Contents

Chapter 1

Introduction

This is a dissertation on evaluating the utility of a set of reasoning heuristics that have been developed to aid an automated reasoner in reasoning about the properties of formal specifications. The focus is on set-theoretic problems and first-order logic resolution-based automated theorem provers. The motivation is presented below.

1.1 Motivation

Mathematical set theory is a building block of a number of formal specification languages, e.g. both Z (Spivey 1992) and B (Abriel 1996) are based on strongly-typed fragments of Zermelo-Fraenkel (Enderton 1977) set theory. One of the advantages in using a formal notation for specifying a system is that the specifier may reason formally about the properties of the system. In particular one may want to prove that the proposed system will behave in a certain way or that some unwanted behaviour will not occur. However, writing out such proofs is a tedious task as may be observed in (Potter *et al*. 1996). Hence of particular interest to a specifier could be the feasibility of using an automated reasoning program (Riazanov & Voronkov 2002, Wos 2006) to reason about such properties.

Set-theoretic problems, however present difficult problems to automated reasoners (Boyer *et al*. 1986, Quaife 1992a, Wos 1988, Wos 1989). Much of the complexity arises from the fact that sets may be elements of other sets. Set-theoretic constructs are strongly hierarchical and could lead to deeply nested constructs that greatly increase a problem's search complexity (Quaife 1992a, Van der Poll & Labuschagne 1999). For example, in the following equality

$$\mathbb{P}(A) = \mathbb{P}(B) \leftrightarrow A = B$$

a reasoner has to move from the level of elements in set A to the level of elements in $\mathbb{P}(A)$ in its search for a proof, but should be prevented from transcending to the level of $\mathbb{P}(\mathbb{P}(A))$ which would greatly and unnecessarily enlarge the search space. This reasoning heuristic tends to come naturally to humans. However, for the automated reasoner to preserve completeness it should still traverse these possibly unlikely search paths when the other paths fail.

It is generally accepted that heuristics are needed to guide reasoners, especially in the context of set-theoretic proofs (Bundy 1999). Van der Poll and Labuschagne developed such a set of heuristics for reasoning about set theory (Van der Poll 2000, Van der Poll & Labuschagne 1999), mainly through observing the behaviour of the resolution-based reasoner, Otter (McCune 2003) in its search for proofs. In total 14 heuristics, based on recognisable patterns, were developed.

1.2 Research Question

The CADE ATP System Competitions (CASC) (Pelletier *et al*. 2002, Sutcliffe & Suttner 2006) is an annual competition that evaluates the performance of automated theorem provers using classical first-order logic. Otter no longer features as a worthy opponent in this competition, since it has to a large extent been superseded by next generation theorem provers e.g. Vampire (Riazanov & Voronkov 2002) and Gandalf (Tammet 1997). Otter is however still used as a relative benchmark for other provers.

The question therefore arises whether the heuristics developed by Van der Poll and Labuschagne (Van der Poll & Labuschagne 1999, Van der Poll 2000) have a wider applicability to other resolution-based reasoners that can be considered state of the art. The research reported on in this dissertation addresses the above research question and leads to the following hypothesis.

1.3 Hypothesis

The Van der Poll-Labuschagne heuristics developed for reasoning with set theory are also applicable to later, state of the art resolution-based automated theorem provers.

For the remainder of this dissertation we shall refer to the Van der Poll-Labuschagne heuristics as the VdPL set of heuristics.

1.4 Approach

To verify the hypothesis we select two theorem provers that can be considered state of the art when using the CASC (Pelletier *et al*. 2002, Sutcliffe & Suttner 2006) competition as a benchmark. The chosen theorem provers should be resolution-based to make the comparison with Otter more direct. Since the VdPL heuristics were developed on set-theoretic problems the chosen provers must also perform generally well with set-theoretic problems to ensure that the heuristics are indeed applicable and useful.

Each heuristic is then tested in turn on a sample set-theoretic problem. Otter is used to discharge the proof. After a failed proof attempt, the relevant VdPL heuristic is applied to the problem specification that enables Otter to find a proof. The original problem is then discharged on the chosen theorem provers. The heuristic is similarly applied to failed proof attempts. If the heuristic is found not to be applicable using the next generation theorem prover, we increase the complexity of the problem, and attempt again.

The use of automated reasoning in formal specification languages was mentioned as one of the motivations for research in reasoning heuristics. The heuristics are further tested on a case study specified in Z (Spivey 1992) and using one of the chosen reasoners to discharge proof obligations that arise.

1.5 Dissertation Layout

Chapter 2 gives an overview of set theory. The Zermelo-Fraenkel axiomatisation of set theory in first-order logic is presented. The use of set theory in formal specification languages is then highlighted followed by the typical issues that arise when reasoning about set-theoretic problems.

An overview of resolution-based theorem proving is presented in Chapter 3. The decision problem and Herbrand's universe is discussed to highlight the theoretical limits

of automated theorem proving. Resolution is presented as a refutation proof procedure followed by a discussion on efficiency enhancements to resolution theorem proving.

The resolution-based automated reasoners Vampire and Gandalf used in this work are presented in Chapter 4 including the motivation for their selection.

The utility of the VdPL heuristics for Vampire and Gandalf is investigated in Chapter 5. For each heuristic a sample problem is presented. The problem is first attempted using Otter. From a failed proof attempt the heuristic is applied to enable a successful refutation. The same problem in then applied to Vampire and Gandalf. In some cases the problem complexity must be increased to illustrate the utility of the heuristic. Some of these results were published in Steyn and Van der Poll (2007).

An order management system case study is presented in Chapter 6 using the Z specification language. Typical proof obligations that arise from Z specifications are presented and discussed.

In Chapter 7 a sample of the proof obligations from the case study is converted to first-order logic and discharged using Vampire. Various heuristics are then applied to some failed proof attempts to facilitate a successful refutation.

Chapter 8 summarises the conclusions to be drawn from the research reported on in this dissertation and indicates directions for further research.

Chapter 2

Introduction to Mathematical Set Theory

Set theory is a foundational theory of mathematics in the sense that many mathematical theorems, including arithmetic and Euclid's geometry, can be formulated as theorems of set theory (Nerode & Shore 1997). The problem of finding the truth of a mathematical statement can therefore be reduced to a problem of showing that its truth can be derived from the axioms of set theory (Enderton 1977).

In this chapter we give an overview of the Zermelo-Fraenkel (ZF) axiomatisation of set theory that allows for the first-order logic representation of set-theoretic problems. In the next section we discuss the use of set theory in formal specification languages. This is followed by a discussion of the limits of the ZF axioms in automated theorem proving due to its infinite axiomatisation. The chapter is concluded with the challenges that are posed by automated set-theoretic reasoning as well as a summary.

2.1 Zermelo-Fraenkel Set Theory

The concept of a set has been used in mathematic writings since ancient times (Enderton 1977). George Cantor's work at the end of the 19th century put set theory on a proper mathematical basis with a series of papers published during the period from 1874 to 1897. He is generally regarded as the father of set theory (Enderton 1977).

This early set theory originated in a non-axiomatic form that relied on an informal understanding of sets as collections of objects. By the turn of the nineteenth century a number of paradoxes were discovered in set theory. One of these is Russell's paradox that was discovered in 1901 by Bertrand Russell (Enderton 1977, Potter *et al*. 1996). He showed that Gottlob Frege's treatment of set theory was contradictory. Frege published a two-volume work in 1893 and 1903 in which he showed how mathematics could be

developed from principles of set theory. Russell's paradox stems from a well defined set in Frege set theory:

$$A = \{x \mid x \notin x\}$$

That is, x is an element of A if and only if x is not an element of itself. The question that arises is whether or not A contains itself. If it does, then by definition it is not a member of A and thus a contradiction. On the other hand if it does not contain itself, then by definition it is a member of A which is also a contradiction.

The paradoxes found in set theory led to the development of axiomatic set theory. This showed that certain assumptions were inconsistent and hence totally flawed. The non-axiomatic approach to set theory is now often referred to as "naive set theory" (Quine 1971).

Ernst Zermelo proposed the first system of axioms for set theory in 1908. The paradoxes that have plagued set theory could not occur under Zermelo's system since the sets required by the paradoxes cannot be constructed using his axioms. However it was discovered that rather simple sets could not be proved to exist based solely on these axioms. Abraham Fraenkel and others proposed the axiom of replacement, discussed below, to enable the creation of such sets (Enderton 1977). This list of set theory axioms, 10 in total, became known as the Zermelo-Fraenkel axioms.

Next we present a brief introduction to the ZF axioms. It is important to note that every object it deals with is a set. Every element of a set is itself a set. Therefore, all mathematical objects must therefore be defined as sets. As an example the non-negative integers (natural numbers) can be represented in set theory as the set of all smaller natural numbers:

$$0 = \varnothing,\ 1 = \{0\} = \{\varnothing\},\ 2 = \{0, 1\} = \{\varnothing, \{\varnothing\}\},\ \ldots.$$

This specific method of encoding the natural numbers was proposed by von Neumann in 1923 (Enderton 1977).

2.1.1 Extensionality Axiom

The action of Extensionality states the condition under which two sets are equal. Two sets are the same if they have the same elements. A set is therefore determined by its elements.

$$\forall A \forall B(\forall x(x \in A \leftrightarrow x \in B) \rightarrow A = B)$$

Note, this axiom only state when two sets are equal, it does not guarantee the existence of any sets. Also, note that equality reasoning in first-order logic requires the axioms presented in section 3.6.1.

2.1.2 Empty set Axiom

There exists a set having no elements called the empty set. The empty set is usually denoted by the symbol $\varnothing$.

$$\exists \varnothing \forall x\ (x \notin \varnothing)$$

The empty set axiom asserts that there exists at least one set, the empty set $\varnothing$. From the axiom of Extensionality it follows that there is only one such set.

2.1.3 Pairing Axiom

If u and v are sets, then there exists a set B containing u and v as its only elements. This set is called the unordered pair of u and v and is denoted by {u, v}.

$$\forall u \forall v \exists B \forall x(x \in B \leftrightarrow x = u \vee x = v)$$

It follows from the axiom of Extensionality that this set is uniquely determined and since the elements in a set are unordered we have {u,v} = {v,u}. Pairing implies the existence of sets containing only one element called singleton sets. For example, given any set v, the singleton set {v} exists and is equal to the unordered pair {v, v}. Repeated application of this axiom asserts the existence of sets of the form {{x}, {x,y}}, which is a standard way of representing the ordered pair (x, y).

2.1.4 Union Axiom

Normally the union axiom is first stated in simpler terms for just two sets (Enderton 1977) and thereafter it is given for the general case. Every set has a union. That is, for any set A there exists a set B whose elements are exactly the elements of the elements of A. For example if A = {a, b, c, d}, then B = ∪{a, b, c, d} = a ∪ b ∪ c ∪ d.

$$\forall A \exists B \forall x (x \in B \leftrightarrow \exists b (b \in A \wedge x \in b))$$

Finite sets like {a, b, c} can be constructed using this axiom and the pairing axiom above. For example, given any a, b and c we can construct sets {a} and {b, c} using the pairing axiom. Set {a, b, c} can then be constructed using the union axiom, that is, {a} ∪ {b, c}.

2.1.5 Subset Axiom

For each formula $\varphi(c, t_1,\ldots, t_n)$ not containing B, the following is an axiom (Enderton 1977):

$$\forall t_1 \ldots \forall t_n \forall c \exists B \forall x (x \in B \leftrightarrow (x \in c \wedge \varphi (x, t_1,\ldots, t_n)))$$

Again from the axiom of Extensionality it follows that the set B is uniquely determined by $c, t_1,\ldots,t_n$. B can be denoted by $\{x \in c \mid \varphi (x, t_1,\ldots, t_n)\}$. It is important to note that the set B being defined is a subset of the given set c, hence the name subset axiom.

As an example, the following formula is an instance of the subset axiom:

$$\forall A \forall C \exists B \forall x (x \in B \leftrightarrow x \in C \wedge x \in A)$$

It asserts the existence of the set intersection operation such that B = A ∩ C. Similarly the existence of the relative complement of C in A, denoted A – C, is asserted by the subset axiom instance:

$$\forall C \forall A \exists B \forall x (x \in B \leftrightarrow x \in A \wedge x \notin C)$$

An unrestricted version of the subset construction axiom was often used to specify sets before the development of axiomatic set theory:

$$\forall t_1 \ldots \forall t_n \forall c \exists B \forall x (x \in B \leftrightarrow \varphi(x, t_1, \ldots, t_n))$$

Here the restricting term $x \in c$ is omitted. This formulation leads directly to Russell's paradox referred to earlier by taking φ to be $x \notin x$. Most of the other axioms can be implied by the unrestricted form, for example the empty set, pairing and union axioms (Enderton 1977). These other axioms must therefore be explicitly stated since they cannot follow from the restricted subset form and the unrestricted form leads to inconsistencies.

2.1.6 Power set Axiom

For any set A there exists a set B whose elements are precisely the subsets of A. B is called the power set of A and is usually denoted by $\mathbb{P}(A)$.

$$\forall A \exists B \forall x (x \in B \leftrightarrow x \subseteq A)$$

The statement "$x \subseteq A$" is unfolded as:

$$\forall t (t \in x \rightarrow t \in A)$$

For example if $A = \{a, b, c\}$, then $\mathbb{P}(A) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$.

2.1.7 Infinity Axiom

There exists a set A such that $\varnothing$ is in A and whenever x is in A, so is the union $x \cup \{x\}$.

$$\exists A[\varnothing \in A \wedge \forall x (x \in A \rightarrow (x \cup \{x\}) \in A)]$$

An infinite set of this form contains a copy of the natural numbers as proposed by von Neumann in 1923 (Nerode & Shore 1997). In this representation the first four natural numbers would be represented as:

$$0 = \varnothing$$

$$1 = \varnothing \cup \{\varnothing\} = \{\varnothing\}$$

$$2 = \{\varnothing\} \cup \{\{\varnothing\}\} = \{\varnothing, \{\varnothing\}\}$$

A proof of these three properties is beyond the scope of this dissertation. Details may be found in (Enderton 1977).

As an example, let $A = \{1, 2\} = \{\{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}$ and $x = \{\varnothing\}$. It is then true that $x \in A$ and $x \cap A = \varnothing$, since $\varnothing \in \{\varnothing\}$ but $\varnothing \notin A$.

2.1.10 Axiom of choice

For any set A of nonempty sets, there is a function f with domain A such that for each $x \in A$, $f(x) \in x$. Function f is called a choice function for A and the range of f is called the choice set of A. In other words, f is a function that chooses one element from each set in A.

$$\forall A[\forall x(x \in A \rightarrow x \neq \varnothing) \rightarrow \exists f(\text{func}(f) \wedge \text{dom}(f) = A \wedge \forall x(x \in A \rightarrow f(x) \in x))]$$

For finite sets A, the axiom of choice is not required since the existence of a choice function can be proved using the other axioms (Enderton 1977). However, for infinite sets A, which are usually uncountable as well, the axiom of choice is needed. This is because it is either impossible or very difficult to construct a rule that makes an uncountable number of selections. In the case where it is very difficult to construct such a rule, the axiom of choice is not required but it makes proofs simpler by postulating that such a rule exists.

The axiom of choice has been controversial ever since Zermelo explicitly stated it as an axiom (Enderton 1977, Nerode & Shore 1997). One of the reasons for this is that it asserts the existence of an object without telling what it is. Objects that are proved to exist using the axiom of choice can generally not be described by any kind of systematic rule. These proofs are therefore non-constructive.

The following example illustrates how one may write a set-theoretic formula using the axioms above.

2.1.11 Example

Consider the following set-theoretic statement:

$$\mathbb{P}\{\{1\}\} = \{\varnothing,\{\{1\}\}\}$$

This statement can be represented in first-order logic with the conjunction of the following list of formulae:

$(\forall A)(\forall B)((\forall X)(X \in A \Leftrightarrow X \in B) \Rightarrow A = B)$	(extensionality)
$(\forall X)(\neg(X \in \text{empty}))$	(empty = $\varnothing$)
$(\forall X)(X \in a \Leftrightarrow X = 1)$	(a = {1})
$(\forall X)(X \in b \Leftrightarrow X = a)$	(b = {a})
$(\forall X)(X \in c \Leftrightarrow (\forall Y)(Y \in X \Rightarrow Y \in b))$	(c = $\mathbb{P}$(b))
$(\forall X)(X \in d \Leftrightarrow X = \text{empty} \vee X = b)$	(d = {empty,b})
$c = d$	(c = d)

2.2 Limitations of ZF Axioms in Automated Theorem Proving

The ZF axioms of subset construction and of replacement are infinite axiom schemas, since any well defined formula φ can be used to yield a relevant axiom. As a result ZF cannot be finitely axiomatised (Montague 1961) and, therefore cannot be input to an automated theorem prover. The user must therefore input the relevant axiom instances from the subset construction and replacement axiom schemas. For example, a proof that would require the premise that the relative complement of two sets exists, A = B – C, must have the following subset axiom instance specified:

$$\forall C \forall B \exists A \forall x(x \in A \leftrightarrow x \in B \wedge x \notin C)$$

There are other axiomatisations of set theory as well. The one used most often in the automated theorem proving community is that of von Neumann-Bernays-Gödel (NBG) (Enderton 1977, Quaife 1992a). NBG differs from ZF in that it makes a separation between concepts of a class and a set. A set has the same meaning as in ZF.

Additionally, any set is a class and any collection of sets is also a class. However, some classes are too large to be sets. An example of such a class is the class of all sets. It is not possible to refer to the class of all classes or the set of all sets which avoids any paradoxes due to self referencing.

Arguably, the most important aspect of NBG set theory for the automated reasoning community is the fact that it can be finitely axiomatised. NBG set theory is therefore mostly used for automated reasoning. It unfortunately suffers from having to deal with two sorts of objects (classes and sets) instead of one (sets). For more information on NBG and automated reasoning, the reader is referred to Boyer *et al*. (1986) and Quaife (1992a).

Formal specification languages like Z and B are based on ZF's set theory despite its infinite axiomatisation. This is because its limitations in theorem proving only become a problem when dealing with advanced mathematical proofs which do not occur in the day-to-day software engineering industry. For example the mathematical toolkit of Z (Spivey 1992) contains a finite number of axioms some of which are instances of the subset axiom. In this work we will therefore also use ZF set theory.

2.3 Summary

In this chapter we gave an overview of ZF set theory, its axioms and an example of specifying a simple set-theoretic statement in first-order logic. We further highlighted the role of set theory in formal specification languages. NBG was mentioned as an alternative axiomatisation to ZF. Unlike ZF, the NBG axioms are finite which makes it attractive for automated reasoning. However, the NBG axiomatisation is more cumbersome to use with little advantage for common set-theoretic problems. As a result, in this work we will go the route of ZF. The chapter concluded by discussing the difficulties of set-theoretic reasoning.

Chapter 3

Resolution

In this chapter we present an overview of resolution-based theorem proving. The chapter starts with a discussion on decidability and Herbrand's universe (Nerode & Shore 1997). These two concepts specify the theoretical limits of automated theorem proving (Leitsch 1997). Resolution is presented as an efficient refutation-based proof procedure. The rest of the chapter is dedicated to efficiency enhancements to resolution theorem proving. These enhancements include resolution refinement, redundancy tests, theory resolution and heuristics.

3.1 Decidability and Herbrand's Universe

At the heart of automated theorem proving lies the "decision problem" (Leitsch 1997). It is the challenge in symbolic logic to find a general algorithm which decides for any first-order statement whether it is universally valid or not. As early as the 17th century, Leibniz had the vision of building a machine that would solve this problem. The problem was revived in the early 20th century by Hilbert who posed it as one of several problems to the mathematical community. He called the decision problem the "fundamental problem of mathematical logic" (Leitsch 1997 p. 212). Progress was made in the following years by several mathematicians who found decidable subclasses of predicate logic.

It was not until the year 1936 that Alonzo Church and Alan Turing independently showed that the problem has no solution (Epstein & Carnielli 2000). Church developed an analysis of computability with his system of the λ-calculus (Church 1936a). He then showed that the λ-definable functions are undecidable. He later applied his conclusions to first-order predicate logic to show that there also exists no effectively calculable procedure to determine the validity of a logical formula (Church 1936b). Turing independently developed his own analysis of computability using the concept of a

machine that can only perform the most elementary operation (Turing 1936). This machine concept is now known as a Turing machine. Turing received a copy of Church's paper in time to include an appendix to show that a function is Turing machine computable if and only if it is λ-definable. By using a diagonal argument, Turing showed that the question of whether a Turing machine will halt on some arbitrarily chosen input is undecidable. This is known as the halting problem.

Herbrand contributed an important approach to mathematical theorem proving in 1930 (Chang & Lee 1973) by proposing a refutation procedure to determine the unsatisfiability of a set of clauses. He associated with each logic formula ¬S an infinite sequence of propositional logic formulas called the Herbrand universe of S (Nerode & Shore 1997). He then showed that ¬S is provable if and only if there is a finite disjunction of formulas in H that is provable. Based hereon he developed an algorithm to find an interpretation that can falsify a given formula. However, if the formula is indeed valid, no such interpretation can exist since it is by definition true under all interpretations. Herbrand's method forms the basis for most modern automatic proof procedures.

The commercial availability of computers during the 1950's enabled Gilmore (Gilmore 1960) to write a program to implement the refutation procedure of Herbrand's theorem. Since a formula is valid if and only if its negation is inconsistent, his program was designed to detect the inconsistency of the negation of the formula. Based on Herbrand's theorem, the unsatisfiability problem is reduced to propositional unsatisfiability and then check for inconsistency. Unfortunately Gilmore's method was only able to prove the simplest of formulas.

Davis and Putnam published a paper in 1960 (Davis & Putnam 1960), shortly after Gilmore's implementation, to improve on Gilmore's method by suggesting a more efficient method to test for the unsatisfiability of the ground sets. Their method was a major improvement but also lacked the necessary efficiency. As with Gilmore's method the generation of ground sets of formulas using a direct implementation of Herbrand's theorem was very inefficient (Leitsch 1997).

3.2 Resolution

All the refutation procedures that are based directly on Herbrand's theorem suffer from the same inefficiency that requires the generation of ground clause sets of the input clause set. It is typical for each successive set to grow exponentially.

In 1965, John Alan Robinson (1965a) published his famous paper on resolution-based theorem proving. It was a major breakthrough since it can be applied directly to any set of first-order logic clauses to test its unsatisfiability without the need to generate successive sets of ground clauses based on Herbrand's theorem.

In the years that followed, many refinements of resolution have been suggested in attempts to further increase its efficiency. Some of these refinements include hyperresolution (Robinson 1965b), set-of-support strategy (Wos 1965), semantic resolution (Slagle 1967) and paramodulation (Robinson & Wos 1969).

3.2.1 Clausal Form

Many computer implementations of first-order logic use the clausal form to represent formulas (Quaife 1992b), which is an apparently quantifier-free conjunctive normal form. This form was introduced by Davis and Putnam (1960). Formulas are therefore represented by a more restricted syntax-type that enables more efficient inference rules to be defined and makes it easier to control proof search. The clausal form of a formula is not necessarily logically equivalent to the original formula. However, the clausal form has the important property that it is unsatisfiable if and only if the original formula is unsatisfiable (Hamilton 1991).

A clause is a finite disjunction of zero or more literals (Chang & Lee 1973). It is sometimes convenient to regard a set of literals as synonymous with a clause. For example $\neg P(x) \vee Q(f(x)) = \{\neg P(x), Q(f(x))\}$. A clause with only one literal is called a unit clause. A clause that contains no literals is called the empty clause and is represented by $\square$. The empty clause is always false since it has no literal and cannot be

satisfied by any interpretation. The following identities hold for empty clauses (Leitsch 1997): $A \vee \square \vee B \equiv A \vee B$ and $\square \vee \square \equiv \square$.

A set S of clauses is regarded as a conjunction of all clauses in S, where every variable in S is considered governed by an implicit universal quantifier (Chang & Lee 1973). For example the formula $(\forall x)[(\neg P(x) \vee Q(f(x))) \wedge (\neg Q(x) \vee P(f(x)))]$ that is represented by the set of clauses $\{\neg P(x) \vee Q(f(x)), \neg Q(x) \vee P(f(x))\}$.

For every formula in first-order predicate logic there exists a procedure that maps it to a set of clauses (Wos *et al*. 1992). There is more than one method of executing this procedure. The first method consists of the following three steps (Chang & Lee 1973).

- The formula is converted into prenex normal form. A formula is in prenex normal form if all the quantifiers appear at the beginning. For example a prenex normal form of $(\forall x)P(x) \leftrightarrow (\exists y)Q(y)$ is $(\forall x)(\exists y)(P(x) \leftrightarrow Q(y))$.

- The formula is then transformed to conjunctive normal form. The above formula then becomes $(\forall x)(\exists y)[(\neg P(x) \vee Q(y)) \wedge (\neg Q(x) \vee P(y))]$.

- The last step is to eliminate all existential quantifiers using Skolem functions. In this step logical equivalence is usually lost however the transformation is still equisatisfiable. For every formula $\forall x_1 \ldots \forall x_n \exists y \psi$ the transformed formula $\forall x_1 \ldots \forall x_n \varphi$ where φ is obtained by replacing every variable y in ψ by the Skolem function $f(x_1,\ldots,x_n)$. By repeating this transformation, every existential quantifier can be eliminated. The last formula then becomes $(\forall x)[(\neg P(x) \vee Q(f(x))) \wedge (\neg Q(x) \vee P(f(x)))]$.

Another method of executing the procedure is given by Leitsch (1997). Here the formula is not required to be transformed to prenex form before the existential quantifiers are eliminated.

3.2.2 Resolution in Propositional Logic

In this section we first discuss how the resolution principle applies to propositional logic. In essence this principle may be viewed as an extension of the one-literal rule of Davis and Putnam (Chang & Lee 1973).

The cut rule (reading from top to bottom) states that:

if P then Q.

P.

therefore Q.

It may also be written in the following format:

$\neg$P, Q P
Q

The top line in the above box is the clausal form of P $\rightarrow$ Q where the comma represents a disjunction. The two clauses above the dividing line represent the premises of the inference rule and the clause below the line represents its conclusion. Q is therefore a logical consequence of ($\neg$P$\vee$Q) and P.

3.2.2.1 Binary Resolution Inference Rule

The propositional resolution principle extends the above rule of modus ponens by allowing any number of additional literals together with P and $\neg$P. The principle states that (Chang & Lee 1973, Leitsch 1997):

Definition 3.1

Let C_1 and C_2 be two clauses where C_1 has the form $L \vee M_1 \vee \ldots \vee M_i$ for $i \geq 0$ and C_2 has the form $\neg L \vee N_1 \vee \ldots \vee N_j$ for $j \geq 0$. From C_1 and C_2 we can then infer $M_1 \vee \ldots \vee M_i \vee N_1 \vee \ldots \vee N_j$.

The resolution inference rule can also be represented as (Eisinger & Ohlbach 1993):

$L, M_1, \ldots, M_i$ $\neg L, N_1, \ldots, N_j$
$M_1, \ldots, M_i, N_1, \ldots, N_j$

The resulting inferred clause is called the **resolvent** of C_1 and C_2. We say that we **resolved on** (the literal) L.

3.2.2.2 Resolution Deduction or Refutation

The next step is to show how a resolution deduction of a clause C can be deduced from a given formula S.

A resolution deduction is defined as (Chang & Lee 1973, Leitsch 1997):

Definition 3.2

Let S be a set of clauses. A resolution deduction of the clause C from S is a finite sequence of clauses $C_1, \ldots, C_n$ such that $C = C_n$ and for all $i=1,\ldots,n$ either C_i is a clause in S or C_i is a resolvent of C_j and C_k for $j, k < i$.

A resolution deduction of the empty clause $\square$ from S is called a resolution refutation of S.

Example 3.1

Consider the clause set $S = \{\neg a \vee b, a, \neg c\}$.

3.2.2.3 Propositional Factoring or Reduction Rule

The resolution inference rule on its own is not sufficient to provide a complete refutational inference system (Leitsch 1997). Take for example the following two clauses:

$C_1 = P \vee P$

$C_2 = \neg P \vee \neg P$

It is clear that these two clauses are contradictory and is unsatisfiable under all interpretations. However, by just employing the resolution inference rule, we cannot deduce the empty clause to show this unsatisfiability. Clauses C_1 and C_2 have resolvent $C_3 = P \vee \neg P$. Resolving C_3 with either C_1 or C_2 will just yield C_1 and C_2 again as resolvents.

We therefore require another inference rule to reduce a clause by getting rid of any redundant literals. This reduction rule states (Leitsch 1997):

Definition 3.3

Let C be a clause. Clause C' is a factor of C if it is obtained by removing any duplicate literals from C.

Applying this rule to the two example clauses C_1 and C_2 above will give us C_3=P and C_4=$\neg$P. The empty clause is then a resolvent of C_3 and C_4.

3.2.2.4 Soundness and Completeness

There are many texts that give the proofs for the refutational soundness and completeness of propositional resolution. The soundness theorem of resolution deduction states that if there is a resolution refutation of a set of clauses S, then S is unsatisfiable (Leitsch 1997, Chang & Lee 1973). The completeness theorem for propositional resolution deduction states that if a set of clauses S is unsatisfiable, then there exists a resolution refutation from S (Leitsch 1997, Nerode & Shore 1997). The proofs of these two properties are

beyond the scope of this dissertation, but details may be observed in Leitsch (1997) and Nerode and Shore (1997).

3.2.3 Resolution in First-order Predicate Logic

3.2.3.1 Substitution and Unification

The resolution principle for predicate logic is similar to that of propositional logic in that one attempts to deduce the empty clause from a set of clauses. However, with predicate logic, the clauses normally contain implicitly quantified variables that must be kept in mind when the resolution rule is applied (Nerode & Shore 1997). Take for example the following clauses:

C_1: $\neg P(x) \vee Q(x)$

C_2: $P(a)$

Variable x in clause C_1 may be unified with any constant e.g. a. We may therefore substitute the constant a for x to obtain the clause:

C_3: $\neg P(a) \vee Q(a)$

C_2 and C_3 can now be resolved upon to obtain resolvent Q(a).

In general a substitution θ can be defined as a set $\{t_1/v_1, \ldots, t_n/v_n\}$ where every v_i is a distinct variable and every t_i is a term other than v_i for $1 \leq i \leq n$, e.g. {f(z)/x, g(a)/y}. Let E be an expression denoting any term, atom or literal. Eθ is then also an expression that is obtained by simultaneously replacing each variable v_i in E with the term t_i (Chang & Lee 1973).

For example, let θ = {u/x, a/y, f(v)/z} and E = Q(x, f(y), z). Then Eθ = Q(u, f(a), f(v)).

Two or more expressions $E_1, \ldots, E_n$ can be unified if there exists a substitution θ such that $E_1\theta = E_2\theta = \ldots = E_n\theta$. The substitution θ is called a unifier of the expressions. Expressions are called unifiable when they have a unifier.

Unification is always applied using the most general unifier to be more effective (Nerode & Shore 1997). A unifier θ of a set of expressions $E_1, \ldots, E_n$ is called a most general unifier if and only if for every unifier σ of the set, there exists a substitution λ such that $E_1\theta = (E_1\sigma)\lambda = E_2\theta = (E_2\sigma)\lambda = \ldots = E_n\theta = (E_n\sigma)\lambda$.

A set of expressions always has a most general unifier if the set is unifiable and the problem of obtaining the most general unifier is decidable (Leitsch 1997). Most texts on resolution provide algorithms to determine the most general unifier.

In the next section we turn our attention to the resolution principle for predicate logic. Examples of the application of substitution and unification are shown.

3.2.3.2 Binary Resolution

Substitution and unification as discussed in the previous section allow us to apply the resolution principle to predicate logic. The resolution principle for predicate logic is stated as (Chang & Lee 1973, Leitsch 1997):

Definition 3.4

Let C_1 and C_2 be two clauses where C_1 has the form $L \vee M_1 \vee \ldots \vee M_i$ for $i \geq 0$ and C_2 has the form $\neg L' \vee N_1 \vee \ldots \vee N_j$ for $j \geq 0$. C_1 and C_2 also have no variables in common. If θ is a most general unifier of L and $\neg L'$, then we can infer clause $C = M_1\theta \vee \ldots \vee M_i\theta \vee N_1\theta \vee \ldots \vee N_j\theta$.

The resolution inference rule can also be represented as (Eisinger & Ohlbach 1993):

$L, M_1, \ldots, M_i$ $\neg L', N_1, \ldots, N_j$
$M_1\theta, \ldots, M_i\theta, N_1\theta, \ldots, N_j\theta$

where θ is the most general unifier of L and $\neg L'$.

The resulting inferred clause C is called the resolvent of C_1 and C_2. We say that we resolved on (the literal) L. Clause C is also called the child clause of parent clauses C_1 and C_2 (Nerode & Shore 1997).

The requirement that the parent clauses have no variables in common is due to the fact that the variables within each clause are local to that clause. This is because the clause is a Skolem standard form (Chang & Lee 1973) of the original formula with implied universal quantifiers for the variables at the beginning of the formula. The variables within different clauses are often renamed to avoid confusion. This renaming is referred to as standardising the variables apart (Nerode & Shore 1997).

A resolution deduction for predicate logic is performed in the same way as for propositional logic. The difference is that the binary inference rule for predicate logic is used (Nerode & Shore 1997).

Example 3.2

Let the clause set S be:

$$S = \{\neg P(x) \vee Q(f(a)), P(a), \neg Q(x)\}$$

We can then show that S is unsatisfiable using the following refutation deduction:

C_1 =	$\neg P(x) \vee Q(f(a))$	Clause in S
C_2 =	P(a)	Clause in S
C_3 =	$\neg Q(x)$	Clause in S
C_4 =	Q(f(a))	Resolvent of C_1 and C_2 Unifier is {a/x}
C_5 =	□	Resolvent of C_3 and C_4 Unifier is {f(a)/x}

Example 3.3: The farmer, goat, cabbage and wolf puzzle

The problem of the farmer, goat, cabbage and wolf is a classic puzzle that is often used to illustrate state space search problems (Eisinger & Ohlbach 1993). These types of problems usually have an initial state and a goal state. The solution to the problem is a path through all the valid states from the initial one to the goal.

The puzzle goes as follows:

> A farmer has a goat, a cabbage and a wolf that he has to take across a river. He has a small boat with which to accomplish this. Unfortunately the boat is very small and can only carry himself and *one* of the goat, cabbage or wolf. In his absence the goat would eat the cabbage and the wolf would eat the goat. How can he cross the river with the goat, cabbage and the wolf?

The above puzzle will be used to illustrate how resolution deduction can be used as a decision procedure to determine whether the puzzle has a solution.

A state in the problem is presented by the predicate symbol S with arity 4. The parameters indicate on which side of the river the farmer, goat, cabbage and wolf are as follows:

fh – farmer here

fa – farmer across

gh – goat here

ga – goat across

ch – cabbage here

ca – cabbage across

wh – wolf here

wa – wolf across

The **initial state** is given by the predicate:

S(fh, gh, ch, wh)

The **goal state** is given by:

S(fa, ga, ca, wa)

A safe state is one in which neither the goat and cabbage nor the goat and wolf are left unsupervised. The predicate symbol SAFE with arity 4 will be used to indicate a safe state. The parameters are similar to those of predicate S. The following are the safe states:

SAFE(fh, gh, ch, wh)

SAFE(fh, gh, ch, wa)

SAFE(fh, gh, ca, wh)

SAFE(fh, gh, ca, wa)

SAFE(fh, ga, ch, wh)

SAFE(fa, gh, ca, wa)

SAFE(fa, ga, ch, wh)

SAFE(fa, ga, ch, wa)

SAFE(fa, ga, ca, wh)

SAFE(fa, ga, ca, wa)

The farmer can cross the river with or without one of the goat, cabbage and wolf if we are in a safe state and the state after the river crossing is also safe. In the light of the resolution deduction, it means we can deduce a new state from an existing state if both

the current and new states are safe. The various river crossing rules are given by the following formulae:

Farmer goes across alone:

$\forall(x,y,z)\ [S(fh,x,y,z) \wedge SAFE(fh,x,y,z) \wedge SAFE(fa,x,y,z) \rightarrow S(fa,x,y,z)]$

Farmer comes back alone:

$\forall(x,y,z)\ [S(fa,x,y,z) \wedge SAFE(fa,x,y,z) \wedge SAFE(fh,x,y,z) \rightarrow S(fh,x,y,z)]$

Farmer takes goat across:

$\forall(y,z)\ [S(fh,gh,y,z) \wedge SAFE(fh,gh,y,z) \wedge SAFE(fa,ga,y,z) \rightarrow S(fa,ga,y,z)]$

Farmer brings goat back:

$\forall(y,z)\ [S(fa,ga,y,z) \wedge SAFE(fa,ga,y,z) \wedge SAFE(fh,gh,y,z) \rightarrow S(fh,gh,y,z)]$

Farmer takes cabbage across:

$\forall(x,z)\ [S(fh,x,ch,z) \wedge SAFE(fh,x,ch,z) \wedge SAFE(fa,x,ca,z) \rightarrow S(fa,x,ca,z)]$

Farmer brings cabbage back:

$\forall(x,z)\ [S(fa,x,ca,z) \wedge SAFE(fa,x,ca,z) \wedge SAFE(fh,x,ch,z) \rightarrow S(fh,x,ch,z)]$

Farmer takes wolf across:

$\forall(x,y)\ [S(fh,x,y,wh) \wedge SAFE(fh,x,y,wh) \wedge SAFE(fa,x,y,wa) \rightarrow S(fa,x,y,wa)]$

Farmer brings wolf back:

$\forall(x,y)\ [S(fa,x,y,wa) \wedge SAFE(fa,x,y,wa) \wedge SAFE(fh,x,y,wh) \rightarrow S(fh,x,y,wh)]$

The resolution deduction is presented in Appendix A.1. It represents one of a number of solutions:

> The farmer takes the goat across and returns. He then takes the wolf across and returns with the goat. He leaves the goat and takes the cabbage across. He then returns and take the goat across.

Referring to Appendix A.1, clauses 1 to 20 are the inputs to the resolution deduction. Clauses 1 to 8 are the above formulae converted to clausal form. Clauses 9 to 18 represent all the safe states. Clause 19 is the initial state. The goal state is given by clause 20 and is the negation of the actual goal since we are using refutation to show that a solution exists.

We deduced the empty clause and thereby a refutation, showing that a solution exists.

3.2.3.3 Factoring

As in the case of propositional resolution, the binary resolution inference rule for predicate logic is sound but not refutation complete (Wos *et al*. 1992). Take the following two clauses as example:

$C_1 = P(a) \vee P(y)$

$C_2 = \neg P(w) \vee \neg P(z)$

Clauses C_1 and C_2 are unsatisfiable, but binary resolution alone is not sufficient to deduce the empty clause. Any deduced clause will still contain two literals.

Factoring is an inference rule that overcomes this problem (Wos *et al*. 1992) and is defined as (Chang & Lee 1973):

Definition 3.5

If θ is a most general unifier of two or more literals of a clause C, then $C\theta$ is called a factor of C.

Returning to the above example, a factor of C_1 is P(a) and a factor of C_2 is $\neg$P(w). The conjunction of these two clauses is then unsatisfiable. Note that unification of P(a) and P(y) in C_1 for example results in P(a) $\vee$ P(a) and not just P(a). However, a clause is considered to be a set of literals and since one does not repeatedly list the same element of a set, unification produces the set {P(a)}.

3.2.3.4 Soundness and Completeness

The combination of the binary resolution and factoring inference rules provides us with a refutational sound and complete inference system (Wos *et al*. 1992). The soundness theorem of resolution deduction states that if there is a resolution refutation of a set of clauses S, then S is unsatisfiable (Nerode & Shore 1997). The completeness theorem of resolution deduction states that if a set of clauses S is unsatisfiable, then there exists a resolution refutation from S (Nerode & Shore 1997, Chang & Lee 1973, Leitsch 1997).

The detailed proof of the above soundness and completeness properties is beyond the scope of this dissertation. Nevertheless the proof starts by showing that the system is complete for ground clauses. The lifting lemma is then the key to proving the completeness of the system for predicate logic. The lifting lemma shows that any instantiation of a deduction can be replaced by a more general one. It is called the lifting lemma because it “lifts” ground deductions to deductions in predicate logic.

3.3 Efficiency Enhancements

The field of automated reasoning concerns itself mainly with searching for the existence of proofs. The size of the search space and the method of traversing the search space are of vital importance to the efficiency of automated theorem proving (Leitsch 1997).

Robinson’s resolution principle (Robinson 1965a) brought about a major advancement to the field of automated reasoning. With each application of the binary inference rule, the search space grows by a bounded number of branches which are generally not too many, compared to methods based on Herbrand’s theorem and other classical methods where the search space could grow at an unbounded rate (Eisinger & Ohlbach 1993).

S_0 are added to S_1 until no more resolutions are possible, that is, until the level is saturated. The resolvents of $S_0 \cup S_1$ are then added to S_2. This process is continued until the empty clause is found.

The level sets are defined as:

$$S_0 = S$$

$$S_n = \{\text{resolvents of } C_1 \text{ and } C_2 \mid C_1 \in \{S_0 \cup \ldots \cup S_{n-1}\}, C_2 \in S_{n-1}\}, n=1,2,\ldots$$

The level-saturation method of resolution is a simple algorithm to implement on a computer but generates an extremely high number of clauses. The example of the farmer, goat, cabbage and wolf puzzle in Appendix A.1 will again be used to illustrate this point. However, we will use a different start state, one where the goat is on this side and the farmer, cabbage and wolf on the other side. Also, only two levels of resolvents will be generated and only an adequate set of input clauses is used. All the initial clauses will keep the same clause numbering as before.

The level-saturation deduction is shown in Appendix A.2. Nine clauses are generated in the first level and 28 in the second level. This shows that the number of resolvents for each level grows at a phenomenal rate.

3.4.1 Linear Resolution

The above level saturation implementation of resolution is not a natural way for people to carry out a proof using resolution. Humans would most likely start with a clause, resolve it with another clause and use the resolvent for the next resolution step, until the empty clause is deduced (Chang & Lee 1973). This method of resolution is a called linear resolution and is a refinement of resolution (Nerode & Shore 1997).

3.4.1.1 Linear Resolution Deduction

We can formally define linear resolution as (Chang & Lee 1973, Leitsch 1997, Nerode & Shore 1997):

Definition 3.6

Let S be a set of clauses and C a clause in S. A linear deduction of D from S with top clause C is a sequence $\langle C_0, B_1, C_1, \ldots, B_n, C_n\rangle$ of clauses (for $n \geq 1$) such that

- $C_0 = C$ and
- $D = C_n$ and
- for $1 \leq i \leq n$, C_i is a resolvent of C_{i-1} and B_i and
- B_i is either in S or is a C_j for some $j < i$.

C is called the top clause, all B_i are called side clauses and all C_i are called centre clauses. There is a linear resolution refutation of S if the empty clause can be deduced from S.

Example 3.4

Let S be the set of clauses $S = \{Q(x) \vee R(x), \neg Q(x) \vee R(f(y)), Q(x) \vee \neg R(f(x)), \neg Q(x) \vee \neg R(x)\}$. The following is a linear resolution refutation of S:

	Clauses in S	
$C_1 =$	$Q(x) \vee R(x)$	
$C_2 =$	$\neg Q(x) \vee R(f(y))$	
$C_3 =$	$Q(x) \vee \neg R(f(x))$	
$C_4 =$	$\neg Q(x) \vee \neg R(x)$	
	Linear refutation with top clause C_2	
$C_5 =$	$\neg Q(x) \vee \neg Q(f(y))$	C_2 and C_4
$C_6 =$	$R(f(y))$	C_5 and C_1
$C_7 =$	$Q(x)$	C_6 and C_3

C_8 =	$\neg R(x)$	C_7 and C_4
C_9 =	$\square$	C_8 and C_6

Note that clause C_6 was obtained after factoring was applied to clause C_5, followed by a resolution step with C_1.

The farmer, goat, cabbage and wolf puzzle that was used as an example of binary resolution (Appendix A.1) is also an example of linear resolution.

3.4.1.2 Soundness and Completeness

Linear resolution is a special case of binary resolution with factoring which is sound, therefore linear resolution is also sound (Nerode & Shore 1997).

Linear resolution is also a complete resolution refutation procedure. A proof is provided by Leitsch (1997). It must be noted however that using the incorrect top clause can cause incompleteness. For example, let S be the set of clauses S = {P(x), $\neg P(y) \vee Q(y)$, $\neg Q(u)$, R(a)}. The following linear deduction shows that S is unsatisfiable:

	Clauses in S	
C_1 =	P(x)	
C_2 =	$\neg P(y) \vee Q(y)$	
C_3 =	$\neg Q(u)$	
C_4 =	R(a)	
	Linear refutation with top clause C_1	
C_5 =	Q(y)	C_1 and C_2
C_6 =	$\square$	C_5 and C_3

However, if clause C_4 is chosen as the top clause, then there are no other clauses that can be resolved with it. C_4 is therefore the only linear deduction and is not a refutation.

3.4.1.3 Refinements of Linear Resolution

There are various refinements for linear resolution (Leitsch 1997). Some of these include clause ordering and literal information (Chang & Lee 1973) as well as input and UR resolution. The latter two will be discussed in the following sections.

3.4.1.4 Input Resolution

Input resolution is a refinement of linear resolution but is not refutation complete (Wos *et al*. 1992). Input resolution is still useful despite its incompleteness. The reason for this is that a large class of theorems can be proved with it and it is very efficient (Chang & Lee 1973).

Input resolution can be defined as (Chang & Lee 1973, Leitsch 1997):

Definition 3.7

Let S be a set of clauses. A clause in S is called an input clause. An input resolution is a resolution in which one of the parent clauses is an input clause. An input deduction is a linear deduction in which all the side clauses are input clauses. An input refutation is an input deduction of the empty clause.

The class of theorems for which input resolution is complete is called Horn logic (Leitsch 1997):

Definition 3.8

Horn logic is the class of all finite sets of Horn clauses, where a Horn clause is a clause with one of the following forms:

1. P
2. $P \vee \neg Q_1 \vee \ldots \vee \neg Q_n$
3. $\neg Q_1 \vee \ldots \vee \neg Q_n$

Form 1 is called a fact, 2 is called rule and 3 is called a goal.

A Horn clause is therefore a clause with at most one positive literal. The terminology of facts, rules and goals comes from the field of logic programming. A proof of the completeness of input resolution on Horn logic is provided by (Leitsch 1997).

The following example serves to show that input resolution is not complete in predicate logic.

Example 3.5

Let S be the set of clauses $S = \{Q(x) \vee R(x), \neg Q(x) \vee R(f(y)), Q(x) \vee \neg R(f(x)), \neg Q(x) \vee \neg R(x)\}$. Note that this is the same set of refutable clauses that was used in Example **3.4**. S does not contain any unit clauses or unit factors of clauses. Also, S contains a non-Horn clause $Q(x) \vee R(x)$. Let $D = \langle C_0, B_1, C_1, \ldots, B_n, C_n \rangle$ be an arbitrary linear input deduction from S (for $n \geq 1$). C_0 must be a clause from S. B_n is also a clause from S. C_n is a resolvent of C_{n-1} and B_n. However, C_n cannot be the empty clause since neither B_n nor any factor of it is a unit clause. D can therefore not be an input refutation of S.

Example 3.6

This example shows an input refutation. It is an example about the relationship between being a parent and grandparent, father and grandfather in this case.

	Clauses in S	
C_1 =	¬FATHER(x, y) ∨ ¬FATHER(y,z) ∨ GRANDFATHER(x, z)	
C_2 =	FATHER(johnSr, johnBoy)	
C_3 =	FATHER(zebulon, johnSr)	
C_4 =	¬GRANDFATHER(zebulon, johnBoy)	
	Input refutation with top clause C_4	
C_5 =	¬FATHER(zebulon, y) ∨ ¬FATHER(y, johnBoy)	C_4 and C_1
C_6 =	¬FATHER(zebulon, johnSr)	C_5 and C_2
C_7 =	□	C_6 and C_3

The farmer, goat, cabbage and wolf puzzle that was used above as an example for binary resolution is also an example of input resolution.

3.4.1.5 Unit Resolution

Unit resolution is a refinement of resolution, but not linear resolution. It is discussed here because it is refutation equivalent to input resolution. Unit resolution can be viewed as an extension of the one-literal rule of Davis and Putnam and may be defined as (Chang & Lee 1973):

Definition 3.9

A unit resolution is a resolution in which at least one parent clause is a unit clause or a unit factor thereof. A deduction in which every resolution step is a unit resolution is called a unit deduction. A unit deduction of the empty clause is called a unit refutation.

Unit resolvents are always smaller as opposed to binary resolution where the resolvents tend to be longer clauses (Quaife 1992b). This property is very important since to deduce the empty clause, shorter clauses ought to be deduced. As a result, unit resolution is a very efficient refinement of resolution (Chang & Lee 1973).

As was stated above, input- and unit resolution are refutation equivalent. That is, a theorem can be proved with input resolution if and only if it can be proved by unit resolution. A proof of this equivalence can be found in (Chang & Lee 1973). This equivalence then implies that unit resolution, as with input resolution, is not refutation complete but is complete for Horn logic (Wos *et al*. 1992).

The proof of the completeness of unit resolution for Horn logic follows from its equivalence with input resolution and the proof referred to in the previous section that input resolution is complete over Horn logic (Leitsch 1997).

Since input resolution and unit resolution are equivalent, Example **3.5** that was used to show that input resolution is not refutation complete also suffices to show that unit resolution is not refutation complete. Recall the given set S = {Q(x) ∨ R(x), ¬Q(x) ∨ R(f(y)), Q(x) ∨ ¬R(f(x)), ¬Q(x) ∨ ¬R(x)}. This time it is easier to see that unit resolution is not sufficient to refute S. This is because there is no unit clause in S that can be used as a parent clause to perform a unit resolution.

Example **3.6** that was used to illustrate an input refutation is also an example of a unit refutation. The farmer, goat, cabbage and wolf puzzle is another example of a unit refutation.

3.4.2 Semantic Resolution

Semantic resolution was proposed by Slagle (1967). It unifies Robinson's hyper-resolution (Robinson 1965b), Meltzer's renamable resolution (Meltzer 1966) and the set-of-support strategy of Wos, Robinson and Carson (1965). These resolution concepts will be discussed below.

3.4.2.1 Splitting into Two Groups

The first method that semantic resolution provides to reduce the number of resolvents is to split a given set S of clauses into two groups S_1 and S_2. Clauses within the same group are not allowed to be resolved with each other. The criterion by which the given set is split in two is determined by a Herbrand interpretation, M (Bachmair & Ganzinger 2001). All clauses that are true under M are put into one group and the rest are put into the other group. It should be noted that if the set of clauses is unsatisfiable, then there is no interpretation that can make all the clauses true. As a result, all interpretations would split the set of clauses in two groups.

Example 3.7

Consider clauses C_1 and C_2 of the puzzle (in Appendix A.1) that are repeated here:

C_1 =	¬S(fh, x, y, z) ∨ ¬SAFE(fh, x, y, z) ∨ ¬SAFE(fa, x, y, z) ∨ S(fa, x, y, z)	Farmer goes across
C_2 =	¬S(fa, x, y, z) ∨ ¬SAFE(fa, x, y, z) ∨ ¬SAFE(fh, x, y, z) ∨ S(fh, x, y, z)	Farmer returns

C_1 and C_2 have the following resolvents:

¬S(fh, x, y, z) ∨ ¬SAFE(fh, x, y, z) ∨ ¬SAFE(fa, x, y, z) ∨ ¬SAFE(fa, x, y, z) ∨ ¬SAFE(fh, x, y, z) ∨ S(fh, x, y, z)	Resolved on S(fa, x, y, z)
¬SAFE(fh, x, y, z) ∨ ¬SAFE(fa, x, y, z) ∨ S(fa, x, y, z) ∨ ¬S(fa, x, y, z) ∨ ¬SAFE(fa, x, y, z) ∨ ¬SAFE(fh, x, y, z)	Resolved on S(fh, x, y, z)

Let M be an interpretation in which every literal is the negation of an atom:

M = {¬S(fh, gh, ch, wh), ¬S(fh, gh, ch, wa), ¬S(fh, gh, ca, wh), …,
¬SAFE(fh, gh, ch, wh), ¬SAFE(fh, gh, ch, wa), ¬SAFE(fh, gh, ca, wh), … }

Further, let all clauses that are true under M go into group S_1 and the rest into group S_2. Both C_1 and C_2 in the puzzle are true under M and therefore belong to the same group S_1. C_1 and C_2 are therefore not allowed to be resolved with each other under the principle of semantic resolution with splitting.

3.4.2.2 Ordering of Predicate Symbols

The second concept of semantic resolution that allows us to cut down on the number of generated resolvents is the ordering of predicate symbols. Given an ordering of predicate symbols, we can only resolve a clause X from S_1 with a clause Y from S_2 if the literal resolved upon contains the largest predicate symbol in X. Such ordering of predicate symbols is specified beforehand.

Example 3.8

Consider clauses C_1 and C_9 of the puzzle that is repeated here:

C_1 =	$\neg S(fh, x, y, z) \vee \neg SAFE(fh, x, y, z) \vee$ $\neg SAFE(fa, x, y, z) \vee S(fa, x, y, z)$	Farmer goes across
C_9 =	SAFE(fh, gh, ch, wh)	Safe state

C_1 and C_9 has the following resolvent:

$\neg S(fh, x, y, z) \vee \neg SAFE(fa, x, y, z) \vee S(fa, x, y, z)$	Resolved on SAFE(fh, gh, ch, wh) Unifier {gh/x, ch/y, wh/z}

We will use the same interpretation M as before in which every literal is the negation of an atom. Also, let clauses that are true under M go into S_1 and the rest into S_2. C_1 is therefore in group S_1 and C_9 is in group S_2. Let the predicate ordering be S > SAFE. C_1 and C_9 are in different groups and therefore the splitting criteria do not prevent them from

	Variant 1	
$C_{1\text{-}21}$ =	¬SAFE(fh, gh, ch, wh) ∨ ¬SAFE(fa, ga, ch, wh) ∨ S(fa, ga, ch, wh)	Resolvent of C_3 and C_{19} Unifier {ch/y, wh/z}
$C_{1\text{-}22}$ =	¬SAFE(fh, gh, ch, wh) ∨ S(fa, ga, ch, wh)	Resolvent of C_{15} and $C_{1\text{-}21}$
$C_{1\text{-}23}$ =	S(fa, ga, ch, wh)	Resolvent of C_9 and $C_{1\text{-}22}$
	Variant 2	
$C_{2\text{-}21}$ =	¬S(fh, gh, ch, wh) ∨ ¬SAFE(fa, ga, ch, wh) ∨ S(fa, ga, ch, wh)	Resolvent of C_3 and C_9 Unifier {ch/y, wh/z}
$C_{2\text{-}22}$ =	¬S(fh, gh, ch, wh) ∨ S(fa, ga, ch, wh)	Resolvent of C_{15} and $C_{2\text{-}21}$
$C_{2\text{-}23}$ =	S(fa, ga, ch, wh)	Resolvent of C_{19} and $C_{2\text{-}22}$

There are at least three other ways in which clause C_{23} may be deduced. All of these deductions use clauses C_3, C_9, C_{15} and C_{19}. The only difference between them is the order in which they use the clauses. The semantic clash avoids this redundant generation of clauses by generating clause C_{23} directly from clauses C_3, C_9, C_{15} and C_{19} without the need of the intermediate clauses like C_{21} and C_{22}. In this scenario the set {C_3, C_9, C_{15}, C_{19}} is called a clash.

A clash can formally be defined as (Slagle 1967):

Definition 3.10

A clash S is a finite set of clauses {E_1, …, E_n, N} for n≥1 such that

1. clause N contains at least n literals L_1, …, L_n
2. for all i = 1, …, n clause E_i contains the complement ¬L_i of literal L_i, but not the complement of any other literal in N nor any literal in E_j for j = 1, …, n.

N is called the nucleus and all E_i are called electrons.

3.4.2.4 Semantic Resolution

Semantic resolution refers to the technique whereby some interpretation M is used to divide a set of clauses into two groups and a resolution step must use clauses from both groups. It is defined as (Leitsch 1997):

Definition 3.11

Let S be a set of clauses and let M be an interpretation of S. Let C and D be clauses in S such that either C or D is false in M. A resolvent with C and D as parent clauses is then called a semantic M-resolvent or simply an M-resolvent.

A semantic deduction is defined as (Leitsch 1997):

Definition 3.12

Let S be a set of clauses and let M be an interpretation of S. A semantic deduction of the clause C from S is a finite sequence of clauses $C_1,\ldots,C_n$ such that $C = C_n$ and for all i=1,…,n either C_i is a clause in S or C_i is an M-resolvent.

3.4.2.5 Semantic Clash Resolution

Semantic resolution can be strengthened by introducing the concept of the semantic clash. This kind of resolution is called semantic clash resolution. It is defined as (Leitsch 1997):

Definition 3.13

Let M be an interpretation of a finite set of clauses $S = \{E_1, \ldots, E_q, N\}$ for $q \geq 1$ that satisfies the following conditions:

1. $E_1, \ldots, E_q$ are false under M.
2. Let $R_1 = N$. There exists a resolvent R_{i+1} of R_i and E_i for $1 \leq i \leq q$.
3. R_{q+1} is false under M.

Set S is then called a semantic clash with respect to M, or simply an M-clash. Clauses $E_1,\ldots, E_q$ are called electrons and clause N is called the nucleus. R_{q+1} is called an M-resolvent of the M-clash S.

A semantic clash deduction is defined as (Leitsch 1997):

Definition 3.14

Let S be a set of clauses and let M be an interpretation of S. A semantic clash deduction of the clause C from S is a finite sequence of clauses $C_1,\ldots,C_n$ such that $C = C_n$ and for all i=1,…,n either C_i is a clause in S or C_i is an M-resolvent of an M-clash.

A proof of the completeness of semantic clash resolution is provided by (Leitsch 1997). The ground completeness is first proved as a lemma. Thereafter, the completeness for first-order logic is proved by using the lifting lemma. Details of the proof are beyond the scope of this dissertation.

3.4.2.6 Semantic Clash Resolution with Predicate Ordering

Semantic Clash Resolution can be strengthened by adding predicate ordering. This is how Slagle (1967) originally proposed semantic resolution. It is defined as (Slagle 1967, Chang & Lee 1973):

Definition 3.15

Let M be an interpretation and P be an ordering of predicate symbols of a finite set of clauses $S = \{E_1, \ldots, E_q, N\}$ for $q \geq 1$ that satisfy the following conditions:

1. $E_1, \ldots, E_q$ are false under M.
2. Let $R_1 = N$. There exists a resolvent R_{i+1} of R_i and E_i for $1 \leq i \leq q$.
3. The literal that was resolved upon in E_i contains the largest predicate symbol in E_i for $1 \leq i \leq q$.
4. R_{q+1} is false under M.

Set S is then called a semantic clash with respect to P and M, or simply a PM-clash. Clauses $E_1,\ldots, E_q$ are called electrons and clause N is called the nucleus. R_{q+1} is called a PM-resolvent of the PM-clash S.

A semantic clash resolution deduction with predicate ordering is then defined as (Slagle 1967, Chang & Lee 1973):

Definition 3.16

Let S be a set of clauses, M an interpretation of S and P an ordering of the predicate symbols appearing in S. A semantic clash resolution deduction with predicate ordering of the clause C from S is a finite sequence of clauses $C_1,\ldots,C_n$ such that $C = C_n$ and for all i=1,…,n either C_i is a clause in S or C_i is a PM-resolvent of a PM-clash.

Proofs of the completeness of semantic clash resolution with predicate ordering are provided by both Slagle (1967) and Chang and Lee (1973). As for semantic clash resolution, the ground completeness is first proved. Thereafter, the completeness for first-order logic is proved by using the lifting lemma.

Next we discuss a number of important subclasses of semantic resolution namely UR-resolution, hyperresolution and set-of-support resolution.

3.4.3 UR-resolution

Unit resulting resolution or simply UR-resolution was proposed in 1967 by McCharen *et al*. (1967). It derives its name from the fact that it produces unit clauses as resolvents.

UR-resolution inference rule can be formally defined as (Eisinger & Ohlbach 1993):

Definition 3.17

Let S be a set of clauses $S = \{E_1, \ldots, E_n, N\}$ for $n\geq 1$. $E_1, \ldots, E_n$ are unit clauses. Clause N has the form $N = L_1 \vee \ldots \vee L_{n+1}$. Let θ be a most general unifier such that $L_i\theta$ and $E_i\theta$ are complementary for all $i = 1, \ldots, n$.

L_{n+1} is called a UR-resolvent of S and is a unit clause. The clause N is called the nucleus. θ is called a simultaneous unifier S.

The inference rule can also be defined in the following format (Van der Poll 2000, Quaife 1992b):

E_1 : E_n $L_1, \ldots, L_{n+1}$
$L_{n+1}\theta$

where θ is a simultaneous unifier such that $L_i\theta$ and $E_i\theta$ are complementary for all i = 1, …, n.

A UR-resolution deduction is defined as (Wos *et al*. 1992, Eisinger & Ohlbach 1993):

Definition 3.18

Let S be a set of clauses. A UR-resolution deduction of the clause C from S is a finite sequence of clauses $C_1,\ldots,C_n$ such that $C = C_n$ and for all i=1,…,n either C_i is a clause in S or C_i is a UR-resolvent of S.

The unit clause resolvent can also be derived using binary resolution (Wos *et al*. 1992). However, in this case binary resolution has some disadvantages (Quaife 1992b). This is because a number of applications of the binary resolution rule are required. As a result intermediate clauses are generated that unnecessarily enlarge the search space. Depending on the search algorithm used e.g. level saturation (Chang & Lee 1973), the same intermediate clauses may be generated more than once because every possible combination of resolution could be attempted. This concept is known as a clash (Slagle 1967) and was discussed under semantic resolution above.

UR-resolution eliminates the unnecessary generation of resolvents by replacing all the individual inferences by just one inference step. For this reason, UR-resolution is referred to as a macro resolution step (Eisinger & Ohlbach 1993).

UR-resolution essentially combines several applications of the unit resolution rule into one macro resolution rule by using the concept of a clash. Unit resolution is not refutation complete, but is complete for Horn logic (Wos *et al*. 1992). As a result UR-resolution is not refutation complete, but is complete for Horn logic (Quaife 1992b). UR-resolution is usually used in conjunction with other inference rules due to its incompleteness.

Appendix A.3 shows an example of the farmer, goat, cabbage and wolf puzzle with UR-resolution applied. Note that the initial clause set is not repeated in the appendix. The use of UR-resolution substantially shortens the proof by simultaneously resolving more than two parent clauses.

3.4.4 Hyperresolution

Hyperresolution was proposed by Robinson (1965b) in the same year that he proposed binary resolution. Hyperresolution is a special case of semantic clash resolution (Leitsch 1997) based on the interpretation that is used. There are two variants, positive and negative hyperresolution. The difference between the two variants is the interpretation that is used. Hyperresolution can be defined in terms of semantic clash resolution as (Chang & Lee 1973):

Definition 3.19

Positive hyperresolution is a special case of semantic clash resolution (with or without predicate ordering) where the interpretation M is chosen such that every literal is negative.

Negative hyperresolution is a special case of semantic clash resolution (with or without predicate ordering) where the interpretation M is chosen such that every literal is positive.

Hyperresolution can also be defined independently of semantic clash resolution (Eisinger & Ohlbach 1993):

Definition 3.20

A clause is called positive if none of its literals has a negation sign. A clause is called negative if all of its literals have a negation sign. A clause is called mixed if it is neither positive nor negative.

Definition 3.21

Let S be a set of clauses $S = \{E_1, \ldots, E_n, N\}$ for $n \geq 1$. Clause N is negative (positive) or mixed and has the form $N = L_1 \vee \ldots \vee L_{n+m}$ for $m \geq 0$. For all $i = 1, \ldots, n$ E_i is positive (negative) and has the form $E_i = K_i \vee H_i$ where K_i is a literal and H_i a possibly empty clause. Let θ be a most general unifier such that $L_i\theta$ and $K_i\theta$ are complementary for all $i = 1, \ldots, n$. Clause $H_1\theta \vee \ldots \vee H_n\theta \vee L_{n+1} \vee \ldots \vee L_{n+m}$ is then called a positive (negative) hyperresolvent of S.

A resolution that yields a positive (negative) hyperresolvent is called a positive (negative) hyperresolution. Clause N is called the nucleus and all E_i are called electrons or satellites. θ is called a simultaneous unifier of S.

The inference rule can also be defined in the following format (Van der Poll 2000, Quaife 1992b):

K_1, H_1 : K_n, H_i $L_1, \ldots, L_{n+m}$
$H_1\theta \vee \ldots \vee H_n\theta \vee L_{n+1}\theta \vee \ldots \vee L_{n+m}\theta$

where all symbols have the same meaning as in Definition **3.21**.

Positive hyperresolution derives its name from the fact that all electrons and hyperresolvents are positive. Negative hyperresolution derives its name similarly namely all electrons and hyperresolvents are negative.

A hyperresolution deduction is defined as (Slagle 1967):

Definition 3.22

A positive hyperdeduction is a semantic clash deduction (with or without predicate ordering) in which the interpretation M is chosen such that every literal is negative.

A negative hyperdeduction is a semantic clash deduction (with or without predicate ordering) in which the interpretation M is chosen such that every literal is positive.

Example 3.9

This example refutation is the same as Example **3.6** about the relationship between being a father and grandfather except for the last deduction step C_6 that makes use of positive hyperresolution.

	Clauses in S	
C_1 =	¬FATHER(x, y) ∨ ¬FATHER(y,z) ∨ GRANDFATHER(x, z)	
C_2 =	FATHER(johnSr, johnBoy)	
C_3 =	FATHER(zebulon, johnSr)	
C_4 =	¬GRANDFATHER(zebulon, johnBoy)	
	Input refutation with top clause C_4	
C_5 =	¬FATHER(zebulon, y) ∨ ¬FATHER(y, johnBoy)	C_4 and C_1
C_6 =	□	C_5, C_2 and C_3

Hyperresolution can be regarded as a generalisation of UR-resolution (Eisinger & Ohlbach 1993). As with UR-resolution, hyperresolution is also a macro inference rule (Leitsch 1997). It has the same advantages as UR-resolution in that it combines more than one inference step into a single step, i.e. it eliminates the generation of intermediate clauses. Therefore, the order in which intermediate resolution steps would have been carried out for the semantic clash becomes irrelevant.

Hyperresolution has the additional advantage that it is refutation complete. Its completeness is implied by the completeness of semantic clash resolution. A direct proof is also provided by (Leitsch 1997) and is beyond the scope of this dissertation.

The input to a theorem-proving attempt is usually given as positive or mixed clauses and the negated conclusion as negative clauses. With negative hyperresolution, the negative conclusion clauses are typically used as electrons. The negative hyperdeduction therefore tends to be suitable for backward reasoning from the conclusion towards the axioms. Similarly, positive hyperdeduction tends to correspond to forward reasoning from the axioms towards the conclusion (Chang & Lee 1973, Eisinger & Ohlbach 1993).

Definition 3.23

Let T be a subset of a set of clauses S. If S – T is satisfiable then T is a set-of-support in S.

A resolution of which the parent clauses are not both from S – T is called a set-of-support resolution.

A deduction in which every resolution is a set-of-support resolution is called a set-of-support deduction.

The set-of-support strategy is refutation complete. Rather lengthy proofs of completeness are given by Wos (1965, 1992). A very concise completeness proof is provided by Slagle (1967) in terms of semantic clash resolution. In this proof, the interpretation M that is used for the semantic clash deduction is any interpretation that satisfies the set S – T. The set S – T is assumed to be satisfiable by definition. Based on this assumption it must have an interpretation that satisfies all of its clauses. It is however possible that the satisfiable set is chosen incorrectly which will fail the assumption the proof is based on. In such a case all proofs might be blocked (Wos *et al*. 1992). The following example illustrates how the wrong choice for the set-of-support could block a refutation:

	Given set
C_1 =	$P(x) \vee Q(y)$
C_2 =	$\neg P(b)$
C_3 =	$\neg Q(c)$
	Set-of-support
C_4 =	$P(a)$

The clause set S = {C_1, C_2, C_3, C_4} is unsatisfiable, but no resolution is possible using the set-of-support strategy. No resolvent is possible starting with clause C_4.

The following two examples show how powerful the set-of-support strategy is in restricting the growth of the search space. The same example (Appendix A.2) that was used to illustrate the level saturation method with using just binary resolution is again used here. The only difference is that the set-of-support strategy is used. The negated goal clause ¬S(fa, ga, ca, wa) is put in the set-of-support and the rest of the clauses in the unsupported set.

The first example is given in Appendix A.4 and shows up to saturation level 3. In this example the first level has only 1 clause instead of 9 clauses in the original example. The second level has only 3 clauses instead of 28. The third level has 8 clauses and the forth level that has not been shown in Appendix A.4 has 21 clauses. The number of clauses per saturation level started to grow very fast in the third and forth levels. The example was therefore not completed since the level growth becomes too large to apply resolution manually.

The second example is given in Appendix A.5. It is the same as the previous one except that this time a predicate ordering is applied such that S > SAFE. This greatly reduced the number of clauses, enabling the example to be extended up to a refutation. Note that the initial set of clauses has been omitted. The first three levels have only one clause as opposed to the previous example's 1, 3 and 8 respectively. Level four has 3 clauses instead of 21. Level five has 6 clauses and a refutation was found immediately on level 6.

3.5 Redundancy and Deletion

The various resolution refinements introduced above could still contain redundancies such as tautologies and circular derivations. Redundancy tests can eliminate these and thereby reduce the search space.

3.5.1 Subsumption

Subsumption is a deletion strategy whereby duplicate clauses or clauses that are more specific than certain other clauses are discarded (Wos *et al*. 1992). This is in line with

the resolution principle that works on the most general level (Leitsch 1997). The case for deletion can be defined as (Eisinger & Ohlbach 1993):

Definition 3.24

A clause C subsumes a clause D if and only if there is a substitution θ such that $C\theta \subseteq D$. D is called a subsumed clause.

The symbol $\subseteq$ in the above definition is used to indicate subsumption of one clause by another. Note that according to this definition a clause D is regarded as redundant not only if it is an instance of C, but also if it contains an instance of C.

For example, let $C = P(x) \vee Q(y)$ and $D = P(a) \vee Q(b) \vee R(a)$. For $\theta = \{a/x, b/y\}$ we get $C\theta = P(a) \vee Q(b)$. But $C\theta \subseteq D$ and therefore C subsumes D. From this example it can be seen that clause C implies clause D and is therefore more general.

Subsumption is often employed as a pre-processing step whereby a set of clauses is first reduced before resolution takes place. Subsumption can also be used during resolution deductions (Leitsch 1997). Forward subsumption is the process that discards any newly generated clauses that are subsumed by previously retained clauses. Backward subsumption occurs when newly generated clauses are used to discard previously retained clauses by subsumption. Lastly, if derived clauses are periodically reduced by subsumption, the process is called replacement.

The pruning of the search space using subsumption is in general refutation complete (Wos *et al*. 1992). Proofs of the completeness and incompleteness of subsumption in combination with some resolution refinements are provided by (Leitsch 1997). An example of incompleteness is the combination of forward subsumption with lock resolution. Another example is the use of subsumption with the set-of-support strategy (Wos *et al*. 1992). A clause D with support can be subsumed by a clause C without support. Clause D might however be required in the final proof, hence the problem can be solved by also giving clause C support.

Appendix A.6 illustrates the use of subsumption combined with the set-of-support strategy. The same example that was used to illustrate the set-of-support strategy without predicate ordering (Appendix A.4) is used here again with the addition of subsumption. Subsumption greatly reduces the size of the search space thereby making it viable to extend it up to a refutation. The number of generated clauses in the first four levels was 1, 2, 4 and 2 as opposed to the original example's 1, 3, 8 and 21 respectively, also in Appendix A.4. Level five has 2 clauses and a refutation was found on level six. The total number of clauses was 18 of which 9 were retained and 9 discarded.

3.5.2 Tautologies

A tautology is a clause that is valid under all interpretations. A clause is a disjunction of literals therefore a clause is a tautology if and only if it is *true* or if it contains a complementary pair of literals (Leitsch 1997). The clause $P(f(x)) \vee Q(y) \vee \neg P(f(x))$ is an example of a tautology. This is because either $P(f(x))$ or its complement will be valid regardless of the interpretation that is used.

The tautology rule states that a clause D that is a tautology can be removed from a clause set S resulting in set $S - \{D\}$. Since D is satisfied by all interpretations it follows that an interpretation satisfies S if and only if it satisfies $S - \{D\}$ (Eisinger & Ohlbach 1993). The two sets S and $S - \{D\}$ are therefore logically equivalent as far as a subsequent proof attempt is concerned.

Clauses that are subsumed are redundant, and this redundancy depends on the other clauses that are present. A tautology is redundant independently of any other clauses that may be present. As an algorithmic test, tautology elimination is therefore simpler and faster than subsumption since the algorithm only needs to check whether or not the clause contains a complementary pair (Chang & Lee 1973).

The tautology rule is in most cases refutation complete (Leitsch 1997). Proofs of the completeness and incompleteness of the tautology rule in combination with some resolution refinements are provided by (Leitsch 1997). Tautology elimination is for example complete when used as pre-processing or in combination with subsumption or

hyperresolution. An example of incompleteness is the combination of forward subsumption with lock resolution (Leitsch 1997).

3.6 Theory Resolution

Any unsatisfiable first-order predicate formula can be refuted by resolution (Robinson 1965a). Resolution is therefore a universal rule of inference. A disadvantage of this generality is that resolution does not have any semantic knowledge of the symbols it manipulates. As a result domain specific knowledge and algorithms cannot be employed to perform macro inference steps. To perform simple addition for example the axioms of number theory must be specified and the correct resolution steps must then be selected to simulate the addition of two numbers. The search space therefore tends to become very big for resolution steps that appear to be trivial.

Tailored inference rules that incorporate the semantic knowledge of a theory have been proposed for specific cases thereby eliminating the need to add the axioms of the relevant theory. These macro inference rules have the advantage of reducing the length of proofs as well as the size of the search space. General theory resolution that incorporates these special cases was proposed by Stickel (1985). A good overview of theory resolution is provided by Eisinger and Ohlbach (1993).

The equality predicate was one of the first symbols for which special inference rules were developed (Eisinger & Ohlbach 1993). One reason for this is that many theorems can be specified more elegantly using the equality relation (Chang & Lee 1973). This is especially the case for mathematical reasoning (Quaife 1992b).

In this section we shall look at how the equality predicate is used in proofs and which axioms must be included to make the decision procedure complete. Paramodulation is thereafter discussed as a special case of theory resolution applied to the equality predicate. Lastly demodulation is discussed.

3.6.1 The Equality Predicate

An equality predicate by convention starts with EQUAL (Wos *et al*. 1992), using prefix notation. For example, to state that a = b the clause EQUAL(a,b) is provided. However, to make clauses more readable the equals symbol '=' will sometimes be used, infix notation instead.

Through inspection we can see that the following clause set is unsatisfiable:

C_1 =	P(a)
C_2 =	EQUAL(a,b)
C_3 =	$\neg$P(b)

However, the unsatisfiability of the above set of clauses cannot be formally proved using only the resolution techniques introduced so far. There is equality involved which is only complete if a number of equality axioms are included in the proof attempt. These extra axioms are (Eisinger & Ohlbach 1993):

$\forall x\ (x = x)$	Reflexivity
$\forall x,y\ (x = y \rightarrow y = x)$	Symmetry
$\forall x,y,z\ (x = y \wedge y = z \rightarrow x = z)$	Transitivity
$\forall x_1,\ldots,x_n,y_1,\ldots,y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n)$	Substitution
$\forall x_1,\ldots,x_n,y_1,\ldots,y_n\ (x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1,\ldots,x_n) \rightarrow P(y_1,\ldots,y_n)$	Substitution

The above substitution rules must be added for every function and predicate symbol appearing among the formulae.

These equality axioms formalise the 'identity of indiscernibles' principle which states that if there is no way of telling two entities apart then they are the same. This principle is also known as Leibniz's law (Eisinger & Ohlbach 1993).

The first example above in the current subsection can now be refuted by adding the relevant substitution clause for the predicate symbol P:

C_1 =	P(a)	
C_2 =	EQUAL(a,b)	
C_3 =	$\neg$P(b)	
C_4 =	$\neg$EQUAL(x,y) $\vee$ $\neg$P(x) $\vee$ P(y)	Axiom of substitution applied to P
C_5 =	$\neg$P(a) $\vee$ P(b)	Resolvent of C_2 and C_4
C_6 =	P(b)	Resolvent of C_1 and C_5
C_7 =	$\square$	Resolvent of C_3 and C_6

3.6.2 Paramodulation

The above axioms provide a logically complete treatment of equality but their use slows down a proof attempt and makes it inefficient (Quaife 1992b). Numerous redundant clauses are generated (Nieuwenhuis & Rubio 2001) resulting in a search space that is rather large for relatively simple problems (Eisinger & Ohlbach 1993).

Many solutions have been proposed (Chang & Lee 1973) of which paramodulation became the most accepted. Paramodulation was introduced by G.A. Robinson and L. Wos in 1969 (Robinson & Wos 1969). The name is derived from the close relationship it has with demodulation (Wos *et al*. 1992). Demodulation is discussed in Section 3.6.3.

Paramodulation can formally be defined as (Chang & Lee 1973):

Definition 3.25

Let C_1 and C_2 be two clauses with no variables in common. C_1 has the form $L[t] \vee M_1 \vee \ldots \vee M_i$ for $i \geq 0$ where $L[t]$ is a literal containing the term t. C_2 has the form $(r = s) \vee N_1 \vee \ldots \vee N_j$ for $j \geq 0$. If θ is a most general unifier of t and r, then we can infer clause

$$C = L\theta[s\theta] \vee M_1\theta \vee \ldots \vee M_i\theta \vee N_1\theta \vee \ldots \vee N_j\theta$$

where $L\theta[s\theta]$ is obtained by replacing a single occurrence of $t\theta$ in $L\theta$ by $s\theta$.

C is called a binary paramodulant of C_1 and C_2. C_1 and C_2 are called the parent clauses of C. The literals L and r = s are called the literals paramodulated upon.

We also say the paramodulation is applied from C_2 into C_1. As a result C_1 is called the 'into' clause and C_2 the 'from' clause.

The paramodulation inference rule can also be represented as (Van der Poll 2000):

$L[t], M_1, \ldots, M_i$	'into' clause
$r = s, N_1, \ldots, N_j$	'from' clause

$$L\theta[s\theta], M_1\theta, \ldots, M_i\theta, N_1\theta, \ldots, N_j\theta$$

where all symbols have the same meaning as in Definition **3.25** above.

An E-model of a set S of clauses is a model of the equality axioms that also satisfies the set S (Chang & Lee 1973). Paramodulation is sound in that if C is a paramodulant of any two clauses in S then any E-model of S is also an E-model of $S \cup \{C\}$ (Eisinger & Ohlbach 1993).

A set S of clauses is E-unsatisfiable if and only if has no E-model otherwise S is called E-satisfiable (Chang & Lee 1973). The use of the paramodulation rule together with resolution is refutation complete for any set of E-unsatisfiable clauses that contains the