1	6	lush	empty	8	2710
		clash	empty	10	8450
			model	4	9480
FPell57		Pelletier Problem 57			
SYN079-1	4	lush	empty	5	3550
			model	5	3600
		clash	empty	9	9630
			model	7	32370
FPell59		Pelletier Problem 59			
SYN081-1	3	lush	empty	29	5180
		clash	empty	7	5890
			model	6	3380
BasisTplgLem2		Topology generated by a basis forms a topological space, part 2			
TOP002-2	3	lush	empty	2	240
		clash	empty	2	550
		clash	model1	2	1960
		clash	model2	2	750

Appendix D

Models and Orderings Used with the Various TPTP Problems

Weak fixed point for L

```
%--------------------------------------------------------------------------
% NUMBERING
%--------------------------------------------------------------------------
numb(numb, equal, 1).
numb(numb, apply, 2).

%--------------------------------------------------------------------------
% MODELS
%--------------------------------------------------------------------------

model_domain(1, l1).
model_domain(1, c1).

model_function(1, [apply, dot1]).
model_function(1, [equal, equal1]).
model_function(1, [l, l1]).
model_function(1, [combinator, c1]).

equal1(X, Y):-
val(X, N1),
val(Y, N2),
eq1(N1, N2).

eq1(l1, c1).
eq1(c1, l1).

val(X, X):-
atomic(X).
val(dot1(X, Z), Y):-
val(X, N1),
val(Z, N2),
dot1(N1, N2, Y).

dot1(X, c1, X).
dot1(c1, X, X).
dot1(l1, l1, l1).


model_domain(2, l1).
model_domain(2, c1).
model_domain(2, a2).

model_function(2, [apply, dot1]).
model_function(2, [equal, equal1]).
model_function(2, [l, l1]).
model_function(2, [combinator, c1]).

equal1(X, Y):-
val(X, N1),
val(Y, N2),
```

```
eq1(N1, N2).

eq1(X, X).
eq1(l1, c1).
eq1(c1, l1).

dot1(_, a2, a2).
dot1(a2, _, a2).
```

Happy Birds

```
%-----------------------------------------------------------------------
%       NUMB
%-----------------------------------------------------------------------

numb(numb, equal, 1).
numb(numb, response, 2).

%-----------------------------------------------------------------------
%       MODELS
%-----------------------------------------------------------------------
model_domain(_, a).
model_domain(_, b).
model_domain(2, c).

model_function(_, [X, X]).

equal(X, Y):-
        val(X, N1),
        val(Y, N1).

val(X, X):-
        atomic(X).
val(response(X, Y), Z):-
        val(X, N1),
        val(Y, N2),
        r(N1, N2, Z).
r(a, a, a).
r(a, b, b).
r(b, a, b).
r(b, b, a).
r(c, c, a).
r(c, a, b).
r(c, b, b).
r(a, c, c).
r(b, c, c).
```

The Alternate Interior Angles in a Trapezoid are Equal

```
%-----------------------------------------------------------------------
% NUMB
%-----------------------------------------------------------------------
numb(1, right_angle, 5).
numb(1, eq, 3).
numb(1, congruent, 4).
numb(1, trapezoid, 2).
numb(1, parallel, 1).

%-----------------------------------------------------------------------
% MODEL
%-----------------------------------------------------------------------
:-dynamic notfalse/2.

notfalse(right_angles_are_equal, _).
notfalse(corresponding_angles_are_equal, _).
notfalse(trapezoid_definition, _).
```

```
notfalse(interior_angles_are_equal, _).
notfalse(a_trapezoid, _).

model_function(_, [right_angle, rightangle]).
model_function(1, [eq, equal1]).
model_function(2, [eq, equal2]).
model_function(_, [congruent, corresponding]).
model_function(_, [trapezoid, trapezoid]).
model_function(1, [parallel, p1]).
model_function(2, [parallel, p2]).
model_function(_, [a, a]).
model_function(_, [b, b]).
model_function(_, [c, c]).
model_function(_, [d, d]).
model_domain(_, a).
model_domain(_, b).
model_domain(_, c).
model_domain(_, d).

equal1(A, B, C, D, E, F):-
val(a1(A, B, C), N1, 0),
val(a1(D, E, F), N2, 0),
N1 == N2.

equal2(A, B, C, D, E, F):-
val(a2(A, B, C), N1, 0),
val(a2(D, E, F), N2, 0),
N1 == N2.

val(a1(a, b, c), 90, _).
val(a1(b, c, d), 90, _).
val(a1(c, d, a), 90, _).
val(a1(d, a, b), 90, _).
val(a1(a, b, d), 45, _).
val(a1(c, b, d), 45, _).
val(a1(c, d, b), 45, _).
val(a1(a, d, b), 45, _).
val(a1(c, a, b), 45, _).
val(a1(c, a, d), 45, _).
val(a1(a, c, d), 45, _).
val(a1(a, c, b), 45, _).
val(a1(X, Y, Z), N, 0):-
val(a1(Z, Y, X), N, 1).

val(a2(a, b, c), 90, _).
val(a2(b, c, d), 135, _).
val(a2(c, d, a), 45, _).
val(a2(d, a, b), 90, _).
val(a2(a, b, d), 60, _).
val(a2(c, b, d), 30, _).
val(a2(c, d, b), 15, _).
val(a2(a, d, b), 30, _).
val(a2(c, a, b), 45, _).
val(a2(c, a, d), 45, _).
val(a2(a, c, d), 90, _).
val(a2(a, c, b), 45, _).
val(a2(X, Y, Z), N, 0):-
val(a2(Z, Y, X), N, 1).

p1(X, Y, Z, W):-
pll1(X, Y, Z, W, 0).

%       pll1/4 runs through all the possible combinations of the four points
%       to see if one has been recorded as parallel.
pll1(a, b, c, d, _).
pll1(a, d, b, c, _).
pll1(X, Y, Z, W, N):-
(N == 0; N == 2; N == 4; N == 6),
N1 is N + 1,
pll1(Y, X, Z, W, N1).
pll1(X, Y, Z, W, N):-
(N == 1; N == 3; N == 5; N == 7),
N1 is N + 1,
pll1(Z, W, X, Y, N1).
```

```
p2(X, Y, Z, W):-
pll2(X, Y, Z, W, 0).
pll2(a, d, c, b, _).
pll2(X, Y, Z, W, N):-
(N == 0; N == 2; N == 4; N == 6),
N1 is N + 1,
pll2(Y, X, Z, W, N1).
pll2(X, Y, Z, W, N):-
(N == 1; N == 3; N == 5; N == 7),
N1 is N + 1,
pll2(Z, W, X, Y, N1).

rightangle(X, Y, Z):-
val(a1(X, Y, Z), 90, 0).

rightangle(X, Y, Z):-
val(a2(X, Y, Z), 90, 0).

corresponding(_, _, _, _, _, _).

trapezoid(a, b, c, d).
```

In semigroups, left and right solutions ⇒ right id exists

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, product, 1).
numb(numb, left_solution, 2).
numb(numb, right_solution, 3).
numb(numb, not_identity, 4).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(1, 0).
model_domain(1, 1).

model_function(1, [product, plus1]).
model_function(1, [left_solution, diff]).
model_function(1, [right_solution, diff]).
model_function(1, [not_identity, other]).

plus1(X, Y, Z):-
val(X, N1),
val(Y, N2),
val(Z, N3),
N4 is N1 + N2,
N3 is N4 mod 2.

val(X, X):-
integer(X).
val(other(X), 1):-
val(X, 0).
val(other(X), 0):-
val(X, 1).
val(diff(X, Y), Z):-
val(X, N1),
val(Y, N2),
N3 is N2 - N1,
N4 is 2 + N3,
Z is N4 mod 2.
```

Cook pigeion-hole problem

```
%------------------------------------------------------------------------
%  MODEL
%-------------------------------------------------------------------------

model_function(model, [pigeon_1_in_hole_1, pigeon_1_in_hole_1]).
model_function(model, [pigeon_2_in_hole_1, pigeon_2_in_hole_1]).

pigeon_1_in_hole_1.
pigeon_2_in_hole_1.
```

If a is a prime and $a = b^2/c^2$ then a divides b

```
%------------------------------------------------------------------------
% NUMBERING
%------------------------------------------------------------------------
numb(numb, product, 1).
numb(numb, square, 2).
numb(numb, divides, 3).
numb(numb, prime, 4).

%------------------------------------------------------------------------
% MODELS
%------------------------------------------------------------------------
model_domain(model, a).
model_domain(model, b).
model_domain(model, c).
model_domain(model, square(a)).
model_domain(model, square(b)).
model_domain(model, square(c)).

model_function(model, [X, X]).

product(X, X, square(X)).
divides(X, square(X)).
divides(X, X).
product(a, square(c), square(b)).
product(square(c), a, square(b)).
divides(a, b).
prime(a).
divides(a, square(b)).
divides(square(c), square(b)).
```

Russell's Paradox

```
%------------------------------------------------------------------------
% NUMBERING
%------------------------------------------------------------------------
numb(numb, element, 1).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_function(model, [element, element]).
model_function(model, [a, a]).

model_domain(model, a).

element(a, a).
```

Anti-Russell Sets

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, element, 1).
numb(numb, f, 2).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(model, a).
model_domain(model, b).

model_function(model, [X, X]).

e(a, b).
e(b, a).
e(b, b).
element(X, Y):-
val(Y, Z),
val(X, N),
e(N, Z).

val(a, a).
val(b, b).
val(f(a), b).
val(f(b), a).
```

All signed combinations of some propositions

SYN001-1.001

```
%--------------------------------------------------------------------------
% NUMBERING
% none required.
%--------------------------------------------------------------------------
% MODEL
%--------------------------------------------------------------------------

model_function(model, [p, p]).
model_domain(model, p).

p.
```

SYN001-1.002

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, p1, 0).
numb(numb, p2, 0).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------
model_function(model, [X, X]).
model_domain(model, p1).
model_domain(model, p2).

p1.

p2.
```

Even Problem

SYN002-1.000

```
%------------------------------------------------------------------------
% NUMBERING
%------------------------------------------------------------------------

numb(numb, p, 1).
numb(numb, f, 2).

%------------------------------------------------------------------------
% MODELS
%------------------------------------------------------------------------

model_function(_, [p, even]).
model_function(1, [f, plus_one1]).
model_function(2, [f, plus_one2]).

model_domain(1, 0).
model_domain(1, 1).
model_domain(2, 0).
model_domain(2, 1).
model_domain(2, 2).
model_domain(2, 3).

even(X):-
val(X, Y),
 0 is Y mod 2.

val(X, X):-
integer(X).
val(plus_one1(X), N):-
val(X, Y),
\+ Y == 1,
N is Y + 1.
val(plus_one2(X), N):-
val(X, Y),
Y == 3,
N is 0.
val(plus_one2(X), N):-
val(X, Y),
\+ Y == 3,
N is Y + 1.
val(plus_one1(X), N):-
val(X, Y),
Y == 1,
N is 0.
```

SYN002-1.002

```
%------------------------------------------------------------------------
% NUMBERING
%------------------------------------------------------------------------

numb(numb, p, 1).
numb(numb, f, 2).

%------------------------------------------------------------------------
% MODELS
%------------------------------------------------------------------------

model_function(_, [p, even]).
model_function(1, [f, plus_one1]).
model_function(2, [f, plus_one2]).

model_domain(1, 0).
model_domain(1, 1).
```

```
model_domain(2, 0).
model_domain(2, 1).
model_domain(2, 2).
model_domain(2, 3).

even(X):-
val(X, Y),
 0 is Y mod 2.

val(X, X):-
integer(X).
val(plus_one1(X), N):-
val(X, Y),
\+ Y == 1,
N is Y + 1.
val(plus_one2(X), N):-
val(X, Y),
Y == 3,
N is 0.
val(plus_one2(X), N):-
val(X, Y),
\+ Y == 3,
N is Y + 1.
val(plus_one1(X), N):-
val(X, Y),
Y == 1,
N is 0.
```

Implications that form a contradiction

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, p1, 0).
numb(numb, p2, 0).
numb(numb, q1, 0).
numb(numb, q2, 0).

%------------------------------------------------------------------------
% MODEL(S)
%------------------------------------------------------------------------

model_domain(model, p1).

model_function(model, [X, X]).

p1.
p2.
q1.
q2.
```

Disjunctions that form a contradiction

SYN005-1.002

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, p1, 1).
numb(numb, p2, 2).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(1, a).
model_function(1, [p1, p1]).
```

```
model_function(1, [p2, p2]).
model_function(1, [a, a]).

p1(a, a).
p2(a, a).
```

SYN005-1.003

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------

numb(numb, p1, 1).
numb(numb, p2, 2).
numb(numb, p3, 3).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(model, a).
model_function(model, [X, X]).

p1(a, a).
p2(a, a).
p3(a, a).
```

SYN005-1.004

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, p1, 1).
numb(numb, p2, 2).
numb(numb, p3, 3).
numb(numb, p4, 4).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(model, a).

model_function(model, [X, X]).

p1(a, a).
p2(a, a).
p3(a, a).
p4(a, a).
```

SYN005-1.005

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, p1, 1).
numb(numb, p2, 2).
numb(numb, p3, 3).
numb(numb, p4, 4).
numb(numb, p5, 5).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------
model_domain(model, a).

model_function(model, [X, X]).
```

```
p1(a, a).
p2(a, a).
p3(a, a).
p4(a, a).
p5(a, a).
```

SYN005-1.006

```
%--------------------------------------------------------------------------
% NUMB
%--------------------------------------------------------------------------
numb(numb, p1, 1).
numb(numb, p2, 2).
numb(numb, p3, 3).
numb(numb, p4, 4).
numb(numb, p5, 5).
numb(numb, p6, 6).

%--------------------------------------------------------------------------
% MODEL
%--------------------------------------------------------------------------

model_domain(model, a).

model_function(model, [X, X]).

p1(a, a).
p2(a, a).
p3(a, a).
p4(a, a).
p5(a, a).
p6(a, a).
```

A problem to demonstrate controlling splits

```
%--------------------------------------------------------------------------
% NUMB
%--------------------------------------------------------------------------
numb(numb, p1, 1).
numb(numb, f, 2).
numb(numb, q1, 3).
numb(numb, q2, 4).
numb(numb, q3, 5).
numb(numb, q4, 6).
numb(numb, g, 7).
numb(numb, p2, 8).
%--------------------------------------------------------------------------
% MODEL
%--------------------------------------------------------------------------
model_domain(_, a).

model_function(_, [a, a]).
model_function(_, [p1, p1]).
model_function(_, [p2, p2]).
model_function(1, [q1, q11]).
model_function(2, [q1, q1]).
model_function(3, [q1, q1]).
model_function(1, [q2, q21]).
model_function(2, [q2, q2]).
model_function(3, [q2, q2]).
model_function(1, [q3, q3]).
model_function(2, [q3, q32]).
model_function(3, [q3, q3]).
model_function(1, [q4, q4]).
model_function(2, [q4, q4]).
model_function(3, [q4, q43]).
model_function(_, [f, f]).
model_function(_, [g, g]).
```

```
p1(a, a).
p1(X, Y):-
val(X, N1),
val(Y, N2),
p1(N1, N2).
q21(a, a).
q21(X, Y):-
val(X, N1),
val(Y, N2),
q21(N1, N2).
q2(_, _):- !, fail.
q32(a, a).
q32(X, Y):-
val(X, N1),
val(Y, N2),
q32(N1, N2).
q3(_, _):-!, fail.
q43(a, a).
q43(X, Y):-
val(X, N1),
val(Y, N2),
q43(N1, N2).
q4(a, a):- !, fail.

q1(a, a).
q1(X, Y):-
val(X, N1),
val(Y, N2),
q1(N1, N2).
q11(a, a):- !, fail.
p2(a, a).
p2(X, Y):-
val(X, N1),
val(Y, N2),
p2(N1, N2).
val(X, X):-
atomic(X).
val(f(X), Y):-
val(X, Y).
val(g(X), Y):-
val(X, Y).
```

A problem to demonstrate the usefulness of relevancy testing

```
%------------------------------------------------------------------------
% NUMBERING
%------------------------------------------------------------------------
numb(numb, _, 0).

%------------------------------------------------------------------------
% MODELS
%------------------------------------------------------------------------
model_domain(model, q).

model_function(model, [X, X]).

r.

p.
q:-fail.
a.
c.
d.
b.
```

Letz problem

```
%-----------------------------------------------------------------------
% NUMBERING
%-----------------------------------------------------------------------
numb(numb, _, 0).

%-----------------------------------------------------------------------
% MODEL
%-----------------------------------------------------------------------

model_domain(model, p).

model_function(model, [X, X]).

p_2_2.
p_2_1.
p_1_2.
p_1_1.
p_0.
```

EW1

```
%-----------------------------------------------------------------------
% NUMBERING
%-----------------------------------------------------------------------
numb(numb, _, 0).

%-----------------------------------------------------------------------
% MODEL
%-----------------------------------------------------------------------

model_domain(_, p).
model_function(1, [q, q1]).
model_function(2, [q, q2]).
model_function(3, [q, q3]).
model_function(1, [r, r1]).
model_function(2, [r, r2]).
model_function(3, [r, r3]).
model_function(1, [s, s1]).
model_function(2, [s, s2]).
model_function(3, [s, s3]).
model_function(1, [t, t1]).
model_function(2, [t, t2]).
model_function(3, [t, t3]).
model_function(_, [p, p]).

q1.
r1.
s1.
t1.

q2:-fail.
r2:-fail.
s2:-fail.
t2:-fail.

q3:-fail.
r3:-fail.
s3.
t3:-fail.

p.
```

EW2

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, p, 0).
numb(numb, q, 0).
numb(numb, r, 0).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(model, p).

model_function(model, [X, X]).

p.
q.
r.
```

MQW

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, g, 1).
numb(numb, f, 2).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(model, a).

model_function(model, [X, X]).

g(a, a).
g(a, f(a)).
g(f(a), a).
```

Pelletier Problem 1

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, p, 0).
numb(numb, q, 0).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(model, p).
model_domain(model, q).

model_function(model, [X, X]).

p.
q.
```

Pelletier Problem 10

```
%------------------------------------------------------------------------
% NUMBERING
%------------------------------------------------------------------------
numb(numb, _, 0).
%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------
model_domain(model, p).

model_function(model, [X, X]).

p.
q.
r.
```

Pelletier Problem 13

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, p, 0).
numb(numb, q, 0).
numb(numb, r, 0).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------
model_domain(_, p).
model_domain(_, q).
model_domain(_, r).

model_function(1, [p, p3]).
model_function(1, [q, q3]).
model_function(1, [r, r3]).

model_function(2, [p, p4]).
model_function(2, [q, q4]).
model_function(2, [r, r4]).

p3.

q3:-fail.
r3:-fail.
p4:-fail.
q4.

r4.
```

Pelletier Problem 15

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, p, 0).
numb(numb, q, 0).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------
model_domain(model, p).
model_domain(model, q).

model_function(model, [X, X]).
```

```
p.
q.
```

Pelletier Problem 17

```
%-------------------------------------------------------------------------
% NUMB
%-------------------------------------------------------------------------
numb(numb, _, 0).

%-------------------------------------------------------------------------
% MODEL
%-------------------------------------------------------------------------

model_domain(model, p).

model_function(model, [X, X]).

p.
q.
r.
s.
```

Pelletier Problem 18

```
%-------------------------------------------------------------------------
%        NUMBERING
%-------------------------------------------------------------------------

numb(numb, big_f, 1).
numb(numb, f, 2).

%-------------------------------------------------------------------------
%        MODELS
%-------------------------------------------------------------------------

model_domain(1, 0).
model_domain(1, 1).

model_function(1, [big_f, integer1]).
model_function(1, [f, plus_one]).

integer1(X):-
val(X, Y),
integer(Y).

val(X, X):-
integer(X).

val(plus_one(X), N):-
val(X, Y),
\+ Y == 1,
N is Y + 1.

val(plus_one(X), N):-
val(X, Y),
Y == 1,
N is 0.
```

Pelletier Problem 19

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------

numb(numb, big_p, 1).
numb(numb, big_q, 2).
numb(numb, f, 3).
numb(numb, g, 4).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(1, a).

model_function(_, [big_p, p]).
model_function(_, [big_q, q]).
model_function(1, [f, no_change]).
model_function(1, [g, no_change]).

p(a).
q(a).

p(X):-
val(X, Y),
p(Y).
q(X):-
val(X, Y),
q(Y).

val(a, a).
val(no_change(X), Y):-
val(X, Y).
```

Pelletier Problem 20

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, big_p, 1).
numb(numb, big_q, 2).
numb(numb, big_r, 3).
numb(numb, f, 4).
numb(numb, big_s, 5).
%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------
model_domain(model, a).
model_domain(model, b).

model_function(model, [X, X]).

big_p(a).
big_q(b).
r(b).
big_r(X):-
val(X, Y),
r(Y).
big_s(a).

val(a, a).
val(b, b).
val(f(_, X), X).
```

Pelletier Problem 21

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------

numb(numb, p, 0).
numb(numb, big_f, 1).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(_, a).
model_domain(_, b).

model_function(_, [a, a]).
model_function(_, [b, b]).
model_function(1, [big_f, f]).
model_function(_, [p, p]).
model_function(2, [big_f, g]).

p.

f(a).
g(b).
g(a).
```

Pelletier Problem 22

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, big_f, 1).
%------------------------------------------------------------------------
% MODEL(S)
%------------------------------------------------------------------------
model_domain(model, a).

model_function(model, [X, X]).

big_f(a).
p.
```

Pelletier Problem 23

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, fig_f, 1).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------
model_domain(model, a).
model_domain(model, b).

model_function(model, [X, X]).

p:-fail.

big_f(a).
big_f(b).
```

Pelletier Problem 31

```
%------------------------------------------------------------------------
% NUMBERING
%------------------------------------------------------------------------
numb(numb, big_f, 1).
numb(numb, big_g, 5).
numb(numb, big_h, 2).
numb(numb, big_i, 3).
nnumb(numb, big_j, 4).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------

model_domain(model, a).
model_function(model, [X, X]).

big_i(a).
big_f(a).
big_j(a).

big_g(a):-fail.
big_h(a):-fail.
```

Pelletier Problem 35

```
%------------------------------------------------------------------------
% NUMBERING
%------------------------------------------------------------------------
numb(numb, big_p, 1).
numb(numb, f, 2).
numb(numb, g, 3).

%------------------------------------------------------------------------
% MODELS
%------------------------------------------------------------------------

model_domain(1, 0).
model_domain(1, 1).

model_function(1, [big_p, both_integers]).
model_function(1, [f, +]).
model_function(1, [g, -]).

both_integers(X, Y):-
N is X mod 2,
M is Y mod 2,
integer(M),
integer(N).
```

Pelletier Problem 37

```
%-------------------------------------------------------------------------
% NUMBERING
%-------------------------------------------------------------------------
numb(numb, big_p, 1).
numb(numb, big_q, 2).
numb(numb, big_r, 3).
numb(numb, f, 4).
numb(numb, g, 5).
numb(numb, h, 6).
numb(numb, i, 7).

%-------------------------------------------------------------------------
% MODEL
%-------------------------------------------------------------------------
model_domain(model, a).

model_function(model, [X, X]).

r(a, a).
p(a, a).
q(a, a).

val(X, X):- atomic(X).
val(f(_, Y), Y).
val(g(Y), Y).
val(h(_, Y), Y).
val(i(_, Y), Y).

big_r(X, Y):-
val(X, N1),
val(Y, N2),
r(N1, N2).

big_p(X, Y):-
val(X, N1),
val(Y, N2),
p(N1, N2).

big_q(X, Y):-
val(X, N1),
val(Y, N2),
q(N1, N2).
```

Pelletier Problem 57

```
%-------------------------------------------------------------------------
% NUMB
%-------------------------------------------------------------------------
numb(numb, big_f, 1).
numb(numb, f, 2).

%-------------------------------------------------------------------------
% MODEL
%-------------------------------------------------------------------------

model_domain(model, a).
model_domain(model, b).
model_domain(model, c).

model_function(model, [X, X]).

big_f(X, X).
big_f(b, c).
big_f(X, Y):-
val(X, N),
val(Y, N1),
```

```
\+ (N == X, N1 == Y),
big_f(N, N1).

val(f(_, X), X).
val(X, X):-
atomic(X).
```

Pelletier Problem 59

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------
numb(numb, big_f, 1).
numb(numb, f, 2).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------
model_domain(model, 0).
model_domain(model, 1).

model_function(model, [big_f, even]).
model_function(model, [f, plus_one]).

even(X):-
val(X, Y),
Y == 0.

val(0, 0).
val(1, 1).
val(plus_one(X), Y):-
val(X, N),
N1 is N + 1,
Y is N1 mod 2.
```

Topology generated by a basis forms a topological space, part 2

```
%------------------------------------------------------------------------
% NUMB
%------------------------------------------------------------------------

numb(numb, element_of_collection, 1).
numb(numb, element_of_set, 2).
numb(numb, top_of_basis, 3).
numb(numb, f11, 4).

%------------------------------------------------------------------------
% MODEL
%------------------------------------------------------------------------
:- dynamic notfalse/2.

model_domain(_, []).
model_domain(_, [(0, 1), (1, 0)]).

model_function(_, [empty_set, []]).
model_function(_, [f, [(0, 1), (1, 0)]]).
model_function(_, [X, X]):-
\+ X == empty_set,
\+ X == f.

notfalse(topology_generated_40, modela).
notfalse(set_theory_6, modela).

element_of_collection(X, Y):-
val(X, N),
set(Y, M),
```

```
member(N, M).
element_of_set(X, Y):-
val(X, N),
set(Y, M),
member(N, M).

basis([(0, 1), (1, 0)]).
set([], []).
set([(0,1), (1, 0)], [(0, 1), (1, 0)]).
set(top_of_basis(X), Y):-
basis(X),
setof((A, B), (member((M, N), X), member((K, L), X), plus(M,
K, A), plus(N, L, B)), F),
append(F, X, Set),
setof(S, subset(S, Set), Y).

val([], []).
val([(0,1), (1, 0)], [(0, 1), (1, 0)]).
val(f11(_, X), X).

plus(X, Y, Z):-
N is X + Y,
Z is N mod 2.

subset([], []).
subset([H|T], D):-
member(H, D),
memberout(H, D, D1),
subset(T, D1).
```