Contents

1 INTRODUCTION 1

1.1 MOTIVATION 1

1.2 RESEARCH QUESTION 2

1.3 HYPOTHESIS 2

1.4 APPROACH 3

1.5 DISSERTATION LAYOUT 3

2 INTRODUCTION TO MATHEMATICAL SET THEORY 5

2.1 ZERMELO-FRAENKEL SET THEORY 5

2.1.1 Extensionality Axiom 7

2.1.2 Empty set Axiom 7

2.1.3 Pairing Axiom 7

2.1.4 Union Axiom 8

2.1.5 Subset Axiom 8

2.1.6 Power set Axiom 9

2.1.7 Infinity Axiom 9

2.1.8 Axiom of replacement 10

2.1.9 Axiom of foundation or regularity 10

2.1.10 Axiom of choice 11

2.1.11 Example 11

2.2 LIMITATIONS OF ZF AXIOMS IN AUTOMATED THEOREM PROVING 12

2.3 SUMMARY 13

3 RESOLUTION 14

3.1 DECIDABILITY AND HERBRAND'S UNIVERSE 14

3.2 RESOLUTION 16

3.2.1 Clausal Form 16

3.2.2 Resolution in Propositional Logic 18

3.2.3 Resolution in First-order Predicate Logic 22

3.3 EFFICIENCY ENHANCEMENTS 29

3.4 REFINEMENTS 30

3.4.1 Linear Resolution 31

3.4.2 Semantic Resolution 37

3.4.3 UR-resolution 44

3.4.4 Hyperresolution 46

3.4.5 Set-of-Support strategy 50

3.5 REDUNDANCY AND DELETION 52

3.5.1 Subsumption 52

3.5.2 Tautologies 54

3.6 THEORY RESOLUTION 55

3.6.1 The Equality Predicate 56

3.6.2 Paramodulation 57

3.6.3 Demodulation 60

3.7 HEURISTICS 63

3.8 SUMMARY 66

4 AUTOMATED THEOREM PROVERS 67

4.1 VAMPIRE 68

4.2 GANDALF ... 72

4.3 SUMMARY ... 75

5 EVALUATION OF SET-THEORETIC REASONING HEURISTICS ... 77

5.1 EQUALITY VERSUS EXTENSIONALITY ... 78

5.2 NESTED FUNCTORS ... 80

5.3 DIVIDE-AND-CONQUER ... 82

5.4 EXEMPLIFICATION ... 84

5.5 MULTIVARIATE FUNCTORS ... 86

5.6 INTERMEDIATE STRUCTURE ... 88

5.7 ELEMENT STRUCTURE ... 90

5.8 REDUNDANT INFORMATION ... 91

5.9 SEARCH-GUIDING ... 93

5.10 RESONANCE ... 96

5.11 TUPLE CONDENSE ... 98

5.12 SUMMARY AND CONCLUSIONS ... 99

6 AN ORDER MANAGEMENT SYSTEM IN Z ... 102

6.1 PROBLEM STATEMENT ... 102

6.2 CONCEPTUAL MODEL ... 103

6.3 THE Z SPECIFICATION LANGUAGE ... 104

6.4 SPECIFYING CLASSES AND THEIR ATTRIBUTES ... 105

6.5 SPECIFYING ASSOCIATIONS ... 107

6.6 SPECIFYING ASSOCIATION CLASSES ... 110

6.7 SPECIFYING OPERATIONS ... 112

6.7.1 Create Operation ... 112

6.7.2 Read Operation *113*

6.7.3 Update Operation *113*

6.7.4 Delete Operation *115*

6.7.5 ProcessOrder *116*

6.8 TOTAL OPERATIONS 117

6.9 SPECIFYING AGGREGATION AND COMPOSITION 118

6.10 SPECIFYING INHERITANCE 119

6.11 SPECIFYING THE SYSTEM STATE 120

6.12 SPECIFYING AN INITIAL STATE 121

6.13 PROOF OBLIGATIONS ARISING FROM THE SPECIFICATION 122

6.13.1 Initialisation Theorem *122*

6.13.2 Precondition Simplification *122*

6.13.3 After State Type *124*

6.13.4 Total Operations *125*

6.13.5 Operation Interaction *126*

6.13.6 Contents of a Set *127*

6.13.7 State Invariant *128*

6.14 CONCLUSION 128

7 DISCHARGING CASE STUDY PROOF OBLIGATIONS 130

7.1 CONVERSION OF Z TO FIRST-ORDER LOGIC 130

7.2 DISCHARGING OF PROOF OBLIGATIONS 132

7.2.1 CreateProduct Invariant *132*

7.2.2 CreateProduct is Total *134*

7.2.3 ProcessOrder set contents *138*

7.2.4 CreateDeleteItem leaves state unchanged *140*

7.2.5 After State Type of CancelOrder *144*

7.3 CONCLUSION 147

8 SUMMARY AND CONCLUSIONS 149

8.1 CONTRIBUTIONS OF THIS DISSERTATION 149

8.2 FUTURE WORK 150

APPENDIX A – RESOLUTION DEDUCTIONS OF THE FARMER, WOLF, GOAT AND CABBAGE (FWGC) PUZZLE 152

A.1 A POSSIBLE REFUTATION DEDUCTION OF THE FWGC PUZZLE 152

A.2 LEVEL SATURATION METHOD DEDUCTION OF FWGC PUZZLE 155

A.3 UR-RESOLUTION DEDUCTION OF FWGC PUZZLE 158

A.4 ILLUSTRATION OF SET-OF-SUPPORT STRATEGY OF FWGC PUZZLE 159

A.5 SET-OF-SUPPORT STRATEGY WITH PREDICATE ORDERING 160

A.6 SET-OF-SUPPORT STRATEGY WITH SUBSUMPTION 161

APPENDIX B - THEOREM PROVERS EVALUATED 164

APPENDIX C – SAMPLE REASONER OUTPUT 167

C.1 VAMPIRE 167

C.2 GANDALF 171

APPENDIX D – Z CASE STUDY OF ORDER PROCESSING SYSTEM 174

D.1 GIVEN SETS (BASIC TYPES) 174

D.2 PRODUCT 174

D.3 ORDER 176

D.4 ITEM 178

D.5 CUSTOMER 179

D.6 COMPANY 181

D.7 PERSON 182

D.8 SYSTEM 183

APPENDIX E – REASONER INPUTS FOR PROOF OBLIGATIONS 185

E.1 CREATEPRODUCT INVARIANT 185

E.2 AFTER STATE TYPE OF CANCELORDER 187

APPENDIX F – VALIDATING REASONING HEURISTICS USING NEXT-GENERATION THEOREM-PROVERS 195

REFERENCES 206

INDEX 213

List of Published Papers

The following paper was published during the research reported on in this dissertation:

Steyn, P.S., Van der Poll, J.A., 2007. Validating Reasoning Heuristics Using Next-Generation Theorem-Provers, In J.C. Augusto, J. Barjis, U. Ultes-Nitsche, eds., *Proceedings of the Fifth Workshop on Modelling Simulation Verification and Validation of Enterprise Information Systems (MSVVEIS'07)*, pp. 43-52, Funchal, Madeira, Portugal, June 2007.

This article appears as Appendix F.

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:

$$0 = \varnothing$$

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

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

$$3 = \{\varnothing, \{\varnothing\}\} \cup \{\{\varnothing, \{\varnothing\}\}\} = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}$$

2.1.8 Axiom of replacement

For each formula $\varphi(x, y)$ not containing B, the following is an axiom (Enderton 1977):

$$[\forall x \forall y_1 \forall y_2(\varphi(x, y_1) \wedge \varphi(x, y_2) \rightarrow y_1 = y_2)] \rightarrow$$
$$\forall A \exists B \forall y[y \in B \leftrightarrow \exists x(x \in A \wedge \varphi(x, y))]$$

This axiom states that if A is a set and the formula φ is a functional mapping, then there exists a set B that is the image of A under φ (Nerode & Shore 1997). The functional property of φ is asserted by the hypothesis of the axiom. The consequent of the axiom states that B is then the set:

$$B = \{y \mid \exists x(x \in A \wedge \varphi(x, y))\}$$

As an example we can show that if the set A exists, then the set B of all power sets of members of A also exists. That is, $B = \{\mathbb{P}(a) \mid a \in A\}$. This is done by taking $\varphi(x, y)$ to be $y = \mathbb{P}(x)$.

2.1.9 Axiom of foundation or regularity

Every non-empty set A contains an element disjoint from A (Enderton 1977).

$$\forall A[A \neq \varnothing \rightarrow \exists x(x \in A \wedge x \cap A = \varnothing)]$$

The axiom of foundation restricts set theory to sets in which the elements of a set must be known or must have been constructed before the set itself can be realised. Some of the consequences of this axiom are (Enderton 1977, Nerode & Shore 1997):

- No set can be a member of itself.
- There exist no sets x and y such that $x \in y$ and $y \in x$.
- There exists no infinite descending sequence of sets e.g. $\ldots \in f(2) \in f(1) \in f(0)$, where f is a function with the domain of the natural numbers.

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 $S = \{\neg a \vee b, a, \neg c\}$.

The following deductions can then take place:

$C_1 =$	$\neg a \vee b$	Clause in S
$C_2 =$	a	Clause in S
$C_3 =$	$\neg c$	Clause in S
$C_4 =$	b	Resolvent of C_1 and C_2

No further application of the resolution rule is possible and the empty clause was not deduced, therefore S is satisfiable, e.g. a = true, b = true and c = false.

Now let S be the following set of clauses:

$S = \{\neg a \vee b, a, \neg c, \neg b \vee c\}$

The following deductions can then take place:

$C_1 =$	$\neg a \vee b$	Clause in S
$C_2 =$	a	Clause in S
$C_3 =$	$\neg c$	Clause in S
$C_4 =$	$\neg b \vee c$	Clause in S
$C_5 =$	b	Resolvent of C_1 and C_2
$C_6 =$	c	Resolvent of C_4 and C_5
$C_7 =$	$\square$	Resolvent of C_3 and C_6

The empty clause was deduced from S and therefore S is unsatisfiable.

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).

Implementations of binary resolution are able to solve much more complex problems. It is however still not efficient enough to solve everyday mathematical problems. One of the problems is that of the unbounded generation of resolvents. Another problem is the presence of redundant clauses and literals. Robinson acknowledged these problems and proposed three principles that automated implementations should employ when searching for a refutation (Robinson 1965a). These principles are those of purity, subsumption and replacement. The reasoning steps taken by binary resolution are also very small (Quaife 1992b) and result in a high number of unnecessary resolvents (Leitsch 1997). Robinson has also seen this as a problem and published a paper on hyper-resolution (Robinson 1965b) that uses more than two clauses simultaneously in a reasoning step. Hyper-resolution is discussed in Section 3.4.4.

Numerous other techniques have been proposed to reduce the search space. Leitsch (1997) lists three categorises of techniques: refinements of resolution, redundancy tests and heuristics. Another category is theory resolution (Eisinger & Ohlbach 1993).

We shall expand on Leitsch's treatment of heuristics and look at it from two perspectives. The first is that of the automatic deduction implementation. The order in which derivations are generated may have a significant impact on the cost of the search. One such heuristic could be to give preference to deductions containing smaller clauses. The second perspective is that of the problem specifier. There are usually many different ways to model a problem in first-order logic. The problem specification provides the initial set of clauses and therefore could have a significant impact on the search space (Van der Poll & Labuschagne 1999, Van der Poll 2000, Wos *et al.* 1992).

The next sections will be dedicated to exploring some of these techniques.

3.4 Refinements

A technique X is a refinement of technique Y if the possible resolution deductions from X are a subset of those of Y. Since X has fewer deductions, the search space is smaller.

A possible implementation of resolution is known as the level-saturation method (Chang & Lee 1973). The first level S_0 is the initial set S of clauses. The resolvents of clauses in

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

being resolved with each other. However, the resolution would be on SAFE(fh, gh, ch, wh) which does not have the largest predicate symbol in clause C_1 based on the chosen predicate ordering. Clauses C_1 and C_9 can therefore not be resolved under the specified ordering. If the predicate ordering was chosen the other way round, then the resolution would have been allowed.

3.4.2.3 The Clash

The final concept of semantic resolution that we introduce is the clash (Slagle 1967). To illustrate the concept, consider the following clauses from the farmer, goat, cabbage and wolf puzzle:

C_3 =	$\neg S(fh, gh, y, z) \vee \neg SAFE(fh, gh, y, z) \vee \neg SAFE(fa, ga, y, z) \vee S(fa, ga, y, z)$	Farmer takes goat across
C_9 =	SAFE(fh, gh, ch, wh)	
C_{15} =	SAFE(fa, ga, ch, wh)	
C_{19} =	S(fh, gh, ch, wh)	Start state
	Take goat across	
C_{21} =	$\neg SAFE(fh, gh, ch, wh) \vee \neg SAFE(fa, ga, ch, wh) \vee S(fa, ga, ch, wh)$	Resolvent of C_3 and C_{19} Unifier {ch/y, wh/z}
C_{22} =	$\neg SAFE(fa, ga, ch, wh) \vee S(fa, ga, ch, wh)$	Resolvent of C_9 and C_{21}
C_{23} =	S(fa, ga, ch, wh)	Resolvent of C_{15} and C_{22}

Clauses C_{21} and C_{22} were intermediate resolvents to allow the resolution of clause C_{23}. This is just one way of generating clause C_{23}. By using the level-saturation method (Section 3.4), clause C_{23} would occur more than once via some other intermediate resolvents. Some of the other ways that duplicates of clause C_{23} could be generated are:

	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).

In the next section we discuss an important advancement in the automated reasoning arena namely the use of a set-of-support.

3.4.5 Set-of-Support strategy

The set-of-support strategy is a widely used and fairly successful restriction strategy (Eisinger & Ohlbach 1993). It was proposed by Wos, Robinson and Carson (Wos 1965) in 1965. Similar to hyperresolution, the set-of-support strategy is a special case of semantic clash resolution (Slagle 1967) based on the interpretation that is used.

The input to a refutation-based proof attempt typically consists of a set of axioms, theorems and a negated conclusion. The set of axioms and theorems should be satisfiable and therefore a refutation should involve the negated conclusion. The set-of-support strategy takes advantage of this general form of refutation-based proofs by preventing clauses from the set of axioms and theorems to be resolved with each other (Eisinger & Ohlbach 1993). The more general case would be to choose any satisfiable subset of the initial clause set. This is the "unsupported" set – no resolutions among its members are allowed. The complement of the "unsupported" set is the "supported" set or *set-of-support*. Any resolution must include a "supported" clause from the set-of-support. The resolvent is also "supported". It therefore prevents the expansion of a set of consistent clauses in a proof procedure where the aim is to find a contradiction (Wos *et al*. 1992).

Linear resolution (Section 3.4.1) is also compatible with the set-of-support strategy (Chang & Lee 1973). That is, its refutation completeness is preserved. A completeness proof is provided by Nerode and Shore (1997).

The set-of-support strategy is defined as (Nerode & Shore 1997, Chang & Lee 1973):

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

reflexivity axiom (Plaisted 1993). The reflexivity axiom is required to be able to refute the E-unsatisfiable set $\{\neg(a = a)\}$.

There are various refinements of paramodulation that preserve completeness when used with resolution. Some of these include hyper, unit, input and linear paramodulation (Chang & Lee 1973) as well as the set-of-support strategy (Wos *et al*. 1992).

The following example shows that the two clauses $x + 0 = x$ and $P((g(y) + 0) + f(b))$ implies the clause $P(z + f(b))$ using paramodulation:

C_1 =	EQUAL(sum(x,0), x)	
C_2 =	P(sum(sum(g(y),0), f(b)))	
C_3 =	¬P(sum(z, f(b)))	
C_4 =	P(sum(g(y), f(b)))	Paramodulant from C_1 into C_2 Unifier {g(y)/x}
C_5 =	□	Binary resolvent of C_3 and C_4

Paramodulation aids in reducing the search space of problems containing equalities. This is because no unnecessary resolution steps can occur with and between the equality axioms (Eisinger & Ohlbach 1993). Unfortunately paramodulation still generates many irrelevant clauses (Quaife 1992b). This is especially the case when the terms t and r in Definition **3.25** above are variables. Demodulation (Wos 1967) is a technique that helps to restrict the number of inferences. Demodulation is discussed in the following section.

Ordered paramodulation (Nieuwenhuis & Rubio 2001) is another restriction technique that only performs replacements of large terms by smaller ones with respect to some ordering. Knuth-Bendix completion (Plaisted 1993) contained the first instances of ordered paramodulation. It is often used successfully in conjunction with other resolution techniques. It provides an algorithm for a class of equational theories that permits the

computation of a set of rewrite rules sufficient to check the truth of every equation of the theory by requiring that equal terms reduce to the same normal form (Quaife 1992b).

3.6.3 Demodulation

Equality relations in many fields of mathematics often tend to be very general with expressions like $(x^{-1})^{-1} = x$. In the presence of the expression $a \cdot b = c$, equality resolutions mechanisms like paramodulation will infer a set of very closely related expressions like $(a^{-1})^{-1} \cdot b = c$. The retention of all these related expressions causes an unacceptable high growth of the search space.

Demodulation was one of the proposed solutions to this uncontrolled repeated application of a given equality (Wos 1967). The aim of demodulation is to simplify the presentation of information. Such simplification is achieved by applying a transformation to relevant clauses that replaces α in some clause by β where $\beta = \alpha$ and β is simpler than α. The original clause is then discarded (see Example **3.10**).

An application of demodulation is defined as (Wos *et al*. 1992):

> **Definition 3.26**
>
> Let C_1 and C_2 be two clauses. 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 is a positive unit equality clause $r = s$ that has been designated to be used to rewrite expressions. C_2 is called a demodulator. Let θ be a substitution such that $r\theta = t$. A clause C is then obtained by replacing t in L with $s\theta$:
>
> $$C = L[s\theta] \vee M_1\theta \vee \ldots \vee M_i\theta$$
>
> Clause C_1 is then discarded and replaced by C. Clause C is called a demodulant of C_1.

Note that one-way matching is used instead of unification. Also, in the above definition, if $s\theta = t$ then a demodulant can be obtained by replacing t in L with $r\theta$. It is however a

common convention to consider only the first argument of the demodulator C_2 (Wos *et al*. 1992).

The application of demodulation can also be represented as (Van der Poll 2000, Quaife 1992b):

$$\frac{L[t], M_1, \ldots, M_i \qquad r = s}{L[s\theta], M_1\theta, \ldots, M_i\theta}$$

where all symbols have the same meaning as in the above Definition **3.26**, e.g. $r\theta = t$.

Demodulation is then defined as (Wos 1967):

Definition 3.27

Let W be a set of positive equality unit clauses. Demodulation is the process of replacing a clause C by a demodulant D of A relative to W. D is obtained by generating a sequence $C_1, \ldots, C_k$ such that $C = C_1$, $D = C_k$, C_{i+1} is a demodulant of C_i as defined above using a demodulator in W for $1 \leq i < k$, C_{i+1} has strictly fewer symbols than C_i and C_k has no demodulant relative to W with fewer symbols.

Example 3.10

In the following example the demodulation rule is applied to clause C_1 using equality W_1 to obtain the simpler demodulant C_2.

$W_1 =$	EQUAL(sum(x,0), x)	An available equality.
$C_1 =$	$P(sum(1,0)) \vee \neg Q(c)$	The given clause.
$C_2 =$	$P(1) \vee \neg Q(c)$	Demodulant of W_1 and C_1 Substitution {sum(1,0)/1}

Demodulation typically applies all demodulators in the system to all relevant terms of any newly generated clause until the clause cannot be simplified any further (Quaife 1992b). In the case where a new demodulator is generated, all the previously retained clauses can be examined for possible demodulation with the new demodulator. This is called back demodulation (Wos *et al*. 1992).

Demodulation and paramodulation are similar in that both cause an equality substitution with a successful application. Demodulation also has equivalent 'from' and 'into' clauses. In fact, paramodulation's name was derived from the close relationship it has with demodulation (Wos *et al*. 1992). Demodulation and paramodulation differs in several aspects though (Wos *et al*. 1992). Unlike paramodulation, demodulation requires the equality literal to be in a unit clause. Demodulation allows for variable replacement only in the argument of the equality literal while paramodulation allows it also in the term into which the substitution is being attempted. Paramodulation retains the parent clauses and the paramodulant whereas demodulation discards the original clause into which the substitution took place.

Example 3.11

The following example from group theory illustrates the simplification of the expression $(e\cdot(e\cdot(a^{-1})^{-1}))\cdot b = c$ to $a\cdot b = c$. Suppose P(x,y,z) represents $x\cdot y = z$, function f(x,y) represents $x\cdot y$ and function g(x) represents x^{-1}. Two equalities are available: $e\cdot x = x$, $(x^{-1})^{-1} = x$.

W_1 =	EQUAL(f(e,x),x)	An available equality.
W_2 =	EQUAL(g(g(x)),x)	An available equality.
C_1 =	P(f(e,f(e,g(g(a)))),b,c)	The given clause.
C_2 =	P(f(e,g(g(a))),b,c)	Demodulant of W_1 and C_1 Substitution {g(g(a))/x}

C_3 =	P(g(g(a)),b,c)	Demodulant of W_1 and C_2 Substitution {g(g(a))/x}
C_4 =	P(a,b,c)	Demodulant of W_2 and C_3 Substitution {a/x}

The following section introduces a further aid to the resolution process, namely heuristics to be used in the search for a proof.

3.7 Heuristics

A general definition of a heuristic is an informal, judgmental knowledge of an application area that constitutes the rules of good judgement in the field (Turban & Frenzel 1992). For example, a stock trading heuristic to reduce risk could be to not buy stocks whose price-to-earnings ratio is larger than 10.

The order in which derivations are generated during the search for a refutation has a strong influence on the cost of the search (Leitsch 1997). A simple but effective heuristic would be to give preference to deductions containing smaller clauses. Smaller clauses could mean clauses containing fewer literals or clauses of smaller term depth, i.e. fewer levels of nesting.

Different types of clause complexity may be combined in a weight function, resulting in a preference for clauses having smaller weight. The weight of a clause is determined primarily by the number of literals in the clause or the term depth of the clause. Other examples of weight function parameters could be to give priorities to variables, function symbols, predicate symbols and terms. An automated reasoning program that employs weighting chooses the clauses with the most favourable weight (e.g. a smaller weight). Using weighting in this manner is referred to as a direction strategy (Wos *et al*. 1992). Weighting can also be used as a restriction strategy. In this case it will cause new clauses whose weight is above some threshold to be deleted.

The use of weighting as a direction strategy only influences the ordering of derivations and therefore does not influence the completeness of the proof procedure. This heuristic differs from a restriction strategy heuristic that could make the proof procedure incomplete. For example, a restriction of producing only clauses with fewer than four literals (say) is an incomplete refinement (Leitsch 1997).

Another heuristic is the selection of the inference mechanism for a specific class of problem. An automated theorem prover could elect to use paramodulation with demodulation when identifying the use of equality. It could identify the problem as a Horn class problem and therefore use the more efficient unit and input resolution strategies that would otherwise be incomplete. The reasoner can also change its strategy when detecting that a specific theory is relevant to the problem for example set theory as is the case in this dissertation. This heuristic could lead to incompleteness for example when forward subsumption is selected in combination with lock resolution (Leitsch 1997).

All the heuristics discussed above are used by modern automated reasoners. Theorem provers also make many of these parameters available to the user for configuration. Some of these parameters include weighting, main loop settings, inference rules, restriction strategies, time and memory limits, maximum number of clauses to retain (McCune 2003, Tammet 1997, Voronkov 2005).

The user can therefore apply his or her own heuristics in an attempt to guide the theorem prover to find a proof by adjusting these parameters. For example the weighting strategy is suggested as a user heuristic by Van der Poll and Labuschagne (1999) whenever the set-of-support contains an equality literal. Various configurations of the weighing strategy are also suggested by Quaife (1992b) when working with problems involving sets and Tarski's geometry. He also proposes that one should not use binary resolution as an inference rule (Quaife 1992b). Wos also provides heuristics with regards to inference rules and strategy selection (Wos *et al*. 1992).

A further heuristic that can be applied by a user is that of problem representation (Wos *et al*. 1992, Van der Poll & Labuschagne 1999). The choice of problem representation can

play an essential role in the theorem prover's chance of succeeding and the time it takes to succeed. Wos refers to the heuristics of problem representation as an art because of its subtleties and the difficulty of characterising the essential concepts that constitute a problem representation. Various problem-representation heuristics are provided by Wos (1992), for example that unit clauses, shorter clauses and equality predicates should be given preference.

Quaife provides heuristics for problem representation in set theory (Quaife 1992a). One of these is to replace the axiom of extensionality that contains a Skolem function when clausified with an equivalent but simpler formula. The Extensionality axiom (Section 2.1.1) states that $\forall A \forall B(\forall x(x \in A \leftrightarrow x \in B) \rightarrow A = B)$ where A and B are sets and x represents elements of these sets (Enderton 1977). The axiom can be replaced with the equivalent formula $\forall A \forall B(A = B \leftrightarrow A \subseteq B \wedge B \subseteq A)$ that requires no Skolem function.

Van der Poll and Labuschagne have done extensive research on heuristics that could be applied to the representation of problems in set theory (Van der Poll & Labuschagne 1999, Van der Poll 2000). In their work they present a set of problem frames or patterns that captures the properties of a specification that can compromise the efficiency of a theorem prover. For each of these problem frames he provides one or more heuristics that can alleviate the problem as well as an example that illustrates its effect. Some of these heuristics are:

- Use the principle of extensionality to replace equality in the set-of-support. For example, if C=D appears in the set-of-support it can be replaced by $(\forall x)(x \in C \leftrightarrow x \in D)$.

- Perform two separate subset proofs whenever the problem at hand requires the theorem prover to prove the equality of two sets. For example, the above goal of C = D can be replaced by a proof showing that $(\forall x)(x \in C \rightarrow x \in D)$ and thereafter a proof that $(\forall x)(x \in C \rightarrow x \in C)$.

- Avoid if possible the use of nested function symbols in definitions.

- Avoid the inclusion of information that is not obviously necessary in the input to the theorem prover.

3.8 Summary

This chapter gave a brief overview of decidability and Herbrand's universe. The next section presented resolution as an efficient refutation procedure. It was shown how a first-order statement may be converted to clausal form which is required for resolution. Resolution in propositional and predicate logic was described. The rest of the chapter focused on efficiency enhancements for resolution. These enhancements were classified into four categories: resolution refinements, redundancy tests, theory resolution and heuristics. The refinements that were addressed were linear resolution, semantic resolution, UR-resolution, hyperresolution and set-of-support strategy. Redundancy tests included subsumption and tautologies. Theory resolution covered paramodulation and demodulation. Finally some heuristics that can be automatically applied by the theorem prover and those that must be applied by the specifier were discussed.

Chapter 4

Automated Theorem Provers

The VdPL heuristics (Section 1.3) were arrived at with the aid of Otter, a first-order automated resolution-based reasoner (McCune 2003). The aim of this work is to determine to what extent other state-of-the-art reasoners can benefit from the said heuristics. The Vampire and Gandalf theorem provers will be used for this purpose and a motivation for using these reasoners is presented. An overview of each of Vampire and Gandalf as well as an example of the input to each and the resulting proofs are given.

Appendix B gives a list of some of the theorem provers that were evaluated for this work. The list is limited to automated theorem provers for first-order logic. Within this list only resolution-based theorem provers were considered since Otter is also based on resolution.

The selection of possible reasoners was further reduced by considering individual performances in the CADE ATP System Competitions (CASC) (Pelletier *et al*. 2002, Sutcliffe & Suttner 2006). CASC is an annual competition that evaluates the performance of automated theorem provers using classical first-order logic. It has 6 main divisions based on the types of problems.

Vampire is our first choice and will be the primary reasoner used in this work. Vampire fits the profile for two reasons. The first is because of its consistent success at the annual CASC competitions. Vampire came first in two of its divisions every year from 2002 to 2007. In 2000 and 2001 it came first in one of the divisions. The second reason stems from the fact that Vampire has solved more set-theoretic problems than any of the other competing provers in the period from 2002 to 2007 across all divisions involving these problems.

Vampire may therefore undoubtedly be considered a state-of-the-art reasoner for set-theoretic problems. If we can show that Vampire benefits from the VdPL heuristics, then it is plausible that other reasoners will benefit from these heuristics as well.

Gandalf was chosen as the secondary reasoner since it was the only other automated, resolution-based reasoner that has recently won in a CASC division.

The rest of this section will be used to describe Vampire and Gandalf in more detail.

4.1 Vampire

Vampire was developed in the Computer Science Department of the University of Manchester by Andrei Voronkov previously together with Alexandre Riazanov. Vampire is coded in C++. The version that was used in this work is version 7.0.

Vampire is described in a number of sources, namely, Voronkov (2001, 2005), Riazanov (2003) and Riazanov & Voronkov (2002, 2001). It is a resolution-based system for fully automatic theorem proving in first-order logic with equality. It implements the calculi of ordered binary resolution and superposition for handling equality. Superposition is a calculus for reasoning in equational first-order logic that combines concepts from first-order resolution with ordering-based equality handling as developed in the context of unfailing Knuth-Bendix completion (Nieuwenhuis & Rubio 2001).

Vampire supports the inference rules of ordered binary resolution (Bachmair & Ganzinger 2001) with negative selection, superposition and a special form of splitting. The splitting rule and negative equality splitting are simulated by the introduction of new predicate definitions and dynamic folding of such definitions.

Vampire makes use of a number of redundancy control and simplification techniques. These include forward and backward subsumption, subsumption resolution, tautology deletion, forward and backward demodulation, rewriting by ordered unit equalities, basic restrictions and irreducibility of substitution terms. The reduction orderings used are the standard Knuth-Bendix ordering and a special non-recursive version of the Knuth-Bendix ordering.

A number of efficient indexing techniques are used to implement the major operations on sets of terms and clauses. Run-time algorithm specialisation is used to accelerate some costly operations for example checks on ordering constraints. Run-time algorithm specialisation originated with Vampire and is described in Riazanov (2003).

Vampire is a saturation-based theorem prover. It implements three different saturation algorithms that can be selected for its main loop for inferring and processing clauses. The three saturation algorithms are an Otter loop with or without the Limited Resource Strategy and the Discount loop. These algorithms belong to the class of given-clause algorithms.

The Otter algorithm used in Vampire is a slight modification of the saturation algorithm used in the Otter reasoner (McCune 2003). A simplified version of the algorithm used by Vampire is given below:

```
input: init: set of clauses;
var active, passive, unprocessed: set of clauses
var given, new: clause;
active := ∅;
unprocessed := init;
loop
  while unprocessed ≠ ∅
    new := pop(unprocessed);
    if new = □ then return unsatisfiable;
    if retained(new) then                          (* retention test *)
      simplify new by clauses in active ∪ passive;(* forward simplification *)
      if new = □ then return unsatisfiable;
      if retained(new) then                        (* another retention test *)
        delete and simplify clauses in active and (* backward simplification *)
                                     passive using new;
        move the simplified clauses to unprocessed;
        add new to passive;
  if passive = ∅ then return satisfiable or unknown;
  given := select(passive);                        (* clause selection *)
  move given from passive to active;
  unprocessed := infer(given, active);             (* generating inferences *)
```

Clause selection in Otter is based on an age-weight ratio which is also known as the pick-given ratio in Otter. The retention test consists of deletion rules plus a weight test. The weight test discards any clause whose weight exceeds a user-defined limit, if specified.

The Limited Resource Strategy aims to improve the effectiveness of the Otter algorithm when a time limit is imposed. Usually when the Otter algorithm reaches the time limit

many clauses remain passive. This means that any computational resources that were used to generate, process and keep these clauses have been wasted. The aim of the Limited Resource Strategy is therefore to identify which passive clauses have little chance to be processed by the time limit and it then discards these clauses.

The set of passive clauses tends to become much larger than the number of active ones. As a result, its use in simplifying inferences slows down the proof search. The Discount algorithm, which is named after the theorem prover Discount (Denzinger *et al*. 1997) aims to solve this problem by not allowing passive clauses to be used at all. As a result any new clauses can be processed faster since only a small subset of all clauses is involved in simplifying inferences. A disadvantage of the algorithm is that a valuable clause might not be generated by a simplification inference, which would have been generated by the Otter algorithm. The Discount algorithm is given below:

```
input: init: set of clauses;
var active, passive, unprocessed: set of clauses
var given, new: clause;
active := ∅;
unprocessed := init;
loop
  while unprocessed ≠ ∅
    new := pop(unprocessed);
    if new = □ then return unsatisfiable;
    if retained(new) then                        (* retention test *)
      simplify new by clauses in active;         (* forward simplification *)
      if new = □ then return unsatisfiable;
      if retained(new) then                      (* another retention test *)
        delete and simplify clauses              (* backward simplification *)
            in active using new;
        move the simplified clauses to unprocessed;
        add new to passive;
  if passive = ∅ then return satisfiable or unknown;
  given := select(passive);                      (* clause selection *)
  simplify given by clauses in active;           (* forward simplification *)
  if given = □ then return unsatisfiable;
  if retained(given) then                        (* retention test *)
    delete and simplify clauses                  (* backward simplification *)
        in active using given;
    move the simplified clauses to unprocessed;
    add given to active;
    unprocessed := infer(given, active);         (* generating inferences *)
```

Various heuristics for Vampire's automatic mode have been derived from empirical data obtained on problems from the TPTP (Thousands of Problems for Theorem Provers) (Sutcliffe & Suttner 1998) problem library.

Vampire is divided into a kernel and a shell component. The kernel works only with clausal normal forms. The shell or pre-processor however can accept a problem in first-order logic syntax, clausify it and perform a number of useful transformations before passing the result to the kernel. The TPTP notation is used as input for Vampire. The advantage of the TPTP syntax is that it is widely used among theorem provers. There are also tools available to convert a problem specified using the TPTP notation into other notations for those theorem provers that do not accept the TPTP notation.

The following is an example TPTP input of a set-theoretic problem to show that $\mathbb{P}\{\{1\}\} = \{\varnothing,\{\{1\}\}\}$.

```
% A TPTP set-theoretic problem:
% Show that P{{1}} = {Empty,{{1}}}.

% Reflexivity
fof(reflexivity, axiom,
  ! [X] : X = X
).

% Extensionality
fof(extensionality, axiom,
  ! [A,B] : ( (![X]:(el(X,A) <=> el(X,B))) => (A = B) )
).

% Empty = {}
fof(empty, axiom,
  ~( ?[X]: el(X, empty) )
).

% A = {1}
fof(a_is_1, axiom,
  ![X]: ( el(X,a) <=> (X = 1) )
).

% B = {A}
fof(b_is_a, axiom,
  ![X]: ( el(X,b) <=> (X = a) )
).

% C = P(B)
fof(c_is_power_b, axiom,
  ![X]: ( el(X,c) <=> (![Y]: (el(Y,X) => el(Y,b)) ) )
).

% D = {Empty,{{1}}}
fof(d_is_empty_or_1, axiom,
  ![X]: ( el(X,d) <=> ( (X = empty) | (X = b) ) )
).

% Goal clause C = D
fof(c_is_d, conjecture,
  c = d
).
```

Vampire then finds a proof for $\mathbb{P}\{\{1\}\} = \{\emptyset,\{\{1\}\}\}$ in 0.2 seconds. The output appears in Appendix C.1. It also lists the proof steps. Vampire tries to make the proof as readable as possible and as a result the proof output makes up the largest part of the output. Finally some global statistics are provided for example the number of clauses generated, subsumed, discarded etc.

Note also that by default in the TPTP notation constants start with a lower-case symbol (e.g. empty above) while variables start with capitals. The documentation (i.e. comments) in the above example follow a more traditional approach, e.g. Empty = {}.

4.2 Gandalf

Gandalf was developed by Tanel Tammet (1997). It is a family of theorem provers for classical first-order logic, intuitionistic first-order logic and propositional linear logic. It also includes a finite model builder. These provers share large parts of their code. The name Gandalf is that of a powerful wizard in the famous fantasy books "The Hobbit" and "The Lord of the Rings" written by J.R.R. Tolkien (1966).

Gandalf is a resolution-based reasoner with equality and implements a large number of inferences and strategies. Some of these include binary-, unit- and hyperresolution, set-of-support, paramodulation, forward and back demodulation, Knuth-Bendix ordering, literal ordering, tautology elimination, forward and backward subsumption and limits on clause length and term depth.

Gandalf is written in Scheme (Dybvig 2003) and compiled to C by the Scheme-to-C compiler Hobbit which was also developed by Tammet (1997). The platforms under which Gandalf has been tested are Linux, Solaris and MS Windows using Cygwin. Gandalf is also optimised for handling problems where large numbers of long clauses are derived. It is freely available under the Gnu Public Licence. A commercial version called G is developed and distributed by Safelogic AB. This version contains numerous additions, strategies and optimisations aimed specifically at the verification of large systems. In this dissertation we used the freely available version c-2.6.r1.

Gandalf implements a large number of search strategies. The usage of these strategies can be either controlled by the human user or by the powerful automatic mode of Gandalf. The automatic mode first selects a set of different strategies that are likely to be useful for a given problem and then attempts all these strategies one after another. It uses time-slicing to limit the time that a specific search strategy is executed. When the strategy's time runs out, the next strategy is executed. During each specific strategy run Gandalf typically modifies its strategy as the time limit for the run starts coming closer. Selected clauses from unsuccessful runs are sometimes used in later runs.

The basic strategies that Gandalf selects from are hyperresolution, binary set-of-support resolution, unit resolution and ordered resolution. Typically Gandalf selects one or two strategies to iterate over term depth limit and one or two strategies to iterate over the selection of equality orderings. During the second half of each strategy run Gandalf will impose additional restrictions, like introducing unit restriction and switching over to strict best-first clause selection.

The strategy selection for a particular problem is based on the following criteria:

- The CASC problem class (Pelletier 2002, Sutcliffe & Suttner 2006). These classes include unit equality (UEQ), pure equality (PEQ), Horn logic with no equality (HNE), Horn with some but not pure equality (HEQ), non-Horn with some but not pure equality (NEQ) and non-Horn with no equality (NNE). These strictly determine the list of basic strategies. The following criteria determine the relative amount of time given to each strategy.
    - The problem size based on the number of clauses in the problem. The classifications are small, medium or large. For bigger problems, the set of support strategy gets relatively more time than other strategies.
    - The percentage of clauses that can be ordered by term depth are small, medium and all. For larger percentages term depth ordering gets relatively more time than other strategies.

Like Vampire, Gandalf is a saturation-based theorem prover. It implements the widely used given-clause saturation algorithm, also used by the Otter and Vampire provers. The main loop for inferring and processing clauses is exactly the same as that of Otter (McCune 2003):

```
while (sos is not empty and no refutation has been found)
  1. Let given_clause be the lightest clause in sos;
  2. Move given_clause from sos to usable;
  3. Infer and process new clauses using the inference rules in effect;
     each new clause must have the given_clause as one of its parents and
     members of usable as its other parents;
     new clauses that pass the retention tests are appended to sos;
end of while loop.
```

A subset of the Otter notation (McCune 2003) is used for problem input. Gandalf does not recognise formula syntax and requires the input to be in *clausal* form. The TPTP utility `tptp2X` (Sutcliffe & Suttner 1998) can be used to convert a problem in TPTP notation to the required Otter notation in clausal form that can be used by Gandalf.

The following example illustrates the use of Gandalf. It is the same example that was used to illustrate Vampire's use. The input is in the Otter clausal form notation and was obtained by a conversion from the TPTP input using the tptp2X utility. The input file has been modified afterwards by adding comments and removing some unnecessary generated comments.

```
assign(max_seconds,1800).
set(prolog_style_variables).
set(tptp_eq).
set(auto).
clear(print_given).

% The usable list
% ---------------
list(usable).

% Reflexivity
  equal(X,X).

% Extensionality
  el($f1(A,B),A)|el($f1(A,B),B)|equal(A,B).
  -el($f1(A,B),A)| -el($f1(A,B),B)|equal(A,B).

% Empty = {}
  -el(X,empty).

% A = {1}
  -el(X,a)|equal(X,1).
  el(X,a)| -equal(X,1).

% B = {A}
  -el(X,b)|equal(X,a).
  el(X,b)| -equal(X,a).
```

```
% C = P(B)
  -el(X,c)| -el(Y,X)|el(Y,b).
  el(X,c)|el($f2(X),X).
  el(X,c)| -el($f2(X),b).

% D = {Empty,{{1}}}
  -el(X,d)|equal(X,empty)|equal(X,b).
  el(X,d)| -equal(X,empty).
  el(X,d)| -equal(X,b).
end_of_list.

% The set of support
% ------------------
list(sos).
% Goal clause C = D
  -equal(c,d).
end_of_list.
```

With the above input Gandalf is given 30 minutes (1800 seconds) to find a refutation by the command `assign(max_seconds,1800)`. Also, no explicit inference strategies have been set. The command `set(auto)` instructs Gandalf to automatically select strategies.

Gandalf finds a proof after 119.21 seconds and produces the output as shown in Appendix C.2. The output also confirms that automatic strategy selection is used and that the time limit is 30 minutes. The problem class has been identified as NEQ, that is non-Horn with some but not pure equality. The problem size is classified as medium. The strategies that have been selected are displayed, followed by the steps of the proof. The strategies that were actually used to obtain the proof are also listed. Finally the output gives some global statistics for example the number of clauses generated, kept, subsumed etc.

Note that the Gandalf algorithm attempts a number of proof strategies one after another. It is possible that only the last strategy attempted produces a proof, hence the work done during the previous strategies is effectively wasted. In reporting the time taken to find a proof, Gandalf does however give the total time taken of which the time for the successful strategy is a part. We observe this in the proof output in Appendix C.2. This phenomenon is similar to other reasoners exploring the consequences of irrelevant information, e.g. the Redundant Information heuristic developed in Section 5.8.

4.3 Summary

The aim of this work is to determine to what extent state-of-the-art resolution-based reasoners may benefit from the VdPL heuristics in the same way that Otter benefited

from these. This chapter identified the two theorem provers that we consider in this dissertation. Vampire and Gandalf were selected because they are also resolution-based reasoners like Otter and performed well in the set theory sections of the CADE ATP System Competitions (CASC) (Pelletier *et al*. 2002, Sutcliffe & Suttner 2006).

A brief introduction to the resolution techniques employed by Vampire and Gandalf was given including a sample input problem and the respective outputs in appendices. In the next chapter we shall investigate the utility of the VdPL heuristics for Vampire and Gandalf.

Chapter 5

Evaluation of Set-Theoretic Reasoning Heuristics

The VdPL heuristics were developed to aid automated reasoners in solving set-theoretic problems. The heuristics were arrived at through the use of the resolution-based reasoner Otter. In this section we measure the utility of these heuristics for Vampire and on a scaled down version also for Gandalf.

We evaluate eleven of the fourteen heuristics initially developed by Van der Poll and Labuschagne (1999) and later extended by Van der Poll (2000). The three heuristics that are not evaluated are weighting, inference rule selection and set of support enlargement. The use of these 3 heuristics would involve modifications of prover default settings e.g. inference rule selection, weighting and inference strategy. A fair amount of time can be consumed by experimenting with the large number of settings and the combinations thereof. For this reason the only default settings that were changed are the time and memory limits.

The experiments reported on in this chapter follow a pattern: First a sample problem is presented and the ZF axioms on which the problem is based are stated. The performance of Otter as researched by Van der Poll and Labuschagne (1999) and Van der Poll (2000) in their attempts to find proofs is then reported. The heuristic identified from a failed proof attempt is then presented. Such heuristic allowed Otter to successfully solve the given problem. Next Vampire and Gandalf are used on the original problem to determine the need for the particular heuristic. In some cases we increase the complexity of the problem as an additional test.

We used Vampire version 8.0 and Gandalf version c-2.6.r1. A time limit of 30 minutes and a memory limit of 128MB were imposed on each. These limits cause Vampire to use

its limited resource strategy, thereby allowing the reasoner to selectively discard passive clauses. No changes were made to the other default settings of Vampire and Gandalf. All the Vampire and Gandalf proofs were done on an AMD Athlon 1700+ machine with 256MB RAM and a clock speed of 1.47GHz. The VdPL heuristics were previously arrived at by running Otter on a slower machine, namely an AMD K6-2 machine with 64MB RAM and a clock speed of 400MHz (Van der Poll 2000).

5.1 Equality versus Extensionality

The first heuristic that we consider is applicable to situations where set-theoretic equality is used in the input to the reasoner. The Zermelo-Fraenkel axiom of extensionality (Enderton 1977) states that two sets are the same if and only if they have the same elements (Section 2.1.1):

$$\forall A \forall B(\forall x(x \in A \leftrightarrow x \in B) \rightarrow A = B) \qquad \textbf{(5.1)}$$

A set is therefore determined by its elements. We can therefore replace any set equality formula with a formula stating that the elements of the sets are the same.

Our first sample problem based on set-theoretic equality and the power set axiom is given by:

$$\mathbb{P}\{\{1\}\} = \{\varnothing, \{\{1\}\}\} \qquad \textbf{(5.2)}$$

Neither Otter nor Vampire accepts formulae in the highly evolved notation of set theory, which is the result of introducing a number of symbols by meta-level definitions. Hence the user has to rewrite set-theoretic formulae like (**5.2**) above in terms of a weaker first-order language having the relevant relations and functions symbols in its alphabet (Van der Poll & Labuschagne 1999). Therefore, our proof obligation is rewritten as:

$$A = \{1\} \wedge B = \{A\} \wedge C = \mathbb{P}(B) \wedge D = \{\varnothing, B\} \rightarrow C = D \qquad \textbf{(5.3)}$$

Further decomposition is required for $\mathbb{P}(B)$ as (refer Section 2.1.6):

$$\forall x(x \in C \leftrightarrow \forall y(y \in x \rightarrow y \in B)) \quad \textbf{(5.4)}$$

Otter found no proof for (**5.3**) in 20 minutes (Van der Poll 2000). Next, if one through extensionality replaces the consequent ($C = D$) by

$$\forall x(x \in C \leftrightarrow x \in D) \quad \textbf{(5.5)}$$

then it allowed Otter to find a proof in 0.03 seconds. These findings led to the following heuristic:

> **Heuristic #1:** Use the principle of extensionality to replace set equality with the condition under which two sets are equal, i.e., when their elements are the same.

When the same problem (**5.3**) above is given to Vampire, it has little difficulty in finding a proof in 1.3 seconds. The application of the above extensionality heuristic #1 leads to a relatively faster proof in 0.1 seconds. These times are however too short to determine the utility of the heuristic for Vampire. However, consider the following, more complex example involving arbitrary intersection:

$$\cap\{\{1,2,3\}, \{2,3,4\}\} = \{2,3\} \quad \textbf{(5.6)}$$

The arbitrary intersection of a set A is defined as $\cap A = \{x \mid (\forall y)(y \in A \rightarrow x \in y)\}$ (Enderton 1977). As before formula (**5.6**) is rewritten to make the relevant constructions explicit:

$$A = \{1,2,3\} \wedge B = \{2,3,4\} \wedge C = \{A,B\} \wedge D = \{2,3\} \rightarrow \cap C = D \quad \textbf{(5.7)}$$

This time Vampire finds no proof within 30 minutes. When we, however, apply the principle of extensionality to the consequent of formula (**5.7**) as in

$$\forall x(x \in \cap C \leftrightarrow x \in D) \quad \textbf{(5.8)}$$

then Vampire finds a short proof in 0.4 seconds. Therefore heuristic #1 appears to be useful for Vampire as well, depending on the complexity of the problem in which the set-theoretic equality occurs.

Gandalf is able to find a proof for the original problem (**5.3**) in 1 minute 57 seconds. When extensionality is applied a proof is found in 43 seconds. However, on the more complex problem Gandalf is unable to find proofs before or after application of the extensionality heuristic. It is possible that the variable-strategy selection algorithm of Gandalf may be responsible for this and more work would be needed to investigate this result.

5.2 Nested Functors

A simple yet effective heuristic is to give preference to deductions containing smaller clauses (Section 3.7), i.e. clauses containing fewer literals or clauses of smaller term depth. The use of nested function symbols (called *functors* in the world of automated reasoning) leads to larger term depth and makes the unification of clauses more difficult.

The nesting of function symbols usually occurs naturally as illustrated by the next example:

$$(A + B) + C = A + (B + C) \qquad (\mathbf{5.9})$$

Formula (**5.9**) states that set-theoretic symmetric difference (denoted by '+') is associative. The symmetric difference (Enderton 1977) of sets A and B is defined as $A + B = (A - B) \cup (B - A) = \{x \mid ((x \in A) \wedge (x \notin B)) \vee ((x \notin A) \wedge (x \in B))\}$. Therefore our sample problem (**5.9**) employs equality as well as a ZF subset axiom as instantiated by set-theoretic difference. A first-order logic definition of the symmetric difference functor (called *symmdiff* below to avoid possible confusion with ordinary set-theoretic difference) is:

$$\forall A \forall B \forall x(x \in symmdiff(A,B) \leftrightarrow ((x \in A \wedge x \notin B) \vee (x \notin A \wedge x \in B))) \qquad (\mathbf{5.10})$$

The conclusion of the proof is then stated as:

$$\forall x(x \in symmdiff(symmdiff(A,B), C) \leftrightarrow \qquad (\mathbf{5.11})$$
$$x \in symmdiff(A, symmdiff(B,C)))$$

With this formulation it took Otter 4 minutes 3 seconds to find a proof of (5.11). The problem can alternatively be formulated as by unfolding (i.e. effectively removing) the nested functors in (5.11):

$$D = A + B \wedge E = D + C \wedge F = B + C \wedge G = A + F \rightarrow \forall x(x \in E \leftrightarrow x \in G) \quad \textbf{(5.12)}$$

The use of (**5.12**) instead allowed Otter to find a proof in only 0.17 seconds (Van der Poll 2000). These results suggest the following heuristic:

Heuristic #2: Avoid, if possible, the use of nested functor symbols in definitions.

Vampire quickly finds a proof of (**5.11**) in less than 0.1 seconds, both with and without the use of the nested functor heuristic #2. We therefore increase the complexity of the problem to further investigate the utility of this heuristic for Vampire. Note that in both problem formulations the extensionality heuristic #1 was already applied to problem conclusions. Rewriting (**5.11**) without using extensionality as

$$symmdiff(symmdiff(A,B), C) = symmdiff(A, symmdiff(B,C)) \quad \textbf{(5.13)}$$

results in Vampire finding no proof after 30 minutes (another illustration of the utility of Heuristic #1 for Vampire). Next we apply the nested functor heuristic #2 by rewriting (**5.13**) using Skolem constants:

$$D = A + B \wedge E = D + C \wedge F = B + C \wedge G = A + F \rightarrow E = G \quad \textbf{(5.14)}$$

Vampire now finds a proof for (**5.14**) after only 0.5 seconds.

The nested functor heuristic #2 does not seem to be useful for Gandalf's algorithm. In fact, it appears to lead the theorem prover astray. Gandalf finds a proof for (5.11) in 41 seconds and for (**5.12**) in 5 minutes 43 seconds. Therefore, the application of the nested functor heuristic #2 resulted in a longer proof time. However, if we test Gandalf on the more complex problem then it finds no proof for (**5.13**) after 30 minutes. Application of the nested functor heuristic as in (**5.14**) results in a proof after 5 minutes 44 seconds. This time corresponds with Gandalf's time for (**5.12**).

One should keep in mind that the VdPL heuristics are guidelines and not hard and fast rules and it is certainly possible that they might not be applicable in every situation as might be expected beforehand. Therefore more work with Gandalf would be needed to determine the cause of the above phenomenon. It is possible, therefore, that the nested functor heuristic #2 may have to be augmented in certain cases.

5.3 Divide-and-Conquer

The heuristic examined in this section is applicable to proofs where the consequence of the proof contains a set-theoretic equality or an if-and-only-if formula. A set-theoretic equality in the conclusion of a proof implies 'if and only if' via the axiom of extensionality. Owing to the if-and-only-if formula, a specifier can perform two separate proofs, one for the only-if part and another proof for the if part.

Consider the following sample problem based on equality and the power set axiom:

$$\mathbb{P}\{0,1\} = \{\varnothing, \{0\}, \{1\}, \{0,1\}\} \qquad \textbf{(5.15)}$$

The formula is rewritten to make the relevant constructions explicit:

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \wedge E = \{\varnothing, A, B, C\} \rightarrow D = E \qquad \textbf{(5.16)}$$

Otter found no proof for (**5.16**) after 30 minutes. Resorting to the extensionality heuristic #1 by changing the conclusion to

$$\forall x(x \in D \leftrightarrow x \in E) \qquad \textbf{(5.17)}$$

allowed Otter to find a proof in 3 minutes 23 seconds (Van der Poll 2000). A further simplification is to perform two separate proofs, one for each half of (**5.17**) and in the two proofs specify the conclusion as

$$\forall x(x \in D \rightarrow x \in E) \qquad \textbf{(5.18)}$$

and

$$\forall x(x \in E \rightarrow x \in D) \qquad \textbf{(5.19)}$$

respectively. Otter then found a proof for (**5.18**) in 0.43 seconds and for (**5.19**) in 0.03 seconds. These results led to the following divide-and-conquer heuristic:

> **Heuristic #3:** Perform two separate subset proofs whenever the problem at hand requires one to prove the equality of two sets.

The divide-and-conquer heuristic #3 is not only applicable to proof consequences containing 'if and only if' formulae but also to a proof conclusion that is a conjunction. In this case a separate proof may be performed for each conjunct in the proof's conclusion.

Vampire is also unable to find a proof for (**5.16**) after 30 minutes. However for (**5.17**), (**5.18**) and (**5.19**) Vampire finds quick proofs in 0.8, 0.3 and 0.1 seconds respectively. These times are again too short to affirm the utility of the divide-and-conquer heuristic #3 for Vampire. As before we increase the complexity of the problem through the equality:

$$\mathbb{P}\{0,1,2\} = \{\varnothing, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}, \{0,1,2\}\} \qquad \textbf{(5.20)}$$

Formula (**5.20**) is again rewritten to make the relevant constructions explicit:

$$A = \{0\} \wedge B = \{1\} \wedge C = \{2\} \wedge D = \{0,1\} \wedge E = \{0,2\} \wedge F = \{1,2\} \wedge \qquad \textbf{(5.21)}$$
$$G = \{0,1,2\} \wedge H = \mathbb{P}(G) \wedge I=\{\varnothing, A, B, C, D, E, F, G\} \rightarrow H = I$$

Vampire terminates without finding a refutation after 8 minutes 53 seconds with the message 'no passive clauses left'. Note that this does not mean that a refutation does not exist. Since Vampire was run with both a time and memory limit, it uses the limited resource strategy (Riazanov & Voronkov 2003), which is not a complete resolution strategy (see Section 4.1). If we apply our extensionality heuristic #1 to (**5.21**) by rewriting the consequent ($H = I$) as

$$\forall x(x \in H \leftrightarrow x \in I) \qquad \textbf{(5.22)}$$

then Vampire finds a proof in 8 minutes 40 seconds which is still too long. Next we apply the divide-and-conquer heuristic #3 by performing two different proofs for each half of (**5.22**) by specifying the proof consequents as

$$\forall x(x \in H \rightarrow x \in I) \qquad (\mathbf{5.23})$$

and

$$\forall x(x \in I \rightarrow x \in H) \qquad (\mathbf{5.24})$$

respectively. Vampire then finds a proof for (**5.23**) in 28 seconds and for (**5.24**) in 2 seconds.

Gandalf is also unable to find a proof of the original problem (**5.16**) after 30 minutes. Application of the extensionality heuristic #1 allows Gandalf to find a proof for (**5.17**) in 1 minute 27 seconds. However, further application of the divide-and-conquer heuristic appears not to be useful for Gandalf in the context of formula (**5.18**) since a proof is found in 1 minute 36 seconds. Similarly a proof for (**5.19**) is found in 1 minute 14 seconds.

On the more complex problem Gandalf is unable to find a proof for (**5.21**) after 30 minutes. Gandalf is still unable to find a proof after applying the extensionality heuristic #1 in (**5.22**). The application of the divide-and-conquer heuristic #3 leads to some degree of success since, although it does not enable Gandalf to find a proof for sub-problem (**5.23**), it enables the reasoner to find a proof for (**5.24**) in 1 minute 15 seconds.

5.4 Exemplification

When writing the contents of sets in list notation one naturally tends to define these sets using one or more levels of indirection by moving from the various elements to a symbol representing the collection of those elements (Van der Poll & Labuschagne 1999). The sample problem used for the divide-and-conquer heuristic will be used here as well:

$$\mathbb{P}\{0,1\} = \{\varnothing, \{0\}, \{1\}, \{0,1\}\} \qquad (\mathbf{5.25})$$

Recall that in the initial formulation

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \wedge E = \{\varnothing, A, B, C\} \rightarrow D = E \qquad (\mathbf{5.26})$$

Otter was unable to find a proof within 30 minutes. Suppose we remove one level of indirection by eliminating symbol *E*, i.e.

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \rightarrow D = \{\varnothing, A, B, C\} \qquad (\mathbf{5.27})$$

where $D = \{\varnothing, A, B, C\}$ is unfolded (repeatedly using the ZF pairing axiom) as

$$\forall x(x \in D \leftrightarrow (x = \varnothing \vee x = A \vee x = B \vee x = C)) \qquad (\mathbf{5.28})$$

in the proof conclusion. With this formulation Otter found a proof in 4 minutes 5 seconds. These results led to the following heuristic:

> **Heuristic #4:** Avoid unnecessary levels of elementhood in formulae by using the elements of sets directly.

The divide-and-conquer heuristic was applied to this last proof attempt to yield proofs in 0.34 and 0.03 seconds for the 'only-if' and 'if' directions respectively. Vampire was also unable to find a proof for (**5.26**) within 30 minutes. However, for (**5.27**) Vampire finds a proof in 0.8 seconds. In this example, therefore, it was not necessary to increase the complexity of the problem to illustrate the utility of the heuristic #4 for Vampire. If we do increase the complexity of the problem by again using formula (**5.20**) as an example, but instead of unfolding it as in (**5.21**) we unfold it as

$$A = \{0\} \wedge B = \{1\} \wedge C = \{2\} \wedge D = \{0, 1\} \wedge E = \{0, 2\} \wedge \qquad (\mathbf{5.29})$$
$$F = \{1, 2\} \wedge G = \{0, 1 ,2\} \wedge H = \mathbb{P}(G) \rightarrow H = \{\varnothing, A, B, C, D, E, F, G\}$$

then Vampire finds a proof in 5 minutes and 50 seconds. The divide-and-conquer heuristic can be applied to this last proof attempt to yield proofs in 31.5 and 1.6 seconds for the 'only-if' and 'if' directions respectively.

Gandalf was able to refute the original problem (**5.26**) after 16 minutes 22 seconds. Application of the exemplification heuristic enables Gandalf to find a proof in 1 minute 14 seconds. On the more complex problem unfolded as in (**5.29**) Gandalf is unable to find a proof with or without application of exemplification. When the divide-and-conquer heuristic is applied then Gandalf only finds a proof for the only-if sub-problem in 12 minutes 56 seconds.

5.5 Multivariate Functors

Terms containing functors may contain both constants and variables as arguments. The number of possible unifications with a clause containing a functor increases with each functor argument that occurs as a variable. As a result more clauses are generated leading to a larger search space. There are two main examples that lead to functors containing variables as arguments. The first is due to the specifier using functors that take variables as arguments, typically because of indirect definitions. The second example is produced by Skolemisation (Hamilton 1991, Section 0). Skolemisation occurs when first-order formulae are clausified to serve as input to the resolution mechanism. An important step is the elimination of existential quantifiers (Van der Poll & Labuschagne 1999). If the existential quantifier occurs after any universal quantifiers, the existential quantifier is replaced by a Skolem functor taking each of the universally quantified variables as an argument (Section 0).

The example problem (**5.16**) will be used again with the extensionality heuristic #1 applied to the conclusion as in (**5.17**). First we define the term $D = \mathbb{P}(C)$ indirectly as

$$\forall x(x \in D \leftrightarrow x \subseteq C) \qquad (\mathbf{5.30})$$

where the subset functor $\subseteq$ is defined as

$$\forall A \forall B(A \subseteq B \leftrightarrow \forall y(y \in A \rightarrow y \in B)) \qquad (\mathbf{5.31})$$

With this formulation Otter found no proof in 30 minutes. The clausification of (**5.31**) results in universally quantified variable *y* being replaced by a Skolem functor of the two variables *A* and *B*. The effect of Skolemisation may be reduced by eliminating one of the

universally quantified variables in (**5.31**), e.g. replace variable *B* by a constant *C* (say) in (**5.30**):

$$\forall A(A \subseteq C \leftrightarrow \forall y(y \in A \rightarrow y \in C)) \qquad (\mathbf{5.32})$$

Otter then found a proof after 4 minutes 5 seconds. Variable *y* in the clausal form of (5.32) is now replaced by a Skolem functor of only one variable as opposed to a functor of two variables in (5.31). The possibility of irrelevant unifications with this Skolem functor has therefore been reduced. It should also be noted that the subset functor $\subseteq$ in both cases has an arity of two, but in (5.31) it contains two variables as opposed to one constant and one variable in (5.32). These results delivered the following heuristic:

> **Heuristic #5:** Simplify terms in sets by either not involving functors, or else functors with the minimum number of argument positions taken up by variables.

Vampire finds proofs with or without heuristic #5 applied. With the subset functor formulated as in (**5.31**) it finds a proof in 21 seconds and for (**5.32**) in 0.1 seconds. The relative improvement in search time is significant. However, the search time for (**5.31**) may still be too low to seriously justify the use of heuristic #5. We therefore increase the complexity of the problem to further test our heuristic. The example problem (**5.21**) that was also used in the divide-and-conquer heuristic has sufficient complexity and will be used again with the extensionality heuristic #1 applied to the conclusion as in (**5.22**). As before, the term $H = \mathbb{P}(G)$ is unfolded as

$$\forall x(x \in H \leftrightarrow x \subseteq G) \qquad (\mathbf{5.33})$$

where the subset functor $\subseteq$ is again defined as in (**5.31**). With this formulation Vampire finds no proof in 30 minutes. We next apply the multivariate functor heuristic by defining the subset functor with variable *B* replaced by the constant *G*:

$$\forall A(A \subseteq G \leftrightarrow \forall y(y \in A \rightarrow y \in G)) \qquad (\mathbf{5.34})$$

Now Vampire finds a proof after 1 minute and 32 seconds. This result can further be improved through divide-and-conquer. The times for the two sub-proofs are 5.2 and 0.3 seconds respectively.

Gandalf is also able to find a proof for the original problem with and without heuristic #5 applied. A proof is found in 8 minutes 11 seconds with the subset functor formulated as in (**5.31**) and in 1 minutes 31 seconds for (**5.32**). The original problem is therefore sufficient to illustrate the usefulness of heuristic #5 in Gandalf's case. For the more complex problem Gandalf is unable to find a proof with or without the multivariate heuristic applied to the subset functor. When the divide-and-conquer heuristic is applied to the conjecture together with the multivariate functor heuristic then Gandalf is only able to find a proof for the only-if direction in 1 minute 16 seconds. Hence, further work would have to investigate possible proofs for the if-direction above.

5.6 Intermediate Structure

The intermediate structure heuristic is applicable to formulae in which the direct definition of intermediate structures leads to complex functor expressions. The intermediate structure heuristic can be regarded as a special case of the Multivariate functor heuristic as outlined below.

Consider the following example problem involving arbitrary union (Section 2.1.4):

$$A \times \cup B \subseteq \cup\{A \times X \mid X \in B\} \qquad \textbf{(5.35)}$$

This set-theoretic formula can be unfolded as:

$$C = \cup B \wedge D = A \times C \wedge E = \{A \times X \mid X \in B\} \wedge F = \cup E \rightarrow D \subseteq F \qquad \textbf{(5.36)}$$

The intermediate structure E still needs to be expanded further. The following formula is a direct definition of E:

$$\forall x(x \in E \leftrightarrow \exists X(X \in B \wedge \forall y \forall z((y,z) \in x \leftrightarrow y \in A \wedge z \in X))) \qquad \textbf{(5.37)}$$

Otter could not find a proof with these inputs and ran out of memory after 9 minutes 51 seconds. Closer inspection of the clausal form of this direct definition of E shows that the ordered pair (y,z) clausifies into the functor $ORD(f1(x1,x2),f2(x1,x2))$. The Skolemisation of the variables y and z results in Skolem functions of two variables that are nested within the ordered pair functor. The nested functor and multivariate functor heuristics showed that this clausal form complicates the resolution process. To avoid this clausal form for E, one may resort to the following definition:

$$\forall x(x \in E \leftrightarrow \exists X(X \in B \wedge \forall y \forall z((y,z) \in x \leftrightarrow (y,z) \in PROD(A,X)))) \qquad (\mathbf{5.38})$$

$PROD(A,X)$ is further defined as:

$$\forall X \forall y \forall z((y,z) \in PROD(A,X) \leftrightarrow y \in A \wedge z \in X) \qquad (\mathbf{5.39})$$

The clausal form for the ordered pair (y,z) in (**5.39**) now remains $ORD(y,z)$. As a result the unification process is simplified and it enabled Otter to find a proof in just 0.06 seconds, leading to the following heuristic:

> **Heuristic #6:** Use an indirect definition for an intermediate structure instead of a direct definition when its clausification results in less complex functors.

Vampire is also not able to find a proof after 30 minutes with the direct definition of E in (**5.37**). However, Vampire is able to find a proof in 40 seconds for the indirect definition of E in (**5.38**) and (**5.39**). Even though Vampire is now able to find a proof in a relatively short time, the time is strikingly larger than that of Otter. Closer inspection of the clausal form of (**5.38**) reveals that it still contains two variable Skolem functors nested in the ordered pair functor. The following reformulation of (**5.38**) uses equality instead of extensionality:

$$\forall x(x \in E \leftrightarrow \exists X(X \in B \wedge x = PROD(A,X))) \qquad (\mathbf{5.40})$$

The clausal form the definition of E in (**5.40**) does not contain any of the complex functors encountered above. As a result Vampire is now able to find a proof in 0.1 seconds.

Gandalf's results are similar to those of Vampire. It is also not able find a refutation after 30 minutes using (**5.37**). Using the indirect definition of *E* in (**5.38**) and (**5.39**) enables Gandalf to find a proof in 1 minute 15 seconds. This result is further reduced to 0.3 seconds by using the equality formulation of (**5.40**).

5.7 Element Structure

The element structure heuristic, like the intermediate structure heuristic, is a special case of the multivariate heuristic by focusing on another situation in which formulae describe the structure of elements of relations and functions.

Consider the following example problem (Van der Poll & Labuschagne 1999):

$$F = \{(\varnothing,a), (\{\varnothing\},b), (a,b)\} \rightarrow F^{-1} = \{(a,\varnothing), (b,\{\varnothing\}), (b,a)\} \quad (\mathbf{5.41})$$

F is firstly defined as (ORD is a functor denoting an ordered pair):

$$\forall x(x \in F \leftrightarrow (x=ORD(\varnothing,a) \vee x=ORD(\{\varnothing\},b) \vee x=ORD(a,b))) \quad (\mathbf{5.42})$$

The relationship between *F* and F^{-1} (the inverse of *F*) is given by:

$$\forall y \forall z(ORD(y,z) \in F \leftrightarrow ORD(z,y) \in F^{-1}) \quad (\mathbf{5.43})$$

The element structure of *F* is defined as:

$$\forall x(x \in F \rightarrow \exists y \exists z(x = ORD(y,z))) \quad (\mathbf{5.44})$$

The following theorem about ordered pairs is also required (Enderton 1977):

$$\forall u \forall v \forall w \forall x(ORD(u,v) = ORD(w,x) \leftrightarrow (u=w \wedge v=x)) \quad (\mathbf{5.45})$$

Finally the goal is specified by:

$$\forall x(x \in F^{-1} \rightarrow (x=ORD(a,\varnothing) \vee x=ORD(b,\{\varnothing\}) \vee x=ORD(b,a))) \quad (\mathbf{5.46})$$

Otter was unable to find a proof after 20 minutes with the above formulation.

Next one may attempt another approach by rather specifying the elements of F and F^{-1} more directly at the level of ordered pairs. Firstly F is defined as

$$\forall y \forall z(ORD(y,z) \in F \leftrightarrow (y=\varnothing \wedge z=a) \vee (y=\{\varnothing\} \wedge z=b) \vee (y=a \wedge z=b)) \qquad (\mathbf{5.47})$$

and the goal by:

$$\forall y \forall z(ORD(y,z) \in F^{-1} \rightarrow (y= a \wedge z=\varnothing) \vee (y= b \wedge z=\{\varnothing\}) \vee (y=b \wedge z=a)) \qquad (\mathbf{5.48})$$

The more direct approach allowed Otter to find a proof in just 0.03 seconds. Closer inspection indicated that the more direct definition does not require formulae (**5.44**) and (**5.45**) to find a proof. The clausal form of these formulae contains nested Skolem functors and two variable arguments to the ORD functor thereby contradicting the multivariate functor heuristic #5. These results led to the following heuristic:

> **Heuristic #7:** Specify elements of relations and functions more directly at the level of ordered pairs or ordered n-tuples whenever the tuples need to be opened during the proof.

The results for Vampire are similar as for Otter without increasing the complexity of the problem. Vampire is also not able to find a proof for (**5.46**) after 30 minutes. However for the direct definition of (**5.48**) Vampire easily finds a proof in 0.1 seconds.

Gandalf is also not able to find a proof for (**5.46**) after 30 minutes, but the direct definition in (**5.48**) enables it to find a refutation in 0.2 seconds. The element structure heuristic #7 therefore appears to be a useful heuristic for the next-generation theorem provers.

5.8 Redundant Information

Redundant formulae that are provided as part of the problem specification can cause many unnecessary unifications that greatly increase the problem's search space. This is especially the case with general formulae like axioms that contain variables instead of constants found in the problem domain. This problem of combinatorial explosion due to

redundant information may seem rather obvious. However, since it is such a common problem the redundant information heuristic serves to emphasise it.

The following example problem is used as illustration:

$$[Fun(f) \wedge Fun(g) \wedge \forall x(x \in dom(f) \cap dom(g) \rightarrow f(x)=g(x))] \rightarrow Fun(f \cup g) \qquad (\mathbf{5.49})$$

The intention of functor *Fun* is that its argument is a function. The unfolding of this formula into a first-order form makes extensive use of the ordered pair functor. One may therefore feel inclined to add the following fact about ordered pair equality:

$$\forall u \forall v \forall w \forall x(ORD(u,v) = ORD(w,x) \leftrightarrow (u=w \wedge v=x)) \qquad (\mathbf{5.50})$$

The inclusion of the ordered pair equality fact prevented Otter from finding a proof within 30 minutes. When the fact is removed Otter was able to easily find a proof in 0.08 seconds. This result led to the following heuristic:

> **Heuristic #8:** Refrain from using formulae in the problem specification that do not contribute to the proof.

It is generally difficult to know beforehand whether a formula or axiom is required for a proof attempt. The next heuristic will attempt to alleviate this problem.

Vampire appears to have no problem in finding quick proofs with or without the inclusion of the ordered pair equality (5.50) in less than 1 second for both cases. To increase the effect of redundant information we add some unnecessary axioms from set theory that are relevant to relations and functions in addition to the equality axiom (**5.1**). Firstly we add a fact about cross products:

$$\forall A \forall B \forall x \forall y(ORD(x,y) \in PROD(A,B) \leftrightarrow (x \in A \wedge y \in B)) \qquad (\mathbf{5.51})$$

Vampire is still able to find a quick proof in 1 second. Next we remove the cross product formula above and instead add the definition for range:

$$\forall R \forall y(y \in RAN(R) \leftrightarrow \exists x(ORD(x,y) \in R)) \qquad (\mathbf{5.52})$$

It now takes Vampire 4.5 seconds to find a proof. The individual addition of these last two formulae therefore did not increase the proof time significantly. However, if we add both formulae at the same time, then Vampire is unable to find a proof in 30 minutes.

Gandalf is not influenced very much by any variation of the above formulae. With the ordered pair equality (5.50) included in the problem it finds a proof in 59 seconds and without it in 57 seconds. Inclusion of both the equality (**5.1**) and cross product (5.51) formulae increases the refutation time slightly to 1 minute 5 seconds. Inclusion of the equality (**5.1**) and range axioms (**5.52**) results in a similar proof time of 1 minute 7 seconds. Finally inclusion of all these unnecessary axioms (**5.1**), (5.51) and (5.52) has a proof time of 1 minute 9 seconds. Gandalf therefore does not seem to be influenced too much with the addition of redundant information. This might be explained by the fact that in Gandalf's automatic mode it first selects a set of different strategies that are likely to be useful for a given problem and then tries all these strategies one after another (Tammet 1997). It uses time-slicing to limit the time that a specific search strategy is executed. When the strategy's time runs out, the next strategy is executed. Redundant information may therefore only have an impact on some of the strategy runs, but not all of them.

5.9 Search-Guiding

The redundant information heuristic suggests that formulae that do not contribute to a proof attempt should be discarded since these unnecessarily enlarge the search space. It is however difficult to know beforehand which formulae and axioms will be required to find a proof.

The purpose of the search-guiding heuristic is to identify which parts of formulae are most probably required for the proof attempt via a technique called resolution by inspection. These parts are called half-definitions. The parts that are less clearly relevant can be provisionally excluded from the proof attempt.

The following example problem for nonempty sets *A* and *B* will be used:

$$\cap(A \cup B) \subseteq (\cap A) \cap (\cap B) \quad \textbf{(5.53)}$$

This formula may be unfolded as:

$$C = A \cup B \wedge D = \cap C \wedge E = \cap A \wedge F = \cap B \wedge G = E \cap F \rightarrow D \subseteq G \quad \textbf{(5.54)}$$

The search-guiding heuristic will be illustrated by tracing a small part of the proof attempt. The goal of the problem is $D \subseteq G$ and must form part of the proof attempt. In its unabbreviated first-order form it is defined by the formula:

$$\forall x(x \in D \rightarrow x \in G) \quad \textbf{(5.55)}$$

The goal of the proof is negated to find a proof by refutation. The negated goal after clausification is given by the following two clauses:

$$c1 \in D \quad \textbf{(5.56)}$$

$$c1 \notin G \quad \textbf{(5.57)}$$

c1 is a Skolem constant that arises due to the negation of the universal quantifier that becomes an existential quantifier. To find a proof both of these clauses must be resolved.

The unabbreviated first-order definition of set *G* is:

$$\forall x(x \in G \leftrightarrow x \in E \wedge x \in F) \quad \textbf{(5.58)}$$

The literal $c1 \notin G$ in (5.57) can only be resolved with a literal of the form $x \in G$ which is found in the "if" direction of the "if-and-only-if" formula (5.58). The "only-if" half can therefore be discarded which gives:

$$\forall x(x \in G \leftarrow x \in E \wedge x \in F) \quad \textbf{(5.59)}$$

The same resolution by inspection technique can be followed for the literal $c1 \in D$ in (5.56) and literal $c1 \notin D$ which is found in the first-order definition for set *D*. The half definition that is required is:

$$\forall x(x \in D \rightarrow \forall b(b \in C \rightarrow x \in b)) \quad \textbf{(5.60)}$$

Otter found a proof for the original unfolding in 8.36 seconds. By using only the half definitions Otter found a proof in 0.12 seconds. Although the original time seems fast enough, the half definitions enabled a 70 times faster proof time. These results inspired the following heuristic:

> **Heuristic #9:** Use half definitions of if-and-only-if formulae via the technique of resolution by inspection to guide the resolution of literals that will form part of a proof starting at the goal.

Vampire is able to find quick proof in less than 1 second with or without the use of the half definitions. To illustrate the utility of the search-guiding heuristic to Vampire we will use the example problem (**5.35**) of the intermediate structure heuristic #6 after the heuristic has been applied. For ease of reference we repeat its unfolding in (**5.36**) here

$$C = \cup B \wedge D = A \times C \wedge E = \{A \times X \mid X \in B\} \wedge F = \cup E \rightarrow D \subseteq F \quad \textbf{(5.61)}$$

with the definition of E as given by (**5.38**). It took Vampire 40 seconds to find a proof for this formulation. The goal of this problem also contains the subset functor which clausifies into clauses similar to (5.56) and (5.57). The search-guiding heuristic #9 is applied by using only half definitions for sets D and F. Vampire is then able to find a proof in 9 seconds.

We can apply the search-guiding heuristic #9 further. The half definition of set D that remained is:

$$\forall x \forall y((x,y) \in D \rightarrow x \in A \wedge y \in C) \quad \textbf{(5.62)}$$

Further note that set C is defined as:

$$\forall x(x \in C \leftrightarrow \exists y(y \in B \wedge x \in y)) \quad \textbf{(5.63)}$$

In its clausal form the literal $y \in C$ in (5.62) will resolve with a clause in a half definition of set *C* in (5.63). By discarding the half definition that is not used for set *C* in (5.63), allows Vampire to find a quick proof in 0.5 seconds.

Gandalf also finds quick proofs for the original problem in less than one second with or without search guiding. For the more complex problem (**5.61**) Gandalf finds a proof after 1 minute 15 seconds. This time is reduced to 1 minute 3 seconds when only half definitions for sets *D* and *F* are used. The time is slightly reduced further to 59 seconds by using a half definition for set *C*. It is plausible, therefore, that a more complicated problem would necessitate the use of the search-guiding heuristic #9 for Gandalf even more.

5.10 Resonance

The resonance heuristic aims to identify formulae that can be rewritten in a format that is syntactically similar to facilitate the resolution process. This heuristic was originally proposed by Wos (1995, 1996). The example used to illustrate it is taken from the area of databases.

Let *Emp* be a partial function from personnel identifications in *ID* to personnel information in *PERSON*:

$$Emp: ID \rightarrow PERSON \qquad (\mathbf{5.64})$$

Suppose the structure of *PERSON* is defined as:

$$PERSON = Name \times Role \times Dept \times Salary \times Address \qquad (\mathbf{5.65})$$

Suppose we want to increase the salary of an employee with personnel number *p* by *amount*. Let *Emp*' represent the updated employee function. The increase operation can then be specified as:

$$\forall x \forall n \forall r \forall d \forall s' \forall a \qquad (\mathbf{5.66})$$
$$(ORD(x, 5TUP(n, r, d, s', a)) \in Emp' \leftrightarrow$$

$$((x \neq p \wedge ORD(x, 5TUP(n, r, d, s', a)) \in Emp) \vee$$
$$(x = p \wedge \exists s(s' = s + amount \wedge ORD(x, 5TUP(n, r, d, s, a)) \in Emp))))$$

A proof obligation (PO) arises from (5.66) above, namely, to show that *Emp*' is (still) a function. This involves showing that each and every element in *Emp*' is a tuple of the correct kind (this PO is not addressed further below) and that *Emp*' is single valued, i.e. functional:

$$Siv(Emp') \tag{**5.67**}$$

where *Siv* is defined as:

$$\forall R(Siv(R) \leftrightarrow \forall u \forall v \forall w(ORD(u,v) \in R \wedge ORD(u, w) \in R \rightarrow (v = w))) \tag{**5.68**}$$

Lastly we add the following two facts about ordered pair equality and 5-tuple equality:

$$\forall u \forall v \forall w \forall x(ORD(u, v) = ORD(w, x) \leftrightarrow (u = w \wedge v = x)) \tag{**5.69**}$$

$$\forall u \forall v \forall w \forall x \forall y \forall u' \forall v' \forall w' \forall x' \forall y' \tag{**5.70**}$$
$$(5TUP(u, v, w, x, y) = 5TUP\ (u', v', w', x', y') \leftrightarrow$$
$$(u = u' \wedge v = v' \wedge w = w' \wedge x = x' \wedge y = y'))$$

Otter failed to find a proof for this formulation after 20 minutes. The resonance heuristic #10 was applied next to formula (5.68) by rewriting it into a form that is syntactically similar to corresponding terms in (5.66):

$$\forall R(Siv(R) \leftrightarrow \tag{**5.71**}$$
$$\forall u \forall v \forall w \forall x \forall y \forall z \forall v' \forall w' \forall x' \forall y' \forall z'$$
$$(ORD(u, 5TUP(v, w, x, y, z)) \in R \wedge$$
$$ORD(u, 5TUP(v', w', x', y', z')) \in R \rightarrow$$
$$5TUP(v, w, x, y, z) = 5TUP\ (v', w', x', y', z')))$$

This reformulation enabled Otter to find a proof in 11.62 seconds. These results led to the following heuristic:

Heuristic #10: Rewrite formulae to give corresponding terms a syntactically similar structure to aid the resolution process.

Vampire is also unable to find a proof for the original formulation of *Siv* in (5.68). After the application of the resonance heuristic to *Siv* Vampire is able to find a proof in less than 1 second.

Gandalf is unable to find any proof after 30 minutes with or without the application of the resonance heuristic, hence further work needs to investigate this phenomenon.

5.11 Tuple Condense

The last heuristic that we consider may also be regarded as a special case of the multivariate functors heuristic #5. The tuple condense heuristic in this section is applicable to tuples containing multiple variable arguments that are not changed or referred to in operations that change some of the other coordinates in the tuples. This heuristic suggests that these irrelevant argument positions be folded up into one argument for the purposes of the proof attempt (Van der Poll 2000).

Example (5.66) from the resonance heuristic #10 will again be used here. The salary increase operation $s' = s + amount$ only changes the salary argument of a tuple of the larger type *PERSON*. We can therefore reorder the argument positions and fold all the irrelevant arguments into one for the purposes of this proof attempt. Formula (5.66) can then be redefined as:

$$\forall x \forall y \forall s' \qquad \textbf{(5.72)}$$
$$(ORD(x, ORD(y, s')) \in Emp' \leftrightarrow$$
$$((x \neq p \wedge ORD(x, ORD(y, s')) \in Emp) \vee$$
$$(x = p \wedge \exists s(s' = s + amount \wedge ORD(x, ORD(y, s)) \in Emp))))$$

In the above formulation the positions for *name, rank, department* and *address* have been abstracted into one position represented by variable *y*.

The resonance heuristic is applied again to the single-valued formula (5.67) to be applicable to the syntactic form of (5.72):

$$\forall R(Siv(R) \leftrightarrow \quad \textbf{(5.73)}$$
$$\forall u \forall v \forall w \forall v' \forall w'$$
$$(ORD(u, ORD(v, w)) \in R \wedge ORD(u, ORD(v', w')) \in R \rightarrow$$
$$ORD(v, w) = ORD(v', w')))$$

With this formulation Otter was able to find a proof in 0.07 seconds as opposed to the 11.62 seconds using only the resonance heuristic #10 of the previous section. These results suggested the following heuristic:

> **Heuristic #11:** Reduce the number of arguments of a functor by folding those arguments that are irrelevant to the proof attempt into one.

Vampire finds quick proofs for the example problem in less than 1 second with or without tuple condensing. A more complex problem is therefore required to illustrate the utility of this heuristic for Vampire. Various problems were used but none was able to show a noticeable difference in refutation time by applying the tuple condense heuristic.

Gandalf on the other hand was unable to find a proof for the example problem after 30 minutes with or without the heuristic. It should be noted therefore that more work is needed to determine the general utility or not, of the tuple condense heuristic #11, either in its current form, or some enhanced version thereof.

5.12 Summary and Conclusions

In this chapter we investigated to what extent the VdPL heuristics may be useful to other reasoners with similar characteristics. The Vampire theorem prover was chosen as the primary reasoner for this task owing to its steadfast superior performance at recent CASC competitions. Gandalf was used as a secondary prover in this evaluation.

We evaluated 11 of the original 14 VdPL heuristics. Table 5.1 below summarises the results of this chapter. A * in the ‘Times Faster’ column indicates a proof found versus

no proof found. The results that best illustrated the applicability of the various heuristics for Vampire and Gandalf were used.

Heuristic	Otter (AMD K6-2 64MB 400MHz)			Vampire			Gandalf		
	Before	After	Times Faster	Before	After	Times Faster	Before	After	Times Faster
Equality vs Extensionality	-	0.03	*	-	0.4	*	117	43	2.7
Nested Functors	243	0.17	1429	-	0.5	*	-	344	*
Divide-and-Conquer	203	0.46	441	520	30	17	87	170	0.5
Exemplification	-	14.74	*	-	0.8	*	982	74	13
Multivariate Functor	-	245	*	-	92	*	491	91	5.4
Intermediate Structure	-	0.06	*	-	0	*	-	75	*
Element Structure	-	0.03	*	-	0.1	*	-	0.2	*
Redundant Information	-	0.08	*	-	0	*	69	57	1.2
Search-Guiding	8.36	0.12	70	40	8.9	4.5	75	59	1.2
Resonance	-	11.62	*	-	0	*	-	-	-
Tuple Condense	11.62	0.07	166	0	0	0	-	-	-

Table 5.1: Summary of theorem-proving results

It was found that Vampire needed 10 of the 11 heuristics that were evaluated. In some cases the original problem had to be enlarged to illustrate the usefulness of the given heuristic using the new reasoner. This is significant for two reasons: Firstly it is evident that Vampire may be considered as a next generation of resolution-based reasoners. Secondly, illustrating the utility of a particular heuristic when the complexity of a problem is increased suggests a real need for the said heuristic when the given problem becomes part of a larger problem and a specifier wants to discharge a proof obligation in a single proof attempt, rather than breaking it up into smaller steps.

Gandalf in general performed better than Otter but not as well as Vampire. Using the theorem-proving defaults of the reasoners throughout, Gandalf was not able to solve

some of the more complex problems that Vampire could and the ones it could solve, usually took longer than Vampire. Of the 11 heuristics evaluated, 9 heuristics were shown to be useful for Gandalf. In all the cases where Gandalf was able to find proofs with and without the relevant heuristics the time gain is diluted by the fact that in any given proof run Gandalf tries various strategies one after the other. For example, if a proof was found during a strategy that started 5 minutes after the proof run was initiated by the human user and such strategy is allocated 1 minute then the best possible time gain can only be 6/5 = 1.2.

In the next chapter we define a case study in Z and identify a number of proof obligations that arise from the specification. Some of these proof obligations will be addressed in Chapter 7.

Chapter 6

An Order management System in Z

In this chapter we present a simplified order management system that caters for order capturing and processing as well as customer and product information. The problem statement of the case study is presented first. Thereafter a high level conceptual model of the problem is given, highlighting the various entities that the case study aims to capture. The goal here is neither to present a treatment of an object-oriented development methodology nor to serve as an exercise in requirements elicitation. Next we examine the patterns that were used to translate the high level object-oriented concepts of the conceptual model to Z. The full Z specification of the problem can be found in Appendix D. Lastly we highlight typical proof obligations that arise from such a Z specification. The next chapter applies the set-theoretic heuristics of the Chapter 5 to proof obligations that arise from the case study.

6.1 Problem Statement

An order management system facilitates the capturing and processing of orders. It could contain various subsystems for handling different stages of the order fulfilment process for example stock, customers, marketing, order entry, financials, processing, and management information. The scope of this case study includes order capturing and processing as well as customer and product information.

The order management system keeps stock of various products. For each product its name, price and quantity in stock is recorded. No two products can have the same name. Free products are also kept in stock. A product can therefore have a price of 0.0. New products can be added. A product's name, price and quantity in stock can be updated. Products can also be deleted. A list of all products that is below a specified threshold quantity can be obtained.

The system has two types of customers namely companies and persons. The addresses and phone numbers of all customers are kept. Additionally, the first and last names of persons and the names and government registration numbers of companies are maintained. New persons and companies can be added. Persons and companies can only be removed if they have not placed orders before. All the information of a customer can be modified.

An order for a customer can be created. The information that is associated with an order includes the customer, date, status and order items. A new order has a status of "pending" and no items. While in pending status an order can be cancelled which will change its status to "cancelled".

New order items can be added to an order that is in a pending status. An order item is for a specific product. No two order items may refer to the same product; instead an item should have a quantity greater than one. The quantity of the product as well as the price of the product at the time the order item was created, is also kept. An order item must have a quantity that is more than zero. An order item can be added regardless of the amount of stock that is available. The quantity of an order item can be updated. Order items may be deleted.

An order with status "pending" can be processed if there are enough products in stock. Processing an order changes its status to "processed" and the quantities of products in stock are reduced by the corresponding item quantities ordered.

6.2 Conceptual Model

The following UML (Booch *et al*. 2005) class diagram is an object-oriented representation of the problem domain. It also shows the various operations for the domain.

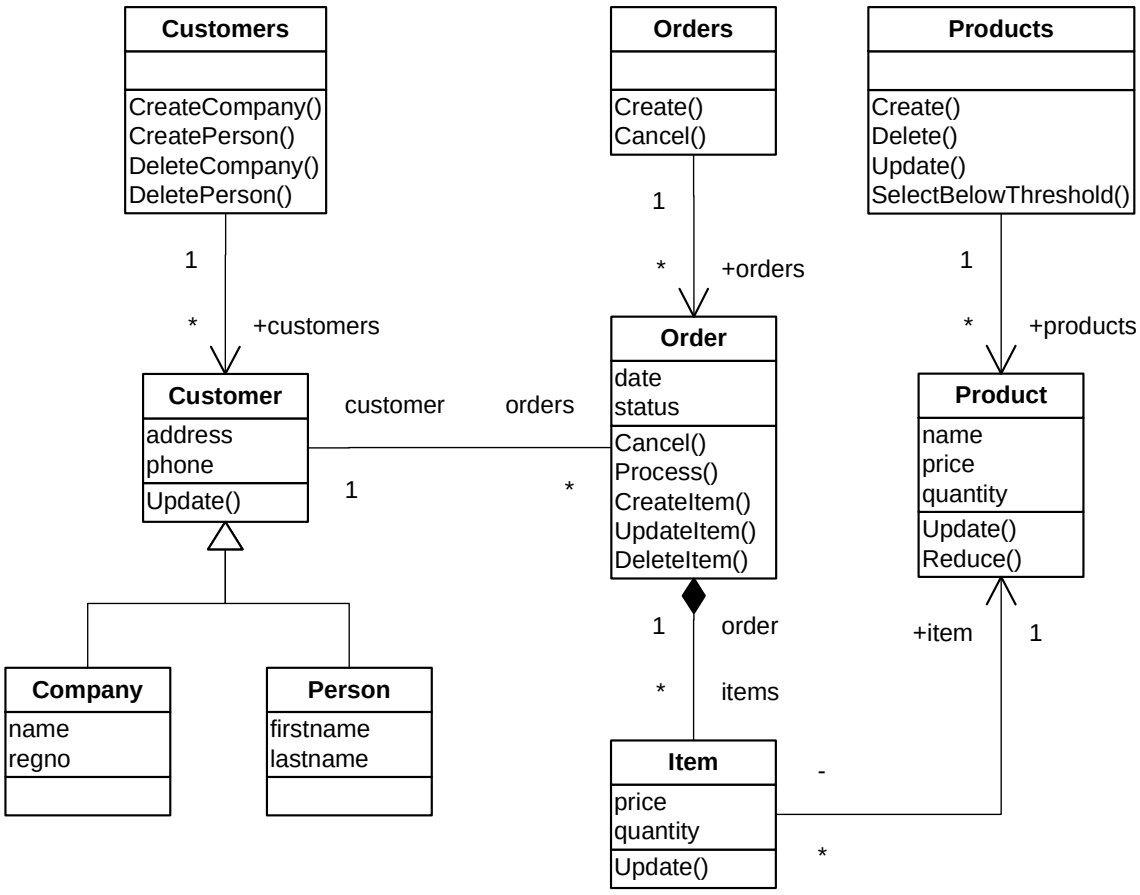

Figure 6.1: A UML class diagram of an order management system

The class diagram contains the main classes of the problem: *Product*, *Customer*, *Company*, *Person*, *Order* and *Item*. It also contains the classes *Products*, *Customers* and *Orders*. The purpose of these latter classes is to provide operations that can manage the collective states of *Product*, *Customer* and *Order*.

6.3 The Z Specification Language

Z (Spivey 1992) is a formal specification language based on first-order logic and a strongly-typed fragment of Zermelo-Frankel set theory. Z attained ISO standardisation in 2002 (ISO 2002).

Since the purpose of this study is not to consider object-oriented aspects per se, the specification of the case study presented above will be done using conventional Z instead of, for example, Object-Z (Derrick & Boiten 2001, Duke *et al*. 1995). One of the main

differences between Z and object-oriented versions is that mathematical functions are used for attributes instead of schemas that encapsulate the class instance state (Wieringa 1998, Amalio & Polack 2003). The "Birthday Book" tutorial that is provided by Spivey (1992) gives a good overview of a specification in standard Z.

The following sections examine the patterns that are used to translate the high level object-oriented concepts of the conceptual model to Z. These are classes, attributes, associations, association classes, aggregation and composition and inheritance.

The full Z specification of the case study is provided in Appendix D.

6.4 Specifying Classes and their Attributes

A UML class describes a set of objects that share the same attributes, operations, relationships and semantics (Booch *et al*. 2005). Objects and object classes are usually identified as nouns in a problem statement. For example, the statement "an order is placed by a customer" refers to two classes of objects that are order and customer. Similarly, the statement "an order is for one or more products" identifies the order class as well as the product class. All orders have the common attributes of order date (the date the order was placed) and a status (whether the order is pending, whether it has been processed, delivered etc.). The objects within a specific class mostly derive their individuality due to differences in attribute values and relationships to other classes. It is possible though that objects can have the same attribute values and relationships. But, as stated earlier, all objects have an implicit identity, which means that in this case they are still separate objects.

This section focuses on the specification of less complicated classes and their attributes. By this is meant classes that do not utilise object-oriented concepts like relationships, inheritance, aggregation and so forth. Also, the class attributes will be pure data values, i.e. an attribute type will not be a class otherwise it would constitute a relationship with that class. Nevertheless, the concepts discussed also apply to more complex cases. The class *Product* in the case study will be used in this section to describe a less complex class.

The *Product* class is an abstraction of the merchandise sold to customers. It has no inheritance associations with another class and it does not refer to any other class (it has no attribute with a class type). The fact that *Product* does not refer to another class does not mean it has no association with another class. It has an association with the *Item* class because *Item* refers to it (see Figure 6.1). This type of association is discussed in the following section.

A product has three attributes: name, price and quantity. The name attribute has a character string type, the price attribute's type is an amount and the quantity is a non-negative (i.e. natural) number. A Nokia 3650 cell phone could be an example of a product object. In this case, the name attribute could have the value "Nokia 3650" and the price attribute could have the value R4495.00 (say).

In Z a UML class may be represented by a single schema. The following example shows how the *Product* class and its attributes may be specified in Z.

Product

$$
\begin{aligned}
&products: \mathbb{P}\ PRODUCT \\
&prodName: PRODUCT \rightarrowtail\!\!\!\!\!\!\rightarrow STRING \\
&prodPrice: PRODUCT \rightarrow\!\!\!\!\!\!\rightarrow AMOUNT \\
&prodQuantity: PRODUCT \rightarrow\!\!\!\!\!\!\rightarrow \mathbb{N} \\
\hline
&dom\ prodName = products \\
&dom\ prodPrice = products \\
&dom\ prodQuantity = products
\end{aligned}
$$

The schema name has been chosen to be the same as that of the class (see Figure 6.1). The schema contains a component that represents the identities of all the available products in the system i.e. *products*: ℙ *PRODUCT*.

Each attribute is declared as a function from an identity to the type of the attribute. For example, the product price attribute is declared by the partial function *prodPrice*: *PRODUCT* ⇸ *AMOUNT*. No two products can have the same name, therefore we use a partial injective function *prodName*: *PRODUCT* ⤔ *STRING*.

The domain of each attribute function must equal the identities collection. This constraint is specified in the predicate section of the schema. For example, the *prodName* function is constrained as: dom *prodName* = *products*.

Finally, any additional constraints can also be specified in the schema's predicate section. For example, if the product name was not defined by an injective function, then the constraint that no two products can have the same name could be specified with the predicate: $\forall p1, p2 : products \bullet p1 \neq p2 \Rightarrow prodName(p1) \neq prodName(p2)$.

A possible state of the Product schema has three products in the *products* identity set, with the corresponding names, prices and quantities recorded by functions *prodName*, *prodPrice* and *prodQuantity*:

products = {38627, 39241, 41189}

prodName = {38627 ↦ "Nokia 3650",
39241 ↦ "Sony Playstation 3",
41189 ↦ "Microsoft Windows Vista Home Premium"}

prodPrice = {38627 ↦ R4495.00,
39241 ↦ R5299.00,
41189 ↦ R1353.18}

prodQuantity = {38627 ↦ 37,
39241 ↦ 29,
41189 ↦ 13}

6.5 Specifying Associations

A link specifies that an object instance of one class is connected to an object instance of another or the same class. For example, the phrase "Ralph (a customer) placed an order on the 11'th of October 2008", describes a link between a *Customer* instance and an *Order* instance.

A link is an instance of an association. An association specifies a set of links with common structure and semantics that exist between two classes. For example, the phrase "a customer places an order", describes an association. It is also possible to have an association between more than two classes, called n-ary associations. This is however not as common and will not be considered further in this dissertation.

An association can also have a minimum and maximum multiplicity at each end. For example, a customer can be associated with zero (minimum) or more (any finite maximum) orders and an order can be associated with one (minimum) and only one (maximum) customer. This is commonly referred to as a one-to-many relationship.

Navigation of associations refers to the ability to navigate from one object to another via the association that exists between their classes. For example, if the one-to-many relationship between customer and order can be navigated only from customer to order, then given a customer instance, one can reference its orders. On the other hand, given an order instance one cannot reference the customer directly.

An association can also have a role at each end. For example, let there be a one-to-many relationship between company and person. The role of person in this association is an employee and the role of the company is employer. There could also be another one-to-one relationship between company and person where the role of person is CEO (Chief Executive Officer).

The association between *Customer* and *Order* will be used to illustrate the various specification styles. As with the specification of a class and its attributes using Z, the specification of associations is not as explicit. However, in the opinion of the author of this dissertation it does not fail to be simple and effective.

Let us have a look at how the association between *Customer* and *Order* may be specified for the case study (see also Appendix D).

```
┌─ Customer ─────────────────────────────────────
│ customers: ℙ CUSTOMER
│ custAddress: CUSTOMER ⇸ STRING
│ custPhone: CUSTOMER ⇸ STRING
├──────────
│ dom custAddress = customers
│ dom custPhone = customers
└────────────────────────────────────────────────
```

```
┌─ Order ────────────────────────────────────────
│ orders: ℙ ORDER
│ orderDate: ORDER ⇸ DATE
│ orderStatus: ORDER ⇸ STATUS
│ orderCustomer: ORDER ⇸ CUSTOMER
├──────────
│ dom orderDate = orders
│ dom orderStatus = orders
│ dom orderCustomer = orders
└────────────────────────────────────────────────
```

The UML model in Figure 6.1 specified the association between *Customer* and *Order* as a one-to-many relationship that can be navigated in both directions while the *Order* schema has a component (*orderCustomer*) that maps the *ORDER* identity to its associated *CUSTOMER* identity. Therefore, given an order identity we can find the related customer identity using the *orderCustomer* function. So we can navigate from an order to the customer who placed the order.

In the Z schema for *Customer* there is no explicit reference to an association between a customer and its orders. However, it is still possible to find the orders of a customer through an operation. *SelectOrdersForCustomer* is an example of an operation that returns all the order identities for a given customer identity.

```
┌─ SelectOrdersForCustomer ──────────────────────
│ ΞOrder
│ customer?: CUSTOMER
│ orders!: ℙ ORDER
├──────────
│ orders! = {o: orders | orderCustomer(o) = customer?}
└────────────────────────────────────────────────
```

It is therefore possible to navigate from a customer to all the orders placed by the customer.

The components in a schema (e.g. *SelectOrdersForCustomer* above) that are decorated with question marks (?) represent input to the system. Components decorated with exclamation marks (!) represent output from an operation. The declaration Ξ*Order* indicates that (1) the *Order* schema is included into the *SelectOrdersForCustomer* schema and (2) the state of *Order* is not changed by the operation. In an expanded version of *SelectOrdersForCustomer*, the *Order* schema is shown twice, thereby representing the operation schema's respective before and after states. The included components of the after state instance are decorated with dash symbols (′) to distinguish them from the corresponding before state components. A predicate is also added to specify that all the before and after state components are equal, thereby stating that the state does not change.

Multiplicity constraints can also be added to the association. For example, the *Order* schema could be extended as follows to specify that a customer may have at most 10 orders (say):

Order

…

…

$\forall c$: ran orderCustomer • #(orderCustomer ▷ {c}) ⩽ 10

Note that the above *Order* schema is not complete; just the relevant predicate specifying that a customer may have no more than 10 orders, is shown.

6.6 Specifying Association Classes

An association that exists between two classes may also contain attributes. Such an association is called an *association class* since it is a class as well. We use the classes *Order* and *Product* as an example. There is an association between them because a product can appear on many orders and an order is for one or more products. However, this association carries more information that is essential for the system i.e. the quantity and price of the product on the order. This information ought not to be stored on product since it is not specific to the product (a normalisation issue in database terms). Neither

can it be stored on the order because we need the information for more than one product. Therefore, it makes sense to store the attributes as part of the association. The name of the association class is *Item* in Figure 6.1. Note that the price attribute on *Item* is the price of the product when the order was created.

The method used for the specification of the association class *Item* is the same as for one-to-many associations used above. The difference here is that *Item* has two one-to-many associations. The first is a one-to-many from *Order* to *Item* and the second is a one-to-many from *Product* to *Item*. A partial schema for *Item* is shown below:

Item

$$
\begin{array}{l}
items: \mathbb{P}\ ITEM \\
itemOrder: ITEM \rightarrow ORDER \\
itemProduct: ITEM \rightarrow PRODUCT \\
\ldots \\
\hline
\ldots \\
\forall i_1, i_2: items \bullet i_1 \neq i_2 \Rightarrow itemOrder(i_1) \neq itemOrder(i_2) \lor itemProduct(i_1) \neq itemProduct(i_2) \\
\ldots
\end{array}
$$

An additional predicate is added to ensure that no two items can reference the same order-product combination. The full version of schema *Item* appears in Appendix D.

The UML model above (Figure 6.1) shows that the association between *Order* and *Item* may be navigated in both directions. This is indicated by the absence of arrows on either side of the association. To facilitate such navigation the operation *SelectItemsForOrder* was defined to allow one to obtain the set of items of an order, given its identity:

SelectItemsForOrder

$$
\begin{array}{l}
\Xi Item \\
order?: ORDER \\
items!: \mathbb{P}\ ITEM \\
\hline
items! = \{i: items \mid itemOrder(i) = order?\}
\end{array}
$$

6.7 Specifying Operations

So far the static aspects of the system have been described. Next, we describe the dynamic aspects of the system. These include the operations that are possible, the relationships between their inputs and outputs and the changes of state that take place as a result of the operations.

The most basic operations that are typically required in most systems are create, read, update and delete (CRUD). The operations of the *Product* class are used to illustrate these. Additionally *ProcessOrder* (see Section 6.7.5) is provided as an example of a more complex operation. An order that is in a pending status can be processed if there are enough products in stock. Processing an order changes its status to *processed* and the quantities of the relevant products in stock are reduced by the order's item quantities.

6.7.1 Create Operation

A create operation adds a new instance of a class to the system. For example, an operation to create a new product is:

CreateProduct

$\Delta Product$
$product?: PRODUCT$
$name?: STRING$
$price?: AMOUNT$
$quantity?: \mathbb{N}$

$product? \notin products$
$name? \notin ran\ prodName$
$products' = products \cup product?$
$prodName' = prodName \cup \{product? \mapsto name?\}$
$prodPrice' = prodPrice \cup \{product? \mapsto price?\}$
$prodQuantity' = prodQuantity \cup \{product? \mapsto quantity?\}$

The declaration $\Delta Product$ indicates that the *Product* schema is included into the *CreateProduct* schema and the state of *Product* may change as a result of the operations specified. In the expanded version, the *Product* schema is included twice, thereby representing the operation schema's respective before and after states. The included

components of the after state instance are decorated with dash symbols (′) to distinguish them from the corresponding before state components.

The first predicate of the schema is a typical precondition of a create operation stating that the new order item may not already be in the system. The following predicates state that the *products*, *prodName*, *prodPrice* and *prodQuantity* functions are extended to map the new name, price and quantity values to the given product identity. Note that the notation $x \mapsto y$ is a graphic way of expressing the ordered pair (x, y).

6.7.2 Read Operation

A read operation finds zero or more objects based on certain criteria and return them. The following schema definition is for a finder operation that returns all products with quantities below the specified threshold:

SelectProductsBelowThreshold

ΞProduct
quantity?: $\mathbb{N}$
products!: $\mathbb{P}$ PRODUCT

---

products! = {p: products | prodQuantity(p) < quantity?}

The output of this operation (*products*!) is a set of product identities. The predicate further states that the output is a subset of product identities of which the quantity is less than the specified input, *quantity*?. The notation $\{x: S \mid E\}$, where S is a set and E a predicate, means the set of values of x taken from S which satisfy E.

6.7.3 Update Operation

An update operation changes the current value of an object in the system. In Z it specifies that the before and after state components are related in certain ways. The update operation for *Product* is specified as:

UpdateProduct

$\Delta Product$
$product?: PRODUCT$
$name?: STRING$
$price?: AMOUNT$
$quantity?: \mathbb{N}$

$product? \in products$
$products' = products$
$prodName' = prodName \oplus \{product? \mapsto name?\}$
$prodPrice' = prodPrice \oplus \{product? \mapsto price?\}$
$prodQuantity' = prodQuantity \oplus \{product? \mapsto quantity?\}$

The first predicate of the schema is a typical precondition of an update operation stating that the product must exist in the system. The following predicates state that the *prodName*, *prodPrice* and *prodQuantity* functions are remapped to associate the new name, price and quantity values to the given product (*product*?). These predicates use the overriding operator $\oplus$. The relation $Q \oplus R$ relates everything in the domain of R to the same objects as R does (Q is overridden by R), and everything else in the domain of Q to the mappings in Q.

The state of the *products* set however does not change i.e.:

$$products' = products$$

The following proof shows that the above restriction could be derived:

$products'$

$= dom\ prodPrice'$ (invariant after)

$= dom\ (prodPrice \oplus \{product? \mapsto price?\})$ (specification of *UpdateProduct*)

$= dom\ prodPrice \cup dom\ \{product? \mapsto price?\}$ (fact about ‘dom’)

$= dom\ prodPrice \cup \{product?\}$ (fact about ‘dom’)

= products ∪ {product?} (invariant before)

= products Predicate: *product*? ∈ *products*

6.7.4 Delete Operation

A delete operation removes an object from the system. The delete operation for *Product* is specified as:

DeleteProduct

ΔProduct
product?: PRODUCT

product? ∈ products
products′ = products \ {product?}
prodName′ = {product?} ⩤ prodName
prodPrice′ = {product?} ⩤ prodPrice
prodQuantity′ = {product?} ⩤ prodQuantity

Again, the first predicate of the schema is a typical precondition of a delete operation stating that the specified product must exist in the system. The remaining predicates state that *prodName*, *prodPrice* and *prodQuantity* functions are changed by removing the mapping for the given product (*product*?). These predicates use the domain anti-restriction operator ⩤. The relation $S \lhd R$ is the set of all tuples (x, y) in R where x is not in the domain of S.

The state of the *products* set also changes to reflect the removal of the product identity i.e.:

products′ = products \ {product?}

The following proof shows that the above restriction could be derived:

$products'$

$= \text{dom } prodPrice'$ (invariant after)

$= \text{dom } (\{product?\} \lhd prodPrice)$ (specification of *DeleteProduct*)

$= \text{dom } ((products \setminus \{product?\}) \lhd prodPrice)$ (fact about $\lhd$)

$= (products \setminus \{product?\}) \cap \text{dom } (prodPrice)$ (fact about 'dom')

$= (products \cap products) \setminus \{product?\}$ (fact about $\cap$ and $\setminus$)

$= products \setminus \{product?\}$ (fact about $\cap$)

The relation $S \lhd R$ above is the set of all tuples (x, y) in R where x is in the domain of S.

6.7.5 ProcessOrder

An operation to process an order is:

ProcessOrder

$\Delta Order$
$\Delta Product$
$\Xi Item$
order?: ORDER

---

order? $\in$ orders
orderStatus(order?) = pending
$\forall i: items \bullet itemOrder(i) = order? \Rightarrow prodQuantity(itemProduct(i)) - itemQuantity(i) \geqslant 0$
$orders' = orders$
$orderDate' = orderDate$
$orderStatus' = orderStatus \oplus \{order? \mapsto processed\}$
$orderCustomer' = orderCustomer$
$products' = products$
$prodName' = prodName$
$prodPrice' = prodPrice$
$prodQuantity' = prodQuantity \oplus$
$\{i: items \mid itemOrder(i) = order? \bullet itemProduct(i) \mapsto prodQuantity(itemProduct(i)) - itemQuantity(i)\}$

The operation schema includes the *Order*, *Product* and *Item* schemas. It further states that it could change the states of the *Order* and *Product* schemas. It takes an order identity as input.

The first three predicates ensure that the order exists, that it is in a pending status and that there are enough products in stock to fulfil the order. If all of these conditions are valid the order's status is changed to *processed* and the quantities of all products referenced by the order's items are reduced by the relevant item quantities.

6.8 Total Operations

All the operations above specify how the system state should change if correct input is given and the relevant preconditions are satisfied. However, the state change is undefined for incorrect inputs. The operations are therefore not total.

As an example we convert the *CreateProduct* operation into a total one as an example of specifying complete operations. The total version will be called *CreateProductTotal*. It has three possible outcomes (some may overlap) depending on the input:

- It could be successful if the input is correct.
- The product could be already known.
- The product name could already exist.

These possible outcomes are specified using the following partial schemas:

Success

result! : REPORT

---

result! = success

ProductAlreadyKnown

ΞProduct
product? : PRODUCT
result! : REPORT

---

product? $\in$ products
result! = already_known

DuplicateProductName

ΞProduct
product? : PRODUCT
name?: STRING
result! : REPORT

---

product? $\notin$ products
name? $\in$ ran prodName
result! = duplicate_name

Using the schema calculus of Z, we can then specify the total create operation as:

$$\text{CreateProductTotal} == (\text{CreateProduct} \wedge \text{Success}) \vee \text{ProductAlreadyKnown} \vee \text{DuplicateProductName}$$

CreateProductTotal is defined for all possible inputs and additionally the *result*! output component specifies whether the operation was successful or otherwise what error occurred.

6.9 Specifying Aggregation and Composition

Aggregation is used to indicate a “whole-part” relationship, in which one class represents a larger entity (the whole), which consists of smaller entities (the parts). Aggregation is often referred to as a “has-a” relationship. Aggregation does not change the meaning of navigation across the association between the whole and its parts, nor does it link the life spans of the whole and its parts (Booch *et al*. 2005).

Composition however is a form of aggregation with strong ownership and a coincident lifetime as part of the whole. This means that in a composite aggregation, an object may be a part of only one composite at a time and once it is created, it lives and dies with the

composite. The whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts.

Aggregation and composition are stronger forms of association (refer Sections 6.5 & 6.6) and therefore the various ways of specifying these are very similar to associations.

Our case study has an instance of aggregation and composition between the classes *Order* and *Item* in Figure 6.1. However, to keep our Z specification relatively simple, we shall not consider aggregation and composition further in this dissertation.

6.10 Specifying Inheritance

A generalisation is the relationship between a more general element (the parent) and a more specific element (the child). The child is fully consistent with the parent and adds additional information. A child inherits the attributes and operations of its parent and is therefore substitutable for the parent, i.e. child objects may be used anywhere the parent may appear. Generalisation is therefore also called an inheritance or an "is-a" relationship. Usually the child has attributes and operations in addition to those found in its parent. Polymorphism occurs when a child overrides an operation of the parent.

The *Customer*, *Company* and *Person* classes can be used to illustrate the different specification approaches for inheritance. All customers have an address, a phone number and a fax number. There are two types of customers, i.e. companies and individuals (called *Person* in this case study). *Company* and *Person* are specialisations of *Customer* since they add some extra attributes.

The identity sets of the child classes are declared as subsets of the customer identity set in the following Z axiomatic definition:

$$
\begin{array}{|l}
COMPANY : \mathbb{P}\ CUSTOMER \\
PERSON : \mathbb{P}\ CUSTOMER \\
\hline
\langle COMPANY, PERSON \rangle\ \text{partition}\ CUSTOMER
\end{array}
$$

The declaration further shows that the customer identity set is *partitioned* by the company and person identity sets. Therefore, any element of *CUSTOMER* is an element of *COMPANY* or *PERSON* but not both.

Customer, *Company* and *Person* are specified below:

Customer

$$
\begin{array}{l}
customers: \mathbb{P}\ CUSTOMER \\
custAddress: CUSTOMER \pfun STRING \\
custPhone: CUSTOMER \pfun STRING \\
\hline
dom\ custAddress = customers \\
dom\ custPhone = customers
\end{array}
$$

Company

$$
\begin{array}{l}
Customer \\
companies: \mathbb{P}\ COMPANY \\
compName: COMPANY \pfun STRING \\
compRegNo: COMPANY \pfun STRING \\
\hline
companies \subseteq customers \\
dom\ compName = companies \\
dom\ compRegNo = companies
\end{array}
$$

Person

$$
\begin{array}{l}
Customer \\
persons: \mathbb{P}\ PERSON \\
perName: PERSON \pfun STRING \\
perSurname: PERSON \pfun STRING \\
\hline
persons \subseteq customers \\
dom\ perName = persons \\
dom\ perSurname = persons
\end{array}
$$

6.11 Specifying the System State

It is conventional in Z to specify a schema representing the whole system state (Potter *et al*. 1996). Operations are defined on the whole state, so that all side effects may be captured and the full invariant could be demonstrated to hold after the operation.

The system state is given by:

System
Product
Order
Item
Customer
Company
Person

6.12 Specifying an Initial State

An initial state of the system is specified and a proof obligation arises to show that such an initial state may be realised. It also provides a base case to show by induction that any operation preserves the invariants of the system state. That is, given any valid system state, one ought to be able to show that such state can be realised from the initial state followed by zero or more operations.

The initial state of a class is specified by an operation schema that only includes the after state components (Potter *et al*. 1996). For example, the initial state of the *Product* scheme is specified as:

InitProduct
Product′

$products' = \varnothing$
$prodName' = \varnothing$
$prodPrice' = \varnothing$
$prodQuantity' = \varnothing$

The initial state of the whole system is then:

InitSystem
InitProduct
InitOrder
InitItem
InitCustomer
InitCompany
InitPerson

6.13 Proof Obligations Arising from the Specification

In this section we highlight a number of proof obligations that arise from Z specifications (Potter *et al.* 1996; Van der Poll 2000; Spivey 1992). Many of these proof obligations occur in the context of operations that change the system state. In Chapter 7 some of these proof obligations will later be converted to first-order logic and discharged using the Vampire theorem prover with the aid of various heuristics presented in Chapter 5.

6.13.1 Initialisation Theorem

Whenever an initial state schema is specified, a proof obligation arises to show that such a state may be realised (Potter *et al*. 1996; Van der Poll 2000). The proof obligation for the *InitProduct* initialisation schema (refer Section 6.12) is:

$$\vdash \exists\, Product' \bullet InitProduct$$

That is, we need to show that there exists an after state such that the predicate of the initialisation schema is valid.

The "turnstile" symbol, $\vdash$, is used to state a theorem (Potter *et al*. 1996). The hypothesis of the theorem is specified on the left hand side of the turnstile and conclusion on the right hand side.

6.13.2 Precondition Simplification

In Z the precondition of an operation is obtained by hiding the after-state components by existentially quantifying them in the schema's predicate (Potter *et al*. 1996). The precondition for the *CreateProduct* operation would therefore be:

PreCreateProduct

Product
product?: PRODUCT
name?: STRING
price?: AMOUNT
quantity?: ℕ

∃ Product′ •
product? ∉ products
name? ∉ ran prodName
products′ = products ∪ {product?}
prodName′ = prodName ∪ {product? ↦ name?}
prodPrice′ = prodPrice ∪ {product? ↦ price?}
prodQuantity′ = prodQuantity ∪ {product? ↦ quantity?}

This precondition can be simplified using, amongst others, the one-point-rule (Potter *et al*. 1996):

PreCreateProduct

Product
product?: PRODUCT
name?: STRING
price?: AMOUNT
quantity?: ℕ

---

product? ∉ products
name? ∉ ran prodName

Whenever a precondition is simplified we need to show that it is equivalent to the original version (Potter *et al*. 1996), i.e. the precondition of *CreateProduct* is indeed schema *PreCreateProduct* above:

⊢ pre CreateProduct =
[Product
product?: PRODUCT
name?: STRING
price?: AMOUNT
quantity?: ℕ
|
product? ∉ products
name? ∉ ran prodName]

The “pre” prefix operator in Z indicates the precondition of a schema (Spivey 1992). Also, note that the right hand side of the above equality uses a linear form of schema definition (Potter *et al*. 1996).

6.13.3 After State Type

Every component of a schema for which a possible state change is specified, a proof obligation arises to show that the corresponding after state component is of the correct type (Van der Poll 2000). As an example consider schema *UpdateProduct* in Section 6.7.3 above and in particular the component *prodName*. Upon the successful completion of *UpdateProduct* one typically has to show that:

(1) Every element of $prodName'$ is an ordered pair, i.e. $prodName'$: $PRODUCT \leftrightarrow STRING$. This effectively verifies the *carrier type* of the component. For the purposes of this dissertation a carrier type is built from applications of the powerset operator, Cartesian products and combinations of these to given sets (basic types). In particular for any two sets X and Y, we define $\mathbb{P}(X \times Y) = X \leftrightarrow Y$ (Potter *et al*. 1996).

(2) Component $prodName'$ is actually more restricted that just its underlying carrier type, i.e. $prodName'$: $PRODUCT \rightarrowtail STRING$).

For reasons of simplicity in this dissertation, we typically do not discharge proof obligations (1) above, but assume the carrier type of the component to be correct. Our decision to not attend to proof obligations involving carrier types stem from the fact that many type checkers for Z like the Community Z Tools (CZT) (Malik & Utting 2005) ensure that the carrier types of components are correct but they do not provide for more restrictive type checking, e.g. the kind mentioned in (2) above.

Proof obligations like those in (2) above are however addressed in our work. Discharging these kind of proof obligations effectively shows that the after state of a component is indeed more restricted as expected. These proof obligations are amongst other things the topic of Chapter 7.

For *UpdateProduct* the following proof obligation needs to be discharged:

Product
products′: ℙ PRODUCT
prodName′: PRODUCT ↔ STRING
prodPrice′: PRODUCT ↔ AMOUNT
prodQuantity′: PRODUCT ↔ ℕ
product?: PRODUCT
name?: STRING
price?: AMOUNT
quantity?: ℕ
|
dom prodName′ = products′
dom prodPrice′ = products′
dom prodQuantity′ = products′
product? ∈ products
name? ∉ ran prodName
products′ = products
prodName′ = prodName ⊕ {product? ↦ name?}
prodPrice′ = prodPrice ⊕ {product? ↦ price?}
prodQuantity′ = prodQuantity ⊕ {product? ↦ quantity?}
⊢
prodName′ ∈ PRODUCT ⤖ STRING
prodPrice′ ∈ PRODUCT ⇸ AMOUNT
prodQuantity′ ∈ PRODUCT ⇸ ℕ

The above notation for stating a proof obligation stems from Potter *et al*. (1996).

6.13.4 Total Operations

For every total operation specified, a proof obligation arises to show that it is indeed total (Van der Poll 2000). An operation is total if its precondition is a partition, i.e. the precondition is total and any two constituent preconditions are pairwise disjoint.

We first need to show that the precondition is total. This is true when the disjunction of all the constituent preconditions is a tautology (Van der Poll 2000). Here is an example for *CreateProductTotal*:

$$\vdash \text{pre (CreateProduct} \land \text{Success)} \lor \text{pre ProductAlreadyKnown} \lor \text{pre DuplicateProductName}$$

An equivalent method of specifying that the precondition is total is:

$$\vdash \text{pre CreateProductTotal} =$$
$$[\text{Product}$$
$$\text{product?: PRODUCT}$$
$$\text{name?: STRING}$$
$$\text{price?: AMOUNT}$$
$$\text{quantity?: } \mathbb{N}$$
$$\mid \text{true}]$$

Secondly we need to show that all the constituent preconditions are pairwise disjoint. This conjecture for *CreateProductTotal* is:

$$\vdash (\text{pre (CreateProduct} \land \text{Success)} \land \text{pre ProductAlreadyKnown}) = \varnothing \land$$
$$(\text{pre (CreateProduct} \land \text{Success)} \land \text{pre DuplicateProductName}) = \varnothing \land$$
$$(\text{pre ProductAlreadyKnown} \land \text{pre DuplicateProductName}) = \varnothing$$

6.13.5 Operation Interaction

The composition of operations leads to various proof obligations (Potter *et al*. 1996; Van der Poll 2000). For example, most specifications of create and delete operations have the property that a create operation followed by a delete operation of the same element results in an unchanged state. This is the case for *CreateProduct* followed by *DeleteProduct*:

$$\text{CreateProduct} \fatsemi \text{DeleteProduct} \vdash \Xi \text{ Product}$$

Similarly, deletion of an element followed by its creation leaves the state unchanged:

$$\text{DeleteProduct} \fatsemi \text{CreateProduct} \vdash \Xi \text{ Product}$$

Another class of interactions is create, update or delete operations followed by a find operation of the corresponding element. The element involved in the state change may or may not be expected to be found, depending on the operation composition. For example, after creating a product with some quantity, one does not expect that product to be below that quantity:

$$\text{CreateProduct} \fatsemi \text{SelectProductsBelowThreshold} \vdash \text{product?} \notin \text{products!}$$

6.13.6 Contents of a Set

A proof obligation arises when adding an element to a set. One has to show that the element is in the set afterwards provided that the necessary precondition holds (Van der Poll 2000). For example the following conjecture could be stated for *CreateProduct*:

$$\text{CreateProduct} \mid \text{product?} \notin \text{products} \land \text{name?} \notin \text{ran prodName}$$
$$\vdash$$
$$\text{product?} \in \text{products}' \land (\text{product?} \mapsto \text{name?}) \in \text{prodName}'$$

In the case of an update we also need to show that such update has been successful, given a valid precondition:

$$\text{UpdateProduct} \mid \text{product?} \in \text{products}$$
$$\vdash$$
$$\text{product?} \in \text{products}' \land (\text{product?} \mapsto \text{name?}) \in \text{prodName}'$$

Similarly, after deleting an element we can show that it is not in the set anymore:

$$\text{DeleteProduct} \mid \text{product?} \in \text{products} \vdash \text{product?} \notin \text{products}' \land \text{product?} \notin \text{dom prodName}'$$

In all of the above cases we need to show that the other elements are not affected (i.e. the operation did not cause any side effects as far as the other elements are concerned). For example:

$$\text{CreateProduct} \mid \text{product?} \notin \text{products} \land \text{name?} \notin \text{ran prodName}$$
$$\vdash$$
$$\forall\, \text{p: products; n: prodName} \mid \text{p} \neq \text{product?} \land \text{n.1} \neq \text{product?} \bullet \text{p} \in \text{products}' \land \text{n} \in \text{prodName}'$$

Note that *n*.1 above is a projection of the first component of the tuple *n*.

6.13.7 State Invariant

The expanded form of any Z operation explicitly specifies the state invariant. It is therefore not required to separately show that the state invariant is preserved. However, to ensure that an operation does not introduce an inconsistency in the state one can prove that the after state invariant is preserved, given the operation with the after state components only declared in terms of their carrier types. For example, the proof obligation for *CreateProduct* would then be:

Product
products′: ℙ PRODUCT
prodName′: PRODUCT ↔ STRING
prodPrice′: PRODUCT ↔ AMOUNT
prodQuantity′: PRODUCT ↔ ℕ
product?: PRODUCT
name?: STRING
price?: AMOUNT
quantity?: ℕ
|
product? ∈ products
name? ∉ ran prodName
products′ = products
prodName′ = prodName ⊕ {product? ↦ name?}
prodPrice′ = prodPrice ⊕ {product? ↦ price?}
prodQuantity′ = prodQuantity ⊕ {product? ↦ quantity?}
⊢
Product′

6.14 Conclusion

In this chapter we presented a simple order management system. The system was specified in Z by translating the high level specification using specific patterns. A

number of typical proof obligations that result from such specifications were also discussed.

In the next chapter we convert some of the case study proof obligations of Section 6.13 to first-order logic and discharge these using the Vampire theorem prover with the aid of various heuristics presented in Chapter 5.

Chapter 7

Discharging Case Study Proof Obligations

In this chapter a number of proof obligations that arose from the case study presented in Chapter 6 are converted from Z to first-order logic and discharged using Vampire. The heuristics discussed in Chapter 5 are finally applied to failed proof attempts in an attempt to find a proof. From the results in Chapter 5 we observed that Vampire generally performed well, hence our decision to use just Vampire as our reasoner in this chapter.

7.1 Conversion of Z to First-Order Logic

To discharge Z proof obligations using Vampire we need to convert the Z notation presented in Chapter 6 to the TPTP (Thousands of Problems for Theorem Provers) notation (refer Section 4.1) used by Vampire. The ISO standardisation of Z (ISO 2002) specifies how a Z specification can be converted to a typed first-order logic. However, first-order logic representation using the TPTP notation is not typed. To avoid the paradoxes of naïve set theory the type information must be incorporated in the TPTP transformation. A typed version of TPTP has been proposed (Claessen & Sutcliffe 2008) but has not been implemented yet. The typed conversion to TPTP notation in this chapter is based on this proposal.

As an example of such a typed conversion we will use the following conjecture:

$$\vdash \mathbb{P}\{\{1\}\} = \{\emptyset, \{\{1\}\}\}$$

Note that this problem was also used in Section 4.1 to illustrate the TPTP notation. In Z the type of the number literal 1 is $\mathbb{A}$ (arithmos) which represents all numbers (ISO 2002). The type of $\{1\}$ therefore is $\mathbb{P}\mathbb{A}$. Similarly the types of $\{\{1\}\}$, $\mathbb{P}\{\{1\}\}$ and $\{\emptyset,\{\{1\}\}\}$ are $\mathbb{P}\mathbb{P}\mathbb{A}$, $\mathbb{P}\mathbb{P}\mathbb{P}\mathbb{A}$ and $\mathbb{P}\mathbb{P}\mathbb{P}\mathbb{A}$ respectively. Since Z is strongly typed, all elements of a set in Z

must be of the same type (ISO 2002). Therefore the type of ∅ is inferred to be ℙℙA. The TPTP representation to show that ℙ{{1}} = {∅,{{1}}} is then given by the following input to a reasoner:

```
% A typed TPTP set-theoretic problem:
% Show that P{{1}} = {Empty,{{1}}}.

% Types
fof(types, axiom,
  el(number_literal_1,t_A) &
  el(empty,t_PPA) &
  el(a,t_PA) &
  el(b,t_PPA) &
  el(c,t_PPPA) &
  el(d,t_PPPA)
).

% Reflexivity
fof(reflexivity, axiom,
  ! [X] : X = X
).

% Extensionality for types PA, PPA and PPPA
fof(extensionality, axiom,
  ![A,B]:((el(A,t_PA) & el(B,t_PA)) =>
      ((![X]:(el(X,t_A) => (el(X,A) <=> el(X,B)))) => A=B)) &
  ![A,B]:((el(A,t_PPA) & el(B,t_PPA)) =>
      ((![X]:(el(X,t_PA) => (el(X,A) <=> el(X,B)))) => A=B)) &
  ![A,B]:((el(A,t_PPPA) & el(B,t_PPPA)) =>
      ((![X]:(el(X,t_PPA) => (el(X,A) <=> el(X,B)))) => A=B))
).

% Empty = {}
fof(empty, axiom,
  ![X]:(el(X,t_PA) => (~el(X,empty)))
).

% A = {1}
fof(a_is_1, axiom,
  ![X]:(el(X,t_A) => (el(X,a) <=> (X = number_literal_1)))
).

% B = {A}
fof(b_is_a, axiom,
  ![X]:(el(X,t_PA) => (el(X,b) <=> (X = a)))
).

% C = P(B)
fof(c_is_power_b, axiom,
  ![X]:(el(X,t_PPA) => (el(X,c) <=> ![Y]:(el(Y,t_PA) => (el(Y,X) => el(Y,b)))))
).

% D = {Empty,{{1}}}
fof(d_is_empty_or_1, axiom,
  ![X]:(el(X,t_PPA) => (el(X,d) <=> (((X = empty) | (X = b)))))
).

% Goal clause C = D
fof(c_is_d, conjecture,
  c = d
).
```

The following should be noted about the above TPTP example. Each variable or constant is specified to be of a specific type. For example el(X,t_PPA) specifies that the type of

variable X is $\mathbb{PPA}$. The same would apply to any functor representing an expression. Lastly, the extensionality axiom must be given for each set type occurring in the problem.

The author of this dissertation wrote a system to translate a large part of Z into first-order logic in TPTP notation. Such system allowed us to automatically convert Z input to the desired notation to be used as input to the reasoner. A number of the heuristics were also applied to the input during this automated translation process. All input to the proof attempts reported on in this chapter were generated with the aid of the above system written in Java.

7.2 Discharging of Proof Obligations

In this section we take some of the proof obligations that arise from the order management system specification in Chapter 6 and show how they can be discharged using Vampire with the aid of the heuristics of Chapter 5.

7.2.1 CreateProduct Invariant

The *CreateProduct* operation adds a new product type to the stock. Its schema contains the following predicate:

$$products' = products \cup \{product?\}$$

The following proof shows that this predicate may also be derived from the other predicates:

$products'$

$= dom\ prodName'$ (invariant after)

$= dom\ (prodName \cup \{product? \mapsto name?\})$ (specification of CreateProduct)

$= dom\ prodName \cup dom\ \{product? \mapsto name?\}$ (fact about 'dom')

= dom prodName ∪ {product?} (fact about ‘dom’)

= products ∪ {product?} (invariant before)

We may, therefore, redefine the *CreateProduct* schema to exclude the *products′* predicate. Also, to keep the schema simple we exclude the price and quantity variables. The final expanded schema is given below:

LeanerCreateProduct

products: ℙ PRODUCT
prodName: PRODUCT ⤖ STRING
products′: ℙ PRODUCT
prodName′: PRODUCT ⤖ STRING
product?: PRODUCT
name?: STRING

---

dom prodName = products
dom prodName′ = products′
product? ∉ products
name? ∉ ran prodName
prodName′ = prodName ∪ {product? ↦ name?}

The proof conjecture can then be stated as:

LeanerCreateProduct ⊢ products′ = products ∪ {product?}

In the conversion to first-order logic all heuristics are applied except for the *products′* predicate to which the extensionality heuristic #1 and exemplification heuristic #4 were not applied.

The exclusion of the exemplification heuristic prevents the union operator’s definition from being used directly instead of a functor. With this transformation Vampire is unable to find a proof by terminating after 7 minutes with no more passive clauses left. We next apply the extensionality heuristic by defining the above set equality *products′* = *products* ∪ {*product*?} in terms of its elements as:

$$\forall x: PRODUCT \bullet x \in products' \Leftrightarrow x \in products \cup \{product?\}$$

With this transformation Vampire is able to find a proof in 1 minute 12 seconds. Next we apply the exemplification heuristic by using a direct definition of the union operator as:

$$\forall x: PRODUCT \bullet x \in products' \Leftrightarrow x \in \{y: PRODUCT \mid y \in products \vee y = product?\}$$

Vampire is then able to find a quick proof in 1 second. The input to this last proof attempt is shown in Appendix E.1.

The following table summarises the above results:

Extensionality (Heuristic #1)	Exemplification (Heuristic #4)	Time to find a proof
No	No	No proof after 7 minutes
Yes	No	72s
Yes	Yes	1s

From the above table we observe that the application of both the extensionality heuristic #1 and the exemplification heuristic #4 leads to a very short proof.

7.2.2 CreateProduct is Total

The *CreateProduct* operation adds a new product type. It is not a total operation since it is not defined for all possible inputs. The after state for example is not defined if the input product already exists. *CreateProductTotal* is an enhanced version that caters for all possible inputs. To show that *CreateProductTotal* is indeed total we need to show that its precondition is a partition (refer Section 6.13.4). This is done as two separate proof obligations (POs):

1. The first is to show that the disjunction of the constituent preconditions is a tautology.

2. Secondly we need to show that all the constituent preconditions are pairwise disjoint.

The following conjecture states that the precondition is a tautology (PO 1 above):

$$\vdash \text{pre } (\text{CreateProduct} \land \text{Success}) \lor \text{pre ProductAlreadyKnown} \lor \text{pre DuplicateProductName}$$

An expanded form of the above conjecture is:

Product
product?: PRODUCT
name?: STRING
price?: AMOUNT
quantity?: $\mathbb{N}$

$\vdash$

$(\exists\ \text{Product}'; \text{result!}: \text{REPORT} \bullet$
$\quad \text{product?} \notin \text{products}$
$\quad \text{name?} \notin \text{ran prodName}$
$\quad \text{products}' = \text{products} \cup \{\text{product?}\}$
$\quad \text{prodName}' = \text{prodName} \cup \{\text{product?} \mapsto \text{name?}\}$
$\quad \text{prodPrice}' = \text{prodPrice} \cup \{\text{product?} \mapsto \text{price?}\}$
$\quad \text{prodQuantity}' = \text{prodQuantity} \cup \{\text{product?} \mapsto \text{quantity?}\}$
$\quad \text{result!} = \text{success})$
$\lor$
$(\exists\ \text{Product}'; \text{result!}: \text{REPORT} \bullet$
$\quad \text{product?} \in \text{products}$
$\quad \text{products}' = \text{products}$
$\quad \text{prodName}' = \text{prodName}$
$\quad \text{prodPrice}' = \text{prodPrice}$
$\quad \text{prodQuantity}' = \text{prodQuantity}$
$\quad \text{result!} = \text{already\_known})$
$\lor$
$(\exists\ \text{Product}'; \text{result!}: \text{REPORT} \bullet$
$\quad \text{product?} \notin \text{products}$

name? ∈ ran prodName

products′ = products

prodName′ = prodName

prodPrice′ = prodPrice

prodQuantity′ = prodQuantity

result! = duplicate_name)

Most heuristics have been applied in the conversion to first-order logic except for the divide-and-conquer heuristic #3. With such transformed input Vampire is unable to find a proof after 30 minutes. Owing to the complexity of the proof, the transformed conjecture is a conjunction of 8 terms. We can therefore apply the divide-and-conquer heuristic #3 to the above conjecture by splitting it into 8 separate conjectures and corresponding proof attempts. Vampire is then still unable to find any proof after 30 minutes for any of the conjectures. These 8 conjectures also consist of conjunctions onto which the divide-and-conquer heuristic may be applied to further. This application results in 56 proofs. Each of these proofs is allocated 5 minutes of which only 1 is found in less than 1 second. The divide-and-conquer heuristic is further applied to the remaining 55 proofs, resulting in 440 proof attempts, all with the aid of the Java system described in Section 7.1 above. Each of these proofs is again allocated 5 minutes of which 438 are found in less than 1 second. The remaining two proofs are further split up into four proof attempts of which three proofs are found in less than one second and the last proof in 5.5 seconds. This one remaining proof is finally split into two proofs for which Vampire finds refutations in 1 second each. The following table summarises the results of these proof attempts:

Proof Attempt #	Number of Proof Obligations	Number of Refutations found	Number of failed attempts remaining
1	1	0	1
2	8	0	8
3	56	1	55

Proof Attempt #	Number of Proof Obligations	Number of Refutations found	Number of failed attempts remaining
4	440	438	2
5	4	3	1
6	1	1	0

As may be observed from the above table, the divide-and-conquer heuristic #3 proved to be very useful.

Lastly we need to show that all the constituent preconditions are pairwise disjoint (PO 2 above) as given by the following conjecture:

$$\vdash (\text{pre } (\text{CreateProduct} \wedge \text{Success}) \wedge \text{pre ProductAlreadyKnown}) = \varnothing \wedge$$
$$(\text{pre } (\text{CreateProduct} \wedge \text{Success}) \wedge \text{pre DuplicateProductName}) = \varnothing \wedge$$
$$(\text{pre ProductAlreadyKnown} \wedge \text{pre DuplicateProductName}) = \varnothing$$

Suppose we exclude the extensionality and divide-and-conquer heuristics in the conversion to first-order logic. Vampire is then unable to find a proof after 30 minutes. The conjecture to be proved is a conjunction that enables us to apply the divide-and-conquer heuristic. This results in three separate proofs. Vampire is now able to find quick refutations for the first and third proofs in 18 and 24 seconds respectively, but is still unable to find a refutation for the second proof attempt after 30 minutes. The remaining proof is a set equality that enables us to apply the extensionality heuristic as:

$$\forall x\ [x \in (\text{pre } (\text{CreateProduct} \wedge \text{Success}) \wedge \text{pre DuplicateProductName}) \Leftrightarrow x \in \varnothing]$$

The application of the extensionality heuristic enables Vampire to find a quick proof in 1 second. We can similarly apply extensionality to the first and third proofs for which Vampire then also find quick proofs in 1 second.

The following table summarises the above results:

Divide-and-conquer (Heuristic #3)	Extensionality (Heuristic #1)	Time to find a proof
No	No	No proof found
Yes	No	18s, no proof, 24s (3 proof attempts)
Yes	Yes	1s, 1s, 1s (3 proof attempts)

7.2.3 ProcessOrder set contents

The *ProcessOrder* operation (refer Section 6.7.5) changes the state of a *pending* order to *processed* and removes the relevant product quantities from stock as indicated on the order line items. The specification of the *ProcessOrder* operation is repeated here for convenience:

ProcessOrder

$\Delta Order$
$\Delta Product$
$\Xi Item$
order?: ORDER

---

order? $\in$ orders
orderStatus(order?) = pending
$\forall i$: items $\bullet$ itemOrder(i) = order? $\Rightarrow$ prodQuantity(itemProduct(i)) – itemQuantity(i) $\geqslant$ 0
orders′ = orders
orderDate′ = orderDate
orderStatus′ = orderStatus $\oplus$ {order? $\mapsto$ processed}
orderCustomer′ = orderCustomer
products′ = products
prodName′ = prodName
prodPrice′ = prodPrice
prodQuantity′ = prodQuantity $\oplus$
{i: items | itemOrder(i) = order? $\bullet$ itemProduct(i) $\mapsto$ prodQuantity(itemProduct(i)) – itemQuantity(i)}

ProcessOrder updates *orderStatus* and *prodQuantity*. A proof obligation therefore arises from *ProcessOrder* to show that:

- Other order statuses are not affected.
- Product quantities not appearing as line items in the order are not affected.

The conjecture for the above double proof obligation is specified as:

$$
\begin{aligned}
&\text{ProcessOrder} \vdash \\
&\quad (\forall\ s: \text{orderStatus} \mid s.1 \neq \text{order?} \bullet s \in \text{orderStatus}') \wedge \\
&\quad (\forall\ q: \text{prodQuantity} \mid \neg(\exists\ i: \text{items} \bullet \text{itemOrder}(i) = \text{order?} \wedge \text{itemProduct}(i) = q.1) \bullet \\
&\qquad q \in \text{prodQuantity}')
\end{aligned}
$$

The divide-and-conquer heuristic has not been applied in the transformation to first-order logic. Furthermore the exemplification heuristic was not applied to the instances of the above override operators ($\oplus$) which are used in the specification of the after state variables *orderStatus*′ and *prodQuantity*′. Functors are therefore used instead of direct definitions of the override operator.

With this transformation Vampire is unable to find a refutation after 30 minutes. We next apply the divide-and-conquer heuristic by doing two separate proofs for *orderStatus*′ and *prodQuantity*′.

Vampire is then able to find a proof for the *orderStatus*′ sub-problem in 16 minutes 22 seconds but is still unable to find a proof for the *prodQuantity*′ sub-problem after 30 minutes.

The *orderStatus*′ sub-problem still contains a definition of the override operator for *prodQuantity*′. Similarly the *prodQuantity*′ sub-problem also contains a definition of the *orderStatus*′ override operator. We can therefore apply the redundant information heuristic by removing these unnecessary definitions. Vampire is then able to find a proof

for the *orderStatus′* sub-problem in 5 minutes 26 seconds and is still unable to find a proof for the *prodQuantity′* sub-problem after 30 minutes.

Lastly we apply the exemplification heuristic to the override operators of the two sub-problems by replacing the indirect definitions via functors with direct definitions. Vampire is then able to find quick proofs for both problems in 1 second.

The following table summarises the above results:

Divide-and-conquer (Heuristic #3)	Redundant information (Heuristic #8)	Exemplification (Heuristic #4)	Time to find a proof
No	No	No	No proof
Yes	No	No	982s, no proof
Yes	Yes	No	326s, no proof
Yes	Yes	Yes	1s, 1s

7.2.4 CreateDeleteItem leaves state unchanged

The *CreateItem* and *DeleteItem* operations respectively add and remove a line item from an order. An operation interaction proof obligation (refer Section 6.13.5) that arises is to show that adding an item and then immediately removing the item will leave the *Item* state unchanged. This proof obligation is specified with the following theorem:

$$\text{CreateItem} \fatsemi \text{DeleteItem} \vdash \Xi \text{Item}$$

An expanded version of this conjecture looks as follows:

$$\Delta \text{Item}$$
$$\Xi \text{Product}$$
$$\text{item?}: \text{ITEM}$$
$$\text{order?}: \text{ORDER}$$

quantity?: $\mathbb{N}_1$

product?: PRODUCT

|

$\exists Item'' \bullet$

$item? \notin items \land$

$items'' = items \cup \{item?\} \land$

$itemOrder'' = itemOrder \cup \{item? \mapsto order?\} \land$

$itemPrice'' = itemPrice \cup \{item? \mapsto prodPrice(product?)\} \land$

$itemQuantity'' = itemQuantity \cup \{item? \mapsto quantity?\} \land$

$itemProduct'' = itemProduct \cup \{item? \mapsto product?\} \land$

$item? \in items'' \land$

$items' = items'' \setminus \{item?\} \land$

$itemOrder' = \{item?\} \lhd itemOrder'' \land$

$itemPrice' = \{item?\} \lhd itemPrice'' \land$

$itemQuantity' = \{item?\} \lhd itemQuantity'' \land$

$itemProduct' = \{item?\} \lhd itemProduct''$

$\vdash$

Ξ Item

The double prime ($''$) decorated components above are the intermediate states that link the outputs of *CreateItem* with the inputs of *DeleteItem*.

In the conversion to first-order logic all heuristics have been applied except for the terms resulting from Ξ*Item* to which the extensionality heuristic has not been applied. With this conversion Vampire is unable to find a proof after 30 minutes.

When fully expanded, the term Ξ*Item* becomes the following conjunction:

$items \in \mathbb{P}\ ITEM \land items' \in \mathbb{P}\ ITEM \land$

$itemOrder \in ITEM \pfun ORDER \land itemOrder' \in ITEM \pfun ORDER \land$

$itemPrice \in ITEM \pfun AMOUNT \land itemPrice' \in ITEM \pfun AMOUNT \land$

$itemQuantity \in ITEM \pfun \mathbb{N}_1 \land itemQuantity' \in ITEM \pfun \mathbb{N}_1 \land$

$itemProduct \in ITEM \pfun PRODUCT \land itemProduct' \in ITEM \pfun PRODUCT \land$

$\text{dom itemOrder} = \text{items} \land \text{dom itemOrder}' = \text{items}' \land$

$\text{dom itemPrice} = \text{items} \land \text{dom itemPrice}' = \text{items}' \land$

$\text{dom itemQuantity} = \text{items} \land \text{dom itemQuantity}' = \text{items}' \land$

$\text{dom itemProduct} = \text{items} \land \text{dom itemProduct}' = \text{items}' \land$

$(\forall i_1, i_2 : \text{items} \bullet i_1 \neq i_2 \Rightarrow \text{itemOrder}(i_1) \neq \text{itemOrder}(i_2) \lor \text{itemProduct}(i_1) \neq \text{itemProduct}(i_2)) \land$

$(\forall i_1, i_2 : \text{items}' \bullet i_1 \neq i_2 \Rightarrow \text{itemOrder}'(i_1) \neq \text{itemOrder}'(i_2) \lor \text{itemProduct}'(i_1) \neq \text{itemProduct}'(i_2)) \land$

$\text{items} = \text{items}' \land$

$\text{itemOrder} = \text{itemOrder}' \land$

$\text{itemPrice} = \text{itemPrice}' \land$

$\text{itemQuantity} = \text{itemQuantity}' \land$

$\text{itemProduct} = \text{itemProduct}'$

Of all these conjuncts above only the last 5 equality conjuncts are required to prove that the before and after states are the same:

$\text{items} = \text{items}' \land$

$\text{itemOrder} = \text{itemOrder}' \land$

$\text{itemPrice} = \text{itemPrice}' \land$

$\text{itemQuantity} = \text{itemQuantity}' \land$

$\text{itemProduct} = \text{itemProduct}'$

We can therefore apply the redundant information heuristic #8 by only keeping the equality conjuncts. In this case the use of the redundant information heuristic is similar to the divide-and-conquer heuristic #3 except that only one half of the divide will be pursued. Vampire is now able to find a proof in 1 minute 44 seconds.

Similarly to term $\Xi Item$, the use of $\Xi Product$ in the problem statement also results in a large number of unnecessary formulae. Of these only *prodPrice* is being used in the function application *prodPrice*(*product*?). We therefore apply the redundant information heuristic again by only keeping the variable declaration of *prodPrice* in $\Xi Product$. Vampire is then able to find a proof in 1 minute 30 seconds.

The next target for the redundant information heuristic is the predicate in schema *Item* that states that *itemOrder* and *itemProduct* combinations must be unique:

$$\forall i_1, i_2 : items \bullet i_1 \neq i_2 \Rightarrow itemOrder(i_1) \neq itemOrder(i_2) \lor itemProduct(i_1) \neq itemProduct(i_2)$$

Removal of this predicate in each of *Item*, *Item′* and *Item″* enables Vampire to find a proof in 1 minute 5 seconds.

We next apply the divide-and-conquer heuristic by finding separate equality proofs for the five state variables of *Item*. Vampire now finds proofs in 0.2, 59, 55, 29 and 22 seconds for the respective conjectures.

The separation of the conjecture into five separate conjectures allows us to apply the redundant information heuristic even further. For example, the proof to show that *itemOrder* = *itemOrder′* does not require any terms that reference the various decorations of *itemPrice*, *itemQuantity* and *itemProduct*. If we remove such unnecessary terms for each of the five conjectures, then Vampire is able to find proofs in 0, 1.5, 6.8, 13.5 and 1.4 seconds respectively.

The following table summarises the above results in order:

Redundant information (Heuristic #8)	Divide-and-conquer (Heuristic #3)	Time to find a proof
Not applied	No	No proof
Keep only equality predicates in Ξ*Item*	No	104s
Keep only declaration of *prodPrice* in Ξ*Product*	No	90s
Remove unique *itemOrder*,	No	65s

Redundant information (Heuristic #8)	Divide-and-conquer (Heuristic #3)	Time to find a proof
itemProduct predicate in *Item*, *Item′* and *Item″*		
	Yes	0.2s, 59s, 55s, 29s, 22s
Remove declarations of unrelated variables and predicates referring to them in *Item*, *Item′* and *Item″*	Yes	0s, 1.5s, 6.8s, 13.5s, 1.4s

7.2.5 After State Type of CancelOrder

An order that has not been processed yet, that is it is in a pending status, can be cancelled. This is done with the *CancelOrder* operation:

CancelOrder

$\Delta Order$
$order?: ORDER$

$order? \in orders$
$orderStatus(order?) = pending$
$orders' = orders$
$orderDate' = orderDate$
$orderStatus' = orderStatus \oplus \{order? \mapsto cancelled\}$
$orderCustomer' = orderCustomer$

Components *orderStatus* and *orderStatus′* are partial functions from *ORDER* to *STATUS* and their carrier types are $ORDER \leftrightarrow STATUS$. Specification tools that do type checking often only ensure that the carrier type of a variable is correct (see Section 6.13.3), but do not cater for more restricted type checking. For the *CancelOrder* operation they,

therefore, do not verify whether *orderStatus′* is a partial function. The following conjecture can be used to show that *orderStatus′* is a partial function:

Order

orders′: ℙ ORDER

orderDate′: ORDER ↔ DATE

orderStatus′: ORDER ↔ STATUS

orderCustomer′: ORDER ↔ CUSTOMER

order?: ORDER

|

dom orderDate′ = orders′

dom orderStatus′ = orders′

dom orderCustomer′ = orders′

order? ∈ orders

orderStatus(order?) = pending

orders′ = orders

orderDate′ = orderDate

orderStatus′ = orderStatus ⊕ {order? ↦ cancelled}

orderCustomer′ = orderCustomer

⊢

orderStatus′: ORDER ⇸ STATUS

In the conversion to first-order logic most heuristics are applied except for the following conjunct to which the exemplification heuristic was not applied to the override operator:

orderStatus′ = orderStatus ⊕ {order? ↦ cancelled}

The exclusion of the exemplification heuristic results in the use of a functor instead of a direct definition of the override operator. This enables us to manipulate the override operator's definition independently to investigate the utility of some other heuristics. We therefore do not use the following generic definition of the override operator in the standard Z toolkit:

$$[X, Y]$$

$$\_ \oplus \_ : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$$

$$\forall r, s : X \leftrightarrow Y \bullet r \oplus s = ((\text{dom } s) \lhd r) \cup s)$$

but rather define it instantiated for *ORDER* and *STATUS*, as well as defining the expression ((dom s) ⩤ r) ∪ s) directly instead of using the domain, domain anti-restriction and union set operations that are evident in the above definition:

$$\_ \oplus \_ : (\text{ORDER} \leftrightarrow \text{STATUS}) \times (\text{ORDER} \leftrightarrow \text{STATUS}) \rightarrow (\text{ORDER} \leftrightarrow \text{STATUS})$$

$$\forall r, s : \text{ORDER} \leftrightarrow \text{STATUS} \bullet r \oplus s = \{x: \text{ORDER} \times \text{STATUS} \mid (x \in r \land \forall p: s \bullet x.1 \neq p.1) \lor x \in s\}$$

Suppose we decide to not use the element structure and multivariate functor heuristics in the definition of the override operator above for the conversion to first-order logic. Vampire is then unable to find a proof after 30 minutes. Next we apply the element structure heuristic to the override operator in its transformation to first-order logic. The heuristic is applied by converting all instances of variables that are Cartesian product types to tuples. One such example is the definition of variable *x* in the above definition of ⊕. It can be rewritten as:

$$\{x_1: \text{ORDER}; x_2: \text{STATUS} \mid ((x_1,x_2) \in r \land \forall p: s \bullet x1 \neq p.1) \lor (x_1,x_2) \in s\}$$

With this transformation of the override operator Vampire is able to find a proof in 2 minutes 47 seconds. However, if the element structure heuristic is only applied to *x* as in the example above and not to all instances of Cartesian product variables then Vampire only finds a proof after 5 minutes 53 seconds. This is an example of the resonance heuristic #10 that requires corresponding terms to have a syntactically similar structure to aid the resolution process.

Next we discard the element structure heuristic and instead apply the multivariate functor heuristic. We do this by replacing the universal variable *s* in the definition of the override operator with the constant {*order*? ↦ *cancelled*}:

$$
\begin{array}{|l}
\_ \oplus \_ : (\mathrm{ORDER} \leftrightarrow \mathrm{STATUS}) \times (\mathrm{ORDER} \leftrightarrow \mathrm{STATUS}) \rightarrow (\mathrm{ORDER} \leftrightarrow \mathrm{STATUS}) \\
\hline
\forall\, r : \mathrm{ORDER} \leftrightarrow \mathrm{STATUS} \bullet r \oplus \{\mathrm{order?} \mapsto \mathrm{cancelled}\} = \\
\quad \{x: \mathrm{ORDER} \times \mathrm{STATUS} \mid \\
\quad\quad (x \in r \wedge \forall\, p: \{\mathrm{order?} \mapsto \mathrm{cancelled}\} \bullet x.1 \neq p.1) \vee x \in \{\mathrm{order?} \mapsto \mathrm{cancelled}\}\}
\end{array}
$$

Vampire now finds a proof in 3 minutes 31 seconds.

Lastly we apply both the element structure and the multivariate functor heuristics to the definition of the override operator. Vampire is then able to find a quick proof in 1 second. The input to this last successful proof attempt appears in Appendix E.2.

The following table summarises the above results:

Element structure (Heuristic #7)	Resonance (Heuristic #10)	Multivariate functor (Heuristic #5)	Time to find a proof
No	No	No	No proof
Yes	Yes	No	167s
Yes	No	No	353s
No	No	Yes	211s
Yes	Yes	Yes	1s

7.3 Conclusion

Chapter 6 introduced a case study of an order management system specified in Z and highlighted typical proof obligations that arise from such specifications. This chapter started by discussing the conversion of Z proof obligations to first-order logic. Finally five proof obligations that arose from the case study were converted to first-order logic with the aid of a Java program written by the author of this dissertation and discharged using the Vampire reasoner with the help of the presented heuristics.

From the successes reported in this chapter we see that the heuristics presented in Chapter 5 are indeed useful, not only in discharging traditional set-theoretic problems, but also problems on a larger scale, typically those present in Z specifications.

The next and final chapter takes stock of what was set out to be achieved in Chapter 1 and to what extent these aims have been met.

Chapter 8

Summary and Conclusions

In this final chapter we revisit our original research question and hypothesis from Chapter 1. We then discuss how the proposed approach in Section 1.4 was applied and what our findings were. The chapter concludes with a discussion of the directions that future work in this area could follow.

8.1 Contributions of this Dissertation

Van der Poll and Labuschagne developed a suite of 14 heuristics (Van der Poll & Labuschagne 1999, Van der Poll 2000) to aid the Otter automated reasoner (McCune 2003) in finding proofs for set-theoretic problems. Otter became dated since the work of Van der Poll and Labuschagne and its performance at the annual CASC (Pelletier *et al*. 2002, Sutcliffe & Suttner 2006) competitions since 2000 indicates that it cannot be considered a state of the art prover anymore. It has also been decommissioned by its author and replaced by Prover 9 (McCune 2009). Nevertheless, Otter was used to arrive at the VdPL heuristics described throughout this dissertation and its use led to the question of whether the VdPL heuristics are applicable to other resolution-based reasoners that have since surpassed Otter in performance.

Our hypothesis was:

> The set-theoretic heuristics developed by Van der Poll and Labuschagne are applicable to state of the art resolution-based automated theorem provers.

We selected Vampire (Riazanov & Voronkov 2002) in Chapter 4 as our primary and state of the art automated theorem prover to verify our hypothesis. Vampire was chosen because it is a resolution-based automated reasoner, performed consistently well in the annual CASC (Pelletier *et al*. 2002, Sutcliffe & Suttner 2006) competitions. Vampire also solved more set-theoretic problems than any of the other competing provers in the

period from 2002 to 2007 across all divisions. Gandalf (Tammet 1997) was chosen as the secondary resolution-based reasoner since it also had reasonable success at the CASC competitions some time ago.

In Chapter 5 each of the VdPL heuristics obtained through the use of Otter was stated and tested in turn on sample set-theoretic problems using Vampire and Gandalf. Otter was used to discharge the proof. After a failed proof attempt the relevant heuristic was applied to the problem specification and it enabled Otter to find a proof. The original problem was then discharged on Vampire and Gandalf. In some of the cases Vampire and Gandalf were able to solve the original problem without the application of the heuristic. In these cases the problem complexity was increased to such an extent that a proof could not be found. The heuristic was then similarly applied to failed proof attempts. It was found that Vampire needed 10 and Gandalf 9 of the 11 heuristics evaluated. However, Vampire generally found proofs quicker, hence it was chosen as the reasoner to be used in the rest of our work.

In Chapter 6 an order management case study was developed using the Z specification language (Spivey 1992) that is based on first-order logic and a strongly-typed set theory. Some of the proof obligations that arose from the case study were selected in Chapter 7, converted to first-order logic and discharged using Vampire. In all these cases various combinations of heuristics were required to enable Vampire to find proofs.

We have therefore provided empirical evidence of the utility of the VdPL heuristics to state of the art resolution-based automated theorem provers in the domain of set-theoretic problems.

8.2 Future Work

The proofs in this work were done using the default settings of Vampire and Gandalf. Three of the 14 heuristics were not evaluated because they require changes to these default settings. Future empirical work involving changing and fine-tuning some of these settings may yield further useful results.

We limited our selection of reasoners to resolution-based provers since Otter is resolution-based. The applicability of the heuristics to other types of automated reasoners could be investigated, for example tableau and term rewriting (Bundy 1999). The problem domain was also limited to set theory. Similarly the utility of the heuristics can be tested on other problem domains, especially those with similar characteristics as set theory, e.g. deeply nested constructs.

The translation of Z specification proof obligations to first-order logic in Chapter 7 had to incorporate the Z type information into the resulting problem. It was found that the additional typed terms in the resulting clauses help to restrict irrelevant search paths resulting from the resolution of clauses with unrelated types. Some resolution of these incompatible clauses can, however, still take place but the resolvents can only be reduced up to the type terms where further reductions are prevented. It is plausible that these undesired unifications may be prevented up front if the type information becomes part of the TPTP notation (Sutcliffe & Suttner 1998) that is used as the input to our proof attempts. The incorporation of type information as manifested in this dissertation coincides with the mechanism proposed by Claessen and Sutcliffe (2008). The implementation of type information into the unification algorithms of next generation theorem provers may therefore create a more efficient class of reasoners against which the applicability of the VdPL heuristics may be tested.

Since the VdPL heuristics appear to be more universally applicable as was known before one may now consider building a library of such recognisable patterns in proof attempts, aimed at automatically transforming a specifier's input to a reasoner prior to an attempt at discharging a proof obligation. Some progress in this regard has already been made as part of this work. The conversion of Z proof obligations in Chapter 7 to first-order logic was largely automated through the use a Java program developed for this purpose. It was found that some of the heuristics could to some extent be applied automatically during the conversion. Further work in this area could result in an automated conversion to first-order logic of the full Z language with the VdPL heuristics applied automatically where applicable.

Appendix A

Resolution Deductions of the Farmer, Wolf, Goat and Cabbage (FWGC) Puzzle

A.1 A Possible Refutation Deduction of the FWGC Puzzle

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_2 =	$\neg S(fa, x, y, z) \vee \neg SAFE(fa, x, y, z) \vee$ $\neg SAFE(fh, x, y, z) \vee S(fh, x, y, z)$	Farmer returns
C_3 =	$\neg S(fh, gh, y, z) \vee \neg SAFE(fh, gh, y, z) \vee$ $\neg SAFE(fa, ga, y, z) \vee S(fa, ga, y, z)$	Farmer takes goat across
C_4 =	$\neg S(fa, ga, y, z) \vee \neg SAFE(fa, ga, y, z) \vee$ $\neg SAFE(fh, gh, y, z) \vee S(fh, gh, y, z)$	Farmer returns goat
C_5 =	$\neg S(fh, x, ch, z) \vee \neg SAFE(fh, x, ch, z) \vee$ $\neg SAFE(fa, x, ca, z) \vee S(fa, x, ca, z)$	Farmer takes cabbage across
C_6 =	$\neg S(fa, x, ca, z) \vee \neg SAFE(fa, x, ca, z) \vee$ $\neg SAFE(fh, x, ch, z) \vee S(fh, x, ch, z)$	Farmer returns cabbage
C_7 =	$\neg S(fh, x, y, wh) \vee \neg SAFE(fh, x, y, wh) \vee$ $\neg SAFE(fa, x, y, wa) \vee S(fa, x, y, wa)$	Farmer takes wolf across
C_8 =	$\neg S(fa, x, y, wa) \vee \neg SAFE(fa, x, y, wa) \vee$ $\neg SAFE(fh, x, y, wh) \vee S(fh, x, y, wh)$	Farmer returns wolf

C_9 =	SAFE(fh, gh, ch, wh)	
C_{10} =	SAFE(fh, gh, ch, wa)	
C_{11} =	SAFE(fh, gh, ca, wh)	
C_{12} =	SAFE(fh, gh, ca, wa)	
C_{13} =	SAFE(fh, ga, ch, wh)	
C_{14} =	SAFE(fa, gh, ca, wa)	
C_{15} =	SAFE(fa, ga, ch, wh)	
C_{16} =	SAFE(fa, ga, ch, wa)	
C_{17} =	SAFE(fa, ga, ca, wh)	
C_{18} =	SAFE(fa, ga, ca, wa)	
C_{19} =	S(fh, gh, ch, wh)	Start state
C_{20} =	¬S(fa, ga, ca, wa)	Goal state negated
	Take goat across	
C_{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_{22} =	¬SAFE(fa, ga, ch, wh) ∨ S(fa, ga, ch, wh)	Resolvent of C_9 and C_{21}
C_{23} =	S(fa, ga, ch, wh)	Resolvent of C_{15} and C_{22}
	Farmer returns	
C_{24} =	¬SAFE(fa, ga, ch, wh) ∨ ¬SAFE(fh, ga, ch, wh) ∨ S(fh, ga, ch, wh)	Resolvent of C_2 and C_{23} Unifier {ga/x, ch/y, wh/z}
C_{25} =	¬SAFE(fh, ga, ch, wh) ∨ S(fh, ga, ch, wh)	Resolvent of C_{15} and C_{24}
C_{26} =	S(fh, ga, ch, wh)	Resolvent of C_{13} and C_{25}

	Take wolf across	
C_{27} =	¬SAFE(fh, ga, ch, wh) ∨ ¬SAFE(fa, ga, ch, wa) ∨ S(fa, ga, ch, wa)	Resolvent of C_7 and C_{26} Unifier {ga/x, ch/y}
C_{28} =	¬SAFE(fa, ga, ch, wa) ∨ S(fa, ga, ch, wa)	Resolvent of C_{13} and C_{27}
C_{29} =	S(fa, ga, ch, wa)	Resolvent of C_{16} and C_{28}
	Bring goat back	
C_{30} =	¬SAFE(fa, ga, ch, wa) ∨ ¬SAFE(fh, gh, ch, wa) ∨ S(fh, gh, ch, wa)	Resolvent of C_4 and C_{29} Unifier {ch/y, wa/z}
C_{31} =	¬SAFE(fh, gh, ch, wa) ∨ S(fh, gh, ch, wa)	Resolvent of C_{16} and C_{30}
C_{32} =	S(fh, gh, ch, wa)	Resolvent of C_{10} and C_{31}
	Take cabbage across	
C_{33} =	¬SAFE(fh, gh, ch, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ S(fa, gh, ca, wa)	Resolvent of C_5 and C_{32} Unifier {gh/x, wa/z}
C_{34} =	¬SAFE(fa, gh, ca, wa) ∨ S(fa, gh, ca, wa)	Resolvent of C_{10} and C_{33}
C_{35} =	S(fa, gh, ca, wa)	Resolvent of C_{14} and C_{34}
	Farmer returns	
C_{36} =	¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ S(fh, gh, ca, wa)	Resolvent of C_2 and C_{35} Unifier {gh/x, ca/y, wa/z}
C_{37} =	¬SAFE(fh, gh, ca, wa) ∨ S(fh, gh, ca, wa)	Resolvent of C_{14} and C_{36}
C_{38} =	S(fh, gh, ca, wa)	Resolvent of C_{12} and C_{37}
	Take goat across	
C_{39} =	¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa) ∨ S(fa, ga, ca, wa)	Resolvent of C_3 and C_{38} Unifier {ca/y, wa/z}

C_{40} =	¬SAFE(fa, ga, ca, wa) ∨ S(fa, ga, ca, wa)	Resolvent of C_{12} and C_{39}
C_{41} =	S(fa, ga, ca, wa)	Resolvent of C_{18} and C_{40}
	Goal achieved	
C_{42} =	□	Resolvent of C_{20} and C_{41}

A.2 Level Saturation Method Deduction of FWGC Puzzle

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_3 =	¬S(fh, gh, y, z) ∨ ¬SAFE(fh, gh, y, z) ∨ ¬SAFE(fa, ga, y, z) ∨ S(fa, ga, y, z)	Farmer takes goat across
C_{12} =	SAFE(fh, gh, ca, wa)	
C_{14} =	SAFE(fa, gh, ca, wa)	
C_{18} =	SAFE(fa, ga, ca, wa)	
C_{19} =	S(fa, gh, ca, wa)	Start state
C_{20} =	¬S(fa, ga, ca, wa)	Goal state negated
	Saturation Level 1	
C_{21} =	¬S(fh, gh, y, z) ∨ ¬SAFE(fh, gh, y, z) ∨ ¬SAFE(fa, ga, y, z)	C_2, C_3 on S(fa, ga, y, z)
C_{22} =	¬SAFE(fh, gh, y, z) ∨ ¬SAFE(fa, ga, y, z) ∨ S(fa, ga, y, z)	C_2, C_3 on S(fh, gh, y, z)
C_{23} =	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ S(fh, gh, ca, wa)	C_2, C_{12}

C_{24} =	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ S(fh, gh, ca, wa)	C_2, C_{14}
C_{25} =	¬S(fa, ga, ca, wa) ∨ ¬SAFE(fh, ga, ca, wa) ∨ S(fh, ga, ca, wa)	C_2, C_{18}
C_{26} =	¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ S(fh, gh, ca, wa)	C_2, C_{19}
C_{27} =	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa) ∨ S(fa, ga, ca, wa)	C_3, C_{12}
C_{28} =	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ S(fa, ga, ca, wa)	C_3, C_{18}
C_{29} =	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_3, C_{20}
	Saturation Level 2	
	¬S(fa, gh, y, z) ∨ ¬SAFE(fa, gh, y, z) ∨ ¬SAFE(fh, gh, y, z) ∨ ¬SAFE(fa, ga, y, z)	C_2, C_{21} on S(fh, gh, y, z)
	¬SAFE(fa, ga, y, z) ∨ ¬SAFE(fh, ga, y, z) ∨ S(fh, ga, y, z) ∨ ¬SAFE(fh, gh, y, z)	C_2, C_{22} on S(fa, ga, y, z)
	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fh, ga, ca, wa) ∨ S(fa, ga, ca, wa)	C_2, C_{27} on S(fh, gh, ca, wa)
	¬SAFE(fa, ga, ca, wa) ∨ ¬SAFE(fh, ga, ca, wa) ∨ S(fh, ga, ca, wa) ∨ ¬S(fh, gh, ca, wa)	C_2, C_{27} on S(fa, ga, ca, wa)
	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ S(fa, ga, ca, wa)	C_2, C_{28} on S(fh, gh, ca, wa)
	¬SAFE(fa, ga, ca, wa) ∨ ¬SAFE(fh, ga, ca, wa) ∨ S(fh, ga, ca, wa) ∨ ¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa)	C_2, C_{28} on S(fa, ga, ca, wa)
	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_2, C_{29} on S(fh, gh, ca, wa)
	¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa) ∨	C_3, C_{23} on

	S(fa, ga, ca, wa) ∨ ¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa)	S(fh, gh, ca, wa)
	¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa) ∨ S(fa, ga, ca, wa) ∨ ¬S(fa, gh, ca, wa)	C_3, C_{24} on S(fh, gh, ca, wa)
	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa) ∨ ¬SAFE(fh, ga, ca, wa) ∨ S(fh, ga, ca, wa)	C_3, C_{25} on S(fa, ga, ca, wa)
	¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa) ∨ S(fa, ga, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa)	C_3, C_{26} on S(fh, gh, ca, wa)
	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{12}, C_{21}
	¬SAFE(fa, ga, ca, wa) ∨ S(fa, ga, ca, wa)	C_{12}, C_{22}
	¬S(fa, gh, ca, wa) ∨ S(fh, gh, ca, wa)	C_{12}, C_{24}
	¬SAFE(fa, gh, ca, wa) ∨ S(fh, gh, ca, wa)	C_{12}, C_{26}
	¬S(fh, gh, ca, wa) ∨ S(fa, ga, ca, wa)	C_{12}, C_{28}
	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{12}, C_{29}
	¬S(fa, gh, ca, wa) ∨ S(fh, gh, ca, wa)	C_{14}, C_{23}
	¬SAFE(fh, gh, ca, wa) ∨ S(fh, gh, ca, wa)	C_{14}, C_{26}
	¬S(fh, gh, y, z) ∨ ¬SAFE(fh, gh, y, z)	C_{18}, C_{21}
	¬SAFE(fh, gh, y, z) ∨ S(fa, ga, y, z)	C_{18}, C_{22}
	¬S(fh, gh, ca, wa) ∨ S(fa, ga, ca, wa)	C_{18}, C_{27}
	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa)	C_{18}, C_{29}
	¬SAFE(fa, gh, ca, wa) ∨ S(fh, gh, ca, wa)	C_{19}, C_{23}
	¬SAFE(fh, gh, ca, wa) ∨ S(fh, gh, ca, wa)	C_{19}, C_{24}

	¬SAFE(fh, gh, y, z) ∨ ¬SAFE(fa, ga, y, z)	C_{20}, C_{22}
	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{20}, C_{27}
	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa)	C_{20}, C_{28}

A.3 UR-resolution Deduction of FWGC Puzzle

	Take goat across	
C_{21} =	S(fa, ga, ch, wh)	N: C_3 E: C_9, C_{15}, C_{19}
	Farmer returns	
C_{22} =	S(fh, ga, ch, wh)	N: C_2 E: C_{13}, C_{15}, C_{21}
	Take wolf across	
C_{23} =	S(fa, ga, ch, wa)	N: C_7 E: C_{13}, C_{16}, C_{22}
	Bring goat back	
C_{24} =	S(fh, gh, ch, wa)	N: C_4 E: C_{10}, C_{16}, C_{23}
	Take cabbage across	
C_{25} =	S(fa, gh, ca, wa)	N: C_5 E: C_{10}, C_{14}, C_{24}
	Farmer returns	
C_{26} =	S(fh, gh, ca, wa)	N: C_2 E: C_{12}, C_{14}, C_{25}
	Take goat across	
C_{27} =	S(fa, ga, ca, wa)	N: C_3 E: C_{12}, C_{18}, C_{26}
	Goal achieved	
C_{28} =	□	C_{20}, C_{27}

A.4 Illustration of Set-of-support Strategy of FWGC Puzzle

	Given set	
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_3 =	¬S(fh, gh, y, z) ∨ ¬SAFE(fh, gh, y, z) ∨ ¬SAFE(fa, ga, y, z) ∨ S(fa, ga, y, z)	Farmer takes goat across
C_{12} =	SAFE(fh, gh, ca, wa)	
C_{14} =	SAFE(fa, gh, ca, wa)	
C_{18} =	SAFE(fa, ga, ca, wa)	
C_{19} =	S(fa, gh, ca, wa)	Start state
	Set-of-support	
C_{20} =	¬S(fa, ga, ca, wa)	Goal state negated
	Saturation Level 1	
C_{21} =	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_3, C_{20}
	Saturation Level 2	
C_{22} =	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_2, C_{21} on S(fh, gh, ca, wa)
C_{23} =	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{12}, C_{21}
C_{24} =	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa)	C_{18}, C_{21}

	Saturation Level 3	
C_{25} =	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_2, C_{23}
C_{26} =	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa)	C_2, C_{24}
C_{27} =	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{12}, C_{22}
C_{28} =	¬S(fh, gh, ca, wa)	C_{12}, C_{24}
C_{29} =	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{14}, C_{22}
C_{30} =	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa)	C_{18}, C_{22}
C_{31} =	¬S(fh, gh, ca, wa)	C_{18}, C_{23}
C_{32} =	¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{19}, C_{22}

A.5 Set-of-support Strategy with Predicate Ordering

	Saturation Level 1	
C_{21} =	¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_3, C_{20}
	Saturation Level 2	
C_{22} =	¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_2, C_{21} on S(fh, gh, ca, wa)
	Saturation Level 3	

C_{23} =	¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{19}, C_{22}
	Saturation Level 4	
C_{24} =	¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{12}, C_{23}
C_{25} =	¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)	C_{14}, C_{23}
C_{26} =	¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa)	C_{18}, C_{23}
	Saturation Level 5	
C_{27} =	¬SAFE(fa, ga, ca, wa)	C_{14}, C_{24}
C_{28} =	¬SAFE(fa, gh, ca, wa)	C_{18}, C_{24}
C_{29} =	¬SAFE(fa, ga, ca, wa)	C_{12}, C_{25}
C_{30} =	¬SAFE(fh, gh, ca, wa)	C_{18}, C_{25}
C_{31} =	¬SAFE(fh, gh, ca, wa)	C_{14}, C_{26}
C_{32} =	¬SAFE(fa, gh, ca, wa)	C_{12}, C_{26}
	Saturation Level 6	
C_{33} =	□	C_{18}, C_{27}

A.6 Set-of-support Strategy with Subsumption

	Given set	
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_3 =	~~¬S(fh, gh, y, z) ∨ ¬SAFE(fh, gh, y, z) ∨~~	Farmer takes goat across

	~~¬SAFE(fa, ga, y, z) ∨ S(fa, ga, y, z)~~	Subsumed by C_{26}
C_{12} =	SAFE(fh, gh, ca, wa)	
C_{14} =	SAFE(fa, gh, ca, wa)	
C_{18} =	SAFE(fa, ga, ca, wa)	
C_{19} =	S(fa, gh, ca, wa)	Start state
	Set-of-support	
C_{20} =	¬S(fa, ga, ca, wa)	Goal state negated
	Saturation Level 1	
C_{21} =	~~¬S(fh, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)~~	C_3, C_{20} Subsumed by C_{23}
	Saturation Level 2	
C_{22} =	~~¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)~~	C_2, C_{21} Subsumed by C_{25}
C_{23} =	~~¬S(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)~~	C_{12}, C_{21} Subsumed by C_{26}
	Saturation Level 3	
C_{24} =	~~¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)~~	C_2, C_{23} Subsumed by C_{22}
C_{25} =	~~¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)~~	C_{12}, C_{22} Subsumed by C_{28}
C_{26} =	¬S(fh, gh, ca, wa)	C_{18}, C_{23}
C_{27} =	~~¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fh, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)~~	C_{19}, C_{22} Subsumed by C_{29}

	Saturation Level 4	
C_{28} =	~~¬S(fa, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)~~	C_{14}, C_{25} Subsumed by C_{30}
C_{29} =	~~¬SAFE(fa, gh, ca, wa) ∨ ¬SAFE(fa, ga, ca, wa)~~	C_{12}, C_{27} Subsumed by C_{31}
	Saturation Level 5	
C_{30} =	¬S(fa, gh, ca, wa)	C_{18}, C_{28}
C_{31} =	¬SAFE(fa, ga, ca, wa)	C_{14}, C_{29}
	Saturation Level 6	
C_{32} =	□	C_{19}, C_{30}

Appendix B

Theorem Provers Evaluated

Prover	**Resolution-Based**	**CASC Division Winner**	**Notes**
Bliksem	Yes	-	Development has been abandoned and was replaced by the Smiley theorem prover.
Carine	Yes	-	E-mail correspondence with the author suggested that Carine would not be suitable for the evaluation described in this dissertation.
Darwin	No	2007	E-mail correspondence with the author confirmed that Darwin only uses resolution in a very limited way.
DCTP	No	2005	Tableau based.
Discount	No	-	Unfailing Knuth-Bendix.
E	No	2000	Equational theorem prover.
E-KRHyper	No	-	Tableau based.
E-Setheo	No	2002	Uses DCTP, E and SETHEO in parallel.
Equinox	No	-	Based on model generation.

Prover	Resolution-Based	CASC Division Winner	Notes
Fampire	Yes	-	Vampire using the SPASS clausifier.
FM-Darwin	No	-	Based on model generation.
Gandalf	Yes	2004	E-mail correspondence with author indicated that Gandalf should be a good candidate for the evaluation described in this dissertation.
Geo	No	-	Based on geometric resolution.
iProver	No	-	Based on instantiation calculus.
LeanCoP	No	-	Connection-driven proof search.
LeanTaP	No	-	Implemented in Prolog.
Mace	No	-	Based on model generation.
Meteor	No	-	Based on model elimination.
Metis	Yes	-	Based on resolution and model elimination.
Octopus	Yes	-	Multiprocessor version of Theo.
OSHL	No	-	Instance based – reduces problems to propositional logic instances.
Otter	Yes	1999	Considered to be the father of many modern automated theorem provers. It is also a good benchmark for improvements in other provers.

Prover	**Resolution-Based**	**CASC Division Winner**	**Notes**
Paradox	No	2007	Based on model generation.
Prover9	Yes	-	Otter's replacement.
RRL	No	-	Based on rewriting techniques.
Setheo	No	-	Tableau based.
Smiley	Yes	-	Not available yet.
SOS	Yes	-	Uses Otter as a sub-program.
SPASS	Yes	1999	
Theo	Yes	-	
Vampire	Yes	2007	Consistent CASC division winner. Solves more set theory problems than other provers.
Waldmeister	No	2007	E-mail correspondence with the author suggested that Waldmeister would not be suitable for our work.

Appendix C

Sample Reasoner Output

C.1 Vampire

```
Refutation found. Thanks to Tanya!
========== Refutation ==========
********** [7, input] **********
(! X0)(el(X0,d) <=> X0=empty \/ X0=b)
********** [7->24, NNF transformation] **********
  (! X0)(el(X0,d) <=> X0=empty \/ X0=b)
------------------------------
  (! X0)((~el(X0,d) \/ (X0=empty \/ X0=b)) & ((~X0=empty & ~X0=b) \/ el(X0,d)))
********** [24->25, flattening] **********
  (! X0)((~el(X0,d) \/ (X0=empty \/ X0=b)) & ((~X0=empty & ~X0=b) \/ el(X0,d)))
------------------------------
  (! X0)((~el(X0,d) \/ X0=empty \/ X0=b) & ((~X0=empty & ~X0=b) \/ el(X0,d)))
********** [25->26, skolemization] **********
  (! X0)((~el(X0,d) \/ X0=empty \/ X0=b) & ((~X0=empty & ~X0=b) \/ el(X0,d)))
------------------------------
  (~el(X0,d) \/ X0=empty \/ X0=b) & ((~X0=empty & ~X0=b) \/ el(X0,d))
********** [26->39, cnf transformation] **********
  (~el(X0,d) \/ X0=empty \/ X0=b) & ((~X0=empty & ~X0=b) \/ el(X0,d))
------------------------------
  el(X0,d) \/ X0!=b
********** [39->42, literal permutation] **********
  el(X0,d) \/ X0!=b
------------------------------
  X1!=b \/ el(X1,d)
********** [42->55, equality resolution] **********
  X1!=b \/ el(X1,d)
------------------------------
  el(b,d)
********** [26->38, cnf transformation] **********
  (~el(X0,d) \/ X0=empty \/ X0=b) & ((~X0=empty & ~X0=b) \/ el(X0,d))
------------------------------
  el(X0,d) \/ X0!=empty
********** [38->43, literal permutation] **********
  el(X0,d) \/ X0!=empty
------------------------------
  X1!=empty \/ el(X1,d)
********** [43->56, equality resolution] **********
  X1!=empty \/ el(X1,d)
------------------------------
  el(empty,d)
********** [3, input] **********
~(? X0)el(X0,empty)
********** [3->14, ENNF transformation] **********
  ~(? X0)el(X0,empty)
------------------------------
  (! X0)~el(X0,empty)
********** [14->15, skolemization] **********
  (! X0)~el(X0,empty)
------------------------------
  ~el(X0,empty)
********** [15->29, cnf transformation] **********
  ~el(X0,empty)
------------------------------
  ~el(X0,empty)
```

```
********** [6, input] **********
(! X0)(el(X0,c) <=> (! X3)(el(X3,X0) => el(X3,b)))
********** [6->10, rectify] **********
  (! X0)(el(X0,c) <=> (! X3)(el(X3,X0) => el(X3,b)))
----------------------------
  (! X0)(el(X0,c) <=> (! X1)(el(X1,X0) => el(X1,b)))
********** [10->20, ENNF transformation] **********
  (! X0)(el(X0,c) <=> (! X1)(el(X1,X0) => el(X1,b)))
----------------------------
  (! X0)(el(X0,c) <=> (! X1)(~el(X1,X0) \/ el(X1,b)))
********** [20->21, NNF transformation] **********
  (! X0)(el(X0,c) <=> (! X1)(~el(X1,X0) \/ el(X1,b)))
----------------------------
  (! X0)((~el(X0,c) \/ (! X1)(~el(X1,X0) \/ el(X1,b))) & ((? X1)(el(X1,X0) & ~el(X1,b))
\/ el(X0,c)))
********** [21->22, rectify] **********
  (! X0)((~el(X0,c) \/ (! X1)(~el(X1,X0) \/ el(X1,b))) & ((? X1)(el(X1,X0) & ~el(X1,b))
\/ el(X0,c)))
----------------------------
  (! X0)((~el(X0,c) \/ (! X1)(~el(X1,X0) \/ el(X1,b))) & ((? X2)(el(X2,X0) & ~el(X2,b))
\/ el(X0,c)))
********** [22->23, skolemization] **********
  (! X0)((~el(X0,c) \/ (! X1)(~el(X1,X0) \/ el(X1,b))) & ((? X2)(el(X2,X0) & ~el(X2,b))
\/ el(X0,c)))
----------------------------
  (~el(X0,c) \/ (~el(X1,X0) \/ el(X1,b))) & ((el(sk1(X0),X0) & ~el(sk1(X0),b)) \/
el(X0,c))
********** [23->36, cnf transformation] **********
  (~el(X0,c) \/ (~el(X1,X0) \/ el(X1,b))) & ((el(sk1(X0),X0) & ~el(sk1(X0),b)) \/
el(X0,c))
----------------------------
  el(X0,c) \/ ~el(sk1(X0),b)
********** [36->45, literal permutation] **********
  el(X0,c) \/ ~el(sk1(X0),b)
----------------------------
  ~el(sk1(X1),b) \/ el(X1,c)
********** [23->35, cnf transformation] **********
  (~el(X0,c) \/ (~el(X1,X0) \/ el(X1,b))) & ((el(sk1(X0),X0) & ~el(sk1(X0),b)) \/
el(X0,c))
----------------------------
  el(X0,c) \/ el(sk1(X0),X0)
********** [35->46, literal permutation] **********
  el(X0,c) \/ el(sk1(X0),X0)
----------------------------
  el(sk1(X1),X1) \/ el(X1,c)
********** [45,46->58, resolution] **********
  ~el(sk1(X1),b) \/ el(X1,c)
  el(sk1(X1),X1) \/ el(X1,c)
----------------------------
  el(b,c)
********** [23->34, cnf transformation] **********
  (~el(X0,c) \/ (~el(X1,X0) \/ el(X1,b))) & ((el(sk1(X0),X0) & ~el(sk1(X0),b)) \/
el(X0,c))
----------------------------
  el(X1,b) \/ ~el(X1,X0) \/ ~el(X0,c)
********** [34->47, literal permutation] **********
  el(X1,b) \/ ~el(X1,X0) \/ ~el(X0,c)
----------------------------
  ~el(X1,X2) \/ ~el(X2,c) \/ el(X1,b)
********** [26->37, cnf transformation] **********
  (~el(X0,d) \/ X0=empty \/ X0=b) & ((~X0=empty & ~X0=b) \/ el(X0,d))
----------------------------
  X0=b \/ X0=empty \/ ~el(X0,d)
********** [37->44, literal permutation] **********
  X0=b \/ X0=empty \/ ~el(X0,d)
----------------------------
  ~el(X1,d) \/ X1=empty \/ X1=b
********** [2, input] **********
(! X1 X2)((! X0)(el(X0,X1) <=> el(X0,X2)) => X1=X2)
********** [2->9, rectify] **********
  (! X1 X2)((! X0)(el(X0,X1) <=> el(X0,X2)) => X1=X2)
```

```
----------------------------
  (! X0 X1)((! X2)(el(X2,X0) <=> el(X2,X1)) => X0=X1)
********** [9->11, ENNF transformation] **********
  (! X0 X1)((! X2)(el(X2,X0) <=> el(X2,X1)) => X0=X1)
----------------------------
  (! X0 X1)((? X2)(el(X2,X0) <~> el(X2,X1)) \/ X0=X1)
********** [11->12, NNF transformation] **********
  (! X0 X1)((? X2)(el(X2,X0) <~> el(X2,X1)) \/ X0=X1)
----------------------------
  (! X0 X1)((? X2)((el(X2,X0) \/ el(X2,X1)) & (~el(X2,X0) \/ ~el(X2,X1))) \/ X0=X1)
********** [12->13, skolemization] **********
  (! X0 X1)((? X2)((el(X2,X0) \/ el(X2,X1)) & (~el(X2,X0) \/ ~el(X2,X1))) \/ X0=X1)
----------------------------
  ((el(sk0(X1,X0),X0) \/ el(sk0(X1,X0),X1)) & (~el(sk0(X1,X0),X0) \/ ~el(sk0(X1,X0),X1)))
\/ X0=X1
********** [13->27, cnf transformation] **********
  ((el(sk0(X1,X0),X0) \/ el(sk0(X1,X0),X1)) & (~el(sk0(X1,X0),X0) \/ ~el(sk0(X1,X0),X1)))
\/ X0=X1
----------------------------
  X0=X1 \/ el(sk0(X1,X0),X1) \/ el(sk0(X1,X0),X0)
********** [27->54, literal permutation] **********
  X0=X1 \/ el(sk0(X1,X0),X1) \/ el(sk0(X1,X0),X0)
----------------------------
  el(sk0(X1,X2),X2) \/ el(sk0(X1,X2),X1) \/ X2=X1
********** [44,54->63, resolution] **********
  ~el(X1,d) \/ X1=empty \/ X1=b
  el(sk0(X1,X2),X2) \/ el(sk0(X1,X2),X1) \/ X2=X1
----------------------------
  el(sk0(X1,d),X1) \/ sk0(X1,d)=empty \/ sk0(X1,d)=b \/ d=X1
********** [47,54->69, resolution] **********
  ~el(X1,X2) \/ ~el(X2,c) \/ el(X1,b)
  el(sk0(X1,X2),X2) \/ el(sk0(X1,X2),X1) \/ X2=X1
----------------------------
  ~el(X1,c) \/ el(sk0(X1,X2),b) \/ el(sk0(X1,X2),X2) \/ X2=X1
********** [63,69->114, resolution] **********
  el(sk0(X1,d),X1) \/ sk0(X1,d)=empty \/ sk0(X1,d)=b \/ d=X1
  ~el(X1,c) \/ el(sk0(X1,X2),b) \/ el(sk0(X1,X2),X2) \/ X2=X1
----------------------------
  el(sk0(sk0(c,d),X1),b) \/ el(sk0(sk0(c,d),X1),X1) \/ sk0(c,d)=empty \/ sk0(c,d)=b \/
X1=sk0(c,d) \/ d=c
********** [58,47,114->403, resolution, forward subsumption resolution] **********
  el(b,c)
  ~el(X1,X2) \/ ~el(X2,c) \/ el(X1,b)
  el(sk0(sk0(c,d),X1),b) \/ el(sk0(sk0(c,d),X1),X1) \/ sk0(c,d)=empty \/ sk0(c,d)=b \/
X1=sk0(c,d) \/ d=c
----------------------------
  el(sk0(sk0(c,d),b),b) \/ sk0(c,d)=empty \/ sk0(c,d)=b \/ d=c
********** [13->28, cnf transformation] **********
  ((el(sk0(X1,X0),X0) \/ el(sk0(X1,X0),X1)) & (~el(sk0(X1,X0),X0) \/ ~el(sk0(X1,X0),X1)))
\/ X0=X1
----------------------------
  X0=X1 \/ ~el(sk0(X1,X0),X1) \/ ~el(sk0(X1,X0),X0)
********** [28->53, literal permutation] **********
  X0=X1 \/ ~el(sk0(X1,X0),X1) \/ ~el(sk0(X1,X0),X0)
----------------------------
  ~el(sk0(X1,X2),X1) \/ ~el(sk0(X1,X2),X2) \/ X2=X1
********** [5, input] **********
(! X0)(el(X0,b) <=> X0=a)
********** [5->18, NNF transformation] **********
  (! X0)(el(X0,b) <=> X0=a)
----------------------------
  (! X0)((~el(X0,b) \/ X0=a) & (~X0=a \/ el(X0,b)))
********** [18->19, skolemization] **********
  (! X0)((~el(X0,b) \/ X0=a) & (~X0=a \/ el(X0,b)))
----------------------------
  (~el(X0,b) \/ X0=a) & (~X0=a \/ el(X0,b))
********** [19->32, cnf transformation] **********
  (~el(X0,b) \/ X0=a) & (~X0=a \/ el(X0,b))
----------------------------
  X0=a \/ ~el(X0,b)
********** [32->49, literal permutation] **********
```

```
  X0=a \/ ~el(X0,b)
----------------------------
  ~el(X1,b) \/ X1=a
********** [49,54->65, resolution] **********
  ~el(X1,b) \/ X1=a
  el(sk0(X1,X2),X2) \/ el(sk0(X1,X2),X1) \/ X2=X1
----------------------------
  el(sk0(X1,b),X1) \/ sk0(X1,b)=a \/ b=X1
********** [49,47,65->92, resolution, forward subsumption resolution] **********
  ~el(X1,b) \/ X1=a
  ~el(X1,X2) \/ ~el(X2,c) \/ el(X1,b)
  el(sk0(X1,b),X1) \/ sk0(X1,b)=a \/ b=X1
----------------------------
  ~el(X1,c) \/ sk0(X1,b)=a \/ b=X1
********** [63,92->160, resolution] **********
  el(sk0(X1,d),X1) \/ sk0(X1,d)=empty \/ sk0(X1,d)=b \/ d=X1
  ~el(X1,c) \/ sk0(X1,b)=a \/ b=X1
----------------------------
  sk0(sk0(c,d),b)=a \/ sk0(c,d)=empty \/ sk0(c,d)=b \/ d=c
********** [403,53,160->745, backward superposition, forward subsumption resolution]
**********
  el(sk0(sk0(c,d),b),b) \/ sk0(c,d)=empty \/ sk0(c,d)=b \/ d=c
  ~el(sk0(X1,X2),X1) \/ ~el(sk0(X1,X2),X2) \/ X2=X1
  sk0(sk0(c,d),b)=a \/ sk0(c,d)=empty \/ sk0(c,d)=b \/ d=c
----------------------------
  ~el(a,sk0(c,d)) \/ sk0(c,d)=empty \/ sk0(c,d)=b \/ d=c
********** [29,54->67, resolution] **********
  ~el(X0,empty)
  el(sk0(X1,X2),X2) \/ el(sk0(X1,X2),X1) \/ X2=X1
----------------------------
  el(sk0(X1,empty),X1) \/ empty=X1
********** [47,67->103, resolution] **********
  ~el(X1,X2) \/ ~el(X2,c) \/ el(X1,b)
  el(sk0(X1,empty),X1) \/ empty=X1
----------------------------
  el(sk0(X1,empty),b) \/ ~el(X1,c) \/ empty=X1
********** [49,103->185, resolution] **********
  ~el(X1,b) \/ X1=a
  el(sk0(X1,empty),b) \/ ~el(X1,c) \/ empty=X1
----------------------------
  ~el(X1,c) \/ sk0(X1,empty)=a \/ empty=X1
********** [63,185->272, resolution] **********
  el(sk0(X1,d),X1) \/ sk0(X1,d)=empty \/ sk0(X1,d)=b \/ d=X1
  ~el(X1,c) \/ sk0(X1,empty)=a \/ empty=X1
----------------------------
  sk0(sk0(c,d),empty)=a \/ sk0(c,d)=b \/ sk0(c,d)=empty \/ d=c
********** [29,745,54,272->1632, backward superposition, forward subsumption resolution]
**********
  ~el(X0,empty)
  ~el(a,sk0(c,d)) \/ sk0(c,d)=empty \/ sk0(c,d)=b \/ d=c
  el(sk0(X1,X2),X2) \/ el(sk0(X1,X2),X1) \/ X2=X1
  sk0(sk0(c,d),empty)=a \/ sk0(c,d)=b \/ sk0(c,d)=empty \/ d=c
----------------------------
  sk0(c,d)=empty \/ sk0(c,d)=b \/ d=c
********** [46,29->62, resolution] **********
  el(sk1(X1),X1) \/ el(X1,c)
  ~el(X0,empty)
----------------------------
  el(empty,c)
********** [62,53,1632->1633, backward superposition, forward subsumption resolution]
**********
  el(empty,c)
  ~el(sk0(X1,X2),X1) \/ ~el(sk0(X1,X2),X2) \/ X2=X1
  sk0(c,d)=empty \/ sk0(c,d)=b \/ d=c
----------------------------
  ~el(sk0(c,d),d) \/ sk0(c,d)=b \/ d=c
********** [56,1632,1633->1664, forward superposition, forward subsumption resolution]
**********
  el(empty,d)
  sk0(c,d)=empty \/ sk0(c,d)=b \/ d=c
  ~el(sk0(c,d),d) \/ sk0(c,d)=b \/ d=c
```

```
----------------------------
  sk0(c,d)=b \/ d=c
********** [58,53,1664->1665, backward superposition, forward subsumption resolution]
**********
  el(b,c)
  ~el(sk0(X1,X2),X1) \/ ~el(sk0(X1,X2),X2) \/ X2=X1
  sk0(c,d)=b \/ d=c
----------------------------
  ~el(sk0(c,d),d) \/ d=c
********** [55,1664,1665->1696, forward superposition, forward subsumption resolution]
**********
  el(b,d)
  sk0(c,d)=b \/ d=c
  ~el(sk0(c,d),d) \/ d=c
----------------------------
  d=c
********** [8, input] **********
~c=d
********** [8->40, cnf transformation] **********
  ~c=d
----------------------------
  c!=d
********** [40->41, literal permutation] **********
  c!=d
----------------------------
  d!=c
********** [1696,41->1863, backward demodulation] **********
  d=c
  d!=c
----------------------------
  #
======= End of refutation =======
========== Statistics ==========
version: 7.41 Civatateo (v7.40 + more docs)
=== General:
time: 0.2s
memory: 18.4Mb
termination reason: refutation found
=== Generating inferences:
resolution: 2716
superposition: 1724
equality_resolution: 4
=== Simplifying inferences:
propositional_tautology: 9
equational_tautology: 718
forward_subsumption: 2120
forward_subsumption_resolution: 269
backward_subsumption: 373
backward_demodulation: 212
=== Generated clauses:
total: 4684
discarded_as_redundant: 2847
=== Retained clauses:
total: 1823
selected: 361
currently_active: 268
currently_passive: 969
======= End of statistics =======
```

C.2 Gandalf

```
Gandalf c-2.6 r1 starting to prove: ./heuristic1/heur1.without.gandalf.in
Using automatic strategy selection.
Time limit in seconds: 1800
```

```
prove-all-passes started

detected problem class: neq
detected subclass: medium

strategies selected:
(hyper 75 #f 2 5)
(binary-unit 28 #f 2 5)
(binary-double 28 #f 2 5)
(binary-double 45 #f)
(binary-double 45 #t)
(binary 151 #t 2 5)
(binary-order 75 #f 2 5)
(binary-posweight-order 304 #f)
(binary-posweight-lex-big-order 75 #f)
(binary-posweight-lex-small-order 28 #f)
(binary-order-sos 151 #t)
(binary-unit-uniteq 75 #f)
(binary-weightorder 151 #f)
(binary-order 151 #f)
(hyper-order 90 #f)
(binary 328 #t)


********* EMPTY CLAUSE DERIVED *********


timer checkpoints:
c(15,40,23,30,0,27,1629,50,968,1644,0,969,11438,4,5847,17751,5,7470,17751,1,7470,17751,50
,7471,17751,40,7471,17766,0,7472,18838,50,7647,18853,0,7647,21683,50,8058,21698,0,8058,24
528,50,8467,24543,0,8467,27373,50,8882,27373,40,8882,27388,0,8882,38896,3,10285,40526,4,1
0983,43308,5,11683,43309,5,11683,43309,1,11683,43309,50,11685,43309,40,11685,43324,0,1168
5)


START OF PROOF
35918 [?] ?
43311 [] el($$f1(X,Y),Y) | el($$f1(X,Y),X) | equal(X,Y).
43312 [] -el($$f1(X,Y),Y) | -el($$f1(X,Y),X) | equal(X,Y).
43313 [] -el(X,empty).
43316 [] -el(X,b) | equal(X,a).
43318 [] -el(X,c) | el(Y,b) | -el(Y,X).
43322 [] -equal(X,empty) | el(X,d).
43323 [] -equal(X,b) | el(X,d).
43324 [] -equal(c,d).
43329 [binary:43313,43311] el($$f1(X,empty),X) | equal(X,empty).
43331 [binary:43324,43311.3,cut:35918] el($$f1(c,d),c).
43351 [binary:43324,43312.3,cut:43331] -el($$f1(c,d),d).
43363 [binary:43351,43322.2] -equal($$f1(c,d),empty).
43366 [binary:43331,43318] -el(X,$$f1(c,d)) | el(X,b).
43378 [binary:43351,43323.2] -equal($$f1(c,d),b).
43380 [binary:43311.3,43378,binarycut:43366] el($$f1($$f1(c,d),b),b).
43419 [binary:43316,43380] equal($$f1($$f1(c,d),b),a).
43420 [binary:43312,43380,demod:43419,cut:43378] -el(a,$$f1(c,d)).
45960 [binary:43329,43366,cut:43363] el($$f1($$f1(c,d),empty),b).
46334 [binary:43316,45960] equal($$f1($$f1(c,d),empty),a).
46539 [para:46334.1.1,43311.1.1,demod:46334,cut:43313,cut:43420,cut:43363] contradiction
END OF PROOF

Proof found by the following strategy:

using binary resolution
not using sos strategy
using double strategy
using dynamic demodulation
using ordered paramodulation
using kb ordering for equality
preferring bigger arities for lex ordering
using clause demodulation
seconds given: 45
```

```
***GANDALF_FOUND_A_REFUTATION***

Global statistics over all passes:

 given clauses:    7037
 derived clauses:   353789
 kept clauses:      33845
 kept size sum:     531726
 kept mid-nuclei:   3933
 kept new demods:   28
 forw unit-subs:    36540
 forw double-subs: 32898
 forw overdouble-subs: 50491
 backward subs:     672
 fast unit cutoff:  7014
 full unit cutoff:  920
 dbl  unit cutoff:  110
 real runtime    :  120.30
 process. runtime:  119.21
specific non-discr-tree subsumption statistics:
 tried:           1208228
 length fails:    48201
 strength fails:  171347
 predlist fails:  249780
 aux str. fails:  69703
 by-lit fails:    142615
 full subs tried: 383264
 full subs fail:  362366
```

Appendix D

Z Case Study of Order Processing System

D.1 Given Sets (Basic Types)

[STRING, AMOUNT, DATE]

[PRODUCT, ORDER, ITEM, CUSTOMER]

STATUS ::= pending | cancelled | processed

D.2 Product

Product

$$
\begin{aligned}
&products: \mathbb{P}\ PRODUCT \\
&prodName: PRODUCT \rightarrowtail\!\!\!\!\!\!\rightarrow STRING \\
&prodPrice: PRODUCT \rightarrow\!\!\!\!\!\!\rightarrow AMOUNT \\
&prodQuantity: PRODUCT \rightarrow\!\!\!\!\!\!\rightarrow \mathbb{N}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{dom}\ prodName = products \\
&\mathrm{dom}\ prodPrice = products \\
&\mathrm{dom}\ prodQuantity = products
\end{aligned}
$$

InitProduct

$$Product'$$

$$
\begin{aligned}
&products' = \varnothing \\
&prodName' = \varnothing \\
&prodPrice' = \varnothing \\
&prodQuantity' = \varnothing
\end{aligned}
$$

CreateProduct

ΔProduct
product?: PRODUCT
name?: STRING
price?: AMOUNT
quantity?: $\mathbb{N}$

---

product? $\notin$ products
name? $\notin$ ran prodName
products′ = products $\cup$ product?
prodName′ = prodName $\cup$ {product? $\mapsto$ name?}
prodPrice′ = prodPrice $\cup$ {product? $\mapsto$ price?}
prodQuantity′ = prodQuantity $\cup$ {product? $\mapsto$ quantity?}

UpdateProduct

ΔProduct
product?: PRODUCT
name?: STRING
price?: AMOUNT
quantity?: $\mathbb{N}$

---

product? $\in$ products
products′ = products
prodName′ = prodName $\oplus$ {product? $\mapsto$ name?}
prodPrice′ = prodPrice $\oplus$ {product? $\mapsto$ price?}
prodQuantity′ = prodQuantity $\oplus$ {product? $\mapsto$ quantity?}

DeleteProduct

ΔProduct
product?: PRODUCT

---

product? $\in$ products
products′ = products $\setminus$ {product?}
prodName′ = {product?} $\lhd$ prodName
prodPrice′ = {product?} $\lhd$ prodPrice
prodQuantity′ = {product?} $\lhd$ prodQuantity

SelectProductsBelowThreshold

ΞProduct
quantity?: $\mathbb{N}$
products!: $\mathbb{P}$ PRODUCT

---

products! = {p: products | prodQuantity(p) < quantity?}

D.3 Order

Order

$$orders : \mathbb{P}\ ORDER$$
$$orderDate : ORDER \pfun DATE$$
$$orderStatus : ORDER \pfun STATUS$$
$$orderCustomer : ORDER \pfun CUSTOMER$$

$$dom\ orderDate = orders$$
$$dom\ orderStatus = orders$$
$$dom\ orderCustomer = orders$$

InitOrder

$$Order'$$

$$orders' = \emptyset$$
$$orderDate' = \emptyset$$
$$orderStatus' = \emptyset$$
$$orderCustomer' = \emptyset$$

CreateOrder

$$\Delta Order$$
$$date? : DATE$$
$$customer? : CUSTOMER$$
$$order! : ORDER$$

$$order! \notin orders$$
$$orders' = orders \cup order!$$
$$orderDate' = orderDate \cup \{order! \mapsto date?\}$$
$$orderStatus' = orderStatus \cup \{order! \mapsto pending\}$$
$$orderCustomer' = orderCustomer \cup \{order! \mapsto customer?\}$$

CancelOrder

$\Delta Order$
order?: ORDER

---

order? $\in$ orders
orderStatus(order?) = pending
orders′ = orders
orderDate′ = orderDate
orderStatus′ = orderStatus $\oplus$ {order? $\mapsto$ cancelled}
orderCustomer′ = orderCustomer

ProcessOrder

$\Delta Order$
$\Delta Product$
$\Xi Item$
order?: ORDER

---

order? $\in$ orders
orderStatus(order?) = pending
$\forall i$: items $\bullet$ itemOrder(i) = order? $\Rightarrow$ prodQuantity(itemProduct(i)) – itemQuantity(i) $\geqslant$ 0
orders′ = orders
orderDate′ = orderDate
orderStatus′ = orderStatus $\oplus$ {order? $\mapsto$ processed}
orderCustomer′ = orderCustomer
products′ = products
prodName′ = prodName
prodPrice′ = prodPrice
prodQuantity′ = prodQuantity $\oplus$
{i: items | itemOrder(i) = order? $\bullet$ itemProduct(i) $\mapsto$ prodQuantity(itemProduct(i)) – itemQuantity(i)}

SelectOrdersForCustomer

$\Xi Order$
customer?: CUSTOMER
orders!: $\mathbb{P}$ ORDER

---

orders! = {o: orders | orderCustomer(o) = customer?}

D.4 Item

Item

$items: \mathbb{P}\ ITEM$
$itemOrder: ITEM \pfun ORDER$
$itemPrice: ITEM \pfun AMOUNT$
$itemQuantity: ITEM \pfun \mathbb{N}_1$
$itemProduct: ITEM \pfun PRODUCT$

$dom\ itemOrder = items$
$dom\ itemPrice = items$
$dom\ itemQuantity = items$
$dom\ itemProduct = items$
$\forall i_1, i_2 : items \bullet i_1 \neq i_2 \implies itemOrder(i_1) \neq itemOrder(i_2) \lor itemProduct(i_1) \neq itemProduct(i_2)$

InitItem

$Item'$

$items' = \emptyset$
$itemOrder' = \emptyset$
$itemPrice' = \emptyset$
$itemQuantity' = \emptyset$
$itemProduct' = \emptyset$

CreateItem

$\Delta Item$
$\Xi Product$
$item? : ITEM$
$order? : ORDER$
$quantity? : \mathbb{N}_1$
$product? : PRODUCT$

$item? \notin items$
$items' = items \cup \{item?\}$
$itemOrder' = itemOrder \cup \{item? \mapsto order?\}$
$itemPrice' = itemPrice \cup \{item? \mapsto prodPrice(product?)\}$
$itemQuantity' = itemQuantity \cup \{item? \mapsto quantity?\}$
$itemProduct' = itemProduct \cup \{item? \mapsto product?\}$

UpdateItem

$\Delta Item$
$item?: ITEM$
$quantity?: \mathbb{N}_1$

$item? \in items$
$items' = items$
$itemOrder' = itemOrder$
$itemPrice' = itemPrice$
$itemQuantity' = itemQuantity \oplus \{item? \mapsto quantity?\}$
$itemProduct' = itemProduct$

DeleteItem

$\Delta Item$
$item?: ITEM$

$item? \in items$
$items' = items \setminus \{item?\}$
$itemOrder' = \{item?\} \lhd itemOrder$
$itemPrice' = \{item?\} \lhd itemPrice$
$itemQuantity' = \{item?\} \lhd itemQuantity$
$itemProduct' = \{item?\} \lhd itemProduct$

SelectItemsForOrder

$\Xi Item$
$order?: ORDER$
$items!: \mathbb{P}\ ITEM$

$items! = \{i: items \mid itemOrder(i) = order?\}$

D.5 Customer

$COMPANY: \mathbb{P}\ CUSTOMER$
$PERSON: \mathbb{P}\ CUSTOMER$

$\langle COMPANY, PERSON \rangle$ partition $CUSTOMER$

Customer

$customers: \mathbb{P}\ CUSTOMER$
$custAddress: CUSTOMER \pfun STRING$
$custPhone: CUSTOMER \pfun STRING$

$dom\ custAddress = customers$
$dom\ custPhone = customers$

InitCustomer

$Customer'$

$customers' = \emptyset$
$custAddress' = \emptyset$
$custPhone' = \emptyset$

CreateCustomer

$\Delta Customer$
$customer?: CUSTOMER$
$address?: STRING$
$phone?: STRING$

$customer? \notin customers$
$customers' = customers \cup \{customer?\}$
$custAddress' = custAddress \cup \{customer? \mapsto address?\}$
$custPhone' = custPhone \cup \{customer? \mapsto phone?\}$

UpdateCustomer

$\Delta Customer$
$customer?: CUSTOMER$
$address?: STRING$
$phone?: STRING$

$customer? \in customers$
$customers' = customers$
$custAddress' = custAddress \oplus \{customer? \mapsto address?\}$
$custPhone' = custPhone \oplus \{customer? \mapsto phone?\}$

DeleteCustomer

ΔCustomer
ΞOrder
customer?: CUSTOMER

---

customer? $\in$ customers
customer? $\notin$ ran orderCustomer
customers$'$ = customers $\setminus$ customer?
custAddress$'$ = {customer?} $\lhd$ custAddress
custPhone$'$ = {customer?} $\lhd$ custPhone

D.6 Company

Company

Customer
companies: $\mathbb{P}$ COMPANY
compName: COMPANY $\rightarrow\!\!\!\!\!\!+$ STRING
compRegNo: COMPANY $\rightarrow\!\!\!\!\!\!+$ STRING

---

companies $\subseteq$ customers
dom compName = companies
dom compRegNo = companies

InitCompany

Company$'$
InitCustomer

---

companies$'$ = $\varnothing$
compName$'$ = $\varnothing$
compRegNo$'$ = $\varnothing$

CreateCompany

ΔCompany
CreateCustomer
name?: STRING
regNo?: STRING

---

companies$'$ = companies $\cup$ {customer?}
compName$'$ = compName $\cup$ {customer? $\mapsto$ name?}
compRegNo$'$ = compRegNo $\cup$ {customer? $\mapsto$ regNo?}

UpdateCompany

ΔCompany
UpdateCustomer
name?: STRING
regNo?: STRING

---

customer? $\in$ companies
companies′ = companies
compName′ = compName $\oplus$ {customer? $\mapsto$ name?}
compRegNo′ = compRegNo $\oplus$ {customer? $\mapsto$ regNo?}

DeleteCompany

ΔCompany
DeleteCustomer

---

customer? $\in$ companies
companies′ = companies \ {customer?}
compName′ = {customer?} $\lhd$ compName
compRegNo′ = {customer?} $\lhd$ compRegNo

D.7 Person

Person

Customer
persons: $\mathbb{P}$ PERSON
perName: PERSON $\pfun$ STRING
perSurname: PERSON $\pfun$ STRING

---

persons $\subseteq$ customers
dom perName = persons
dom perSurname = persons

InitPerson

Person′
InitCustomer

---

persons′ = $\emptyset$
perName′ = $\emptyset$
perSurname′ = $\emptyset$

CreatePerson

ΔPerson
CreateCustomer
name?: STRING
surname?: STRING

---

persons′ = persons $\cup$ customer?
perName′ = perName $\cup$ {customer? $\mapsto$ name?}
perSurname′ = perSurname $\cup$ {customer? $\mapsto$ surname?}

UpdatePerson

ΔPerson
UpdateCustomer
name?: STRING
surname?: STRING

---

customer? $\in$ persons
persons′ = persons
perName′ = perName $\oplus$ {customer? $\mapsto$ name?}
perSurname′ = perSurname $\oplus$ {customer? $\mapsto$ surname?}

DeletePerson

ΔPerson
DeleteCustomer

---

customer? $\in$ persons
persons′ = persons \ customer?
perName′ = {customer?} $\lhd$ perName
perSurname′ = {customer?} $\lhd$ perSurname

D.8 System

System

Product
Order
Item
Customer
Company
Person

InitSystem

InitProduct
InitOrder
InitItem
InitCustomer
InitCompany
InitPerson

Appendix E

Reasoner Inputs for Proof Obligations

E.1 CreateProduct Invariant

```
% If we didn't specifiy that products? = products ? {product?}, then it could be deduced

fof(anonymous, axiom,
  % t_Product type
  el(t_Product,t_PProduct) &
  % t_String type
  el(t_String,t_PString)
).

% +..
%  products: P PRODUCT
%  prodName: PRODUCT >-|-> STRING
%  products': P PRODUCT
%  prodName': PRODUCT >-|-> STRING
%  product?: PRODUCT
%  name?: STRING
% |
%  dom prodName = products
%  dom prodName' = products'
%  product? /e products
%  name? /e ran prodName
%  prodName' = prodName u { product? |--> name? }
% ---
% ---------------------- rewritten to -----------------------------
% +..
%  products: P PRODUCT
%  prodName: P { X31: PRODUCT; X32: STRING }
%  products': P PRODUCT
%  prodName': P { X33: PRODUCT; X34: STRING }
%  product?: PRODUCT
%  name?: STRING
% |
%  A X60: PRODUCT; X61: STRING; X62: PRODUCT; X63: STRING @
%    not (X60,X61) e prodName \/ not (X62,X63) e prodName \/ not X61 = X63 \/ X60 = X62
%  A X60: PRODUCT; X61: STRING; X62: PRODUCT; X63: STRING @
%    not (X60,X61) e prodName \/ not (X62,X63) e prodName \/ not X60 = X62 \/ X61 = X63
%  A X68: PRODUCT; X69: STRING; X70: PRODUCT; X71: STRING @
%    not (X68,X69) e prodName' \/ not (X70,X71) e prodName' \/ not X69 = X71 \/ X68 = X70
%  A X68: PRODUCT; X69: STRING; X70: PRODUCT; X71: STRING @
%    not (X68,X69) e prodName' \/ not (X70,X71) e prodName' \/ not X68 = X70 \/ X69 = X71
%  A EL37: PRODUCT @
%    (A X104: PRODUCT; X105: STRING @
%        not (X104,X105) e prodName \/ not EL37 = X104) \/ EL37 e products
%  A EL37: PRODUCT @ not EL37 e products \/ (E X109: STRING @ (EL37,X109) e prodName)
%  A EL38: PRODUCT @
%    (A X112: PRODUCT; X113: STRING @
%        not (X112,X113) e prodName' \/ not EL38 = X112) \/ EL38 e products'
%  A EL38: PRODUCT @ not EL38 e products' \/ (E X117: STRING @ (EL38,X117) e prodName')
%  not product? e products
%  A X93: PRODUCT; X94: STRING @ not (X93,X94) e prodName \/ not name? = X94
%  A X85: PRODUCT; X86: STRING @
%    not (X85,X86) e prodName' \/ (X85,X86) e prodName \/ X86 = name?
%  A X85: PRODUCT; X86: STRING @
%    not (X85,X86) e prodName' \/ (X85,X86) e prodName \/ X85 = product?
```

```
%  A X89: PRODUCT; X90: STRING @
%    (X89,X90) e prodName' \/ not X89 = product? \/ not X90 = name?
%  A X89: PRODUCT; X90: STRING @ (X89,X90) e prodName' \/ not (X89,X90) e prodName
% ---
fof(anonymous, axiom,
  el(products, t_PProduct) &
  el(prodName, t3) &
  el(products_, t_PProduct) &
  el(prodName_, t3) &
  el(productI, t_Product) &
  el(nameI, t_String) &
  ![X60,X61,X62,X63]:
   ((el(X60,t_Product) & el(X61,t_String) & el(X62,t_Product) & el(X63,t_String)) =>
    (~el(ord_t2(X60,X61),prodName) | ~el(ord_t2(X62,X63),prodName) | ~(X61 = X63) |
     X60 = X62)) &
  ![X60,X61,X62,X63]:
   ((el(X60,t_Product) & el(X61,t_String) & el(X62,t_Product) & el(X63,t_String)) =>
    (~el(ord_t2(X60,X61),prodName) | ~el(ord_t2(X62,X63),prodName) | ~(X60 = X62) |
     X61 = X63)) &
  ![X68,X69,X70,X71]:
   ((el(X68,t_Product) & el(X69,t_String) & el(X70,t_Product) & el(X71,t_String)) =>
    (~el(ord_t2(X68,X69),prodName_) | ~el(ord_t2(X70,X71),prodName_) | ~(X69 = X71) |
     X68 = X70)) &
  ![X68,X69,X70,X71]:
   ((el(X68,t_Product) & el(X69,t_String) & el(X70,t_Product) & el(X71,t_String)) =>
    (~el(ord_t2(X68,X69),prodName_) | ~el(ord_t2(X70,X71),prodName_) | ~(X68 = X70) |
     X69 = X71)) &
  ![EL37]: (el(EL37,t_Product) =>
    ((![X104,X105]: ((el(X104,t_Product) & el(X105,t_String)) =>
      (~el(ord_t2(X104,X105),prodName) | ~(EL37 = X104)))) | el(EL37,products))) &
  ![EL37]: (el(EL37,t_Product) => (~el(EL37,products) |
    (?[X109]: (el(X109,t_String) & el(ord_t2(EL37,X109),prodName))))) &
  ![EL38]: (el(EL38,t_Product) =>
    ((![X112,X113]: ((el(X112,t_Product) & el(X113,t_String)) =>
      (~el(ord_t2(X112,X113),prodName_) | ~(EL38 = X112)))) | el(EL38,products_))) &
  ![EL38]: (el(EL38,t_Product) => (~el(EL38,products_) |
    (?[X117]: (el(X117,t_String) & el(ord_t2(EL38,X117),prodName_))))) &
  ~el(productI,products) &
  ![X93,X94]: ((el(X93,t_Product) & el(X94,t_String)) =>
    (~el(ord_t2(X93,X94),prodName) | ~(nameI = X94))) &
  ![X85,X86]: ((el(X85, t_Product) & el(X86, t_String)) =>
    ((~el(ord_t2(X85,X86),prodName_) | el(ord_t2(X85,X86),prodName)) | X86 = nameI)) &
  ![X85,X86]: ((el(X85,t_Product) & el(X86,t_String)) =>
    (((~el(ord_t2(X85,X86),prodName_) | el(ord_t2(X85,X86),prodName))) | X85=productI)) &
  ![X89,X90]: ((el(X89,t_Product) & el(X90,t_String)) =>
    ((el(ord_t2(X89,X90),prodName_) | ~(X89 = productI)) | ~(X90 = nameI))) &
  ![X89,X90]: ((el(X89,t_Product) & el(X90,t_String)) =>
    (el(ord_t2(X89,X90),prodName_) | ~el(ord_t2(X89,X90),prodName)))
).

fof(anonymous, axiom,
  % set equality t_PPProduct
  ![VAL197,VAL198]:((el(VAL197,t_PPProduct) & el(VAL198,t_PPProduct)) =>
    ((![EL199]:(el(EL199,t_PProduct) =>
      (el(EL199,VAL197) <=> el(EL199,VAL198)))) => VAL197=VAL198))
).

fof(anonymous, axiom,
  % t_PProduct membership
  ![X196]:(el(X196,t_PProduct) => (el(X196,t_PProduct) <=>
    ![Y]:(el(Y,t_Product) => (el(Y,X196) => el(Y,t_Product))))) &
  % t_PProduct type
  el(t_PProduct,t_PPProduct)
).

fof(anonymous, axiom,
  % tuple equality t2
  ![X201,X203,X202,X204]:
   ((el(X201,t_Product) & el(X203,t_String) & el(X202,t_Product) & el(X204,t_String)) =>
     (ord_t2(X201,X203)=ord_t2(X202,X204) <=> (X201 = X202 & X203 = X204))) &
  % tuple type t2
```

```
  ![X201,X203]:((el(X201,t_Product) & el(X203,t_String)) => (el(ord_t2(X201,X203), t2)))
).

fof(anonymous, axiom,
  % set equality t3
  ![VAL205,VAL206]:((el(VAL205,t3) & el(VAL206,t3)) => ((![EL207]:(el(EL207,t2) =>
    (el(EL207,VAL205) <=> el(EL207,VAL206)))) => VAL205=VAL206))
).

% { X31: PRODUCT; X32: STRING }
fof(anonymous, axiom,
  % t2 membership
  ![EL200]:(el(EL200,t2) => (el(EL200,t2) <=>
   (?[X31,X32]: ((el(X31,t_Product) & el(X32,t_String)) & (EL200=ord_t2(X31,X32)))))) &
  % t2 type
  el(t2,t3)
).

fof(anonymous, axiom,
  % set equality t1
  ![VAL209,VAL210]:((el(VAL209,t1) & el(VAL210,t1)) =>
   ((![EL211]:(el(EL211,t3) =>
     (el(EL211,VAL209) <=> el(EL211,VAL210)))) => VAL209=VAL210))
).

% P { X31: PRODUCT; X32: STRING }
fof(anonymous, axiom,
  % t3 membership
  ![X208]:(el(X208,t3) =>
    (el(X208,t3) <=> ![Y]:(el(Y,t2) => (el(Y,X208) => el(Y,t2))))) &
  % t3 type
  el(t3,t1)
).

% |-? products' = products u { product? }
% ---------------------- rewritten to -----------------------------
% |-? A EL135: PRODUCT @ not EL135 e products' \/ EL135 e products \/ EL135 = product?
%     A EL135: PRODUCT @ EL135 e products' \/ not EL135 = product?
%     A EL135: PRODUCT @ EL135 e products' \/ not EL135 e products
fof(conjecture, conjecture,
  ![EL135]: (el(EL135, t_Product) =>
    (~el(EL135,products_) | el(EL135,products) | EL135 = productI)) &
  ![EL135]: (el(EL135, t_Product) => (el(EL135,products_) | ~(EL135 = productI))) &
  ![EL135]: (el(EL135, t_Product) => (el(EL135,products_) | ~el(EL135,products)))
).
```

E.2 After State Type of CancelOrder

```
fof(anonymous, axiom,
  % t_Order type
  el(t_Order,t_POrder) &
  % t_Customer type
  el(t_Customer,t_PCustomer) &
  % t_Date type
  el(t_Date,t_PDate)
).

% STATUS ::= pending | cancelled | processed
fof(anonymous, axiom,
  % t_Status type
  el(t_Status,t_PStatus) &
  % pending, cancelled, processed: STATUS
  el(pending,t_Status) & el(cancelled,t_Status) & el(processed,t_Status)
).

% +..
%  orders: P ORDER
%  orderDate: ORDER -|-> DATE
```

```
%  orderStatus: ORDER -|-> STATUS
%  orderCustomer: ORDER -|-> CUSTOMER
%  orders': P ORDER
%  orderDate': P (ORDER x DATE)
%  orderStatus': P (ORDER x STATUS)
%  orderCustomer': P (ORDER x CUSTOMER)
%  orderI: ORDER
% |
%  dom orderDate = orders
%  dom orderStatus = orders
%  dom orderCustomer = orders
%  dom orderDate' = orders'
%  dom orderStatus' = orders'
%  dom orderCustomer' = orders'
%  orderI e orders
%  orderStatus (orderI) = pending
%  orders' = orders
%  orderDate' = orderDate
%  orderCustomer' = orderCustomer
% ---
% ---------------------- rewritten to ----------------------------
% +..
%  orders: P ORDER
%  orderDate: P {X31: ORDER; X32: DATE}
%  orderStatus: P {X33: ORDER; X34: STATUS}
%  orderCustomer: P {X35: ORDER; X36: CUSTOMER}
%  orders': P ORDER
%  orderDate': P {X37: ORDER; X38: DATE}
%  orderStatus': P {X39: ORDER; X40: STATUS}
%  orderCustomer': P {X41: ORDER; X42: CUSTOMER}
%  orderI: ORDER
% |
%  A X63: ORDER; X64: DATE; X65: ORDER; X66: DATE @
%    not (X63,X64) e orderDate \/ not (X65,X66) e orderDate \/ not X63 = X65 \/ X64 = X66
%  A X73: ORDER; X74: STATUS; X75: ORDER; X76: STATUS @
%    not (X73,X74) e orderStatus \/ not (X75,X76) e orderStatus \/ not X73=X75 \/ X74=X76
%  A X83: ORDER; X84: CUSTOMER; X85: ORDER; X86: CUSTOMER @
%    not (X83,X84) e orderCustomer \/ not (X85,X86) e orderCustomer \/
%    not X83=X85 \/ X84=X86
%  A EL49: ORDER @ (A X134: ORDER; X135: DATE @
%    not (X134,X135) e orderDate \/ not EL49 = X134) \/ EL49 e orders
%  A EL49: ORDER @ not EL49 e orders \/ (E X139: DATE @ (EL49,X139) e orderDate)
%  A EL50: ORDER @ (A X142: ORDER; X143: STATUS @
%    not (X142,X143) e orderStatus \/ not EL50 = X142) \/ EL50 e orders
%  A EL50: ORDER @ not EL50 e orders \/ (E X147: STATUS @ (EL50,X147) e orderStatus)
%  A EL51: ORDER @ (A X150: ORDER; X151: CUSTOMER @
%    not (X150,X151) e orderCustomer \/ not EL51 = X150) \/ EL51 e orders
%  A EL51: ORDER @ not EL51 e orders \/ (E X155: CUSTOMER @ (EL51,X155) e orderCustomer)
%  A EL52: ORDER @ (A X158: ORDER; X159: DATE @
%    not (X158,X159) e orderDate' \/ not EL52 = X158) \/ EL52 e orders'
%  A EL52: ORDER @ not EL52 e orders' \/ (E X163: DATE @ (EL52,X163) e orderDate')
%  A EL53: ORDER @ (A X166: ORDER; X167: STATUS @
%    not (X166,X167) e orderStatus' \/ not EL53 = X166) \/ EL53 e orders'
%  A EL53: ORDER @ not EL53 e orders' \/ (E X171: STATUS @ (EL53,X171) e orderStatus')
%  A EL54: ORDER @ (A X174: ORDER; X175: CUSTOMER @
%    not (X174,X175) e orderCustomer' \/ not EL54 = X174) \/ EL54 e orders'
%  A EL54: ORDER @ not EL54 e orders' \/
%    (E X179: CUSTOMER @ (EL54,X179) e orderCustomer')
%  orderI e orders
%  orderStatus (orderI) = pending
%  A EL55: ORDER @ not EL55 e orders' \/ EL55 e orders
%  A EL55: ORDER @ not EL55 e orders \/ EL55 e orders'
%  A X118: ORDER; X119: DATE @ not (X118,X119) e orderDate' \/ (X118,X119) e orderDate
%  A X122: ORDER; X123: DATE @ not (X122,X123) e orderDate \/ (X122,X123) e orderDate'
%  A X126: ORDER; X127: CUSTOMER @
%    not (X126,X127) e orderCustomer' \/ (X126,X127) e orderCustomer
%  A X130: ORDER; X131: CUSTOMER @
%    not (X130,X131) e orderCustomer \/ (X130,X131) e orderCustomer'
% ---
fof(anonymous, axiom,
  el(orders,t_POrder) &
```

```
  el(orderDate,t_P_OrderxDate_) &
  el(orderStatus, t4) &
  el(orderCustomer, t7) &
  el(orders_,t_POrder) &
  el(orderDate_,t_P_OrderxDate_) &
  el(orderStatus_, t4) &
  el(orderCustomer_, t7) &
  el(orderI,t_Order) &
  ![X63,X64,X65,X66]:
   ((el(X63,t_Order) & el(X64,t_Date) & el(X65,t_Order) & el(X66,t_Date)) =>
    ((((((~el(ord_t__OrderxDate_(X63,X64),orderDate) |
     ~el(ord_t__OrderxDate_(X65,X66),orderDate))) | ~(X63=X65))) | X64=X66))) &
  ![X73,X74,X75,X76]:
   ((el(X73,t_Order) & el(X74,t_Status) & el(X75,t_Order) & el(X76,t_Status)) =>
    ((((((~el(ord_t3(X73,X74),orderStatus) |
     ~el(ord_t3(X75,X76),orderStatus))) | ~(X73=X75))) | X74=X76))) &
  ![X83,X84,X85,X86]:
   ((el(X83,t_Order) & el(X84,t_Customer) & el(X85,t_Order) & el(X86,t_Customer)) =>
    ((((((~el(ord_t6(X83,X84),orderCustomer) |
     ~el(ord_t6(X85,X86),orderCustomer))) | ~(X83 = X85))) | X84 = X86))) &
  ![EL49]: ((el(EL49,t_Order)) =>
    ((((![X134,X135]: ((el(X134,t_Order) & el(X135,t_Date)) =>
     ((~el(ord_t__OrderxDate_(X134,X135),orderDate) |
      ~(EL49 = X134))))) | el(EL49,orders)))) &
  ![EL49]: ((el(EL49,t_Order)) => ((~el(EL49,orders) |
    (?[X139]: (el(X139,t_Date) & el(ord_t__OrderxDate_(EL49,X139),orderDate)))))) &
  ![EL50]: ((el(EL50,t_Order)) =>
    ((((![X142,X143]: ((el(X142,t_Order) & el(X143,t_Status)) =>
     ((~el(ord_t3(X142,X143),orderStatus) | ~(EL50 = X142))))) | el(EL50,orders)))) &
  ![EL50]: (el(EL50,t_Order) => ((~el(EL50,orders) |
    (?[X147]: (el(X147,t_Status) & el(ord_t3(EL50,X147),orderStatus)))))) &
  ![EL51]: (el(EL51,t_Order) =>
    ((((![X150,X151]: ((el(X150,t_Order) & el(X151,t_Customer)) =>
     ((~el(ord_t6(X150,X151),orderCustomer) | ~(EL51 = X150))))) | el(EL51,orders)))) &
  ![EL51]: (el(EL51,t_Order) => ((~el(EL51,orders) |
    (?[X155]: (el(X155,t_Customer) & el(ord_t6(EL51,X155),orderCustomer)))))) &
  ![EL52]: (el(EL52,t_Order) => ((((![X158,X159]: ((el(X158,t_Order) & el(X159,t_Date)) =>
    ((~el(ord_t__OrderxDate_(X158,X159),orderDate_) |
     ~(EL52 = X158))))) | el(EL52,orders_)))) &
  ![EL52]: (el(EL52,t_Order) => ((~el(EL52,orders_) | (?[X163]: (el(X163,t_Date) &
    el(ord_t__OrderxDate_(EL52,X163),orderDate_)))))) &
  ![EL53]: (el(EL53,t_Order) =>
    ((((![X166,X167]: ((el(X166,t_Order) & el(X167,t_Status)) =>
     ((~el(ord_t3(X166,X167),orderStatus_) | ~(EL53 = X166))))) | el(EL53,orders_)))) &
  ![EL53]: ((el(EL53,t_Order)) => ((~el(EL53,orders_) | (?[X171]: (el(X171,t_Status) &
    el(ord_t3(EL53,X171),orderStatus_)))))) &
  ![EL54]: (el(EL54,t_Order) =>
    ((((![X174,X175]: ((el(X174,t_Order) & el(X175,t_Customer)) =>
     ((~el(ord_t6(X174,X175),orderCustomer_) | ~(EL54 = X174))))) | el(EL54,orders_)))) &
  ![EL54]: (el(EL54,t_Order) => ((~el(EL54,orders_) |
    (?[X179]: (el(X179,t_Customer) & el(ord_t6(EL54,X179),orderCustomer_)))))) &
  el(orderI,orders) &
  orderStatus(orderI) = pending &
  ![EL55]: (el(EL55,t_Order) => ((~el(EL55,orders_) | el(EL55,orders)))) &
  ![EL55]: (el(EL55,t_Order) => ((~el(EL55,orders) | el(EL55,orders_)))) &
  ![X118,X119]: ((el(X118,t_Order) & el(X119,t_Date)) =>
    ((~el(ord_t__OrderxDate_(X118,X119),orderDate_) |
    el(ord_t__OrderxDate_(X118,X119),orderDate)))) &
  ![X122,X123]: (el(X122,t_Order) & el(X123,t_Date) =>
    ((~el(ord_t__OrderxDate_(X122,X123),orderDate) |
     el(ord_t__OrderxDate_(X122,X123),orderDate_)))) &
  ![X126,X127]: ((el(X126,t_Order) & el(X127,t_Customer)) =>
    ((~el(ord_t6(X126,X127),orderCustomer_) | el(ord_t6(X126,X127),orderCustomer)))) &
  ![X130,X131]: ((el(X130,t_Order) & el(X131,t_Customer)) =>
    ((~el(ord_t6(X130,X131),orderCustomer) | el(ord_t6(X130,X131),orderCustomer_))))
).

fof(anonymous, axiom,
  % set equality t_PPOrder
  ![VAL524,VAL525]:((el(VAL524,t_PPOrder) & el(VAL525,t_PPOrder)) =>
    ((![EL526]:(el(EL526,t_POrder) =>
```

```
     (el(EL526,VAL524) <=> el(EL526,VAL525)))) => VAL524=VAL525))
).

% P ORDER

fof(anonymous, axiom,
  % t_POrder membership
  ![X523]:(el(X523,t_POrder) =>
    (el(X523,t_POrder) <=> ![Y]:(el(Y,t_Order) => (el(Y,X523) => el(Y,t_Order))))) &
  % t_POrder type
  el(t_POrder,t_PPOrder)
).

fof(anonymous, axiom,
  % tuple equality t__OrderxDate_
  ![X528,X530,X529,X531]:
    ((el(X528,t_Order) & el(X530,t_Date) & el(X529,t_Order) & el(X531,t_Date)) =>
     (ord_t__OrderxDate_(X528,X530)=ord_t__OrderxDate_(X529,X531) <=>
       (X528 = X529 & X530 = X531))) &
  % tuple type t__OrderxDate_
  ![X528,X530]:((el(X528,t_Order) & el(X530,t_Date)) =>
    (el(ord_t__OrderxDate_(X528,X530),t__OrderxDate_)))
).

fof(anonymous, axiom,
  % set equality t_P_OrderxDate_
  ![VAL532,VAL533]:((el(VAL532,t_P_OrderxDate_) & el(VAL533,t_P_OrderxDate_)) =>
    ((![EL534]:(el(EL534,t__OrderxDate_) =>
     (el(EL534,VAL532) <=> el(EL534,VAL533)))) => VAL532=VAL533))
).

% {X31: ORDER; X32: DATE}

fof(anonymous, axiom,
  % t__OrderxDate_ membership
  ![EL527]:(el(EL527,t__OrderxDate_) =>
    (el(EL527,t__OrderxDate_) <=> (?[X31,X32]: ((el(X31,t_Order) &
      el(X32,t_Date)) & (EL527 = ord_t__OrderxDate_(X31,X32)))))) &
  % t__OrderxDate_ type
  el(t__OrderxDate_,t_P_OrderxDate_)
).

fof(anonymous, axiom,
  % set equality t1
  ![VAL536,VAL537]:((el(VAL536,t1) & el(VAL537,t1)) =>
    ((![EL538]:(el(EL538,t_P_OrderxDate_) =>
     (el(EL538,VAL536) <=> el(EL538,VAL537)))) => VAL536=VAL537))
).

% P {X31: ORDER; X32: DATE}
fof(anonymous, axiom,
  % t_P_OrderxDate_ membership
  ![X535]:(el(X535,t_P_OrderxDate_) => (el(X535,t_P_OrderxDate_) <=>
    ![Y]:(el(Y,t__OrderxDate_) => (el(Y,X535) => el(Y,t__OrderxDate_))))) &
  % t_P_OrderxDate_ type
  el(t_P_OrderxDate_,t1)
).

fof(anonymous, axiom,
  % tuple equality t3
  ![X540,X542,X541,X543]:
   ((el(X540,t_Order) & el(X542,t_Status) & el(X541,t_Order) & el(X543,t_Status)) =>
    (ord_t3(X540,X542)=ord_t3(X541,X543) <=> (X540 = X541 & X542 = X543))) &
  % tuple type t3
  ![X540,X542]:((el(X540,t_Order) & el(X542,t_Status)) => (el(ord_t3(X540,X542), t3)))
).

fof(anonymous, axiom,
  % set equality t4
  ![VAL544,VAL545]:((el(VAL544,t4) & el(VAL545,t4)) =>
    ((![EL546]:(el(EL546,t3) => (el(EL546,VAL544) <=> el(EL546,VAL545)))) =>
```

```
     VAL544=VAL545))
).

% {X33: ORDER; X34: STATUS}
fof(anonymous, axiom,
  % t3 membership
  ![EL539]:(el(EL539,t3) => (el(EL539,t3) <=>
    (?[X33,X34]: ((el(X33,t_Order) & el(X34,t_Status)) & (EL539 = ord_t3(X33,X34)))))) &
  % t3 type
  el(t3,t4)
).

fof(anonymous, axiom,
  % set equality t2
  ![VAL548,VAL549]:((el(VAL548,t2) & el(VAL549,t2)) =>
    ((![EL550]:(el(EL550,t4) =>
     (el(EL550,VAL548) <=> el(EL550,VAL549)))) => VAL548=VAL549))
).

% P {X33: ORDER; X34: STATUS}
fof(anonymous, axiom,
  % t4 membership
  ![X547]:(el(X547,t4)=>(el(X547,t4) <=> ![Y]:(el(Y,t3) => (el(Y,X547) => el(Y,t3))))) &
  % t4 type
  el(t4,t2)
).

fof(anonymous, axiom,
  % tuple equality t6
  ![X552,X554,X553,X555]:
    ((el(X552,t_Order) & el(X554,t_Customer) & el(X553,t_Order) & el(X555,t_Customer)) =>
     (ord_t6(X552,X554)=ord_t6(X553,X555) <=> (X552 = X553 & X554 = X555))) &
  % tuple type t6
  ![X552,X554]:((el(X552,t_Order) & el(X554,t_Customer)) => (el(ord_t6(X552,X554), t6)))
).

fof(anonymous, axiom,
  % set equality t7
  ![VAL556,VAL557]:((el(VAL556,t7) & el(VAL557,t7)) =>
    ((![EL558]:(el(EL558,t6) =>
     (el(EL558,VAL556) <=> el(EL558,VAL557)))) => VAL556=VAL557))
).

% {X35: ORDER; X36: CUSTOMER}
fof(anonymous, axiom,
  % t6 membership
  ![EL551]:(el(EL551,t6) => (el(EL551,t6) <=>
    (?[X35,X36]:((el(X35,t_Order) & el(X36,t_Customer)) & (EL551=ord_t6(X35,X36)))))) &
  % t6 type
  el(t6,t7)
).

fof(anonymous, axiom,
  % set equality t5
  ![VAL560,VAL561]:((el(VAL560,t5) & el(VAL561,t5)) => ((![EL562]:(el(EL562,t7) =>
    (el(EL562,VAL560) <=> el(EL562,VAL561)))) => VAL560=VAL561))
).

% P {X35: ORDER; X36: CUSTOMER}
fof(anonymous, axiom,
  % t7 membership
  ![X559]:(el(X559,t7) => (el(X559,t7) <=> ![Y]:(el(Y,t6) => (el(Y,X559) => el(Y,t6)))))
&
  % t7 type
  el(t7,t5)
).

% orderStatus (orderI)
fof(anonymous, axiom,
  % orderStatus(A563) equality
  ![A563,VAL564]:((el(A563,t_Order) & el(VAL564,t_Status)) =>
```

```
    ((el(ord_t3(A563,VAL564),orderStatus)) => (orderStatus(A563)=VAL564))) &
  % orderStatus(A563) type
  ![A563]:(el(A563,t_Order) => (el(orderStatus(A563),t_Status)))
).

% +..
%  override: (ORDER <=--> STATUS) x (ORDER <=--> STATUS) --> (ORDER <=--> STATUS)
% |
%  A r: ORDER <=--> STATUS @
%    override (r,{orderI |--> cancelled}) =
%      {x: ORDER x STATUS | (x e r /\ (A p: {orderI |--> cancelled} @ x . 1 /= p . 1)) \/
%        x e {orderI |--> cancelled}}
% ---
% ---------------------- rewritten to ----------------------------
% +..
%  override: P {X194: {X202: P {X210: ORDER; X211: STATUS};
%                X203: P {X212: ORDER; X213: STATUS}};
%                X195: P {X204: ORDER; X205: STATUS}}
% |
%  A X220: P {X229: ORDER; X230: STATUS}; X221: P {X231: ORDER; X232: STATUS} @
%   E y: P {X240: ORDER; X241: STATUS} @ ((X220,X221),y) e override /\
%    (A y': P {X256: ORDER; X257: STATUS} @ not ((X220,X221),y') e override \/
%     (A X287: ORDER; X288: STATUS @ not (X287,X288) e y' \/ (X287,X288) e y)) /\
%      (A y': P {X256: ORDER; X257: STATUS} @ not ((X220,X221),y') e override \/
%       (A X291: ORDER; X292: STATUS @ not (X291,X292) e y \/ (X291,X292) e y'))
%  A r: P {X207: ORDER; X208: STATUS}; X242: ORDER; X243: STATUS @
%    not (X242,X243) e override (r,{(orderI,cancelled)}) \/
%     (A X272: ORDER @ not X272 = orderI \/ not X242 = X272) \/ X243 = cancelled
%  A r: P {X207: ORDER; X208: STATUS}; X242: ORDER; X243: STATUS @
%    not (X242,X243) e override (r,{(orderI,cancelled)}) \/
%     (A X272: ORDER @ not X272 = orderI \/ not X242 = X272) \/ X242 = orderI
%  A r: P {X207: ORDER; X208: STATUS}; X242: ORDER; X243: STATUS @
%    not (X242,X243) e override (r,{(orderI,cancelled)}) \/
%     (X242,X243) e r \/ X243 = cancelled
%  A r: P {X207: ORDER; X208: STATUS}; X242: ORDER; X243: STATUS @
%    not (X242,X243) e override (r,{(orderI,cancelled)}) \/
%     (X242,X243) e r \/ X242 = orderI
%  A r: P {X207: ORDER; X208: STATUS}; X246: ORDER; X247: STATUS @
%    (X246,X247) e override (r,{(orderI,cancelled)}) \/
%      not X246 = orderI \/ not X247 = cancelled
%  A r: P {X207: ORDER; X208: STATUS}; X246: ORDER; X247: STATUS @
%    (X246,X247) e override (r,{(orderI,cancelled)}) \/
%      not (X246,X247) e r \/ X246 = orderI
% ---
fof(anonymous, axiom,
  el(override, t10) &
  ![X220,X221]: (el(X220, t4) & el(X221, t4) =>
   (?[Y]: (el(Y, t4) & (el(ord_t9(ord_t11(X220,X221),Y),override) &
    ![Y_]: (el(Y_, t4) => (~el(ord_t9(ord_t11(X220,X221),Y_),override) |
     (![X287,X288]: ((el(X287,t_Order) & el(X288,t_Status)) =>
      (~el(ord_t3(X287,X288),Y_) | el(ord_t3(X287,X288),Y)))))) &
       ![Y_]: (el(Y_, t4) => (~el(ord_t9(ord_t11(X220,X221),Y_),override) |
        (![X291,X292]: ((el(X291,t_Order) & el(X292,t_Status)) =>
         ((~el(ord_t3(X291,X292),Y) | el(ord_t3(X291,X292),Y_)))))))))) &
  ![R,X242,X243]: ((el(R, t4) & el(X242,t_Order) & el(X243,t_Status)) =>
    ((~el(ord_t3(X242,X243),override(R,const9)) |
     (![X272]: (el(X272,t_Order) =>
      ((~(X272 = orderI) | ~(X242 = X272)))))) | X243 = cancelled)) &
  ![R,X242,X243]: ((el(R, t4) & el(X242,t_Order) & el(X243,t_Status)) =>
    ((~el(ord_t3(X242,X243),override(R,const9)) |
     (![X272]: (el(X272,t_Order) =>
      ((~(X272 = orderI) | ~(X242 = X272)))))) | X242 = orderI)) &
  ![R,X242,X243]: ((el(R, t4) & el(X242,t_Order) & el(X243,t_Status)) =>
    ((~el(ord_t3(X242,X243),override(R,const9)) |
     el(ord_t3(X242,X243),R)) | X243 = cancelled)) &
  ![R,X242,X243]: ((el(R, t4) & el(X242,t_Order) & el(X243,t_Status)) =>
    ((~el(ord_t3(X242,X243),override(R,const9)) |
      el(ord_t3(X242,X243),R)) | X242 = orderI)) &
  ![R,X246,X247]: ((el(R, t4) & el(X246,t_Order) & el(X247,t_Status)) =>
    ((el(ord_t3(X246,X247),override(R,const9)) |
     ~(X246 = orderI)) | ~(X247 = cancelled))) &
```

```
  ![R,X246,X247]: ((el(R, t4) & el(X246,t_Order) & el(X247,t_Status)) =>
    ((el(ord_t3(X246,X247),override(R,const9)) |
     ~el(ord_t3(X246,X247),R)) | X246 = orderI))
).

fof(anonymous, axiom,
  % tuple equality t11
  ![X566,X569,X567,X570]:((el(X566,t4) & el(X569,t4) & el(X567,t4) & el(X570,t4)) =>
    (ord_t11(X566,X569) = ord_t11(X567,X570) <=> (![EL568]:(el(EL568,t3) =>
     (el(EL568,X566) <=> el(EL568,X567))) &
      ![EL571]:(el(EL571,t3) => (el(EL571,X569) <=> el(EL571,X570)))))) &
  % tuple type t11
  ![X566,X569]:((el(X566,t4) & el(X569,t4)) => (el(ord_t11(X566,X569), t11)))
).

fof(anonymous, axiom,
  % set equality t12
  ![VAL572,VAL573]:((el(VAL572,t12) & el(VAL573,t12)) =>
    ((![EL574]:(el(EL574,t11) =>
     (el(EL574,VAL572) <=> el(EL574,VAL573)))) => VAL572=VAL573))
).

% {X202: P {X210: ORDER; X211: STATUS}; X203: P {X212: ORDER; X213: STATUS}}
fof(anonymous, axiom,
  % t11 membership
  ![EL565]:(el(EL565,t11) => (el(EL565,t11) <=>
    (?[X202,X203]: ((el(X202, t4) & el(X203, t4)) & (EL565 = ord_t11(X202,X203)))))) &
  % t11 type
  el(t11,t12)
).

fof(anonymous, axiom,
  % tuple equality t9
  ![X576,X578,X577,X579]:((el(X576,t11) & el(X578,t4) & el(X577,t11) & el(X579,t4)) =>
    (ord_t9(X576,X578) = ord_t9(X577,X579) <=>
     (X576=X577 & ![EL580]:(el(EL580,t3) => (el(EL580,X578) <=> el(EL580,X579)))))) &
  % tuple type t9
  ![X576,X578]:((el(X576,t11) & el(X578,t4)) => (el(ord_t9(X576,X578), t9)))
).

fof(anonymous, axiom,
  % set equality t10
  ![VAL581,VAL582]:((el(VAL581,t10) & el(VAL582,t10)) =>
    ((![EL583]:(el(EL583,t9) =>
     (el(EL583,VAL581) <=> el(EL583,VAL582)))) => VAL581=VAL582))
).

% {X194: {X202: P {X210: ORDER; X211: STATUS}; X203: P {X212: ORDER; X213: STATUS}};
X195: P {X204: ORDER; X205: STATUS}}
fof(anonymous, axiom,
  % t9 membership
  ![EL575]:(el(EL575,t9) => (el(EL575,t9) <=>
    (?[X194,X195]: ((el(X194, t11) & el(X194,t11) & el(X195, t4)) &
       (EL575 = ord_t9(X194,X195)))))) &
  % t9 type
  el(t9,t10)
).

fof(anonymous, axiom,
  % set equality t8
  ![VAL585,VAL586]:((el(VAL585,t8) & el(VAL586,t8)) =>
    ((![EL587]:(el(EL587,t10) =>
     (el(EL587,VAL585) <=> el(EL587,VAL586)))) => VAL585=VAL586))
).

% P {X194: {X202: P {X210: ORDER; X211: STATUS}; X203: P {X212: ORDER; X213: STATUS}};
%    X195: P {X204: ORDER; X205: STATUS}}
fof(anonymous, axiom,
  % t10 membership
  ![X584]:(el(X584,t10) =>
    (el(X584,t10) <=> ![Y]:(el(Y,t9) => (el(Y,X584) => el(Y,t9))))) &
```

```
  % t10 type
  el(t10,t8)
).

% {X588: ORDER x STATUS | X588 = (orderI,cancelled)}
fof(anonymous, axiom,
  % const9 membership
  ![X588]:(el(X588,t3) => (el(X588,const9) <=>
    (el(X588, t3) & el(X588,t3) & X588 = ord_t3(orderI,cancelled)))) &
  % const9 type
  el(const9,t4)
).

% override (r,{(orderI,cancelled)})
fof(anonymous, axiom,
  % override(A591,A592) equality
  ![A591,A592,VAL593]:((el(A591,t4) & el(A592,t4) & el(VAL593,t4)) =>
    ((el(ord_t9(ord_t11(A591,A592),VAL593),override)) =>
     (override(A591,A592)=VAL593))) &
  % override(A591,A592) type
  ![A591,A592]:((el(A591,t4) & el(A592,t4)) => (el(override(A591,A592),t4)))
).


% +..
% |
%  orderStatus' = override (orderStatus,{(orderI,cancelled)})
% ---
% ---------------------- rewritten to ----------------------------
% +..
%  X464: ORDER
%  X465: STATUS
%  X466: ORDER
%  X467: STATUS
% |
%  A X312: ORDER; X313: STATUS @ not (X312,X313) e orderStatus' \/
%                        (X312,X313) e override(orderStatus,{(orderI,cancelled)})
%  A X316: ORDER; X317: STATUS @
%    not (X316,X317) e override (orderStatus,{(orderI,cancelled)}) \/
%    (X316,X317) e orderStatus'
% ---
% |-? not (X464,X465) e orderStatus' \/ not (X466,X467) e orderStatus' \/
%     not X464 = X466 \/ X465 = X467
fof(anonymous, axiom,
  el(x464,t_Order) &
  el(x465,t_Status) &
  el(x466,t_Order) &
  el(x467,t_Status) &
  ![X312,X313]: ((el(X312,t_Order) & el(X313,t_Status)) =>
    (~el(ord_t3(X312,X313),orderStatus_) |
    el(ord_t3(X312,X313),override(orderStatus,const9)))) &
  ![X316,X317]: ((el(X316,t_Order) & el(X317,t_Status)) =>
    (~el(ord_t3(X316,X317),override(orderStatus,const9)) |
    el(ord_t3(X316,X317),orderStatus_)))
).
fof(conjecture, conjecture,
  ~el(ord_t3(x464,x465),orderStatus_) | ~el(ord_t3(x466,x467),orderStatus_) |
  ~(x464 = x466) | x465 = x467
).
```

Appendix F

Validating Reasoning Heuristics Using Next-Generation Theorem-Provers

Validating Reasoning Heuristics Using Next-Generation Theorem-Provers

Paul S. Steyn[1] and John A. van der Poll[2]

School of Computing, University of South Africa (UNISA)
paulsteyn@gmail.com[1] vdpolja@unisa.ac.za[2]

Abstract. The specification of enterprise information systems using formal specification languages enables the formal verification of these systems. Reasoning about the properties of a formal specification is a tedious task that can be facilitated much through the use of an automated reasoner. However, set theory is a corner stone of many formal specification languages and poses demanding challenges to automated reasoners. To this end a number of heuristics has been developed to aid the Otter theorem prover in finding short proofs for set theoretic problems. This paper investigates the applicability of these heuristics to a next generation theorem prover Vampire.

1 Introduction

Mathematical set theory is a building block of a number of formal specification languages, e.g. both Z [13] and B [1] are based on strongly-typed fragments of Zermelo-Fraenkel (ZF) [3] set theory. One of the advantages in using a formal notation for specifying an enterprise information system is that the specifier may formally reason 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 [8]. Hence of particular interest to a specifier could be the feasibility of using an automated reasoning program [12, 17] to reason about such properties.

When reasoning about the properties of a specification language based on set theory, one inevitably has to move to the level of sets and the various operations defined on them. These operations in turn are based on the underlying axioms of the particular set theory in question.

1.2 Set-Theoretic Reasoning Heuristics

Set theoretic reasoning brings about a number of problems, especially if one opts for a resolution-based reasoner like Otter [6]. Much of the complexity arises from the fact that sets may be elements of other sets. Constructs in set theory are often strongly hierarchical and may lead to deeply nested structures that greatly increase a problem's search complexity [9]. In the following equality

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

a reasoner has to transcend 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. It is generally accepted that heuristics are needed to guide reasoners, especially in the context of set-theoretic proofs [2]. One such set of heuristics for reasoning about set theory has been developed previously [15, 16], mainly through observing the behaviour of the resolution-based reasoner, Otter in its search for proofs. In total 14 heuristics, based on recognisable patterns, were developed and the question arises whether these heuristics have a wider applicability to other resolution-based reasoners, e.g. Vampire [12] and Gandalf [14]. This paper investigates the utility of the said heuristics for Vampire.

1.3 Layout of this Paper

Section 2 gives a brief introduction and justification of the use of the Vampire prover in this work. Section 3 presents the main results of our work, namely, the extent to which Vampire also needs the heuristics previously arrived at through the use of Otter. A case study in section 4 illustrates the utility of some of the heuristics on a small Z specification. We conclude with an analysis and pointers for future work.

2 The Vampire Theorem Prover

We chose Vampire [10, 12], a resolution-based automated reasoner for first-order logic with equality for evaluating the wider applicability of the 14 heuristics mentioned above for two reasons: The first is because of its consistent success at the annual CADE ATP System Competitions (CASC) [7]. The second reason stems from the fact that Vampire has solved more set-theoretic problems than any of the other competing provers in the period from 2002 to 2005 across all CASC divisions involving these problems. If we can show that Vampire benefits from the heuristics developed before, then it is plausible that other reasoners may benefit from these heuristics as well.

Vampire is a saturation-based reasoner and implements three different saturation algorithms that can be selected for its main loop for inferring and processing clauses. The three saturation algorithms are an Otter loop with or without the Limited Resource Strategy and the Discount loop. These algorithms belong to the class of given-clause algorithms. Vampire's algorithm is a slight modification of the saturation algorithm used by Otter [6].

The Limited Resource Strategy [11] aims to improve the efficiency of the Otter algorithm when a time limit is imposed by identifying and discarding passive clauses that have little chance to be processed within the time limit. The Limited Resource Strategy is therefore not a complete proof procedure.

3 Evaluation of Set-Theoretic Reasoning Heuristics

In this section we measure the utility of some previously developed heuristics [15, 16] for Vampire. Fourteen heuristics were originally developed, but for reasons of space we evaluate 5 heuristics. Our experiments follow a pattern: First we present our sample problem and the ZF axiom(s) on which the problem is based. Then we report the performance of Otter in an attempt to solve the problem. From a failed proof attempt we define a heuristic that allows Otter to successfully solve the problem. Next Vampire is used on the original problem to determine its need for the particular heuristic. In some cases we increase the complexity of the problem as an additional test.

We used Vampire version 8.0 that was also used at the CADE ATP System Competition [7] in 2005 (CASC-20). A time limit of 30 minutes and a memory limit of 128MB were imposed which causes Vampire to use its limited resource strategy. No changes were made to Vampire's other default settings.

3.1 Equality versus Extensionality

Our first sample problem based on equality and the power set axiom is given by:

$$\mathbb{P}\{\{1\}\} = \{\emptyset, \{\{1\}\}\} \quad (\mathbf{1})$$

Currently neither Otter nor Vampire accept formulae in the highly evolved notation of ZF set theory, hence the user has to rewrite set-theoretic formulae like (1) above in a weaker first-order language. Therefore, proof obligation in (1) is rewritten as:

$$A = \{1\} \wedge B = \{A\} \wedge C = \mathbb{P}(B) \wedge D = \{\emptyset, B\} \rightarrow C = D \quad (\mathbf{2})$$

Further decomposition is required for $\mathbb{P}(B)$ as follows:

$$\forall x(x \in C \leftrightarrow \forall y(y \in x \rightarrow y \in B)) \quad (\mathbf{3})$$

Otter finds no proof for (2) in 20 minutes. Next, using the extensionality axiom we replace the consequent $(C=D)$ by

$$\forall x(x \in C \leftrightarrow x \in D) \quad (\mathbf{4})$$

and this allows Otter to find a proof in 0.03 seconds. These findings lead to the following heuristic (for the sake of this paper we call it Heuristic #1):

> **Heuristic #1:** Use the principle of extensionality to replace set equality with the condition under which two sets are equal, i.e., when their elements are the same.

When the same problem (2) is given to Vampire, it has no difficulty in finding a proof in 1.3 seconds. The application of the above extensionality heuristic leads to an equally fast proof in 0.1 seconds. These times are too short to determine the utility of the heuristic for Vampire. However, consider the following, more complex example involving a subset axiom of arbitrary intersection:

$$\cap \{\{1,2,3\}, \{2,3,4\}\} = \{2,3\} \quad (5)$$

As before formula (5) is rewritten to make the relevant constructions explicit:

$$A = \{1,2,3\} \wedge B = \{2,3,4\} \wedge C = \{A,B\} \wedge D = \{2,3\} \rightarrow \cap C = D \quad (6)$$

This time Vampire finds no proof within 30 minutes. When we however apply the principle of extensionality to the consequent of formula (6) as in

$$\forall x(x \in \cap C \leftrightarrow x \in D) \quad (7)$$

then Vampire finds a short proof in *0.4 seconds*. Therefore Heuristic #1 appears to be useful for Vampire as well, depending on the complexity of the problem.

3.2 Nested Functors

An effective heuristic is to give preference to deductions containing smaller clauses [5], i.e. clauses containing fewer literals or clauses of smaller term depth. The use of nested function symbols (called *functors*) leads to larger term depth and complicates unification. The nesting of function symbols occurs often, e.g.:

$$(A + B) + C = A + (B + C) \quad (8)$$

Formula (8) states that set-theoretic symmetric difference (denoted by '+') is associative. The symmetric difference of sets A and B is defined as $A + B = (A - B) \cup (B - A) = \{x \mid ((x \in A) \wedge (x \notin B)) \vee ((x \notin A) \wedge (x \in B))\}$. Therefore formula (8) employs equality as well as a ZF subset axiom as instantiated by set-theoretic difference. A first-order definition of the symmetric difference functor is:

$$\forall A \forall B \forall x(x \in \text{symmdiff}(A,B) \leftrightarrow ((x \in A \wedge x \notin B) \vee (x \notin A \wedge x \in B))) \quad (9)$$

The conclusion of the proof is then stated as:

$$\forall x(x \in \text{symmdiff}(\text{symmdiff}(A,B), C) \leftrightarrow x \in \text{symmdiff}(A, \text{symmdiff}(B,C))) \quad (10)$$

With this formulation it takes Otter 4 minutes 3 seconds to find a proof of (10). Unfolding, and thereby effectively removing, the nested functors as

$$D = A + B \wedge E = D + C \wedge F = B + C \wedge G = A + F \rightarrow \forall x(x \in E \leftrightarrow x \in G) \quad (11)$$

allows Otter to find a proof in only 0.17 seconds, suggesting:

Heuristic #2: Avoid, if possible, the use of nested functor symbols in definitions.

Vampire quickly finds a proof of (10) in less than 0.1 seconds, both with or without the use of the nested functor heuristic. We therefore increase the complexity of the problem to further investigate the utility of Heuristic #2 for Vampire. Note that in both problem formulations the extensionality heuristic was already applied to problem conclusions. Rewriting (10) without using extensionality as

$$\text{symmdiff}(\text{symmdiff}(A,B), C) = \text{symmdiff}(A, \text{symmdiff}(B,C)) \quad (12)$$

results in Vampire finding no proof after 30 minutes. Next we apply the nested functor heuristic by rewriting our problem using Skolem constants:

$$D = A + B \wedge E = D + C \wedge F = B + C \wedge G = A + F \rightarrow E = G \quad (13)$$

Vampire now finds a proof after only *0.5 seconds*.

3.3 Divide-and-Conquer

The heuristic examined in this section is applicable to proofs where the consequence of the proof contains a set equality or an if-and-only-if formula. A set equality in the conclusion of a proof implies 'if and only if' via the axiom of extensionality. Owing to the if-and-only-if formula, a specifier can perform two separate proofs, one for the only-if part and another proof for the if part. Consider the following sample problem based on equality and the power set axiom:

$$\mathbb{P}\{0,1\} = \{\emptyset, \{0\}, \{1\}, \{0,1\}\} \quad (14)$$

The formula is rewritten to make the relevant constructions explicit:

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \wedge E = \{\emptyset, A, B, C\} \rightarrow D = E \quad (15)$$

Otter terminates without finding a refutation after 30 minutes. We resort to our extensionality heuristic by changing the conclusion to:

$$\forall x(x \in D \leftrightarrow x \in E) \quad (16)$$

Otter now finds a proof in 3 minutes 23 seconds. An alternative approach is to perform two separate proofs, one for each half of (16) and in the two proofs specify the conclusions as in (17) and (18) below.

$$\forall x(x \in D \rightarrow x \in E) \quad (17)$$

$$\forall x(x \in E \rightarrow x \in D) \quad (18)$$

Otter proves (17) and (18) in 0.43 and 0.03 seconds respectively, leading to:

Heuristic #3: Perform two separate subset proofs whenever the problem at hand requires one to prove the equality of two sets.

Vampire is also unable to find a proof for (15) after 30 minutes. However for (16), (17) and (18) Vampire finds quick proofs in 0.8, 0.3 and 0.1 seconds respectively. These times are too short to affirm the utility of the divide-and-conquer heuristic for Vampire. As before we increase the complexity of the problem through the equality:

$$\mathbb{P}\{0,1,2\} = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}, \{0,1,2\}\} \quad (19)$$

Formula (19) is again rewritten to make the relevant constructions explicit:

$$A = \{0\} \wedge B = \{1\} \wedge C = \{2\} \wedge D = \{0,1\} \wedge E = \{0,2\} \wedge F = \{1,2\} \wedge G = \{0,1,2\} \wedge H = \mathbb{P}(G) \wedge I=\{\emptyset, A, B, C, D, E, F, G\} \rightarrow H=I \quad (20)$$

Vampire terminates without finding a refutation after 8 minutes 53 seconds with the message 'no passive clauses left'. Note that this does not mean that a refutation does not exist. Since Vampire was run with both a time and memory limit, it uses the limited resource strategy [11], which is not a complete search strategy. Applying our extensionality heuristic by rewriting (H = I) above as

$$\forall x(x \in H \leftrightarrow x \in I) \quad (21)$$

allows Vampire to find a proof after 8 minutes 40 seconds which is still too long. By applying divide-and-conquer to (21) in the usual way allows Vampire to find short proofs in 28 secs and 2 secs respectively, illustrating the utility of the heuristic.

3.4 Exemplification

When writing the contents of sets in list notation one naturally tends to define these sets using one or more levels of indirection by moving from the various elements to a symbol representing the collection of those elements. The sample problem used for the divide-and-conquer heuristic will be used here as well, viz:

$$\mathbb{P}\{0,1\} = \{\emptyset, \{0\}, \{1\}, \{0,1\}\} \quad (22)$$

Recall that Otter failed to find a proof in 30 minutes for the initial unfolding in (15). Suppose we remove one level of indirection by eliminating symbol E, i.e.

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \rightarrow D = \{\emptyset, A, B, C\} \quad (23)$$

where D = {∅, A, B, C} is unfolded (repeatedly using the ZF pairing axiom) as

$$\forall x(x \in D \leftrightarrow (x = \emptyset \vee x = A \vee x = B \vee x = C)) \quad (24)$$

in the proof conclusion. With this formulation Otter finds a proof in 4 minutes 5 seconds. These results lead us to the following heuristic:

Heuristic #4: Avoid unnecessary levels of elementhood in formulae by using the elements of sets directly.

The divide-and-conquer heuristic can be applied to this last proof attempt to yield proofs in 0.34 and 0.03 seconds for the 'only-if' and 'if' directions respectively. Vampire was also unable to find a proof for (15) within 30 minutes. However, for (23) Vampire finds a proof in *0.8 seconds*. In this example, therefore, it was not necessary to increase the complexity of the problem to illustrate the utility of the heuristic for Vampire. If we do increase the complexity of the problem by again using formula (19) as an example, but instead of unfolding it as in (20) we unfold it as

$$A = \{0\} \wedge B = \{1\} \wedge C = \{2\} \wedge D = \{0, 1\} \wedge E = \{0, 2\} \wedge F = \{1, 2\} \wedge G = \{0, 1 ,2\} \wedge H = \mathbb{P}(G) \rightarrow H = \{\emptyset, A, B, C, D, E, F, G\} \quad (25)$$

then Vampire finds a proof in 5 minutes and 50 seconds. The divide-and-conquer heuristic can be applied to this last proof attempt to yield proofs in *31.5* and *1.6 seconds* for the 'only-if' and 'if' directions respectively.

3.5 Multivariate Functors

Functors containing variables as arguments lead to more unifications, which in turn lead to a larger search space. Functors are often introduced by Skolemisation [4], which occurs when first order formulae are clausified to serve as input to the resolution mechanism. If an existential quantifier occurs within the scope of any universal quantifiers, the existential quantifier is replaced by a Skolem functor taking each of the universally quantified variables as an argument.

The example problem (15) will be used again with the extensionality heuristic applied to the conclusion as in (16). First we define the term $D = \mathbb{P}(C)$ indirectly as

$$\forall x(x \in D \leftrightarrow x \subseteq C) \quad \textbf{(26)}$$

where the subset functor $\subseteq$ is defined as

$$\forall A \forall B(A \subseteq B \leftrightarrow \forall y(y \in A \rightarrow y \in B)) \quad \textbf{(27)}$$

With this formulation Otter finds no proof in 30 minutes. The clausification of (27) results in variable *y* being replaced by a Skolem function of the two variables *A* and *B*. The effect of Skolemisation may be reduced by eliminating one of the universally quantified variables in (27), e.g. replace variable *B* by the constant *C* in (26):

$$\forall A(A \subseteq C \leftrightarrow \forall y(y \in A \rightarrow y \in C)) \quad \textbf{(28)}$$

Now Otter finds a proof after 4 minutes 5 seconds. Variable *y* in the clausal form of (28) is now replaced by a Skolem functor of only one variable as opposed to a functor of two variables in (27). The possibility of irrelevant unifications with this Skolem functor has therefore been reduced. It should also be noted that the subset functor $\subseteq$ in both cases has an arity of two, but in (27) it contains two variables as opposed to one constant and one variable in (28). These results lead to:

> **Heuristic #5:** Simplify terms in sets by either not involving functors, or else functors with the minimum number of argument positions taken up by variables.

Vampire finds quick proofs with or without the heuristic applied. With the subset functor formulated as in (27) it finds a proof in 21 seconds and for (28) in 0.1 seconds. The relative improvement in search time is significant. However, the search time for (27) may still be too low to seriously justify the use of the heuristic. We therefore increase the complexity of the problem to further test our heuristic. The example problem (20) that was also used in the divide-and-conquer heuristic has sufficient complexity and will be used again with the extensionality heuristic applied to the conclusion as in (21). As before, the term $H = \mathbb{P}(G)$ is unfolded as

$$\forall x(x \in H \leftrightarrow x \subseteq G) \quad \textbf{(29)}$$

where the subset functor $\subseteq$ is again defined as in (27). With this formulation Vampire finds no proof in 30 minutes. We next apply the multivariate functor heuristic by defining the subset functor with variable *B* replaced by the constant *G*:

$$\forall A(A \subseteq G \leftrightarrow \forall y(y \in A \rightarrow y \in G)) \qquad \textbf{(30)}$$

Now Vampire finds a proof after 1 minute and 32 seconds. This result can further be improved through divide-and-conquer. The times for the two sub-proofs are 5.2 and 0.3 seconds respectively.

4 Case Study: Football Fan Register

The following case study serves as a very small example of the specification of an enterprise information system using Z and the subsequent reasoning about one of its properties using the heuristics of the previous section.
A Football Identity Scheme allocates each fan a single unique identity code. It also keeps a list of troublemakers who have been banned from attending matches. *PERSON* and *ID* are two given sets and represent the set of people and the set of all possible identity codes. The system state is recorded by *FIS* [8]*:*

FIS

$members : ID \rightarrowtail PERSON; \quad banned : \mathbb{P}\ ID$

$banned \subseteq \text{dom}\ members$

The partial injective function *members* maps identity codes to fans. The set *banned* is a set of banned identity codes and is a subset of the domain of *members*.

Schema *AddMember* adds members to the system. It takes a person as input and returns a newly allocated identity code.

AddMember

ΔFIS
$person? : PERSON; \quad id! : ID$

$person? \notin \text{ran}\ members \land id! \notin \text{dom}\ members$
$members' = members \cup \{id! \mapsto person?\} \land banned' = banned$

A Proof Obligation

Next we show how some of the above heuristics may be used to successfully discharge a proof obligation arising from the specification. We want to show that *members´* is still an injective function. The following are given as input to Vampire:

$$\text{members} \in \text{rel}(\text{id},\text{person}) \wedge \text{isSiv}(\text{members}) \wedge \text{isInjective}(\text{members}) \quad (\mathbf{31})$$

$$\text{banned} \in \mathbb{P}(\text{id}) \wedge \text{banned} \subseteq \text{dom}(\text{members}) \wedge \text{person?} \in \text{person} \wedge \text{id!} \in \text{id} \quad (\mathbf{32})$$

$$\text{person?} \notin \text{ran}(\text{members}) \quad (\mathbf{33})$$

$$\text{id!} \notin \text{dom}(\text{members}) \wedge ![M]: (M \in \text{newMembers} \Leftrightarrow M=\text{ord}(\text{id!},\text{person?})) \quad (\mathbf{34})$$

$$\text{members'} = \text{members} \cup \text{newMembers} \wedge \text{banned'} = \text{banned} \quad (\mathbf{35})$$

These facts represent the state *FIS* and operation *AddMember* Formula (31) states that *members* is a relation that is single valued and injective, i.e. a partial injective function [13]. The axioms for *rel*, *isSiv*, *isInjective*, *dom*, *ran*, *subset*, *union* etc. are not shown here but are part of the input to Vampire.

The proof obligation is stated as:

$$\text{members'} \in \text{rel}(\text{id},\text{person}) \wedge \text{isSiv}(\text{members'}) \wedge \text{isInjective}(\text{members'}) \quad (\mathbf{36})$$

Vampire finds no proof for (36) in 30 minutes. The divide-and-conquer heuristic can be applied to (36), resulting in three separate sub-proofs with consequents:

$$\text{members'} \in \text{rel}(\text{id},\text{person}) \quad (\mathbf{37})$$

$$\text{isSiv}(\text{members'}) \quad (\mathbf{38})$$

$$\text{isInjective}(\text{members'}) \quad (\mathbf{39})$$

Vampire finds proofs for (38) and (39) in 14 minutes 48 seconds and 14 minutes 24 seconds respectively, but fails to find a proof for (37) after 30 minutes. Next we apply the multivariate functor heuristic by removing axioms for union, domain, injectivity, single valued ness, power set, range, relation and subset and replace them by instances of the same axioms where some variables are replaced by constants. For example, (33) requires the following definition for the range of a relation:

$$\forall R\ \forall Y[Y \in \text{ran}(R) \Leftrightarrow \exists(X)(\text{ord}(X,Y) \in R)] \quad (\mathbf{40})$$

A replacement instance of (40) is therefore added to the proof attempt where variable *R* is replaced with constant *members*:

$$\forall Y[Y \in \text{ran}(\text{members}) \Leftrightarrow \exists(X)(\text{ord}(X,Y) \in \text{members})] \quad (\mathbf{41})$$

Vampire now finds quick proofs for (38) and (39) in 4 and 7 seconds respectively. Vampire still cannot find a proof for (37) in 30 minutes. We finally apply the nested functor heuristic to all the introduced axiom instances like (41). For example, (41) contains the nested functors *el(Y,ran(members))* and is replaced by:

$$\text{ranMems} = \text{ran}(\text{members}) \wedge \quad (\mathbf{42})$$
$$\forall Y[Y \in \text{ranMems} \Leftrightarrow \exists(X)(\text{ord}(X,Y) \in \text{members})]$$

Vampire finds a solution for sub-proof (37) in 9 minutes and 18 seconds. Solutions for (38) and (39) are also found slightly faster in 2 and 4 seconds respectively.

5 Conclusions and Future Work

In this paper we investigated to what extent a previously developed set of heuristics to facilitate proofs in set theory for a resolution-based automated reasoner are applicable to another reasoner with similar characteristics. The Vampire theorem prover was chosen for this task owing to its steadfast performance at recent CASC competitions. We evaluated 5 heuristics and found that all these heuristics are indeed needed, even though the original problem often had to be enlarged to illustrate the utility of the given heuristic using the new reasoner. Our heuristics appear to have an even larger support base since we also tested these on another reasoner, namely, Gandalf [14] and comparable results as reported on in this paper were witnessed.

Future work in this area may include an investigation into the applicability of the rest of our heuristics. Preliminary results indicate that at least 11 of the original 14 heuristics are useful, some addressing the challenge of tuples and functors with arity 6 or more [15].

References

1. Abriel, J-R. 1996. The B Book: Assigning Programs to Meanings. Cambridge University Press.
2. Bundy, A. 1999. A Survey of Automated Deduction. Tech. Rep. EDI-INF-RR-0001, Division of Informatics, University of Edinburgh. April.
3. Enderton, H. 1977. Elements of Set Theory. Academic Press, Inc.
4. Hamilton A. G. 1991. Logic for Mathematicians. Revised edition. Cambridge University Press.
5. Leitsch A. 1997. The resolution calculus. Springer-Verlag, New York.
6. McCune, W. W. 2003. OTTER 3.3 Reference Manual. Argonne National Laboratory, Argonne, Illinois. ANL/MCS-TM-263.
7. Pelletier, F. J., Sutcliffe, G., and Suttner, C. 2002. The development of CASC. AI Communications 15(2), 79-90.
8. Potter, B., Sinclair, J. and Till D., An Introduction to Formal Specification and Z, Prentice Hall, 1996.
9. Quaife, A. 1992. Automated Development of Fundamental Mathematical Theories. Automated Reasoning Series. Kluwer Academic Publishers.
10. Riazanov, A. 2003. Implementing an Efficient Theorem Prover. Ph.D. thesis, University of Manchester.
11. Riazanov, A. and Voronkov, A. 2003. Limited resource strategy in resolution theorem proving. Journal of Symbolic Computing 36(1-2), 101-115
12. Riazanov, A. and Voronkov, A. 2002. The design and implementation of VAMPIRE. AI Communications 15(2), 91-110.
13. Spivey, J. M. 1992. The Z Notation: A Reference Manual, 2nd ed. Prentice-Hall, London.
14. Tammet, T. 1997. Gandalf. Journal of Automated Reasoning 18(2), 199-204.
15. van der Poll, J. A. 2000. Automated Support for Set-Theoretic Specifications. Ph.D. thesis, University of South Africa.
16. van der Poll, J. A. and Labuschagne, W. A. 1999. Heuristics for Resolution-Based Set-Theoretic Proofs. South African Computer Journal Issue 23, 3 – 17.
17. Wos, L. 2006. Milestones for automated reasoning with OTTER. International Journal on Artificial Intelligence Tools, 15 (1): 3 – 19, February.

References

1) Abriel, J-R. 1996. The B Book: Assigning Programs to Meanings. Cambridge University Press.

2) Amalio, N., Polack F., Comparison of formalisation approaches of UML class constructs in Z and Object-Z, Proceedings, ZB 2003, Turku, Finland. LNCS 2651, Springer, June 2003.

3) Bachmair, L., Ganzinger, H., 2001. Resolution Theorem Proving. In: A. Robinson and A. Voronkov, eds. Handbook of Automated Reasoning, Vol. I, Elsevier Science, 2001, chapter 2, pp. 19-99.

4) Booch, G., Rumbaugh, J., Jacobson, I., 2005. The Unified Modeling Language User Guide, 2nd Edition, Addison-Wesley.

5) Boyer, R. et al, 1986. Set theory in first-order logic: clauses for Gödel's axioms. J. Automated Reasoning, 2(3), pp 287-327. Hingham, MA, USA: Kluwer Academic Publishers.

6) Bundy, A. 1999. A Survey of Automated Deduction. Tech. Rep. EDI-INF-RR-0001, Division of Informatics, University of Edinburgh. April.

7) Chang, C.L., Lee, R.C.T., 1973. Symbolic Logic and Mechanical Theorem Proving. Boston: Academic Press.

8) Church, A., 1936a. An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics, 58, 345-363.

9) Church, A., 1936b. A Note on the Entscheidungsproblem. Journal of Symbolic Logic, 1, 40-41.

10) Claessen K., Sutcliffe G., 2008, Department of Computer Science, University of Miami, viewed 13 November, 2008, <http://www.cs.miami.edu/~tptp/TPTP/Proposals/TypedFOF.html>.

11) Davis, M., Putnam, H., 1960. A Computing Procedure for Quantification Theory. J. ACM, 7(3), pp 201-215.New York, NY, USA: ACM Press.

12) Davis, M., Logemann, G., Loveland, D., 1962. A machine program for theorem-proving. Commun. ACM, 5(7), pp 394-397.New York, NY, USA: ACM Press.

13) Denzinger, J., Kronenburg, M., Schulz, S., 1997. DISCOUNT: A Distributed and Learning Equational Prover. Journal of Automated Reasoning (18), pp 189-198.

14) Derrick, J., Boiten, E., 2001. Refinement in Z and Object-Z, Foundations and Advanced Applications. Springer-Verlag.

15) Duke, R., Rose, G., Smith, G., 1995. Object-Z: a Specification Language Advocated for the Description of Standards. Computer Standards and Interfaces, 17, pp 511-533. North-Holland.

16) Dybvig, R.K., 2003. The Scheme Programming Language. MIT Press.

17) Enderton, H. 1977. Elements of Set Theory. Academic Press, Inc.

18) Eisinger, N., Ohlbach, H.J., 1993. Deduction systems based on resolution. New York, NY, USA: Oxford University Press, Inc..

19) Epstein, R.L., Carnielli, W.A., 2000. Computability. Functions, Logic, and the Foundations of Mathematics, 2nd Edition. Wadsworth/Thomson Learning.

20) Gilmore, P.C., 1960. A proof method for quantification theory: its justification and realization. j-IBM-JRD, 4, pp 28-35.

21) Hamilton A. G. 1991. Logic for Mathematicians. Revised edition. Cambridge University Press.

22) ISO 2002, ISO 13568:2002, Information technology - Z formal specification notation - Syntax, type system and semantics, International Organization for Standardization.

23) Leitsch, A., 1997. The resolution calculus. New York, NY, USA: Springer-Verlag New York, Inc..

24) Malik, P. and Utting, M., 2005. CZT: A Framework for Z Tools. In: Treherne, H., King, S., Henson, M., Schneider, S. (Eds.), LNCS, 3455. Springer-Verlag. pp. 65-84.

25) McCharen, J., Overbeek, R., Wos, L., 1967. Complexity and related enhancements for automated theorem proving programs. Computers and Mathematics with Applications, 2, pp 1-16.

26) McCune, W., 2003. OTTER 3.3 Reference Manual. CoRR, cs.SC/0310056.

27) McCune W., 2009. Prover9 Manual. Argonne National Laboratory, viewed 3 February, 2009, <http://www.prover9.org>.

28) Meltzer, B., 1966. Theorem-proving for computers: Some results on resolution and renaming. TCJ, 8, pp 341-343".

29) Montague, R., 1961. Semantic Closure and Non-Finite Axiomatizability I, in Infinitistic Methods, Proceedings of the Symposium on Foundations of Mathematics, (Warsaw, 2-9 September 1959). Oxford, England: Pergamon, pp. 45-69, 1961.

30) Nerode, A., Shore, R., 1997. Logic for Applications. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

31) Nieuwenhuis, R., Rubio, A., 2001. Paramodulation-Based Theorem Proving. In: A. Robinson and A. Voronkov, eds. Handbook of Automated Reasoning, Vol. I, Elsevier Science, 2001, chapter 7, pp. 371-443.

32) Pelletier, F.J., Sutcliffe, G., Suttner, C.B., 2002. The Development of CASC. AI Communications, 15(2-3), pp 79-90.

33) Plaisted, D.A., 1993. Equational Reasoning and Term Rewriting Systems. In: Dov Gabbay, Christopher Hogger and J. A. Robinson, eds. The Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 1: Deductive Methodologies, Oxford: Oxford University Press, 1993, pp. 274-367.

34) Potter, B., Sinclair, J. and Till D., An Introduction to Formal Specification and Z, Prentice Hall, 1996.

35) Quaife, A., 1992a. Automated deduction in von Neumann-Bernays-Gödel set theory, Journal of Automated Reasoning, Volume 8, Issue 1, Feb 1992.

36) Quaife, A., 1992b. Automated Development of Fundamental Mathematical Theories. Norwell, MA, USA: Kluwer Academic Publishers.

37) Quine, W. V. 1971. Set Theory and its Logic. Harvard University Press, Cambridge, MA.

38) Riazanov, A., Voronkov, A., 2001. Vampire 1.1 (System Description). IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning, London, UK: Springer-Verlag, 2001, pp. 376-380.

39) Riazanov, A., Voronkov, A., 2002. The design and implementation of VAMPIRE. AI Commun., 15(2), pp 91-110. Amsterdam, the Netherlands: IOS Press.

40) Riazanov, A. 2003. Implementing an Efficient Theorem Prover. Ph.D. thesis, University of Manchester.

41) Robinson, J.A., 1965a. A Machine-Oriented Logic Based on the Resolution Principle. J. ACM, 12(1), pp 23-41.New York, NY, USA: ACM Press.

42) Robinson, J.A., 1965b. Automatic deduction with hyper-resolution. International Journal of Computer Mathematics, 1, pp 227-234.

43) Robinson, G., Wos, L., 1969. Paramodulation and theorem-proving in first-order theories with equality. In: Bernard Meltzer and Donald Michie, eds. Machine Intelligence 4, Edinburgh, Scotland: Edinburgh University Press, 1969, pp. 135-150.

44) Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W. Object-Oriented Modelling and Design. Prentice-Hall, 1991.

45) Slagle, J.R., 1967. Automatic Theorem Proving With Renamable and Semantic Resolution. J. ACM, 14(4), pp 687-697. New York, NY, USA: ACM Press.

46) Steyn, P.S., Van der Poll, J.A., 2007. Validating Reasoning Heuristics Using Next-Generation Theorem-Provers, In J.C. Augusto, J. Barjis, U. Ultes-Nitsche, eds., Proc. 5th MSVVEIS'07, pp. 43-52, Funchal, Madeira, Portugal, June 2007.

47) Spivey, J. M. 1992. The Z Notation: A Reference Manual, 2nd ed. Prentice-Hall, London.

48) Stickel, M.E., 1985. Automated deduction by theory resolution. J. Autom. Reason., 1(4), pp 333-355.Hingham, MA, USA: Kluwer Academic Publishers.

49) Sutcliffe, G., Suttner, C.B., 1998. The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning, 21(2), pp 177-203.

50) Sutcliffe, G., Suttner, C., 2006. The state of CASC. AI Communications, 19(1), pp 35-48.

51) Tammet, T., 1997. Gandalf. Journal of Automated Reasoning, 18(2), pp 199-204.

52) Tammet, T., OCT 1997. Gandalf version c-1.0c Reference Manual.

53) Tolkien, J.R.R., 1966. The Lord of the Rings, 2nd ed. Allen & Unwin, London.

54) Turban, E., Frenzel, L.E., 1992. Expert Systems and Applied Artificial Intelligence. Prentice Hall Professional Technical Reference.

55) Turing, A.M., 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, series 2, 42 (1936-37), 230-265.

56) Van der Poll, J. A. and Labuschagne, W. A. 1999. Heuristics for Resolution-Based Set-Theoretic Proofs. South African Computer Journal Issue 23, 3 – 17.

57) Van der Poll, J. A. 2000. Automated Support for Set-Theoretic Specifications. Ph.D. thesis, University of South Africa.

58) Voronkov, A., 2001. Algorithms, Datastructures, and other Issues in Efficient Automated Deduction. IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning, London, UK: Springer-Verlag, 2001, pp. 13-28.

59) Voronkov, A., Mar 2005. Vampire 7 Reference Manual and Guide. School of Computer Science, University of Manchester.

60) Wieringa, R., A survey of structured and object-oriented software specification methods and techniques. ACM Computing Surveys, 30(4):459-527, December 1998.

61) Wos, L., Robinson, G.A., Carson, D.F., 1965. Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. J. ACM, 12(4), pp 536-541.New York, NY, USA: ACM Press.

62) Wos, L. et al, 1967. The Concept of Demodulation in Theorem Proving. J. ACM, 14(4), pp 698-709.New York, NY, USA: ACM Press.

63) Wos, L. et al, 1984. Automated Reasoning: Introduction and Applications. Prentice Hall Professional Technical Reference.

64) Wos, L., 1988. Research Problem #8: An Inference Rule for Set Theory. In Automated Reasoning: 33 Basic Research Problems, pp. 137 - 138. Prentice-Hall.

65) Wos, L., 1989. The problem of finding an inference rule for set theory, Journal of Automated Reasoning, Volume 5, Issue 1, pp. 93 – 95.

66) Wos, L. et al, 1992. Automated reasoning (2nd ed.): introduction and applications. New York, NY, USA: McGraw-Hill, Inc.

67) Wos, L., Veroff, R., 1994. Logical basis for the automation of reasoning: case studies. New York, NY, USA: Oxford University Press, Inc.

68) Wos, L., 1995. The Resonance Strategy. Computers and Mathematics with Applications, 29(2):133-178. (Special issue on Automated Reasoning)

69) Wos, L., 1996. The Power of Combining Resonance with Heat. Journal of Automated Reasoning, 17(1):23-81.

70) Wos, L. 2006. Milestones for automated reasoning with OTTER. International Journal on Artificial Intelligence Tools, 15 (1): 3 – 19, February.

Index

aggregation, 118

association, 107

association class, 110

automated theorem prover. *See* theorem prover

binary resolution, 23

  example, 24

  soundness & completeness, 21, 29

cade atp system competition. *See* casc

casc, 2, 67

class, 105

  attribute, 105

clausal form, 16, 74

completeness, 21, 29, 33

composition, 118

decidability, 14

decision problem, 14

demodulation, 60

discount algorithm, 70

dissertation

  contributions, 149

  future work, 150

  research question, 2

equality predicate, 56

factoring, 28

farmer goat cabbage wolf puzzle, 25

gandalf, 72, 77

  example input, 74

given sets, 174

Herbrand’s universe, 14

heuristics

  divide-and-conquer, 82, 136, 137, 139, 143

  element structure, 90, 146

  equality vs extensionality, 78, 133, 137

  exemplification, 84, 134, 140

  intermediate structure, 88

  multivariate functors, 86, 146

  nested functors, 80

  redundant information, 91, 139, 142

  resonance, 96, 146

  search-guiding, 93

tuple condense, 98
hyperresolution, 46
inheritance, 119
input resolution, 34
level-saturation method, 30
linear resolution, 31
soundness & completeness, 33
Neumann-Bernays-Gödel, 12
operations, 112
total, 117
otter, 2, 69, 77
otter algorithm, 69
paramodulation, 57
prenex normal form, 17
proof obligations, 132
case study, 102
CancelOrder after state type, 144
CreateDeleteItem leaves state unchanged, 140
CreateProduct invariant, 132
CreateProduct is total, 134
ProcessOrder set contents, 138
typical, 122
after state type, 124
contents of a set, 127
initialisation theorem, 122
operation interaction, 126
precondition simplification, 122
state invariant, 128
total operations, 125
reasoner. *See* theorem prover
refutation, 15
resolution, 16
binary resolution, 18, 23
factoring, 28
heuristics, 63
in predicate logic, 22
in propositional logic, 18
redundancy and deletion, 52
subsumption, 52
tautologies, 54
refinements, 30
hyperresolution, 46
input resolution, 34
linear resolution, 31

semantic resolution, 37
set-of-support, 50
unit resolution, 36
ur-resolution, 44
theory resolution, 55
demodulation, 60
equality predicate, 56
paramodulation, 57
unification, 22
resolvent, 24
Russell’s paradox, 6
schema
CancelOrder, 144, 177
Company, 120, 181
CreateCompany, 181
CreateCustomer, 180
CreateItem, 178
CreateOrder, 176
CreatePerson, 183
CreateProduct, 112, 175
CreateProductTotal, 117
CreateProductZen, 133
Customer, 109, 120, 180
DeleteCompany, 182
DeleteCustomer, 181
DeleteItem, 179
DeletePerson, 183
DeleteProduct, 115, 175
DuplicateProductName, 118
InitItem, 178
InitOrder, 176
InitProduct, 121, 174, 180, 181, 182
InitSystem, 121, 184
Item, 111, 178
Order, 109, 110, 176
Person, 120, 182
PreCreateProduct, 123
ProcessOrder, 116, 138, 177
Product, 106, 174
ProductAlreadyKnown, 118
SelectItemsForOrder, 111, 179
SelectOrdersForCustomer, 109, 177
SelectProductsBelowThreshold, 113, 175
Success, 117

System, 121, 183
UpdateCompany, 182
UpdateCustomer, 180
UpdateItem, 179
UpdatePerson, 183
UpdateProduct, 114, 175
semantic resolution, 37
predicate ordering, 39
splitting, 38
the clash, 40
set theory, 5
naive set theory, 6
Neumann-Bernays-Gödel, 12
paradoxes, 5
Zermelo-Fraenkel, 5
set-of-support, 50
Skolem functor, 17
Skolemisation, 17
soundness, 21, 29, 33
standardising variables apart, 24
subsumption, 52
system state, 120
tautologies, 54
theorem prover, 67
gandalf. *See* gandalf
otter. *See* otter
vampire. *See* vampire
tptp, 70, 130
unification, 22
most general unifier (mgu), 23
Unified Modelling Language (UML), 103
unit resolution, 36
ur-resolution, 44
vampire, 68, 77
example input, 71
Z, 1, 104
conversion to 1st order logic, 130
Zermelo-Fraenkel, 5
axioms
choice, 11
empty set, 7
extensionality, 7
foundation/regularity, 10
infinity, 9

pairing, 7
power set, 9
replacement, 10
subset, 8
union, 8
limitations, 12