Department of Computer Science

Technical Report

Undergraduate Dissertation: Incremental Answer Set Programming

Martin John Brain

Technical Report CSBU-2004-05 **May 2004**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Incremental Answer Set Programming

Martin John Brain

May 2004

MMath in Mathematics

Incremental Answer Set Programming

Submitted by Martin John Brain

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the degree Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution or learning. Except where specifically acknowledged, it is the work of the author.

Signed: .....

Date: .....

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purpose of consultation.

Signed: .....

Date: .....

Abstract

This document presents a series of algorithms for computing the answer sets of an *AnsProlog** programs. They focus on the efficient calculation of the effect that small changes on a logic program's rule base have on the programs answer sets rather than computing from no initial knowledge. In application domains where the program is developed or altered in stages and solutions are required for each stage this provides a more efficient approach than recomputing the program completely each time. Also discussed is an implementation of this theory in an interactive answer set computation tool. Finally a discussion of quantitive assessment methods and a qualitative assessment of the algorithms and implementation are presented and future development discussed.

Acknowledgements

The author would like to express his thanks to Dr Marina De Vos for her support, suggestions and tireless dedication to the role of supervisor for this project. Thanks also to Professor John ffitch for helping provide inspiration and motivation.

In the 'world + dog' thanks section the author would like to express gratitude to Richard M. Stallman for his work in inspiring and creating the GNU project, without whom very little of the technology used in this project would exist; my housemates, friends and family for their tolerance and support; Liam Howlett, Maxim Reality, Leeroy Thornhill and Keith Flint of The Prodigy and Lee Chaos, Kelly, Robin and Huw of Chaos Engine without who's music none of this would have got written in time and the Day Chocolate company. To everyone who has helped - thanks.

The creation of this document and the project it describes were achieved using free, open source software; specifically no Microsoft products were used.

Contents

Chapter 1

Introduction

If economists and modern historians are to be believed, we are now living in the information age. The changes in how information is created, stored and transmitted that motivate this definition are largely down to advanced in digital electronics, primarily the computer. For anyone with at least a passing familiarity with this technology (including everyone who has used a computer in the past 20 years) it should be clear that the problem of representing information in the digital domain is largely solved. Computer hardware and software for inputing, representing and outputting text, graphics, video, analogue data (such as sound) and even 3 dimensional objects is available to all (modulo the usual economic inequalities). Storage and transmission of digital information, although still active areas of both academic and commercial research, are basically speaking. Computers with the capacity to store digital representations of tens of thousands of books (or countless million stone tablets) are sold as entertainment devices, while the Internet lets anyone with access to suitable equipment share any information with anyone else on the planet (and beyond) given enough time. The main problems of handling information with digital and computer technology have been mainly solved.

The next layer of problems focuses on how to process digital information, critically how to represent and manipulate the links between bits[1] of information. Partial solutions to this problem will be apparent to any user of modern computer equipment. Abstractions such as files and file meta data(type, owner, creation date, etc.), hierarchical file systems and relational databases allow the storage of pieces of information and the links between them given certain criteria. Each of them has their own strengths and weaknesses, for example a hierarchical file system is a natural choice for information with a clear sort order, but when files need to be accessed using a variety of different sorting system quickly become unwieldy (hence the current development of relational database based file systems). Relational databases perform very well when handling information in a given structured form but have difficult representing more free form information and encoding ideas based on the lack of particular bits of knowledge. To some degree it is possible to express any set of information and links using any of these abstractions, in many cases this is not practical. Logic programming offers one of the most expressive and yet simplistic systems for representing pieces of information and the links between them. A large number of categories of information structures that are difficult (if not practically impossible) to represent using any of the more well known abstractions can be expressed trivially using logic programs.

Another motivating factor for using data abstraction mechanisms is the ability to query the information contained and derive conclusions from it. Humans do not simply store knowledge[2], they can use the links between the items of knowledge to reason, plan, conjecture and interact with the world, in short they think. Applying processes to data stored in a particular abstract form to develop new data is, in essence attempting to teach computers how to think in some capacity[3]. The particular logic programming systems the work will address is $AnsProlog^*$, the semantics of this system (the definition of what each statement in the language actually means) define a concept of answer sets. These are sets of pieces of information which are logically consistent with the logic program (one set of information and links, stored according the abstraction), depending on the context of the program these may be conclusions, solutions or possible world views. A number of algorithms for computing the answer sets

[1] As in 'and pieces' not as in 'and bytes'.

[2] Although anyone involved in education at any level might argue that 'Humans do not store knowledge simply' is a more apt comment

[3] Whether this can actually be called thinking or not is a very complex issue and partially case of philosophical belief. Many argue that there is a fundamental difference between human and machine 'intelligence'; humans are able to create new ideas, where as intelligent machines can only create information from what they already have. This of course requires a belief in a non deterministic universe...

of a given logic program (in this system) exist, these can be thought of as being like the process a detective might take working through the items of evidence and the links between them.

This work describes a new algorithm for computing the answer sets of an *AnsProlog** program. It is different from existing algorithms in that it calculates the answer sets of a program using the answer sets of a program that is slightly different, containing one more or one less link. The target application domain for this algorithm is situations where the logic program develops over the course of the application (for example, information is added over time). Currently if the program changes then the answer sets have to be recomputed from scratch, which is inefficient and slow if there are a large number of changes. Just as a detective would not restart an investigation when a new piece of evidence is found, or when a new link between suspects is uncovered, this algorithm modifies the views of the world rather than starting again from scratch. The aim of the project that this document describes was to find a faster and more efficient way of handling these situations.

This document is not structured as a traditional software project, the bulk of the work is in developing theory, the implementation is more of a proof of concept than the focus of the project. The theory presented is all new and to the best of the author's knowledge there are no previously published works that discuss this topic at all. There are no clear requirements in the strict software engineering sense so the traditional approach of identifying stake-holders, analysing requirements, developing designs, implementing them, user testing and evaluation against the initial specification is simply not relevant. Chapter 2 provides a brief introduction to Logic as a formal subject, this is intended to introduce key concepts as well as setting the scene for later theoretical content. Chapter 3 formalises the ideas of logic programming that have been touched on in this introduction, presents the formal definition of *AnsProlog**, discusses algorithms for computing answer sets and their implementation and applications, as well a brief discussion of other logical systems. Next chapter 4 presents the theoretical basis and the content of the incremental computation algorithm. Initially only the addition of ground[4] rules to the system is considered, however results for subtracting rules and handling unground rule are also presented. Chapter 5 discusses the implementation of this algorithm, the IASAI system for providing a programming interface to a variety of answer set solver algorithms and the IDEAS tool which uses this to provide a intuitive, command line environment for manipulating logic programs and computing their answer sets. Chapter 6 presents some of corollaries of section 4.2 in a different light, the technology used to incrementally compute answer sets can also provide debugging and analysis features in an interactive development environment. It is possible to produce metrics for the significance of atoms within a program and the relevance of rules. Chapter 7 discuss the issues surrounding benchmarking of the IDEAS tool using a variety of different solver algorithms (including the incremental algorithm discussed in chapter 4) and characterises the type of environment required to perform any form of meaningful quantitive analysis. Finally chapter 8 presents some possible directions for future research in this field as well as concluding the project.

[4]Rules that do not contain variables, this term is defined and explained in chapter 3

Chapter 2

Logic

This chapter provides background information on logic and aims to dispel some of the common myths about logic systems and logic research. This provides a formal background and conceptual context for the definitions of logic programming presented in chapter 3. Without a base understanding of logic no real understanding of logic programming, what problems it attempts to solve and how it dose so can be developed. In essence, logic programming is the implementation and automation of logic.

2.1 What is Logic?

The Oxford English Dictionary [1] defines logic as

> The branch of philosophy that treats of the forms of thinking in general, and more especially of inference and of scientific method. (Prof. J. Cook Wilson.) Also, since the work of Gottlob Frege (1848-1925), a formal system using symbolic techniques and mathematical methods to establish truth-values in the physical sciences, in language, and in philosophical argument.

A slightly less cumbersome definition that we will use in this work is "a formal system for representing and manipulating the relations between concepts".

2.2 Why is it Useful?

Structured reasoning based on logic underpins the concepts of mathematical proof and the scientific method. Without it neither science or mathematics could be considered to be rigorous subjects - neither would have any firm basis for claiming any results to be true. Although not clearly applicable [1] in 'everyday life', logic is fundamental to how we understand the world and how we build on our current understanding.

But why continue to study it? Most of the significant problems have been solved, logic as used in mathematics and science is regarded to be well understood by many. Most of the research work within logic is based on applying logic and logical systems to new areas of human endeavour. Knowledge representation, intelligent inference engines and automatic diagnostic systems are all prime examples.

2.3 Types of Logic

A common misunderstanding is that there is only one form of logic: A proposition can either be proved or not; an action is either 'logical' or 'illogical'. Bizarrely[2] little could be further from the truth. The academic study of logic covers many different logic systems. A logic system is a set of literals (the most basic building blocks of information) and connectives (operators that form a new statement from one or more literals). More complex logic systems also have more complex objects such as functions, variables and functors. This gives different ranges of expressiveness, both in terms of how much and what can be expressed.

Choosing which logical system to use is very significant and potentially a complex decision. The system has to be expressive enough to allow everything to be said (and hopefully to be said in a concise

[1] ...to the average person. Mathematicians and logicians are considered non average for the purposes of this argument.
[2] Illogically?

and simple way[3]): this has to be traded off against the other properties of the system. For example is there a useful and meaningful concept of 'solution' with respect to the problem, are there algorithms and/or methods for computing these, are the algorithms efficient or even deterministic.

Obviously a logic system has to be able to do more than just represent collections of concepts - it has to have a way of creating new statements in a rigorous manner. The system for doing this is referred to as a logical calculus. These can be developed in a variety of ways. Semantic calculi are based on model theory. The statement in the logical language is mapped into another set which models the atoms and connectives. Truth tables are a prime example of this sort of calculus - literals are given a 'truth value' and connectives map into simple operators. The key concept in semantic calculi is the idea of satisfaction (denoted as $\vDash$ e.g. $p \vDash_u q$ is the statement given statement p is satisfied in model u then so is statement q), essentially this is 'is this statement true with respect to this model'. Semantic calculi are essentially assigning a form of meaning to the logical statements. Another major class of logical calculi are syntactic calculus, which are based on how the statements and their connectives can be structured. Natural deduction, tableau and sequent calculus are all examples of this form of structured reasoning. The corresponding key idea is inference - 'can a proof this statement be constructed' (denoted as $\vdash$ E.G. $p \vdash_u q$ is the statement there is a proof in calculus u from statement p to statement q).

Calculi of the same kind can be related; shown to be equivalent, disjoint or contained within another calculus. To speak about equivalence between other kinds of calculi requires the concepts of soundness and completeness. A syntactic calculus is sound with respect to a semantic calculus if for every inference from statement p to q, q is satisfied given p is satisfied. Completeness is the converse statement. Proofs of this form are useful as they show the validity of a particular calculus with respect to another and give an alternative method for proving or disproving a given relation. In general creating a proof is easier in a syntactic calculus while showing that a proof does not hold is easiest in a semantic calculus.

Here we present some of the most common and fundamental logic systems, there are many more but it is hoped that this small selection will give the ideas of the different degrees of expression and levels of difference between them.

2.3.1 Classical Propositional Logic

Classical logic is the system that comes closest to the popular perception of logic. It consists of atoms (often thought of as concepts or statements) and four connectives, AND, OR, $IMPLIES$ and NOT (notated as $\wedge$, $\vee$, $\Rightarrow$ and $\neg$. The semantic calculus most commonly associated with this system is that of truth tables. The idea that literals are two valued - thought of as 'true' and 'false' may seem strange at first but occurs in other calculi on this logical system. A consequence of natural deduction (the syntactic calculus of classical logic) is that '$p \vee \neg p$' can always be asserted - nicely parallel the assignment to a two valued system. This concept is referred to as the Law of the Excluded Middle - if we know that something is "not not true" ($\neg\neg p$) then it must be "true" (p).

2.3.2 Intuistic Logic

Although the law of the excluded middle is a great advantage in constructing proofs (proof by contradiction relies on this principle) - it can be argued that it is an excessive and unnecessary simplification. Intuitistic logic is an alternative set of calculi that can be used with the same logical language as classical logic. In refuting the law of the excluded middle the structure of natural deduction (notably the role of negation and implication) has to be altered. This leads to the need for a new semantic calculus known as Kripke possible worlds.

Whether classical logic or intuistic logic is the 'correct' way of doing things is something of a philosophical debate. Intuistic logic has some useful features but is much more difficult to actually develop proofs in, although the proofs that are constructed tend to be of more use as they not only showing that an object can exist - they demonstrate how to find it.

When applying a logical system to modelling knowledge the existence of the law of the excluded middle gives rise to a phenomenon called the closed world assumption. In a system with the laws of the exclude middle, not knowing something is true is the same as knowing it is not true. The contrast with this can the 'real world' intuition that these two things should be different lead some systems to define two types of negation, classical (with law of the exclude middle style semantics) and negation as failure (with intuistic style negation semantics).

[3]Clarity of expression is a very important feature of a logic system as many of the major problems with using logic come with subtle misunderstandings of how to formalise a particular concept.

2.3.3 Predicate Logic

The logical systems described so far have no real way of talking about groups of similar objects. For example "Epimenides is a Cretan" or "Not all Cretan are liars". To express this the logical languages need to be extended with the concepts of predicates, variables, functions and quantifiers. Predicates take one of more objects from a set and act as literals. They are used to represent information about the objects, for example a predicate taking two names ($older(X, Y)$) could represent the concept 'is older than'. If the two names are 'fred' and 'bob' the predicate would represent 'fred is older than bob' ($older(`fred', `bob')$). Variable are place holders for the values that a predicate takes, they are introduced by qualifiers. Functions create new objects from a finite set of old ones (for example addition). Quantifiers are new connectives allowing statements to be built talking about classes of predicates. They represent the concepts of 'there exists' (notated as $\exists$) and 'for every' (notated as $\forall$).

Predicates vastly increase the expressive power of both classical and intuistic logical systems. However there is a problem, there can exist no method that can prove whether a statement in predicate logic is valid[4] (satisfied with no starting assumption) in the general case, i.e. there are undecidable propositions. Predicate logic is expressive enough to formalise what we want to say but too expressive to actually use in the arbitrary case.

[4]This is a corollary of the famous Godel's Incompleteness Theorem[2]

Chapter 3

Logic Programming

This chapter introduces the concepts of logic programming from its foundations to the problems that motivate answer set programming and it's implementation.

3.1 What is Logic Programming?

The fundamental building blocks of modern digital computers are based on classical logic[3]. Data is represented as a series of zeros and ones which could be viewed as truth values, operations on these can then be characterised as logical formulae. In turn this means that building computer models of logical systems is not only possible but often comparatively simple, natural and straight forward.

Early in the development of digital computers[4] it was observed that the building of models of logical systems could easily be automated[1]. Computers could be programmed to perform logical inference between statements. In answering some forms of query and proving some kinds of propositions were possibly more suitable than humans. Their speed of operation and ability to perform exactly the same task repeatedly made them a more viable tool.

Logic programming addresses designing, building and using automated inference systems. These use information structured in a particular (computer) language representing a given logic system to answer queries and provide logical conclusions from the data. For example given structured information on the relations between friends and family it would be possible to use logic programming tools to find how many seating plans there are for Christmas dinner that don't cause undue strife; certain tools may also be able to pick the optimum solution (assuming that the user can decide on and give suitable information on preference).

There are fundamental differences between logic programming and imperative based programming paradigms; there is no distinction between the concepts of data and functions or between programs and specifications. Rather than telling the computer how to accomplish the task ('find kettle, pour water in, heat up water') you tell it what it needs to know about the world ('to make tea you need boiling water, tea, milk and sugar', 'a kettle can boil water', 'the kettle is in the cupboard') and what you want ('make me a cup of tea') and let it figure out what needs to be done. Obviously it is not a suitable tool for all programming tasks, some things are more naturally described as imperative tasks (e.g. operating system kernels) but a surprisingly high number of tasks can be naturally described as logic programs.

3.2 Why is it useful?

Perhaps the most high profile usage of logic programming to date has been in artificial intelligence. To create a system that appears to behave in an 'intelligent' fashion, you must have a way of representing what the system 'knows' about the world it is in and a way of drawing conclusions from this information to guide its actions. Clearly the best way of expressing these is to use a language and tools that use 'chunks'[2] of knowledge as building blocks and logical relation and inference as their main operators. Much of the early work on AI and expert systems was attempting to structure the knowledge of human experts about a given system into a form that would be suitable for use in logic programming. This

[1] Arguably the automation of logical calculations predates the digital computer. The first mechanical logic machine was created by William Stanley Jevons in 1869!

[2] Exactly what constitutes an atomic unit of knowledge is clearly both a difficult and an application specific concept. For the process of representing and working with the relations between these atoms the choice is largely irrelevant.

approach is used in many modern automatic diagnostic systems; the designers of the system create a logical description of the system and tests which can be used to determine where problems are. The resulting program can then be used interactively to diagnose (manually or semi automatically) problems and suggest solutions.

However logic programming and AI are far from being synonymous. AI encompasses many other areas such as modelling how natural systems work, pattern recognition and machine learning techniques. Logic programming also has other applications. Machine planning systems - such as timetabling software is often based on logic programming ideas. Given the natural correlation between structured information stored in a database and the structured knowledge used in logic programming there has been quite a lot of interest in combining the two in advanced data manipulation work. Logic programming has been used in data reconstruction, integrity, combination of non uniform knowledge sources[5] and data mining[6].

Arguably one of the most interesting developing uses for logic programming is in Intelligence Augmentation (IA). Where as AI seeks to create intelligent systems to reduce the amount of human interaction needed, IA aims to reduce the amount of information that the humans in a system need to be dealing with[7]. Rather than the computer attempting to work out what to do, and IA system would work out possible scenarios of the current and future state of the system and then present these to the human operators. Essentially the computer is doing the dull, repetitive thinking and summarising it so that human interaction is only needed to make the important decisions. From helping traders on the stock market make quick decisions correct to asking the ships computer to find possible escape routes, this is the sort of application has the potential to change how people think about complex systems in the same way as robotics revolutionised precision engineering.

3.3 PROLOG

PROLOG (standing for 'PROgramming in LOGic') is one of the earliest and perhaps best known of logic programming languages. It is based on a restricted set of first order predicate calculus known as Horn clauses. Horn clauses are statements of the form:

$$p_1 \wedge \ldots \wedge p_n \wedge \neg p_{n+1} \wedge \ldots \wedge p_m \rightarrow q$$

Where p_1, ..., p_n, p_{n+1}, ..., p_m and q are predicates.

While retaining much of the expressive power of first order predicate logic, checking the validity of Horn clauses is possible in finite time[8]. PROLOG interpreters use a unification algorithm to answer queries on the data / program that has been entered into them.

3.4 Problems with PROLOG

There are a variety of problems, both philosophical and practical that make PROLOG unsuitable for certain tasks.

- For efficient computation, several non logical operators and ideas are needed. The prime examples of this are the significance of order in the body and the cut operator.
- Semantics of querying are built into the semantic calculus used which leads to a non intuitive blurring between the data and the usage of the data.
- It is relatively easy to create programs that cause the interpreter to enter an infinite loop due to the rule selection algorithm needed.
- PROLOG traditionally only contains one form of negation, negation as failure and thus uses a closed world assumption, i.e. not knowing that a statement is true is the same as knowing it is not true. In many knowledge representation applications this is not appropriate; not knowing whether a cable is live and knowing that it is not are far from the same thing.

Consequently the field of logic programming has been looking towards developing new tools and logic systems that are better suited to efficient and expressive knowledge representation tasks.

3.5 *AnsProlog**

Semi negative logic programs have been characterised in a variety of different ways (see section 3.7.1), with several different sets of semantics. Here we shall use the notation used by Chitta Baral[9]; *AnsProlog**. Throughout this document we consider a language without function symbols and disjunction in the heads of rules, a sub class referred to as $AnsDatalog^{\neg,\perp}$ rather[3] than full *AnsProlog**, see section 8.2 for comments on extending the presented results to *AnsProlog**. The definition is presented in the style of a Backus-Naur Form (BNF) for a context-free (or type 2) language[4]; this is not only to aid comprehension but to emphasis that answer set semantics is not a sub set of classical logic; it is a different logical system.

Definition 3.5.1 *$AnsDatalog^{\neg,\perp}$ programs are expressed in a language $\mathscr{L}$ made up of letters from the following alphabet:*

- ***Object constants**. These are the identifiers for a series of objects; the things we want to talk about in our logic program. Conventionally these are notated as lower case words using underscore instead of spaces, i.e. thing_one, thing_two, box_with_hook.*
- ***Predicate symbols**. These represent a statement about a number of objects. The number of objects the refer to is called their arity. Notated as lower case letters with their arguments in brackets, (e.g.. $in(box\_with\_hook, thing\_one)$ which could then be given the intuitive meaning[10] "thing_one is in box_with_hook"). Non predicated statements ("today is thursday") are simply 0-arity predicates.*
- ***Variables**. These allow predicates to be used to make a statement about a group of objects, they are notated as capital letters; X, Y, Z, ... For example $in(box\_with\_hook, X)$.*
- ***Connectives**[5]. To construct more complex statements from predicates, logical connectives are needed. '$\leftarrow$', '**not**', '$\neg$' and ',' are valid connectives.*
- ***Punctuation**. To clarify the expression of relations, '(', ')' and '.' are used as punctuation.*
- *The symbol $\perp$.*

For the benefit of readers previously unfamiliar with logic programming it may be advantageous to clarify several things at this point. All object constants (often shortened to constants) are treated in the same fashion - there is no concept of type or class. If the program requires a concept of type then it is constructed using predicates to state that a particular object has a given type. There is no handling of the meaning of constants or predicates, one of the biggest cause of 'bugs' in logic programs is confusion and mixing of the meaning of predicates and constants, for example a predicate as simple as $in(A, B)$ could be viewed as set inclusion or statements of the form "A is in B" or "B is in A", mixing which of these it represents within a program will give results other than what was intended.

Terminology is introduced to make referring to particular combinations of letters in a generic language simpler.

Definition 3.5.2

- ***Term** is used to refer to a variable or a constant. If it is a constant it is said to be a ground term.*
- ***Atoms** are of the form $p(t_1, t_2, \ldots, t_n)$ where p is an n-ary predicate and t_i are terms. If all of the terms in an atom are ground then the atom is said to be ground. From the previous definition, if we take in to be a predicate with arity 2 and thing_one and box_with_hook to be constants then $in(X, thing\_one)$ is an atom and $in(box\_with\_hook, thing\_one)$ is a ground atom.*
- *The **Herbrand Base** of a language is the set of all ground terms that can be constructed using the predicates and object constants. These are all of the pieces of information that we can speak about using using language , notated $\mathcal{B}_l$.*

[3] Also referred to as $Datalog^{\mathbf{not},\neg,\perp}$

[4] For readers unfamiliar with the formal descriptions of languages in the context of computing science, this is the system used to describe the syntax of most programming languages. A full formal description of $AnsDatalog^{\neg,\perp}$ is not given as the syntax is fairly minimal

[5] Numerous subsets of *AnsProlog** exist and the convention to use superscript to note what connectives are permitted in the language, $AnsDatalog^{\neg}$ is *AnsDatalog* where connective $\neg$ is allowed. The usage of $*$ is intended to refer to all possible subsets / variations on the language.

- *A **literal** is an atom optionally preceded with the classical negation operator, '$\neg$'. It is ground if the atom is ground.*
- *A **naf-literal** is an atom optionally preceded with the negation as failure operator, '**not**'. It is ground if the atom is ground.*

Literals can be thought of individual statements or indivisible pieces of information. The next step is to use connectives to join them together to describe the links between information.

Definition 3.5.3 *A logic program is made up of a set of rules in language l. Each rule has the following form:*

$$L_0 \leftarrow L_1, \ldots, L_n, \mathbf{not}\ L_{n+1}, \ldots, \mathbf{not}\ L_m.$$

Where L_0 is a literal or $\bot$ and L_i for $i \in [1, m]$ are literals.

The intuitive meaning[6] of this is "if we know $L_1, \ldots, L_n$ and we do not know $L_{n+1}, \ldots, L_m$ then we may conclude L_0". '$\leftarrow$' is used instead of '$\Rightarrow$' and ',' instead of '$\wedge$' are although they are doing a similar thing, their semantics are different. Using the usual logical connectives would serve to confuse matters. For example, contraposition, the equivalence of $a \Rightarrow b$ and

The Herbrand Base of a program P is the Herbrand Base of the language from which the rules in P are formed from. The set containing L_0 is referred to as the *head* of the rule and the set containing $L_1, \ldots, L_n, \mathbf{not} L_{n+1}, \ldots, \mathbf{not} L_m$ is referred to as the *body*. These are notated $H(r)$ and $B(r)$, clearly $H(r) \in \mathcal{B}_P$ and $B(r) \subset \mathcal{B}_P \cup \mathbf{not}\ \mathcal{B}_P$.

Rules are informally broken down into three groups. Rules with an empty body are called *facts* as they state something about the base knowledge in the system. Commonly they are notated without *gets*. Rules with $\bot$ in the head (often notated without a head) are referred to as *constraints* as they describe a circumstance in which a contradiction occurs, they constrain possible solutions. Finally rules with non empty head and body are referred to as *implications* as they represent a situation where new information can be inferred given certain conditions.

In the following is a fragment from a program that implements graph colouring:

$$\begin{aligned} colour\_of(N, C) &\leftarrow node(N), col(C), \mathbf{not}\ another\_colour(N, C). \\ \bot &\leftarrow col(C), link(N, M), colour\_of(N, C), colour\_of(M, C). \\ node(a). \\ node(b). \\ node(c). \\ link(b, c). \\ link(a, X) &\leftarrow node(X). \end{aligned}$$

by convention, constants are notated in lower case and variables in upper case. $node(a)$ is a ground atom and the rule that contains it is a fact. $link(a, X) \leftarrow node(X)$. is an example of an implication which doesn't contain ground atoms. Finally $\bot \leftarrow col(C), link(N, M), colour\_of(N, C), colour\_of(M, C)$. is a constraint (again not containing any ground atoms).

3.6 Answer Set Semantics

There have been several sets of semantics proposed for programs using the syntax outlined above (again, see section 3.7.1 for an example of an alternative system). One of these systems is answer set semantics (originally known as stable model semantics[12], due to the initial use of fix point based characterisations.). The semantics of $AnsDatalog^{\neg,\bot}$ will be developed by considering several functional subsets. Unless otherwise noted all logic programs in this text will be assumed to only contain ground rules (i.e. rule made up from ground literals). The description presented here is based on the description in "Knowledge Representation, Reasoning and Declarative Problem Solving"[9], which provides an accurate and comprehensive summary of the state of answer set semantic research. Readers interested in looking further into the topic are strongly advised to use this as a starting point.

[6]The term 'meaning' has to be used with great care in the context of logic programming[11]. What is presented here is the motivating idea behind the definition, it is presented to help the reader grasp the fundamental concepts rather than as any form of formal explanation.

Grounding is the process of converting a non ground program into a ground program which will have the same answer sets. In all practical cases the set of object constants is finite and so a non ground program can be converted to a ground program by replacing each non ground rule with every possible substitution of each variable. A number of techniques exist to achieve this in the most efficient manner possible (both in terms of space and time) so it will largely speaking be disregarded as a theoretical issue. See section 4.5 for a further discussion of approaches to implementing grounding and handling unground programs.

3.6.1 $AnsDatalog^{-\mathbf{not}}$

This is the simplest sub set of $AnsDatalog^{\neg,\bot}$, the $\neg$ and **not** connectives are not used. Also the head of each rule must contain an atom, not $\bot$. Various names (and alternative semantics) for this subclass of programs have been used, for example definite, Horn logic and positive programs can all be found in older literature.

There are several equivalent characterisations of answer set semantics for $AnsDatalog^{-\mathbf{not}}$, presented here are two that are of particular use when developing algorithms for calculating the semantics of a program. These are the model theoretic and the iterated fix point characterisations (which gave rise to the term stable model semantics). Intuitively the two approaches to working out the semantics of a logic program is to find all sets of atoms which are consistent with all of the rules and to start with the facts and work out what can be concluded by using the rules. From a certain point of view these could be referred to as top down and bottom up approaches. The first is an intuitive description of the model theoretical characterisation, while the second applies to the iterated fixed point characterisation.

3.6.1.1 Model Theoretical Characterisation

A Herbrand Interpretation I of a AnsProlog program P is a subset of the program's Herbrand base ($I \subseteq \mathcal{B}_P$). It is said to satisfy a rule:

$$L_0 \leftarrow L_1, \ldots, L_n.$$

If and only if:

$$\{L_1, \ldots, L_n\} \subseteq I \Rightarrow L_0 \in I$$

If it satisfies all the rules in a program then it is a Herbrand model. It is said to be minimal if there are no other models that are a strict subset of it.

Definition 3.6.1 *The answer set of a program is its minimal Herbrand model*

In the case of $AnsDatalog^{-\mathbf{not}}$ programs, every program has exactly one answer set[9]. When this definition is used in relation to calculating answer sets some additional terminology is used. A rule is said to be *applicable* with respect to a set S if and only if $\{L_1, \ldots, L_n\} \subseteq I$ and is *applied* if it is applicable and $L_0 \in S$. Herbrand models are Herbrand interpretations with respect to which every rule is either applied or not applicable.

3.6.1.2 Iterated Fixed Point Characterisation

The iterated fixed point characterisation defines a function T_P^0 on a program P. It maps from the set of Herbrand Interpretations (notated as $2^{\mathcal{B}_P}$) to itself ($T_P^0 : 2^{\mathcal{B}_P} \rightarrow 2^{\mathcal{B}_P}$) as follows:

$$T_P^0(S) = \{L_0 \in \mathcal{B}_P \mid L_0 \leftarrow L_1 \ldots L_p \in P \text{ and } L_1 \ldots L_p \in S\}$$

where P is the set of rules. This function is referred to as the immediate consequence operator. Intuitively it is adding to the set anything that can be inferred from a rule in P. This can trivially be shown to be monotone and thus over a finite Herbrand base it has a fixed point. This fixed point, notated $lpf(T_P^0)$ is the answer set of P.

3.6.2 *AnsDatalog*

AnsDatalog is a superset of $AnsDatalog^{-\mathbf{not}}$ in which the **not** connective is allowed. This vastly increases the expressive power of the language but makes it significantly more complex. Where as the answer sets of an $AnsDatalog^{-\mathbf{not}}$ program are reasonably intuitive, those of *AnsDatalog* may not be. For example, the answer set of:

$$\begin{aligned} b &\leftarrow a. \\ c &\leftarrow a. \\ d &\leftarrow b, c. \\ f &\leftarrow d, e. a. \end{aligned}$$

is reasonably obvious[7]. However the same cannot be said of Q:

$$\begin{aligned} b &\leftarrow a. \\ c &\leftarrow \mathbf{not}\ d, a. \\ d &\leftarrow \mathbf{not}\ c, b. \\ f &\leftarrow d, e. \\ a &\leftarrow \mathbf{not}\ f. \end{aligned}$$

Introducing **not** creates a problem for either of the previous two characterisation of semantics as the answer set is no longer unique and some of the possible answer sets are nonsensical. This problem is solved by the usage of the Gelfond-Lifschitz reduct (or transformation)[12].

Definition 3.6.2 *The transform takes a set of atoms $S \subset \mathcal{B}_P$ and an AnsProlog program P and reduces it to an $AnsProlog^{-\mathbf{not}}$ program P^S by:*

- *Removing every rule that contains* **not** *p in the body if $p \in S$*
- *Removing all remaining negative literals (i.e.* **not** *q) from the rules*

The answer sets of P are the sets of literals S such that S is the answer set of P^S.

Using the preceding example program, Q, reducts with respect to $\{a, b, c, d\}$ and $\{a, b, d\}$ (respectively $Q^{\{a,b,c,d\}}$ and $Q^{\{a,b,d\}}$) are:

$$\begin{aligned} b &\leftarrow a. \\ f &\leftarrow d, e. \\ a. & \end{aligned}$$

and

$$\begin{aligned} b &\leftarrow a. \\ d &\leftarrow b. \\ f &\leftarrow d, e. \\ a. & \end{aligned}$$

Which have answer sets $\{a, b\}$ and $\{a, b, d\}$, therefore giving $\{a, b, d\}$ as an answer set of Q.

The definition of applicable has to change in this environment. A rule is said to be applicable with respect to a set S if and only if $\{L_1, \ldots, L_n\} \subseteq I$ and $\{L_{n+1}, \ldots, L_m\} \cap I = \emptyset\}$. Models are defined as before, Herbrand interpretations with respect to which every rule is either applied or not applicable. Answer sets are Herbrand models but models are not automatically answer sets, for example

$$\begin{aligned} a &\leftarrow b. \\ b &\leftarrow a. \end{aligned}$$

has a model of $\{a, b\}$ but no answer sets. Models that are not answer sets either contain atoms that are unsupported (i.e. they are not minimal models) or contain circular dependencies, as in the given example. Rules that are not applicable are either rules that would be removed by the reduct or rules that depend on (i.e. contain in their body) atoms which are not in the model.

[7] For those without the time to work through the definitions, it's $\{a, b, c, d\}$.

3.6.3 $AnsDatalog^{\perp}$

A smaller extension that adding the **not** connective, this allows constraints - rules with $\perp$ in the head. The simplest way of handling the semantics for these is to remove any answer sets of the program (minus constraints) with respect to which these are applicable. Formally this is handled by altering the definitions of satisfaction and the immediate consequence operator, this gives exactly the same semantics but is not as conceptually clean, as unlike $AnsDatalog^{-\mathbf{not}}$ programs $AnsDatalog^{-\mathbf{not},\perp}$ programs do not necessarily have an answer set at all.

3.6.4 $AnsDatalog^{\neg,\perp}$

The final extension, allowing $\neg$ in the language is again comparatively trivial. $\neg$ is the connective used to represent classical negation, i.e. a and $\neg a$ cannot both be simultaneously true. This is handled by mapping the $AnsDatalog^{\neg,\perp}$ program to $AnsDatalog^{\perp}$ by replacing each negative literal (i.e. $\neg a$) with a new atom and adding a constrain to stop the new atom and the positive literal both being simultaneously true. In essence classical negation in *AnsProlog* is just 'syntactic sugar'.

Unless explicitly stated it is assumed that this transformation has already been applied to all programs in this document. Although in the formal sense this work applies to $AnsDatalog^{\neg,\perp}$ results and examples will only be produced for $AnsDatalog^{\perp}$ (i.e. programs after the transformation has been made).

3.6.5 Alternative notations for answer sets

Generally in relation to algorithm for computing answer sets, there are a number of other ways of notating answers sets. These are completely equivalent and semantically no different, they just allow for cleaner notation. An answer set $A \subset \mathcal{B}_P$ of program P can be written as $< A, B >$ where $B \subset \mathcal{B}_P$ and $A \cup B = \mathcal{B}_P$ and $A \cap B = \emptyset$. This notation, in which an answer set is referred to as a total interpretation, is used in well founded semantics (see section 3.7.1) and in some computation algorithms. Alternatively an answer set can be written as a subset of the set of all naf-literals ($\mathcal{B}_P \cup \mathbf{not}\ \mathcal{B}_P$) such that $C = A \cup \mathbf{not}\ B$.

3.7 Algorithms

The definition of answer set semantics naturally gives a very basic algorithm for calculating stable models.

- Ground the initial rules
- Iterate through all possible Hebrand Interpretations
  - Reduce the rule set with respect to the interpretation
  - Compute the answer set of the reduced set of rules using the immediate consequence operator
  - Add to the list if the result equals the interpretation the program was reduced by
- Return the list of answer sets

However this algorithm is exponential in the number of atoms in the rule set. It is clear that there are many improvements that could be made to this; facts and constraints can be used to cut the search space and rules that are already in AnsProlog^{-not} form can be resolved before any particular Interpretation is selected.

Most of the existing answer set calculation algorithms use the idea of partial interpretations.

Definition 3.7.1 *A Partial Interpretation of a program P is a pair $< T, F >$ such that $T, F \subset \mathcal{B}_P$ and $T \cap F = \emptyset$. An atom a is true with respect to the partial interpretation $\Leftrightarrow a \in T$, it is false $\Leftrightarrow a \in F$. Correspondingly* **not** *a is true $\Leftrightarrow a \in F$ and false $\Leftrightarrow a \in T$.*

The atoms in T are thought of as being known to be true / in an answer set, while those in F are thought of as being known to be false / not in an answer set. The algorithms extend these (to give an interpretations $< T', F' >$ such that $T \subset T'$ and $F \subset F'$) as much as is possible, if an answer set has not been reached (i.e. $T \cup F \neq \mathcal{B}_P$) then a naf-literal a is chosen and the algorithm branches, taking each possible value of a. Key to computing and extending these interpretations is the concept of the well-founded semantics of a program.

3.7.1 Well-Founded Semantics

Well-founded semantics are an alternative semantic system for $AnsProlog^*$. For reasons that will be discussed later they can be thought of as an approximation of the answer sets of a program. The characterisation here is for $AnsDatalog$ but as with the discussion of answer set semantics it could easily be expanded to $AnsProlog^*$. There are a large number of equivalent characterisations of well founded semantics, for example Van Gelder's alternating fixpoint characterisation [13], argumentation based approaches[14] and 3-valued based semantics[15] the one presented here the fixed point characterisation used by T. Przymusiński[16].

Definition 3.7.2 *Let P be a program, $I =< T, F >$ and $T', F' \subset \mathcal{B}_P$ then define:*

$$\begin{aligned}
\mathscr{T}_I(T') &= \{p \in \mathcal{B}_P \mid p \notin T \wedge \exists r \in P \\
&\quad s.t.\ p = H(r) \wedge \forall b \in B(r)\ (b \text{ is true with respect to } I) \vee (b \in T')\} \\
\mathscr{F}_I(F') &= \{p \in \mathcal{B}_P \mid p \notin T \wedge \forall r \in P \\
&\quad s.t.\ p = H(r) \wedge \exists b \in B(r)\ (b \text{ is false with respect to } I) \vee (b \in F')\}
\end{aligned}$$

Intuitively $a \in \mathscr{T}_I(T')$ if there is a rule P with a as it's head and that is applicable with respect to I and T', likewise $b \in \mathscr{F}_I(F')$ if there are no applicable rules (with respect to I and F') with b as their head. $\mathscr{T}_I(T')$ is what can be conclude / is known to be true from I and T' and $\mathscr{F}_I(F')$ is corresponding what is definitively known to be not true. These functions can be shown to be monotonic and as their co-domain is finite they have fixed points. T_I is the fixed point of $\mathscr{T}_I$ (starting from $\emptyset$ and with respect to a partial interpretation I), likewise F_I is the fixed point of $\mathscr{F}_I$. These are essentially the most that can be concluded from the partial interpretation I. These are then used in the next definition:

Definition 3.7.3 *$\mathscr{I}$ is a function from the set of partial interpretations to itself. For $I =< T, F >$ it is given as:*

$$\mathscr{I}(I) = I \cup < T_I, F_I >$$

Essentially this takes a partial interpretation and adds to it everything that can be concluded form it. This is intuitively monotonic and as again it's co-domain is finite, it has a fixed point. The fixed point of $\mathscr{I}$ starting from $< \emptyset, \emptyset >$ is the well-founded semantic of a given program [16].

Critically the[8] well-founded semantic of a program is sound with respect to the answer sets of that program. If $W =< T, F >$ is the well founded semantic of a program and S be the set of answer sets. If $a \in T$ then $\forall A \in Sa \in A$ and if $b \in F$ then $\forall A \in Sa \notin A$. Alternatively using the well founded semantic style characterisation of an answer set, $\forall A =< T_A, F_A > \in S(T \subset T_A) \wedge (F \subset F_A)$. Thus well-founded semantics serve as an approximation to answer set semantics.

3.7.2 Branch and Bound Algorithm

This is a simple example of the a branch and bound example, presented in "Knowledge Representation, Reasoning and Declarative Problem Solving"[9] many other answer set calculation algorithms follow this pattern, just using slightly different steps for extending the partial interpretations (such as Fitting's operator[17] in smodels) and (different) heuristics for selecting the atom to add.

```
/* Global variables */
P     // set of rules
new   // set of answer sets

/* Local variables */
list              // A list of triples, each of which is a
                  //  set of rules and two sets of atoms
                  //  (a partial interpretation)
currentTriple     // A triple as described above
selectedAtom      // A temporary atom
tmpTriple         // A temporary triple
```

[8] The as well founded semantics are unique

```
tmpInterpretation // A temporary partila interpretation

/* Operators */
U                 // Union
^                 // Intersection

/* Functions */
reduct(P,atom)
                  // Reduces the rule set P by removing
                  //  all rules from P is they contain
                  //  'not atom' and removing 'atom'
                  //   from the bodies of the other rules

extendPartialInterpretation(T,F)
                  // Extends the partial implementation as
                  //  described above

heuristic(T,F)
                 // Selects an atom in the Herbrand base
                 //  but not in T or F



/* Start algorithm */

/* Start with one entry to the list */
list.append(triple(P,emptySet,emptySet));

while (list.size > 0) {

    /* Take one triple off the list */
    currentTriple = list.removeFirstElement();

    /* See if there is a smaller answer set */
    /* Happens if an assumed atom is unfounded */
    if ( ! (exists T in new s.t.
                        T < currentTriple.true) ) {

        /* Select an atom  that is not already *
         * accounted for                       */
        selectedAtom = heuristic(currentTriple.true,
                                 currentTriple.false);

        /* Assume that it is true */
        tmpInterpretation =
          extendPartialInterpretation(currentTriple.true,
                                      currentTriple.false);

        tmpTriple =
           (reduce(currentTriple.rules,selectedAtom),
            currentTriple.true U tmpInterpretation.true
                               U selectedAtom,
            currentTriple.false U tmpInterpretation.false);

        /* See if an answer set has been created */
        if ( ! (exists T in new s.t.
                           T < tmpTriple.true) ) {

            if ( tmpTriple.true ^
                  tmpTriple.false == emptyset) {

                if ( tmpTriple.true U
                       tmpTriple.false == HerbrandBase(P)) {
                    /* Have created an answer set */
                    new.append(tmpTriple.true);
```

```
            } else {
                /* More work needed */
                list.append(tmpTriple);
            }

        } else {
            /* Inconsistant set - discard */
        }
    }

    /* Assume that it is false */
    tmpInterpretation =
      extendPartialInterpretation(currentTriple.true,
                                  currentTriple.false);

    tmpTriple =
       (reduce(currentTriple.rules,not selectedAtom),
        currentTriple.true U tmpInterpretation.true,
        currentTriple.false U tmpInterpretation.false
                            U selectedAtom);

    /* See if an answer set has been created */
    if ( ! (exists T in new s.t.
                        T < tmpTriple.true) ) {

        if ( tmpTriple.true ^
              tmpTriple.false == emptyset) {

            if ( tmpTriple.true U
                  tmpTriple.false == HerbrandBase(P)) {
                /* Have created an answer set */
                new.append(tmpTriple.true);
            } else {
                /* More work needed */
                list.append(tmpTriple);
            }

        } else {
            /* Inconsistant set - discard */
        }
    }

  }
}
/* End algorithm */
```

3.8 Tools

There are a variety of tools available for calculating the answer sets of various sub sets of $AnsProlog^*$. Most of the major centers of research into Answer Set Semantics have their own, 'in house' solver, however two solvers, smodels[18] and DLV[19] are cited the most frequently.

Smodels was developed at the Helsinki University of Technology. It is composed of two main components, lparse[20] and smodels[21]. The program smodels is just the an answer set solver - it only works on ground programs, lparse provides a front end with grounding and a variety of syntactic extension to make programming easier. Smodels also implements cardinality and weight constraints (as well as appropriate optimisations). It is distributed in source form[18] (C++) and licensed under the GPL[22].

DLV is a disjunctive datalog system. Unlike smodels it doesn't support functions but does support AnsProlog^{or}, i.e. disjunctions in the head of a rule. It is developed by a team based at the University of Calabria, Italy. It is distributed in binary form[19] without licence.

Also of note is the noMoRe[23] created by the University of Potsdam. Rather than using well founded semantics and a branch and bound style of approach, it creates a dependency graph and then uses graph colouring techniques to calculate the answer sets of the input program. lparse is used to handle grounding.

Another interesting case is cmodels[24], it uses an alternative logic formalism, SAT solvers to compute the answer sets semantics of the input program. Essentially this is the opposite of the derivate logic systems described below.

At the time of writing the author is not aware of the existence of any solvers that can change the answer sets of a program as the program is changed; what this project seeks to do.

3.9 Applications

Although *AnsProlog** is still a very new research field and many questions remain unanswered, there have already been a wide variety of applications.

3.9.1 Derivative Logic Systems

A wide variety[25] of logic systems have been implemented using answer set solvers as a basis or a back end. These include:

- OCLP - Ordered Choice Logic Programming. This provides and alternative semantic for multiple head atoms as an exclusive choice and a system for expressing preference between rules.It works by mapping OCLP rule systems into $AnsProlog^{\perp}$ rules and using a conventional answer set solver.
- Nested Programs. A generalisation of *AnsProlog** still under a lot of development. It allows the heads and bodies of the rule to contain arbitrary boolean formulae. Example solvers include nlp [28] and noMoRe [23].
- Finitary programs. These support function in a symbolic way and recursion. This extension is mainly motivated by the need of representing and reasoning about recursive data structures (such as XML documents) in a natural and uniform way, without resorting to external preprocessors. Traditionally recursion has been a problematic area as it tends to give logic systems which are undecidable.

3.9.2 Direct Applications

The most quoted and highest profile use of answer set semantics is in developing a decision support system for the Reaction Control System (propulsion and manoeuvring) of the US Space Shuttle[29]. The system is computer controlled on launch and landing and under the control of the astronauts. There are well established procedures for performing certain manoeuvres in space, even in the event of failure of some of the components of the system. However due to the size and complexity of the RCS, it is simply not practical to devise procedures and train for all possible circumstances. The decision support system allows the outcome of a suggested plan to be tested but also can create a detailed plan of how to achieve a certain manoeuvre given certain conditions (i.e. failure of electrical components, leaks from piping, etc.).

Prototype applications in agent systems[30], data mining[5] and semantic web knowledge extraction[6] have also been developed and are envisaged to become prominent technologies as the areas around them move from theoretical to applied research.

Chapter 4

Theory

In this chapter we present the theoretical background and algorithms for incremental computation of answer sets. The main focus is on handling the addition of one, grounded rule, however results to cover subtraction and unground rules are also presented.

4.1 Background and Motivation

Very few of the potential applications of answer set programming are static. For example a diagnostic system will not only be used once on any given problem. An initial computation will give a variety of potential scenarios, additional test can then add to the knowledge base and determine the exact status of the system. Almost all applications of answer set semantics involve computing the answer sets of a sequence of similar rule sets. The current solution is to simply run each rule set through a traditional answer set solver independently. However this approach takes time, making interactive applications with larger rule sets difficult and cumbersome to use.

For example, given the following program:

$$\begin{aligned} a &\leftarrow b. \\ c &\leftarrow \textbf{not } d, a. \\ d &\leftarrow \textbf{not } c, a. \\ b. & \\ e &\leftarrow d. \end{aligned}$$

which has answer sets $\{a, b, c\}$ and $\{a, b, d, e\}$ there are some rules that can be added and removed with minimal effort. For example, adding $e \leftarrow b.$ will only add e to every answer set that doesn't already contain it as e is not used in the body of any rule. Likewise removing $e \leftarrow d.$ will have a minimal effect. Adding $\leftarrow e, d.$ top the initial program would simply remove the second answer set. However removing the first rule, $a \leftarrow b$, would cause major changes as the exclusive choice between c and d will not be made.

It is important to note that the presented algorithms in no way changes the syntax or the semantics of the programs. The answer sets of a new rule set given by the incremental algorithm is exactly the same as the one given by a traditional answer set solver. The algorithms also do not attempt to produce a solution method of lower theoretical complexity; in the worst case scenario all answer sets have to be recomputed.

4.2 Preliminary Results

In this section some background results and notations are presented. These are presented separately to increase both the clarity of the following algorithms and the proof of the results.

4.2.1 Self Referential Rules

A rule is self referential if it directly refers to itself, i.e. the head of the rule appears in it's body. For example $a \leftarrow a, b.$ or $b \leftarrow \textbf{not } b, c.$ It is formalised as:

Definition 4.2.1 *A rule r is positively self referential if $H(r) \in B(r)$. It is negatively self referential if* **not** $H(r) \in B(r)$.

Self referential rules pose something of a problem when combined with ideas of the implication and dependency developed later. In the absence of self referential rules deciding whether a rule applies in a given context and how it changes the context can be handled independently, making the computation of answer sets much simpler. Thus they must be removed before the start of the algorithm. Fortunately this can be done with a simple mapping.

Definition 4.2.2 *Let P be a program and rsr be a mapping from the power set of the set of rules*[1] *to itself defined as:*

$$\begin{aligned} rsr(P) &= \{r \in P | r \text{ is non self referential}\} \cup \\ &\quad \{\bot \leftarrow B(r). | \forall r \in P \cdot \text{that are negatively self referential}\} \end{aligned}$$

This mapping is essentially the intuitive treatment of self referential rules; positive rules are removed as they do not add to the information known at any point (the head of the rule is required to be known for the rule to be applicable which then gives the head of the rule) and negative self referential rules become constraints (if the rule is applicable then the head cannot be true, however if the rule is applicable then the head is true).

Proposition 4.2.1 *Let P be a program and $A \subset \mathcal{B}_P$, then:*

$$A \text{ is an answer set of } P \Leftrightarrow A \text{ is an answer set of } rsr(P)$$

Proof 4.2.1 *This proof considers removing firstly one positively self referential rule and then one negatively self referential rule, the result then follows from inducting this principle.*

Let Π be a program without self referential rules and r a positively self referential rule. If A is an answer set of $\Pi \cup \{r\}$, consider Π^A (the reduct of Π with respect to A) and $Q = (\Pi \cup \{r\})^A$. If $r \notin Q$ then $Q = \Pi^A$ and thus A is trivially an answer set of Π^A. If $r \in Q$ then $lpf(T^0_{\Pi^A}) = lpf(T^0_Q)$ as using r in the immediate consequence operator doesn't alter the output set (r is only used when $B(r) \subset S$, then $H(r) \in S$ trivially), thus $A = lpf(T^0_{\Pi^A})$ is an answer set of Q. This means A is an answer set of Π, i.e. applying rsr does not remove any answer sets.

Conversely if B is an answer set of Π, consider Π^B and $T = (\Pi \cup \{r\})^B$. If $r \notin T$ then $T = \Pi^B$ and B is an answer set of $\Pi \cup \{r\}$ as before. If $r \in T$ then $lpf(T^0_{\Pi^B}) = lpf(T^0_T)$ by the previous reasoning, thus applying rsr does not allow any new answer sets to become possible.

To handle the case of negatively self referential rules, again let Π be a program without self referential rules and r a negatively self referential rule. If A is an answer set $Q = (\Pi \cup \{r\})^A$ and $S = \Pi \cup \leftarrow B(r)$. If $r \notin Q^A$ then $S^A = Q^A$ as the constraint will be removed as well (as it has the same body), thus A is an answer set of S. If $r \in Q^A$ then $H(r) \notin A$, thus r is never used by the immediate consequence operator (as it's body is never contained by a previous step, or $H(r) \in lpf(T^0_{Q^A}) = A$) and $lpf(T^0_{Q^A}) = lpf(T^0_{S^A})$ and A is an answer set of S. Thus applying rsr to remove a negatively self referential rule does not remove any answer sets.

If B is an answer set of S, we must have $B(r) \not\subset B$. If $H(r) \in B$ then after performing the reduct both programs give the same rule set, and thus B is an answer set of S. Alternatively we can conclude (as before) that as $B(r) \not\subset of B$, r is not used by the immediate consequence operator, thus giving B as an answer set of S. This shows that no new answer sets are added.

By inducting this principle the result follows.

4.2.2 Rule Set Augmentation and Conditional Answer Sets

At several points in the following algorithms there is a requirement to search for answer sets of a program that fulfil certain conditions. These conditions, in the form of simple assertions about which atoms are contained in the answer set, are formed from the context of the algorithm or from assertions that can be made about the behaviour of answer sets. For example handling the addition of the rule $a \leftarrow b, c.$ could be handled as search for answer sets which contain $\{a, b, c\}$ (to generate all of the answer sets with respect to which the rule is applicable) and inheriting all of the rule sets of the old program which do not contain $\{a, b, c\}$ (all of the answer sets with respect to which the rule is not applicable). This section presents results which allow the computation of such conditional answer sets in a more efficient way.

Proposition 4.2.2 *Let S be an answer set of program P and $A = \{a_1, \ldots, a_n, \textbf{not } a_{n+1}, \ldots, \textbf{not } a_m\}$ a set of naf-literals*[2] *such that $A \subset S$. Then S is also an*

[1] Technically this should be the power set of the set of rules created from a given language. However in most applications of this theory, the language is considered a product of a set of rules; rather than the other way around as it is defined formally

[2] Here the answer set is characterised as a (non contradictory) subset of the set of all naf-literals

answer set of the augmented program $P * A$:

$$P * A = P \cup \{a_1 \leftarrow .\ , \ldots,\ a_n \leftarrow .\} \cup \{\bot \leftarrow a_{n+1}.\ , \ldots,\ \bot \leftarrow a_m.\}$$

Proof 4.2.2 *It is clear that by applying the reduct with respect to S the following is derived:*

$$(P * A)^S = P^S \cup \{a_1 \leftarrow .\ , \ldots,\ a_n \leftarrow .\} \cup \{\bot \leftarrow a_{n+1}.\ , \ldots,\ \bot \leftarrow a_m.\}$$

As $\{\mathbf{not}\ a_{n+1}, \ldots, \mathbf{not}\ a_m\} \subset S$ then $\{a_{n+1}, \ldots, a_m\} \cap S = \emptyset$ (or S would be contradictory having $\mathbf{not}\ a_k \in S$ and $a_k \in S$ for some $k \in [n+1, m]$), so given $\{a_1, \ldots, a_n\} \subset S$, we may conclude:

$$lpf(T^0_{(P*A)^S}) = lpf(T^0_{P^S})$$

And thus S is an answer set of $P * A$.

Thus by computing the answer sets of the augmented program we form a list of models / interpretations which contains all of the answer sets of the original program which satisfy the given criteria. It is important to note that this set contains all such answer sets but is not equal to it. For example if this technique is used to search for the answer sets of the following program

$$a \leftarrow b.$$
$$b \leftarrow a.$$

which contain $\{a, b\}$ gives[3] $\{\{a, b\}\}$ while the initial program clearly has no answer sets at all.

This technique is of interest because using most branch and bound based computation algorithms, computing the answer sets of the augmented program is quicker[4] than computing the answer set of the original program. The set of atoms A can be used as the starting set rather than starting from $\emptyset$. Given facts and constraints with one body element are accounted for on the first iteration of the algorithm the behaviour will be the same. Also testing the list of candidate answer sets is reasonably computationally efficient. This can be done by performing the reduct and checking the support of each of the atoms in the starting set A. Thus giving an efficient method of computing answer sets that fit simple criteria using existing solving algorithms.

4.2.3 Usage and Support

A couple of conditions are used repeatedly in the following definitions and proofs and are formalised here for simplicity.

Definition 4.2.3 *Let P be a set of rules and $a \in \mathcal{B}_P$ then:*

$$\begin{aligned} support(a, P) &= \{r \in P | H(r) = a\} \\ usage(a, P) &= \{r \in P | a \in B(r)\ \vee\ \mathbf{not}\ a \in B(r)\} \end{aligned}$$

Intuitively the support of an atom in a given set of rule is the set of rules which could support it within a given answer set. Usage is the number of rules in which it appears. They can be used to easily infer some basic facts; for example if $support(a, P) = \emptyset$ then a will not appear in any answer set.

Both of these can easily be maintained as summary information in the implementation of an answer set solver. Doing so can provide considerable speed increases in the implementation of the following algorithms.

4.2.4 Implications

Unfortunately adding rules to a rule set is not as simple as just adding the head atom in answer sets where the rule is applicable. Adding atoms to an answer set may in turn make or stop other rules being applicable and thus require other atoms to be added or removed. These are the implications of adding / removing the original atom. The positive implications are the atoms that have to be added and the negative implications are the atoms that need to be removed.

[3] Calculating the answer sets of a program formally gives a set of answer sets, hence the slightly bazaar notation

[4] In real terms. Obviously the theoretical complexity is the same.

Definition 4.2.4 *Let P be a program, $A \subset \mathcal{B}_P$ and $a \in \mathcal{B}_P \setminus A$ is the atom to be added into A. Let $P_{A,+} \subset P$ be the set of rules that are applicable with respect to A and $P_{A,-} = P \setminus P_{A,+}$ (The complementary set of rules that are not applicable).*

The positive implications I^+ and negative implications I^- are defined as:

$$\begin{aligned} I(+,P,A,+a) &= \{H(r) | \exists r \in usage(a,P) \cdot r \in P_{A,-} \ \wedge \ r \in P_{A\cup\{a\},+}\} \\ I(-,P,A,+a) &= \{H(r) | \exists r \in usage(a,P) \cdot r \in P_{A,+} \ \wedge \ r \in P_{A\cup\{a\},-}\} \end{aligned}$$

Intuitively these are the set of heads of any rule that becomes applicable by adding a and anything that can only be concluded by a rule depending on not a.

Correspondingly the implications of removing an atom are defined as:

Definition 4.2.5 *Let P be a program, $A \subset \mathcal{B}_P$ and a ($\in A$ is the atom to be removed. $P^{A,+}$ and $P^{A,-}$ are defined as before.*

The positive implications I^+ and negative implications I^- are defined as:

$$\begin{aligned} I(+,P,A,-a) &= \{H(r) | \exists r \in usage(a,P) \cdot r \in P_{A,-} \ \wedge \ r \in P_{A-\{a\},+}\} \\ I(-,P,A,-a) &= \{H(r) | \exists r \in usage(a,P) \cdot r \in P_{A,+} \ \wedge \ r \in P_{A-\{a\},-}\} \end{aligned}$$

These are the heads of any rule that become applicable by removing a and any atom that can only be got by a rule depending on a.

For compactness the following notation is introduced:

$$\begin{aligned} I(P,A,+a) &:= (\ I(+,P,A,+a), I(-,P,A,+a)\ ) \\ I(P,A,-a) &:= (\ I(+,P,A,-a), I(-,P,A,-a)\ ) \end{aligned}$$

Handling implications is a non trivial problem as each one can give rise to further implications and they may also be linked so order of resolution can matter. For example given the program P :

$$\begin{aligned} a &\leftarrow c, \textbf{not}\ b. \\ b &\leftarrow c, \textbf{not}\ a. \\ c &\leftarrow d. \end{aligned}$$

Which has only one answer set $A = \emptyset$, adding the rule $d \leftarrow$ gives the following sequence of implications:

$$\begin{aligned} I(P,\emptyset,+d) &= (\{c\},\emptyset) \\ I(P,\{d\},+c) &= (\{a,b\},\emptyset) \end{aligned}$$

Adding d leads to a further implication, c must be added. However adding c leads to a more complex situation. Adding a will stop b being an implication and vice versa. The final answer sets of $P \cup \{d \leftarrow\}$ are $\{a,c,d\}$ and $\{b,c,d\}$. Implications are the main complexity in incremental algorithms, handling implications is mostly solving the incremental problem.

4.2.5 Dependency Graphs

Dependency graphs showing the relation between rules are a common tool in answer set semantic research. Past uses have included the theoretical basis for stratified logic programs and some other functional[5] subclasses of $AnsProlog^*$ and as the basis for computing answer sets[23]. They have been characterised in a variety of ways with different degrees and displays of aggregation of information[6], thus we give a definition of the exact type required here rather than referencing a standard definition.

[5] As opposed to syntactic subclasses such as $AnsProlog^{\neg,\perp}$

[6] The link between rules, heads and bodies is essentially three non orthogonal directions of information, graphs are essentially a projection of this information onto a two dimensional paradigm, hence the number of fundamentally equivalent ways of expressing the relation. The choice between them is based entirely on which way the information is to be accessed.

Definition 4.2.6 *Let P be a program, it's dependency graph $D = (N, L)$ where N is a set of nodes and $L \subset N \times N \times P$ is a set of directed links annotated with rules, is constructed as follows:*

$$\begin{aligned} N &= \{n_a | \forall a \in \mathcal{B}_P\} \\ L &= \{(n_a, n_b, r) | \forall r \in usage(a, P) \cap support(b, P)\} \end{aligned}$$

Intuitively a link is created from a to b if there is a rule that has a or **not** a in the body and b in the head. Nodes are annotated with atoms and atoms will be used interchangeably with 'the node representing the atom'. This means that for every rule r there will be $|B(r)|$ links to $H(r)$. As all self referential rules have been removed using the mapping described in section 4.2.1 no node will link directly to itself.

For these to be of any use, there have to be ways of extracting information from them. Thus a definition of reachability is presented

Definition 4.2.7 *Let $D = (N, L)$ be the dependency graph of program P and let $a \in \mathcal{B}_P$. The reachability set after n steps from atom a is notated as $R^n(a)$ and defined as*

$$\begin{aligned} R^1(a) &= \{b \in \mathcal{B}_P | \exists (n_a, n_b, r) \in L\} \\ R^n(a) &= \{b \in R^1(c) | c \in R^{n-1}(a)\} \cup R^{n-1}(a) \end{aligned}$$

This is the obvious definition, the set of nodes which are reachable after travelling along n links. The following propositions give basic properties of the reachability function and link it to the concept of implication.

Proposition 4.2.3 *Let P be a program, D its dependency graph and a and atom then there exists n_a s.t. $R^{n_a}(a) = R^{n_a+1}(a)$*

Proof 4.2.3 *$R^n(a)$ is trivially monotonic, it's co-domain ($2^{\mathcal{B}_P}$) is finite, thus there exists a fixed point as described.*

For clarity some simple notation is introduced. In the context of the preceding proposition $R^{n_a}(a)$ is notated as $R^\omega(a)$. If $b \in R^1(a)$, b is said to be directly reachable from a, while if $b \in R^\omega(a)$ it is just said to be reachable from a.

Proposition 4.2.4 *Let P be a program, $D = (N, L)$ it's dependency graph and $a \in \mathcal{B}_P$, $A \subset \mathcal{B}_P$ then:*

$$\begin{aligned} I(+, P, A, +a) &\subset R^1(a) \\ I(+, P, A, -a) &\subset R^1(a) \\ I(-, P, A, +a) &\subset R^1(a) \\ I(-, P, A, -a) &\subset R^1(a) \end{aligned}$$

Proof 4.2.4 *This is proved for $I(+, P, A, +a)$, the proof for other implications is entirely analogous.*

$$\begin{aligned} b \in I(+, P, A, +a) &\Rightarrow \exists r \in usage(a, P) \cdot b = h(r) \\ &\Rightarrow \exists (n_a, n_b, r) \in L \\ &\Rightarrow b \in R^{\{1\}}(a) \end{aligned}$$

Thus all of the implications of altering atom a and all of their knock on implications will be contained within R^ω, giving a technique for bounding the possible changes caused by adding a rule to a program. To use this and to infer that atoms cannot be altered and thus their status propagated to new answer sets results that relate the concept of reachability to the answer sets of the corresponding program are needed.

Definition 4.2.8 *Let P be a program and $a, b \in \mathcal{B}_P$. Then a may effect the status of b in an answer set $\Leftrightarrow$ there exists a chain of rules $(r_1, r_2, \ldots, r_n)$ with $r_1, \ldots, r_n \in P$ such that $(a \in B(r_1)) \vee (\textbf{not } a \in B(r_1))$, $(H(r_n) \in B(r_{n+1})) \vee (\textbf{not } H(r_n) \in B(r_{n+1}))$ and $H(r_n) = b$.*

Although complex to write down this definition is conceptual simple. If there are rules $(r_1, \ldots, r_n)$ such that a can effect whether r_1 is applicable, r_n can effect whether b appears in an answer set and each rule can influence the next then it is fair to say that a can influence whether b is in an answer set. After applying the reduct, this chain may allow the immediate consequence operator to conclude b if the status of a is known. This allows the concepts of the dependency graph (and thus implications) to be related to answer sets.

Proposition 4.2.5 *Let P be a program and $a, b \in \mathcal{B}_P$.*

$$b \in R^\omega(a) \quad \Rightarrow \quad a \textit{ may effect the status of } b$$

Proof 4.2.5 *As $b \in R^\omega(a)$ there is at least one path from a to b in the dependency graph of P. Form $(r_1, \ldots, r_n)$ such that r_x is the rule that annotates the link of the graph used at step x of the path. The 'chaining' dependencies of the rules are satisfied trivially by the definition of the dependency graph. Clearly $(a \in B(r_1)) \vee (\textbf{not } a \in B(r_1))$ as r_1 is the first step in a path from a, likewise $H(r_n) = b$ is trivial as r_n is the last step in a path to b.*

Perhaps more usefully, the 'opposite' can be concluded

Theorem 4.2.1 *Let P be a program and $a, b \in \mathcal{B}_P$.*

$$b \notin R^\omega(a) \quad \Rightarrow \quad \neg(a \textit{ may effect the status of } b)$$

Proof 4.2.6 *This is proved by contraposition. If a may effect the status of b then there exists a chain $(r_1, \ldots, r_n)$. By the definition of this chain $H(r_1) \in R^1(a)$, $H(r_j) \in R^1(H(r_{j-1}))$ and $b \in R^1(H(r_{n-1}))$. Thus $b \in R^{n-1}(a)$ which implies $b \in R^\omega$, from which the result follows.*

Thus there exists a link between the maximum implications of a change and the reachability set from the changed atom. Also shown is the equivalence of being reachable and being able to effect another atoms existence in an answer set. This allows the conclusion in phase two (see section 4.3.2), that if an atom cannot be reached by the changed atom then it will have the same status in the new answer set that it had in the original.

4.3 Addition

The algorithm for handling adding rules is divided into three phases. The first section removes all cases where the new rule can be easily shown to have no effect, or it's effect is trivial. After this phase, if there are answer sets which require non trivial changes, a traditional answer set solver can be used or the second section can be used to further reduce the program. It is possible that the first phase handles all answer sets in which case the algorithm finishes. The second section of the algorithm works by 'bounding' the changes caused by adding the rule, if it can be shown that a particular conclusion is not effected by the addition of the new rule then it will be the same in the new answer set as it was in the old one. Again if there are still sets that need to be handled after this phase is completed the program can be passed to a traditional answer set calculation algorithm (with some extra assertions on what will not be changed) or the final phase can be used to complete the answer sets. Again it is possible that phase two handles all cases and there is no more work to be done. The third section of the algorithm is a modified version of the smodels[21] algorithm which dynamically cuts out sections of the program if it is clear that they will not be affected by the changes.

4.3.1 Phase One

If the answers sets of a program are viewed in the model theoretical characterisation then it is clear that adding a rule r to an answer set S will not change anything if r is either applied or not applicable with respect to S. In these cases the rule will not alter the eventual conclusion of the direct consequence operator or be removed by the reduct or contain unsupported atoms respectively. Only if the rule is applicable but not applied does it stop A being an answer set, i.e. cause a change. Phase one of the algorithm reduces the problem to cases where r is applicable but not applied and handles some edge cases (i.e. the introduction of new atoms to the system).

4.3.1.1 Description

The first and most important distinction that has to be made is whether the existing program has any answer sets or not. If it doesn't in some sense there is no information to work with. There are a few criteria that can be applied; the rule must allows the alteration of existing information (i.e. it is not a constraint and the head is used in the body of another rule) and all of the positive atoms in the body have non empty support (this will remove positive dependency on atoms that are new to the system). Apart from these is nothing that can be done to resolve cases without completely recomputing the answer sets.

On the other hand if there are answer sets already the algorithm has something to work with. Constraints are handled first as the can only remove answer sets. If the rule being added is a constraint it is simply a case of removing all answer sets with respect to which it is applicable. Handling other types of rules is more complex as they may lead to implications. First the list of answer sets is split into three lists on the basis of the status of the rule with respect to the answer set. One list contains all of the answer sets with respect to which the new rule is not applicable, one for applied and one for applicable but not applied. These are labelled A, B and C respectively. As previously noted answer sets in lists A and B are unchanged and become answer sets of the new program. Thus if list C is empty then the first part of the addition is complete. If B is non empty we already have a list of the candidates that every answer set in C will become, thus answer sets in C are removed and again the first part of this phase is complete. The final case is when both B and C are non-empty in which case the list C and a reference to the rule being added must be passed to phase two of the algorithm to detect and handle any possible implications rising from adding the head of the rule to the answer sets.

Finally if the body of the rule contains any negative atoms that appear in any of the old answer sets it is possible that adding this rule during the computation of that answer set could have created a scenario in which this rule is applicable. Thus a search for answer sets in which the new rule is applied must be made. Without this it is impossible to add exclusive choices to the system. For example the following program:

$$a \leftarrow \textbf{not}\ b.$$
$$b \leftarrow \textbf{not}\ a.$$

has two answer sets $\{a\}$ and $\{b\}$. Adding the first rule to an empty rule set creates[7] a single answer set $\{a\}$, but without this additional search the second answer set will not be created when the second rule is added.

The analysis performed at this stage can be seen as determining how relevant the rule being added is. This is further considered in section 6.2.2.

4.3.1.2 Pseudo Code

As well as the written description of each algorithm, pseudo code has been included to help make the flow of control within each phase of the algorithm more readily apparent. The code uses similar syntax to C++ and may also be a useful reference for understanding the implementation described in chapter 5.

In the presented pseudo code, each atom is regarded as a positive integer value. Answer sets are sets of integers, and an atom is in an answer set if its integer is in the set and not if minus its integer belongs to the set. Thus the 'value' of the atom in the set is either +1 or -1. This representation is useful in the later stages of the algorithm where answer sets are constructed, thus as well as knowing that an given atom appears (or does not appear) in the answer set, there are also steps where the status of the atom is unknown - in this case neither atom or -atom are in the set.

```
/* Global variables */
P     // set of rules
old   // set of answer sets of P

/* Arguments */
r     // rule to be added

/* Local Variables */
A     // potential new answer sets where r is not
      //  applicable
```

[7] An empty rule set is assumed to have a single, empty answer set

```
B     // potential new answer sets where r is applied
C     // potential new answer sets where r is
      //  applicable but not applied

/* Return values */
new   // set of answer sets of P U {r}

/* Functions */
search(atoms,rules)
      // A function that uses a conventional solver
      // to find answer sets of rules given atoms



/* Start of phase one */

if (old.size == 0) {
    /* Check to see if the rule can alter *
     * the existing information           */

    if ((r.type != CONSTRAINT) &&
         (usage(r.head,P).size > 0)) {

        /* Check to see if it is possible to *
         * use this rule                     */

        for b in r.body {
            if ((b.positive == true) &&
                 (support(b,P).size == 0)) {
                return;
            }
        }

        /* Have to look for possible solutions */
        new = search(r.head U r.body, P U {r});

        return;
    } else {
        /* Cannot change the contradictory nature of *
         * the rule set                              */
        return;
    }

} else {
    /* There are answer sets */

    /* Handle constraints */
    if (r.type == CONSTRAINT) {
        for s in old {
            if (r.applicable(s) == true) {
                /* Drop this answer set */
            } else {
                new.add(s);
            }
        }
        return;
    }


    /* Work out the cases involved */

    for s in old {
        if (r.applicable(s) == true) {
            if (r.applied(s) == true) {
                B.add(s);
```

```
            } else {
                C.add(s);
            }
        } else {
            A.add(s);
        }
    }

    /* If r is not applicable or applied then the *
     * answer sets are maintained                 */
    new = A U B;

    /* Handle implications */
    if (C.size != 0) {
        new = new U phaseTwo(C,r,+r.head);
    }

    /* Search for any extra answer sets */
    for b in r.body {
        if ((b.positive == false) &&
             (support(b,P).size > 0)) {
            search(r.head U r.body, P U {r});
            break;
        }
    }

    return;
}

/* End of phase one */
```

4.3.1.3 Proofs

Phase one of the algorithm is shown to be finite, complete and correct given the corresponding results for phase two and phase three of the algorithm (which are proved later). The search function is also required to be finite, complete and correct but by the results in section 4.2.2 and the proofs of standard answer solver algorithms these results may be taken to be true.

Theorem 4.3.1 *Phase one terminates in finite time.*

Proof 4.3.1 *The only looping statements are over finite sets that are not increased by the body of the loop, thus as phase two and search are finite, phase one is trivially finite.*

Theorem 4.3.2 *Phase one is correct; everything it produces is an answer set.*

Proof 4.3.2 *The proof of correctness of this phase of the algorithm is broken into two sections, corresponding to the case of some and no answer sets of the original program.*

In the case where there were not initially any answer sets, answer sets are only produced by the search function. Given the search function is taken to be correct, in this case all results produced will be answer sets.

In the case where there are existing answer sets there is slightly more to prove. The handling of constraints can be trivially seen to be correct. If the rule being added is not a constraint then how the sets are produced must be considered. If the new rule is not applicable with respect to an answer set of the existing program then either it is removed by the reduct or it is not used by the immediate consequence operator, either way the answer set is clearly an answer set of the new program. Likewise if the rule is applied with respect an old answer set, it will be in the rule set after the reduct but will not effect the fix point of the immediate consequence operator (both the head and the body of the rule will be reached by the operator as they were with the old program), so it is also an answer set of the new program. Given the previous comments on the search function and under the assumption (which will be proved later) that the second phase of the algorithm is correct, this phase of the algorithm is correct.

Theorem 4.3.3 *Phase one is complete; it produces all answer sets of the new program.*

Proof 4.3.3 *In the case where there are no existing answer sets it is necessary to show that the tests do not discount a rule that may potentially create an answer set and that searching starting with the set containing the head and body of the rule does not rule out answer sets. As previously discussed constraints can only remove rule sets, thus if a constraint is being added it cannot change the contradictory nature of the rule set. Likewise if the head of the rule is not used in any other rule in the program then it will not alter the contradictory nature of the rule set. Answer sets of the new program must be such that the added rule is applied, if not then the added rule would be removed by the reduct or not used by the immediate consequence operator, if either of these occur then it would have been an answer set of the original program, giving a contradiction. This justifies searching for answer sets containing both the head and the body of the added rule. It also justifies checking for the support of all positive atoms in the body, if there are no rules that support them, regardless of the reduct that atom will not be in the fix set of the immediate consequence operator and thus the new rule can never be used.*

To show that the algorithm is correct (i.e. everything it produces is an answer set), how the sets are produced must be considered. If the new rule is not applicable with respect to an answer set of the existing program then either it is removed by the reduct or it is not used by the immediate consequence operator, either way the answer set is clearly an answer set of the new program. Likewise if the rule is applied with respect an old answer set, it will be in the rule set after the reduct but will not effect the fix point of the immediate consequence operator (both the head and the body of the rule will be reached by the operator as they were with the old program), so it is also an answer set of the new program. Given the previous comments on the search function and under the assumption (which will be proved later) that the second phase of the algorithm is correct, this phase of the algorithm is correct.

On the other hand if there are answer sets of the old program the situation is more complex. An arbitrary answer set of the new program is considered and shown to be given by the algorithm. If the rule being added is a constraint then this is trivially true. Constraints only remove answer sets so an answer set of the new program must have also been an answer set of the old program (in which the constraint wasn't true). Thus by iterating through all of the old answer sets and removing those with respect to which the constraint is applicable generates the full list of new answer sets. For other rules let h be the head of rule r which is being added and A and answer set of the new program. If $h \notin A$ then r is not applicable with respect to A and so A is an answer set of the old program, thus A is given by this algorithm. Likewise if there is more than one rule supporting a in A then A was an answer set of the original program and is thus produced by this algorithm. In the remaining case (where a is only supported by r in A) two sub cases must be addressed, whether the body of r can be concluded without r. If this is the case then A can be derived from an answer set of the old program and is thus produced (given phase two produces all modified answer sets). If not then r must depend on another atom not being true, and the support of that atom must depend on h not being true. The final search catches these cases (such as adding the second rule in an exclusive choice), thus all of the answer sets of the new program are produced by this algorithm.

4.3.2 Phase Two

Phase two of the algorithm focuses on cases when the rule to be added is applicable but not applied with respect to a subset of the answer sets of the program. In this case the conditions for the rule to be 'true' hold and what the rule concludes is not already known - it actually makes a difference. This stage aims to resolve this difference in simple cases and to bound the maximum effect it can have before handing off to a traditional answer set calculation algorithm or phase three if necessary.

Given that implications are defined for removal of atoms as well as addition, this phase can be used after the first phase of the subtractive algorithm (see section 4.4)

4.3.2.1 Description

The first step of the algorithm is to check to see if there are any implications. A trivial sub case of this is when the atom given by the head of the rule is not used elsewhere in the program (for example if it is new to the system). In these cases the atom given by the head can simply be added to the answer set. If this handles all of the answer sets in the input list then the algorithm is finished. At this stage the remaining answer sets can be generated by a search (using the previously outlined theory) using a conventional solver. Alternatively the set of atoms reachable from the head is calculated. By theorem 4.2.1 anything outside this cannot be changed by the addition of the rule. This information can then be passed to a conventional solver (as the search for answer sets in which the unchanged information of each answer set, the head and the body comprise the set of known information) or used to construct a

subset of the reachability graph in which only 'unknown' (i.e. reachable) nodes and the links between them are present. A copy of this - along with the rule being added and the answer set in question is passed to the third phase of the algorithm. It is worth noting that the body of the rule as well as the value of all unreachable nodes is passed to phase three. If the head of the rule can effect it's own body (i.e. exclusive choice) then if the body is not assumed to be true, phase three could waste time generating answer sets in which the new rule is not applicable (all of which are already known).

Phase one of the algorithm essentially tests the relevance of the rule being added, i.e. can it be used to add information to the current context. Phase two is then testing the significance of the information added, how much does it change the context it is used in. This idea is explored further in section 6.2.4.

4.3.2.2 Pseudo Code

Again, pseudo code is presented

```
/* Global Variables */
P    // The program (before addition or after subtraction)
D    // The dependency graph of program P

/* Arguments */
S       // The set of answer sets to be altered.
r       // The rule to be added / removed
change  // The alteration to be made to the answer sets

/* Local Variables */
reachable // The set of atoms reachable from H(r)
sub       // A subgraph of D

/* Return Value */
new  // The answer sets of the augmented program

/* Functions */
implications(positive,ruleSet, atomSet, change)
    // Calculates the implications of the given change
    // in atom set and the given rules

generateReachability(atom,graph)
    // Generate the set of things reachable from atom
    // in graph

generateSubGraph(atomSet,graph)
    // Generates the subgraph of graph containing only
    // nodes in atomSet

augmentSubGraph(answerSet,graph)
    // Labels each node of the graph with the value it
    // takes in answerSet



/* Start of phase two */

/* First check to see if any of the answer sets have *
 * trivial implications                              */
for a in S {

    if ((implications(true,P,a,change).size() == 0) &&
         (implications(false,P,a,change).size() == 0)) {

        /* Then the changes do not progress and further *
           and a U r.head is an answer set              */

        a.applyChange(change);
        new.add(a);
        S.remove(a);
```

```
    }
}

/* Check to see if there are any case unhandled */
if (S.size() == 0) {
    return;
}

/* The changes in the remainder of S can be handled *
 * with an conventional solver as described before  */

reachable = generateReachability(r.head,D);
sub = generateSubGraph(reachable,D);

/* Handle the rest of the potential answer sets with *
 * phase three                                       */

for a in S {

    /* Augment the sub graph with the values each *
     * node held previously                       */
    augmentSubGraph(a,subGraph);

    new = new U phaseThree((a-(reachable U neg(reachable)))
                           U r.body U {r.head} ,graph,P,0);
}

return;

/* End of phase two */
```

4.3.2.3 Proofs

Again, phase three is assumed to be finite, complete and correct for the purposes of this proof.

Theorem 4.3.4 *Phase two terminates in finite time.*

Proof 4.3.4 *As with phase one, the proof of this is trivial. The only loops are over finite, unchanging sets, thus it must be finite.*

Theorem 4.3.5 *Phase two is correct; everything it produces is an answer set.*

Proof 4.3.5 *Given the proof of correctness of phase three, this reduces to only having to prove that the handling of cases where there are no implications is correct. If P is a program, to which r is being added (the proof for subtraction is analogous), giving the addition of a to answer set A, which has been shown to have no implications. The lack of positive implications gives $lpf(T^0_{(P\cup\{r\})^A}) = A \cup \{a\}$, because as r is applicable with respect to A, $r \in (P \cup \{r\})^A$ and $B(r) \subset A$ thus $a \in lpf(T^0_{P^A})$, the lack of positive implications means that this is the only thing that has to be added. Correspondingly the lack of negative implications gives $(P \cup \{r\})^{A\cup\{a\}} = (P \cup \{r\})^A$ because adding a does not stop any rules being applicable which is a stronger condition than not making any extra rules removed via the reduct. Combining the two gives $lpf(T^0_{(P\cup\{r\})^{A\cup\{a\}}}) = A \cup \{a\}$, thus $A \cup \{a\}$ is an answer set. Given phase three is taken to be correct, phase two is correct.*

Theorem 4.3.6 *Phase two is complete; it produces all answers sets based on the input set of answer sets.*

Proof 4.3.6 *When there are no implications, phase two clearly produces the only answer set based on a given original answer set. If there are implications and phase three produces all answer sets based on the answer set given to it (as direct information (everything in `(a - (reachable U neg(reachable))) U r.body U r.head`) and as indirect information (values of the notated graph)), as is assumed then phase two produces all of the answer sets based on the list of answer sets it is given.*

4.3.3 Phase Three

Phase three of the algorithm resolves the implications and dependency graph that have been discovered in phase two, to produce the answer sets of the modified program. It uses of a modified version of the standard bound / reduce / branch approach. When atoms within the program are shown to be unmodified by the changes caused by adding rule r, it dynamically cuts the graph of atoms that need to be resolved, thus reducing the size of the problem further. Sections of the graph that become unreachable from any of the nodes that can still be changed take the same value they had in the original answer set; there is no longer any way that they could be anything else.

4.3.3.1 Description

This section is based on a standard branch and bound algorithm. The known set is augmented until all rules are either applied or not applicable with respect to it. This is handled by looping through the rule sets, removing rules that are applied and not applicable. If a rule is applied but not applicable, the known set is checked for the negative version of the rule head (i.e. it is known that the rule's head is not in the known set), if so a contradiction has been found and nothing is returned. Alternatively the head of the rule is added and any changes resolved. At the end of each run through the rule sets, any atoms that have no supporting rules left in the set of undecided rules is set to be negative (IE it does not appear in the answer set). Finally when all changes have been made, if there are still rules left in the list then the algorithm branches.

The main difference from a standard branch and bound algorithm is the use of the augmented graph. When the value of an atom is decided, the resolve function is called. It removes the node and all links from it. If this is the last link to any node, i.e. there is only one rule left that dictates the status of that node and it entirely dependent on the current node, then that node will have the same relation to it's old value that the current node does. For example if the current node takes the same value that it did in the old answer set and this is the only thing required to determine another node, then that node will also behave as before. This allows a faster method of determining the value of nodes; sections of the graph that are cannot be affected by any of the changes to the answer set are removed and set to their previous value.

4.3.3.2 Pseudo Code

For clarity of control flow, pseudo code is given.

```
/* Global Variables */
P    // The program (before addition or after subtraction)

/* Arguments */
known    // A set of atoms with known values
graph    // An augmented graph s.t.
           // 1. {atoms in graph} U positive{known} U
           //                       negative{known} = H{P}
           // 2. the rule corresponding to every link
           //    is in rules
rules    // A list of rules with unknown status with
         //  respect to known
rules_a  // A list of rules that are applied with respect
         //  to known

/* Local Variables */
changes     // A simple boolean flag of whether there
            //  have been any changes on one pass of the
            //  algorithm
supported   // A list of pairs of atom and binary flag,
            //  flag is set to true if there are still
            //  rules that support this atom
tmp         // A temporary atom that is used in branching
known_copy  // A copy of known, used for branching
graph_copy  // A copy of the graph, used for branching

/* Return Value */
newAns   // The new answer set
```

```
/* Functions */
checkSupport(set,ruleSet)
            // Checks that every atom in set and no others
            // are supported by the rules in ruleSet



/* Start phase three */

/* Work until all rules have been accounted for */
while (rules.size > 0) {

    do {

        /* Note that nothing has changed so far */
        changes = false;

        /* Clear supported flags */
        for s in supported {
            supported.clearFlag(s);
        }

        /* Attempt to resolve the status of as many *
         * rules as possible                        */
        for r in rules_u {

            /* Work out the status of each rule with *
             * respect to the known set of atoms     */
            switch (r.status(known)) {
                applied : rules.remove(r);
                    rules_a.add(r);
                    break;

                applicable : rules.remove(r);
                    /* Check for contradictions */
                    if ((r.type() == CONTRADICTION) ||
                        (known.contains(-r.head)) == true) {
                        return;
                    } else {
                        resolve(r.head,+1,&known,&graph);
                    }

                    changed == true;
                    rules_a.add(r);
                    supported.remove(supported.
                                      lookupByAtom(r.head));

                    break;

                not_applicable : rules.remove(r);
                    break;

                unknown : supported.setFlag(
                            supported.lookupByAtom(r.head));
                    break;
            }
        }


        /* Set every unsupported atom to -1 */
        for s in supported {
            if (s.flag == false) {
                resolve(s.atom,-1,&known,&graph);
                changed == true;
```

```
            }

        }

        resolve(r.head,-1,&known,&graph);

    } while (changes == true);

    /* Branch */
    tmp = graph.first();
    known_copy = known;
    graph_copy = graph;

    resolve(tmp,+1,&known,&graph);
    resolve(tmp,-1,&known_copy,&graph_copy);

    phaseThree(known_copy,graph_copy,rules,rules_a);
}

if (checkSupport(known,rules_a) == true) {
newAns = known;
}

/* End of phase three */


resolve(atom,value,atomSet,graph) {

    /* Update the atom set */
    atomSet.add(value);

    /* Remove all of the links from the node */
    for l in graph.lookup(atom).linksFrom {

        graph.removeLink(l);

        /* If that was the last link to the node */
        if (graph.lookup(l.target).linksTo.size == 0) {
            /* It's value must have the relation to it's *
             * old one that this one does                */
            if (graph.lookup(atom).oldValue == value) {
                resolve(l.target,l.target.oldValue,
                                atomSet,graph);
            } else {
                resolve(l.target,-l.target.oldValue,
                                atomSet,graph);
            }
        }
    }
    graph.removeNode(atom);
}
```

4.3.3.3 Proofs

Theorem 4.3.7 *Phase three terminates in finite time.*

Proof 4.3.7 *The only real question over whether this is finite is whether all rules are removed from* `rules` *. In the body of the loop naf-literals are only ever added to* `known` *, which is a subset of the set of all naf-literals which is finite thus eventually for every atom a, a or* **not** *a will be in* `known` *. In this case every rule will either be applied, applicable but giving a contradiction or not applicable, thus all rules will be removed from* `rules` *and the algorithm will terminate.*

Theorem 4.3.8 *Phase three is correct; everything it produces is an answer set.*

Proof 4.3.8 *The first step to proving this is to show that the set produced,* `known` *is a model the program and* `rules_a` *contains a list of the applied rules with respect to it. The later point is obvious*

as rules are only added to `rules_a` *when they are known to be applied with respect to it. That* `known` *is a model can easily be seen as rules are only removed from* `rules` *when they are known to be applied or not applicable with respect to it (in the case of applicable but not applied,* `known` *is changed so it is applied), and the loop only finishes when all rules have been removed, thus all rules are either applied or not applicable with respect to* `known` *. (It is also necessary to prove that* `known` *is not contradictory - which is obvious as naf-literals are only added when they are not known).*

Finally if all of the atoms in `known` *are supported and nothing else is (as given by* `checkSupport` *) then it is an answer set. Thus everything created is an answer set.*

Theorem 4.3.9 *Phase two is complete; it produces all answers sets based on the input answer set.*

Proof 4.3.9 *By branching each time there is a choice to make between including and not including an atom all possible derivate of the given answer set will be calculated. Thus the algorithm produces all of the new answer sets required if it does not remove any valid answer sets from the computation. Answer sets are only discounted (the function returns without setting the return value) if there has been explicitly found to be a contradiction, either a constraint has been found to be applicable or a contradictory atom has to be added - in which case it would not be an answer set or if* `known` *contains unsupported atoms, in which case it is not minimal and thus a model but not an answer set.*

4.4 Subtraction

Although naively the diametric opposite of adding a rule, the algorithm for handling subtraction is very similar to the additive case. Again it can be divided into three phases. The first phase has the same structure but rather than finding circumstances in which the rule is applicable but not applied it searches for answer sets where the rule is applied and the only way of concluding the head atom; these are the only circumstances in which removal is likely to change the answer sets at all. The second two phases are the same as in the additive case; the problem is still resolving the implications of a change.

4.4.1 Phase One

Analogously with the additive phase one, this phase removes answer sets that are unchanged by the removal of the given rule. The criteria required to change the answer set is similar if slightly more complex, the rule must have been applied with respect to the answer set and been the only rule that was capable of making the atom given by the head true. For example using the following program:

$$\begin{aligned} a &\leftarrow b. \\ c &\leftarrow \textbf{not}\ d, a. \\ d &\leftarrow \textbf{not}\ c, a. \\ \bot &\leftarrow c, d. \\ b&. \\ e &\leftarrow b. \\ e &\leftarrow d. \end{aligned}$$

which has answer sets $\{a, b, c, e\}$ and $\{a, b, d, e\}$ could have $e \leftarrow d.$ removed without the answer sets needing any alteration, however removing $e \leftarrow b.$ would require alteration of the first answer set (it becomes $\{a, b, c\}$).

4.4.1.1 Description

Again this splits into two cases, for when there are not and are any existing answer sets. The treatment of each is very similar to the additive case.

In the case when there are no existing answer sets, very little is known about the rule set, aside from a few trivial tests there is very little that can be done to avoid needing to recompute entirely. If the head of the rule is not used in any of the remaining rules then it certainly doesn't lead to any contradictions, implicit or explicit so removing it will not alter anything. Likewise if there are positive atoms in the body that have no support in the program then there is no way the rule could have been used, and thus no way it could have given rise to a contradiction. Apart from that the only improvement over a complete re-computation is to search for answer sets where the removed rule would have been applicable. Any answer sets of the new program must have been possible to apply the removed rule, otherwise they would have been answer sets of the original.

When there are answer sets there is slightly more work to do. First constraints are handled as a search for answer sets containing the body of the constraint - the things that adding the constraint would remove. Next the answer sets of the old program are checked, those in which the rule was not applied or was not the only applied rule allowing the conclusion of the rules head are become answer sets of the new program. The other answer sets are passed off to phase two of the algorithm to resolve the implications of removing the head of the removed rule. Likewise, as before a search for answer sets can be used instead of phase two.

Proof of this can be derived from the proof of phase one in the additive case, in section 4.3.1.3.

4.4.1.2 Pseudo Code

```
/* Global variables */
P      // set of rules after the removal
old    // set of answer sets of P U {r}

/* Arguments */
r      // rule that has been removed

/* Local Variables */
A      // old answer sets where r was needed

/* Return values */
new    // set of answer sets of P

/* Functions */
search(atoms,rules)
    // A function that uses a conventional solver to
    //  find answer sets of rules given atoms



/* Start phase one */
if (old.size == 0) {
    /* Check to see if the rule could have effected *
     *  anything beyond the head of the rule        */
    if (usage(r.head,P).size > 0) {

        /* Check to see if it is possible to use this *
         * rule                                        */
        for b in r.body {
            if ((b.positive == true) &&
                 (support(b,P).size == 0)) {
                return;
            }
        }

        /* Have to look for possible solutions that *
         * would have been effected by r            */
        new = search(r.body, P);

        return;
    } else {
        /* Cannot change the contradictory nature of *
         * the rule set                               */
        return;
    }

} else {
    /* There are answer sets */

    /* Handle constraints */
    if (r.type == CONSTRAINT) {

        /* Search for answer sets that were removed by r */
```

```
        new = old U search(r.body, P);

        return;
    }


    /* Look for answer sets in which r was applied */

    for s in old {
        if (r.applied(s) == true) {

            /* And see if it was the only thing that *
             * could make it's head true             */

            for q in support(r.head,P) {
                if ( q.applied(s) == true ) {
                    new.add(s);
                    goto nextRule;
                }
            }

            A.add(s);

        } else {
            new.add(s);
        }

    nextRule :

    }

    /* Handle implications */
    if (A.size != 0) {
        new = new U phase2(A,r,-r.head);
    }


    /*** Not sure about this bit ***/
    /* Search for any extra answer sets */
    for b in r.body {
        if (b.positive == false) {
            for
            /* Not happy with this condition */
        }
    }

    return;
}
/* End phase one */
```

4.5 Grounding

So far only the addition and subtraction of ground rules has been considered, clearly being able to add non ground rules would make an implementation even more flexible and useful. The obvious method for implementing this is to apply a standard grounding algorithm to the new rule set and add / subtract ground rules from the old ground program until it becomes the new ground program. This is an equivalent approach to recomputing answer sets completely for each small change - there is no need to run a complete grounding algorithm.

The theoretical treatment of grounding is to create a family of rules from each non ground rule, with every possible combination of objects substituted for each variable. This is clearly problematic as the run time of existing solver algorithms are polynomial in the number of rules in the program. Thus most grounding systems actually remove rules when it can be shown they they will clearly never be used in the ground program. The key observation is that any grounding of an atom for which there

are no rules in the ground program that have it as their head will never appear in any answer sets, so any rule that depends on it can never be used and need not be created. Thus the key to grounding is to work out which groundings of a given atom may be possible.

A grounding of a particular atom can be valid if one of two conditions hold, either it directly appears in the unground program (directly valid) or it appears as the head of a ground rule that depends only on valid, ground atoms. To start with only rules that contain positive atoms in the body will be considered, negated atoms containing atoms are quite strong and complex statements and will be considered later. This again leverages the link between the head and the body of the rule[8]. For example if the rule $p(X) \leftarrow q(X), r(X)$ is considered, for a given object constant c it is simple to conclude that $p(c)$ is a valid grounding if both $q(c)$ and $r(c)$ are valid. Thus a grounding dependency graph may be formed by the following procedure[9]:

- For every atom, create a node for each term in the atom. For example if p is a trinary (3-ary) atom then three nodes, p_1, p_2 and p_3 are created. 0-ary atoms create no nodes as they do not need to be considered when grounding.

- For every unground rule consider every variable that appears in the head of the rule. If it does not appear in any of the positive atoms in the body then it is disregarded. If it appears in one then create a links from the node representing the atom and term in which it appears in the body to the relevant node for it's appearances in the head. Finally if it appears in several positive atoms in the body create an 'intersection' node, link the nodes corresponding to the variables position in the atoms of the body to the intersection node and the intersection node to the nodes that represent the positions of the variable in the head of the rule.

- Form a list for each normal (i.e. non intersection) node containing all of the directly valid groundings for the rule.

A grounding of an atom is valid if, for every term in the atom, the grounding of that term is valid. A grounding of a term is valid if the constant fulfil one of these conditions:

- It is contained in the list corresponding to that node (i.e. it is directly valid).

- There is a link to the node representing the term from a node for which the constant is a valid grounding.

- There is a link to the node representing the term from an intersection node and the constant is a valid grounding for all of the nodes that link to it.

Thus by iterating through the set of all object constants and for each one for each term (in each atom) recording if it is a valid grounding or not; a complete list of which groundings are valid for each term can be constructed in finite time.

The justification for ignoring negative atoms in the bodies of each rule is that this is looking at the head value that could possibly be true. Considering the ground rules without the negated atoms is a weaker condition, thus if a grounding is possible with the full rules then it will also be possible with the positive versions of the rule, all that is lost is a small scope for producing more compact lists of valid groundings.

To ground the rules work through each variable in each rule and construct a set of possible substitutions. Then produce a ground copy of the rule for each possible combination taking one element from each set. The set of possible substitutions is created as follows:

- If the variable only appears in the head of the rule then it creates a list of possible substitutions that is equal to the intersection of the valid grounding lists of all of the terms in which is appears.

- If it appears in the positive atoms in the body and optionally the head, then the set of possible substitutions is equal to the intersection of the grounding lists of the the body atoms in which it appears.

[8] This serves as yet another example of how the syntax and semantics of *AnsProlog** reflect each other, key semantic concepts can usually be expressed cleanly and simply using basic syntactic constructs.

[9] As this algorithm handles atoms with an arbitrary number of terms the simplicity of the concept may be lost in the wording. Considering a program in which atoms only ever have one term reduces this to using nodes that correspond to predicate and may make the algorithm more comprehensible.

- If the variable appears in both some positive and some negative atoms in the body and optionally the head of the rule, then the set of possible substitutions is equal to the intersection of the possible grounding lists of the terms in the positive atoms in body that it appears in.
- If the variable appears in the head of the rule and only in negative atoms in the body then the set of possible substitutions is equal to the intersection of the grounding lists of all of the terms in the head in which it appears.
- Finally if the variable only appears in negative atoms in the body of the rule then the possible substitutions list is equal to the intersection of the lists corresponding to the terms in which it appears. It is also necessary to produce groundings for the rule with all of the atoms in which this variable appears removed.

4.5.1 Additive

Given the above approach to grounding there are two steps to adding a unground rule, working out what other groundings it makes possible and grounding the rule itself. First the grounding dependency graph must be updated and any changes propagated through it. Then all of the rules that contain an atom which has at least one term that's valid grounding list has changed must be considered to see if any extra ground versions are produced. Finally the new rule is ground as described above.

4.5.2 Subtractive

To remove an unground rule the intuitive action is to remove all of the ground versions of it and any other groundings that it makes possible. Removing the ground versions is trivial if a link between ground and unground versions is kept [computationally this is trivial if implemented in anything approximating a sensible fashion, for example keeping a list of pointers to the ground versions of a rule with the original rule]. Removing all groundings that it makes possible is slightly more difficult. It should also be noted that removing all of the ground rules enabled by the removed rule is non essential, any that remain will simply be non applicable. An efficient way of removing most of the ground rules made possible by the removed rule is to consider the support of the heads of each ground version of the removed rule. If they are not supported by another rule then any rule that uses them can be removed. The same holds true about updating the grounding dependency graph, it is trivial to remove the links created by a rule and any object constants that are only given in the removed rule, performing more alterations requires more time and is not strictly necessary.

Chapter 5

Implementation

5.1 Requirements

It is difficult to discuss the requirements of this project in a traditional software engineering context. The stated aim of the project was to produce a faster and more efficient[1] approach to computing answer set semantics of a series of similar programs. Thus an implementation is necessary for a practical demonstration but is clearly not the main focus of this work. It lies somewhere between proof of concept code and a prototype of a useful tool. Many of the requirements (in a formal, software engineering sense) are undefined, left to the common sense and discretion of the implementor. This section aims to support these choices rather than attempt to build a formal framework on such ephemeral ideals.

Given that this technology will largely only be of interest to people who already use, or intend to use answer set solvers, some guidance can be taken from the existing answer set solver implementations. The target audience is highly intelligent, mostly with a formal background in logic to at least undergraduate if not doctorate level. Technical expertise is much greater than both the average individual and others with comparable qualifications in other fields. Access to expert level expertise in most fields of computing technology is also likely. For example it is fair to assume that a user is capable of compiling source code for most common languages, using a command line interface and if they cannot modify the program themselves they will have access to others who can. The amount of time available to them to learn the control interface will probably be limited, however they are perfectly capable of following instructions, reading an overview of the functionality and aided by a couple of examples, figuring it out for themselves.

Most current answer set solvers are developed for Unix[2] and / or Microsoft Windows. It is fair to assume that the average user will have access to either platform. The hardware and performance requirements obviously vary with the size of the input program, but it is fair to say that most answer set solvers will handle trivial and small (up to 100 atoms) in a reasonable time when run on a modern desktop computer. Users are assumed to have access to such hardware and probably significantly higher performance systems (multi processor server, clusters and possibly mainframes) if required for a particular task.

In general answer set solvers are distributed in source form, often licensed under the GPL[22]. DLV is a notable exception to this, although binary versions are freely[3] offered for download for the project website[19]. C or C++ is a common choice of implementation language and generally loose compliance to ANSI standards can be expected - i.e. no software beyond the relevant platform's usual tool chain is required for compilation.

Given this environment it was decided to develop the implementation on a desktop PC running GNU/Linux. GNU/Linux is one of the most widely used and popular Unix like systems used in academia and the availability of free[4] GNU/Linux based operating system such as Debian[32] was felt to somewhat mitigate the disadvantage of developing for one platform. C++ was chosen as the implementation language as it provide high performance, powerful abstraction mechanisms such as

[1] Efficiency in the terms of computer systems is somewhat of a difficult concept. There are a large number of possible definitions, many of which overlap, some of which are contradictory. In this case it is meant in a non rigours manner, "not performing calculations that can be trivially shown to be unnecessary". To some degree the same could also be said about 'faster', but this is addressed in chapter 7.

[2] Used here to refer to POSIX compliant and near POSIX compliant clones of UNIX™. This includes but is not limited to GNU/Linux, Solaris, OS X and BSD variants

[3] Free as in beer[31]. No licences is offered / provided / required.

[4] As in speech, not as in beer[31].

templates and compatibility with other solvers at source level. The implementation was developed to follow the ANSI C++ standard to decrease the amount of work required to port it other platforms (other UNIX like systems should be trivial, especially if they have the GNU tool chain installed). Finally distribution of source code licensed using the GPL[22] was chosen, to give the users maximum flexibility and to encourage further development and integration of the implementation.

5.2 Design

The implementation of the ideas presented here is divided into two sections. IASAI (Incremental Answer Set Algorithm Infrastructure) provides an abstract framework for developing answer set solver algorithms and applications that use them. IDEAS (Interactive Development & Evaluation tool for Answer Set programs) uses the IASAI interface to provide a text based, interactive system for manipulating, developing and using answer set programs.

The focus of IASAI is creating a common interface for different answer set algorithms. Modules that implement the IASAI interface[5] are under development for the algorithm presented in this paper, the smodels algorithm[18] and a generic external answer set solver (aimed at compatibility with DLV[19]. It provides basic operations on rule sets such as adding and subtracting rules, calculating answer sets, etc. as well as functional descriptions of the representations of rules and atoms. As well as providing support for use of answer set programming with incremental knowledge bases it is hoped that this will give a uniform interface for implementing any logic engine that uses answer set semantics without needing to tailor it for a particular system or algorithm.

IDEAS is a command line tool that provides a user interface to the functionality of IASAI. It is intended to be used as a stand alone tool as well as being easily integrated into larger systems for rapid prototyping of logical systems that use these semantics. As well as running in a traditional interactive fashion it also supports scripting in the style of Unix command shells.

5.3 Implementation techniques

In this section some of the features of the implementation are discussed. This is not intended as a complete or through description of the the implementation, more an outline of some of the major implementational decisions.

5.3.1 Sets

Set data types are used extensively throughout both the algorithms and the supporting code. Tests for inclusion, alteration and iteration through sets are some of the most fundamental operations of the implementation, thus considerable effort was put into the selection of the correct storage algorithms for the set data types. Five approaches were considered:

[5] It is not fully in described in this document as it is still under development.

Algorithm	Inclusion	Addition	Subtraction	Iteration	Memory	Other
Unsorted array	$O(n)$	$O(n)$	$O(n)$	$O(1)$	Efficient	None
Sorted array	$O(ln(n))$	$O(n)$	$O(n)$	$O(1)$	Efficient	Addition and subtraction both require moving blocks of memory
Link list	$O(n)$	$O(n)$	$O(n)$	$O(1)$	More costly than array	Moving between entries is more costly than an array based implementation
Balanced binary tree	$O(ln(n))$	$O(ln(n))$	$O(ln(n))$	$O(1)$	More costly than array	Moving between entries is more costly plus requires code and run time to balance after insert. In the worst case performs no better than a link list
Bit set	$O(1)$	$O(1)$	$O(1)$	$O(n)$	Wasteful for sparse sets but very efficient for dense sets	None

Both sorted arrays and bit sets are used in the implementation. Sorted arrays are used for the bodies of rules. These typically contain only a small fraction of the possible range of values. In these applications the cost of moving elements within the set after alterations is not great and after construction is not a frequent operation. Also the lower theoretical complexity of balanced trees is countered by the increase in overheads of moving between elements. Bit sets are used for answer set and potential answer sets. These contain a significant subset of their possible values making so they are not a waste of memory. Also iteration through an answer set is a comparatively rare operation (largely used for outputting to the user) while tests for inclusion and alterations are very common. The implementation of bit sets was also designed to allow the data in them to be addressed as a variety of different data types. This allows the use of hardware vector support such as SSE on Intel processors, VIS on SPARC and AltiVec on PowerPC to handle set-wise operations such as union, intersection and subtraction at 64 or 128 bits per operation. For example the union two subsets of the range[6] $[-1000, 1000]$ could be computed in around 60 machine instructions. Bit sets also give fast ways of computing some more complex operations, for example checking that a set does not contain x and $-x$ can be implemented with bit masks and shifts, which will perform potentially over 100 times faster than the equivalent calculation on other sets.

5.3.2 Support and Usage

The support and usage functions are widely used in the incremental algorithm. Thus in the implementation lists containing the support and usage of each atom are maintained with the atoms. Updating these after altering the rule set is trivial as the lists that contain the rule identifier can be inferred directly from the rule.

5.3.3 Task lists

The incremental IASAI back end is designed to perform the minimum amount of work required to satisfy the request made to it. This is implemented as a FIFO list of task structures. Each task structure contains an indication of which function should be used (phase two, phase three, conventional solver, etc.) and the arguments. Where the algorithms call these key functions (such as handing off between the phases and branching in the conventional solver) a task is added to the list. When an answer set is requested the structures are in turn removed from the list and executed until an answer set has been generated. This allows the computation to effectively be 'paused' after each answer set has been discovered, thus only doing work when it is required. It is also required to support multi-threading and slack time computations, as described below.

5.3.4 Multi-threading

The division of work into task has been carefully chosen so that the only interaction between them is consequence. Executing a task may give more tasks but will not effect any existing tasks and the order of execution is non critical. This allows a custom version of the task dispatch function to be used which assigns tasks to one of a set of threads. Existing answer set solver implementations[18][19] use recursion of lists to handle branching, however this approach allows the simultaneous execution of both branches if run on suitable hardware. Given the increasing trend towards thread and process based parallelism in modern CPU design, this sort of approach is required to make maximum use of computing resources and drastically cut the real time of such computations. See chapter 7.

5.3.5 Slack time computation

The just in time computation approach outlined above clearly helps the initial response time of an interactive program. Returning after the computation of the first answer set will give less delay than computing all answers sets before returning control to the user (assuming a non categorical rule set). However if a second answer set is requested the response time of the just in time method will be less. Slack time computation is the implementational work around to solve this problem.

Slack time computation allows tasks to be dispatched while the front end is waiting and processing user input. IASAI back ends have a function that dispatches tasks while a given semaphore remains clear. To make use of this feature front ends are required to use two threads, a main thread and a slack thread. Only one of these threads will make use of the IASAI back end or in fact heavy use of the CPU at any given time making it suitable for single processor hardware. Before the main thread is

[6] Which could carry the status for up to 1000 atoms

about to start processing user input (for an interactive program this is clearly very light work for the system) it wakes up the slack thread, which then calls the slack function from the IASAI back end. This dispatches tasks within the back end while the main thread is processing the user input. When the user input functions returns, the main thread marks the semaphore controlling the slack function, this causes it to finish computation and return, the slack thread of the front end then sleeps and the main thread continues as normal. When the system is being used interactively this gives the a better distribution of the compute load across the running time of the program and minimises the latency of commands to the program.

5.4 Usage overview

IDEAS serves not only as a user friendly front end to the IASAI system but also as an example of how the interface should be used. It's source code provides an ample reference on how to use an IASAI back end , thus this section is concerned only with the direct usage of IDEAS.

5.4.1 Commands

IDEAS supports a minimal but functional set of commands:

add rule	Adds a rule to the system
doze [number]	Explicitly allows the given number of seconds slack time computation
help	Displays help message
list rules	Displays a list of rules currently in the program
read [filename]	Adds rules in from the given file
solve [number]	Produces the given number of solutions, 0 computes all
write [filename]	Writes the current rule set to the given file

5.4.2 Example

A transcript of an example interactive session

```
IDEAS version 0.1.0, Copyright (C) 2004 Martin Brain
IDEAS comes with ABSOLUTELY NO WARRANTY
This is free software, covered by the GNU General
Public Licence, and you are welcome to redistribute
it under certain conditions
See the file COPYING for details

Built with incrementalSolver from iasai-0.1.0
IDEAS> add rule a :- b.
IDEAS> solve 0
{ not a, not b }
IDEAS> add rule c :- not d, a.
IDEAS> add rule d :- not c, a.
IDEAS> solve
{ not c, not d, not a, not b }
IDEAS> add rule b.
IDEAS> solve
{ a, b, not c, d }
IDEAS> solve
{ c, not d, a, b }
IDEAS> solve
Starting list of answer sets again
{ not c, d, a, b }
IDEAS> add rule e :- d.
IDEAS> solve 0
{ c, not d, not e, a, b }
{ not c, d, e, a, b }
```

```
IDEAS> add rule e :- b.
IDEAS> solve 0
{ not c, d, e, a, b }
Implications handled
{ c, not d, e, a, b }
IDEAS> add rule <> :- c, e.
IDEAS> solve 0
{ not c, d, e, a, b }
IDEAS> list rules
0 :     a :- b.
1 :     c :- not d, a.
2 :     d :- not c, a.
3 :     b .
4 :     e :- d.
5 :     e :- b.
6 :     <> :- c, e.
IDEAS>
```

Chapter 6

Debugging and Analysis

This chapter presents a series of results and definitions derived from the algorithms in chapter 4. These are aimed at users of answer set semantic based systems and the developers of interactive development tools rather than further developing the theory. They provide some intuitive information about the answer sets and rules within a system. Without an application context they are of little value, however with some user defined value context they provide qualitative feedback on the information in the system. For example in an automatic diagnostic system, qualitative hints on how significant or relevant the results of a variety of tests are in the current environment can provide a very powerful heuristic for the over all system behaviour.

Also presented are some initial methods for providing debugging support in the context of an interactive answer set solver. Again these are of comparatively little independent theoretical interest but have appreciable practical value.

6.1 Debugging

6.1.1 Classification of Errors

The most primary question in the design of any debugging or interactive fault finding system is what constitutes an error. The academic study of software engineering and human computer interaction are awash with classification schemes for error. Different approaches categories by what causes them to occur, what allows them to occur, what they effect and how they do so. This variety of approaches is not just restricted to the analysis of 'systems', it can be found in classification systems of programming errors. Security implications of broken code are categorised by effect (information leakage, escalation of privileges, DOS, etc.) while memory use analysers classifies errors by causes (uninitialised memory, unallocated memory, etc.). Here we shall use a standard[37] classification of errors in a programming language by what level of the formal formation of the language they appear in.

- **Lexical** - Errors that occur when a part of the statement in the given language is not a valid word. For example in the English language, "The cat is fghjk." contains a lexical error as the fourth word isn't a valid English word. In the case of programming languages these can be found by the lexical analysis (or tokeniser) phase of the language compiler / interpreter.

- **Syntactic** - The words in the statement are valid by they do not form a valid sentences. For example "The the is cat mat on." is made of valid words but not in a correct order. Again these errors can be found automatically in programming languages by the compiler.

- **Semantic** - The statement is well constructed but doesn't have a valid meaning. For example[1] "The cat is gaseous" is syntactically valid but doesn't mean anything. Depending on both the language and the error these may be possible to detect at compile time or only detectable at run time (if at all).

- **Logical** - The statement is technically correct but not what the user wanted. For example "The dog is brown." is valid English but doesn't say anything useful about how the cat is. These are

[1] Unlike programming languages, the English language is semantically context dependent so as a point of pedantry it might be possible to construct a context in which this is not a semantic error. For the sake of this example and the health of the cat(s) potentially involved the reader is strongly dissuaded from attempting this.

traditionally the hardest type of bug to find. Software engineering is, in part an attempt to stop this sort of error by introducing procedures, such as verification against a standard.

In the case of *AnsProlog** the content of these categories is somewhat different to procedural languages. Object constants, predicates and variables are introduced as they are needed, thus there are comparatively few types of lexical error possible. Also the syntax of the language is very simple so the scope for syntactic errors is also small. Both of these class of error can be (and are in IDEAS) found automatically and thus are not of interest. The semantics of *AnsProlog** are defined for all syntactically valid programs so in theoretical terms there are no semantic errors. The techniques for debugging presented here are an attempt to handle logical errors, and are thus subject to the same problems and vagaries that any attempts at the automatic location of logical errors has. Given that some researchers[9] view logic programs as specifications, or equivalent to them, this categorisation is not entirely surprising. Errors are divided between rule level and program level.

6.1.2 Language Modification

By altering the language syntax and semantics slightly it is possible to remove the possibility to make certain types of lexically and syntactically based mistakes at the rule level. These will not eliminate logical errors at the rule level, a user can still write $a \leftarrow b.$ when they meant $a \leftarrow b, \textbf{not}\ c.$ and there is really no way of finding this automatically[2].

- Introduce a notation, `<>` for the symbol $\bot$ and mandate it's use. If rules with $\bot$ in the head are notated without a head then any rule in which the programmer has neglected to fill in a head atom will become a constraint. For example there is no way of telling automatically if `:- a, b.` is notation for $\bot \leftarrow a, b.$ or $c \leftarrow a, b.$ without the head. With an explicit $\bot$ notation these errors by omission can be found automatically.
- Introduce a notation for declaring the existence of a predicate symbol and create a syntax error if a variable has been used before it has been defined. This will catch slight misspellings and typos in the names of predicates, implementations may also wish to use technology for spelling checking algorithms to suggest replacement predicates in an interactive context.
- Introduce notation for declaring sets of object constants and require that when predicates are declared, the range of each term is specified. As well as making programs clearer this information can be checked by the grounder to check that no unintended ground naf-literals can occur.

6.1.3 Query based Debugging

In this section two techniques are presented in outline for answering queries about why answer sets have particular combinations of atoms. The users is assumed to have some knowledge of what the answer sets should be. This is not an unreasonable assumption as if the user didn't know what the answer sets were supposed to be, how would they know that what they had was incorrect. These techniques can then help them locate which sets of rules give the invalid conclusions, thus locating some classes of program level logical errors.

Exactly how the results of these queries are represented to the user (especially in the case of multiple failed options) is left as an open ended question.

6.1.3.1 Why is Set S Contained in Answer Set A?

The first requirement is to check that S is indeed a subset of A. Then the problem breaks into showing why each positive atom in A is supported and then why each negative atom is not supported. The former is given by providing a list of which rules are applied with respect to S and support each of the positive atoms in A. A similar list is used for the second section, although it contains each rule that has a negative member of A as it's head (an optionally a note informing the user that note exist in appropriate cases) and which atoms in S stop the rule being applicable.

For example given the program:

$$\begin{aligned} a &\leftarrow b. \\ c &\leftarrow \textbf{not}\ d, a. \\ d &\leftarrow \textbf{not}\ c, a. \\ b. & \end{aligned}$$

[2]It could be argued that by putting the . at the end of the rule, the programmer has explicitly said "This is the complete rule" thus the error is entirely outside the language interpretation software

asking[3] "Why is $\{a, \textbf{not}\ d\}$ contained in answer set $\{a, b, c\}$?" is envisaged to produce the following computer generated explanation[4]:

> a is in $\{a, b, c\}$ as $a \leftarrow b.$ is applied with respect to it. d is not in $\{a, b, c\}$ as only $d \leftarrow \textbf{not}\ c, a.$ supports d and c is in the answer set so it is not applicable.

6.1.3.2 Why is Set S not Contained in any Answer Set?

Again the first requirement is to make sure the request actually makes sense, i.e. there is no answer set containing S. The task then breaks down into first justifying why the atoms in S could be supported (or not supported) in a consistent fashion, if this is possible then conclusions can be drawn until a contradiction is reached. This is computed by creating a list of scenarios. Each time there is a choice of rule, a new scenario is created and each time a scenario is removed, how it was constructed and why it was removed is returned to the user. For the first section of the algorithm work through each positive atom in S and each rule that supports it, each creates a scenario (and may add to the list of the atoms that has to be justified). If any of the rules is inconsistent with the set of atoms required (S or a superset there of) in a given scenario then it is removed. Care must be taken to ensure that the atoms in the required set that have been justified and those that haven't are marked, thus allowing the detection circular dependencies. The negative atoms in the required set of each scenario are then considered. If any of the rules that support it are applicable with respect to the set of required atoms then that scenario is removed. Finally each scenario has rules that are applicable with respect to the required set added until a contradiction is reached.

It is difficult to give a trivial and comprehensive example of everything that this approach could detect. However it is envisaged that asking "Why is $\{c, d\}$ not contained in any answer sets" of the previous example would give the following explanation:

> c is can only be supported by $c \leftarrow \textbf{not}\ d, a.$ but d is assumed to be in the answer set.

6.2 Analysis

The section use t technology and techniques developed for the incremental answer set calculation to provide some intuitive information about a program and it's rules. These are intended to aid programmers and users of answer set semantic systems in gaining a conceptual grasp of what a program is doing and how it is working without overloading them with information. For example the values returned by the relevance test can be used to provide a heuristic for what information to attempt to discover next, while the significance tests can easily highlight what are the key concepts within a knowledge base. The computation of the quantities referenced in these definitions has been outlined in section section:backgroundresults, thus any implementation of the incremental solver algorithms can be trivially extended to be able to answer questions posed by the user regarding these definitions.

6.2.1 Redundance

Adding a rule to a base of knowledge does not necessarily alter the 'sum'[5] of the knowledge. For example if we believe that where there is smoke there is fire ($fire \leftarrow smoke$) and we learn that where there is smoke and the sound of a fire alarm then there is a fire ($fire \leftarrow smoke, firealarm$) we have learnt something new but it doesn't add to what we know about locating fires. We formalise this concept as redundance[6].

Definition 6.2.1 *Rule r_1 is redundant with respect to rule $r_2 \Leftrightarrow H(r_1) = H(r_2)$ and $B(r_2) \subset B(r_1)$*

A rule is redundant if there is a way of concluding the same thing from less information; what the rule tells you is already said more succinctly elsewhere. This concept easily generalises to the following.

Definition 6.2.2 *Rule r is redundant with respect to rule set $P \Leftrightarrow \exists r' \in P$ s.t. r is redundant w.r.t. r'*

[3] The user is assumed to have requested the computation of the answer sets of the program before asking this.
[4] Some form of graphical display might be an alternative option for more complex results
[5] For some sense of sum!
[6] Previous works have referred to this as subsumption, the alternative name is given to lend the concept a more intuitive base

set does not change the answer sets. If one rule in a set is redundant with respect to the rest of the set then removing it will not change the answer sets either.

The following proposition shows that a redundant rule does not effect the answer sets of a rule set.

Proposition 6.2.1 *Let P be a program and r a rule that is redundant with respect it. Also let $A \subset \mathcal{B}_P$.*

$$A \text{ is an answer set of } P \Leftrightarrow A \text{ is an answer set of } Q = P \cup \{r\}$$

Proof 6.2.1 *First consider the forwards ($\Rightarrow$) implication. If $r \notin Q^A$ then $Q^A = P^A$ so A is an answer set of Q. Otherwise ($r \in Q^A$), let $r' \in P$ be the rule that make r redundant. As $B(r') \subset B(r)$ then $r' \in Q^A$ and also if $B(r) \subset lpf(T^0_{Q^A})$ then $B(r') \subset lpf(T^0_{Q^A})$ so $lpf(T^0_{Q^A} = lpf(T^0_{P^A})$ and A is an answer set of Q. The proof of the reverse ($\Leftarrow$) is entirely analogous.*

By testing rules for redundance, an application using an incremental solver as a knowledge store can remove rules that are clearly no longer needed.

6.2.2 Relevance

To motivate this definition, the Oxford English Dictionary [1] entry for relevant is given:

> Bearing upon, connected with, pertinent *to*, the matter in hand

In the context of a rule's relation to a program and it's answer sets this gives the following definitions:

Definition 6.2.3 *Let P be a program, S is the set of answer sets of that program, $r \notin P$ is a rule and $A \in S$.*

- *r is relevant with respect to $A \Leftrightarrow r$ is applicable but not applied with respect to A*
- *The relevance of r is the proportion of S with respect to which it is relevant.*

This is intuitive as only links between information (rules) that can be used and infer something that is not already known are considered to be relevant. In the light of this definition phase one of the algorithm is restricting the work of the solver to cases where the new rule is relevant. This definition can trivially extended to to cover the relevance of rules within the program, although this requires calculating the answer sets of the program without that rule.

For example given the following program :

$$\begin{aligned} a &\leftarrow b. \\ c &\leftarrow \textbf{not } d, a. \\ d &\leftarrow \textbf{not } c, a. \\ e &\leftarrow \textbf{not } f, a. \\ f &\leftarrow \textbf{not } e, a. \\ b. & \end{aligned}$$

which has the answer sets $\{a, b, d, f\}$, $\{e, a, b, d\}$, $\{c, f, a, b\}$ and $\{c, e, a, b\}$. The rule $g \leftarrow a.$ has relevance 1 as it is applicable with respect to all answer sets but not applied with respect to any of them. $g \leftarrow \textbf{not } c, b$ have relevance $\frac{1}{2}$, while $g \leftarrow \textbf{not } c, \textbf{not } f$ and $f \leftarrow \textbf{not } c, b$ have relevance $\frac{1}{4}$ and $c \leftarrow \textbf{not } g, h.$ has relevance 0.

Relevance can be used by an application as a metric for working out which rules from a set will change the most / least of the current 'world views' when added.

6.2.3 Independence

Proposition 4.2.5 and Theorem 4.2.1 showed that the ability of an atom to influence the status of another atom within any answer set was equivalent to whether it was reachable from the node corresponding to that atom (in the program's dependency graph). This is presented to the user of the system as the concept of independence.

Definition 6.2.4 *Let P be a program, D it's dependency graph and $a, b \subset \mathcal{B}_P$.*

$$a \text{ is independent of } b \Leftrightarrow a \notin R^\omega(b)$$

It is worthwhile noting that this definition of independence is not reflexive. For example in the following program:

$$\begin{aligned} a &:- b. \\ b &:- c. \\ c. & \end{aligned}$$

c is independent of a but a is not independent of c. Although independence is commonly used in a reflexive context in natural language, this is not necessarily automatically the case. The Machevellian school of politics would suggest that being independent of others but able to control them (i.e. they are not independent of you) is an optimum scenario - in this context assuming independence to be reflexive could be a costly mistake.

Independence can be used by a programmer to help find and understand unintended logical links between concepts or investigate if certain patterns in answer sets are co-incidental or not.

6.2.4 Significance

A generalisation of the concept of independence. Again this is motivated by it's definition in the Oxford English Dictionary [1]

1. The meaning or import of something, 2. Importance, consequence

The less atoms that are in the system that are independent of a particular atom the more significant it is. This is formalised as:

Definition 6.2.5 *Let P be a program, D it's dependency graph and $a \subset \mathcal{B}_P$.*

$$significance(a, P) \quad = \quad \frac{|R^\omega(a)|}{|\mathcal{B}_P|}$$

Using the example program in section 6.2.2, b has significance of $\frac{5}{6}$, a has significance $\frac{2}{3}$ and c, d, e and f have significance $\frac{1}{3}$ (as they can effect one other atom, which can then effect them).

In a constraint satisfaction problem such as timetabling, using significance can give non obvious information on the relative significance of resources, which can then be used by an intelligent planning application to try to remove bottlenecks in future.

6.2.5 Possible Groundings

In a system which implements the addition of non ground rules it is simple to provide the lists of possible groundings for a given predicate to the user. By tracing back through the dependency graph it is also possible to display why each one is a valid, i.e. which rule or combination of rules is required for it to be valid. For example the following program (a subsection of a classification system) contains a flaw:

$$\begin{aligned} data(unicycle, 1, no, light). \\ data(bicycle, 2, no, light). \\ data(cart, 4, no, light). \\ data(wagon, 4, no, heavy). \\ data(motorbike, 2, yes, medium). \\ data(gocart, 4, yes, light). \\ data(mini, 4, yes, medium). \\ data(landrover, 4, yes, heavy). \end{aligned}$$

$$\begin{aligned} motorVechile(X) &:- data(X, Y, yes, Z). \\ car(X, Z) &:- motorVechile(X), data(Y, 4, yes, Z). \end{aligned}$$

The last rule should read $car(X, Z) :- motorVechile(X), data(X, 4, yes, Z).$ as the existing program gives $car(landrover, light)$ and $car(mini, heavy)$ as possible groundings. By displaying which rules a grounding is dependent upon this form of error can be traced.

Chapter 7

Evaluation

This chapter contains a discussion of the issues involved in producing a fair and representative series of tests for comparing the incremental and external solver IASAI back ends. This was originally intended to be used as a criteria of the project's success however during the construction of these tests it became clear there is a more fundamental issue involved. In short no tests can be performed as there no typical / representative applications of an incremental solver. Converting a 'typical' application of a conventional solver into an incremental program does not produce a 'typical' or even a meaningful application of an incremental solver.

7.1 Introduction

Before the results or even the tests can be considered there must be a discussion of exactly what is to be measured. The motivating problem for the development of the theory was to provide a 'faster' and 'more efficient' way of dealing with applications that develop rule sets over time. These are, for the most part, interactive applications of some sort. Interaction is either directly with a user or with another software component. (If the application is not interacting - i.e. receiving new information from external sources then the whole rule set is known before run time and even if solutions are required to sub problems this could be split between multiple instances of conventional solver. This is still an interesting problem but a much less likely frequently encountered style of application.) Thus the key issues are how well the program interacts; how quickly and how consistently it responds to requests for solutions. The response time for non solutions commands is of little interest as it is fairly implementation dependent and not the actual difference between the two approaches. An additional complications is that the response time will be strongly related to the number of rules in the program at that given point. Thus simple collection aggregate data (such as mean solution request length or standard deviation to test consistency) will not give an accurate reflection of the true relation.

7.2 Existing Benchmarks

For a comparatively new technology, answer set solvers have already acquired a wide variety of benchmarking systems. However as there is very little syntax shared between solvers and certainly nothing close to a standard input language, standard benchmarks are a long way off. The Dagstuhl initiative[33] divided benchmarks into three categories, ground (all ground using lparse[20] or the DLV[19] grounders) unground and free style (in which a text description of the program was given and any solver specific techniques could be used). An alternative approach is the University of Potsdam benchmark generator[34] which can generate standard problems (finding Hamiltonian cycles in graphs, N-Queens problems, graph colouring, etc.). Hybrid approaches such as providing data and a text description also exist[35]. Even random generation of input programs has been used[36] as a comparison strategy.

However none of these approaches is really suitable for a quantitive evaluation of this project as they all focus on measuring the performance of just one calculation in terms of elapsed time, CPU time and sometimes memory usage. As previously outlined these are not the types of value that are appropriate for this project.

7.3 Parameters

In this section description of the main variables in any benchmark and how they could be address is presented.

7.3.1 Input Variables

The first and most obvious input variable in the benchmark test of IDEAS is which IASAI back end is to be used. Four different were considered; the external solver back end (using lparse[20] and smodels[18]), the uni-thread incremental back end and the multi-thread incremental back end with 2 and 4 threads[1]. This gives a good comparison between two algorithms as well as illustrating what advantages multi-threading gives.

The next obvious variable is the input program. Logic programming is very free form compared to procedural languages, which poses a problem when benchmarking. Traditionally procedural languages have a small number of constructs around which most programs are based, for example functions, switch statements, for loops, etc. Additionally by these tend to be used in a comparatively small number of ways. Thus benchmarking a system on a set of typical components (i.e. a quick sort, an FFT, balanced binary tree operations and so on) can give a reasonably representative view of the predicted behaviour of any program by matching the programs attributes to the relevant benchmarks (i.e. this program uses a lot of floating point operations and has lots of non linear memory access, thus the FFT benchmark will give a good idea). In the case of logic programming this approach is much harder to achieve. There are very few non trivial constructs that are common across a significant number of programs. Instead a similar approach to benchmarking applications is used, some program representative of typical tasks are created and measured. Then by finding which category of 'typical' program a certain task falls into the performance can be predicted. As the main focus of benchmarking is the comparative performance of the different solvers only a few different programs will be considered. Although the results could be argued to be more representative if a larger number of programs were used it is also important to keep the number of tests and results down when benchmarking or risky any useful conclusions being lost in a sea of data. Issue relating to the choice of representative problem are further discussed in section 7.4.

A subsidiary issue of what program to use is what order the rules should be within the program. When benchmarking traditional answer set solvers this is largely trivial however in the case of the incremental algorithm this is not the case. Although the link between the ordering of the rules and the performance of the incremental algorithm is complex it is comparatively easy to come up with cases in which it has a clear effect. Putting all of the constraints last leads to a large number of potential answer sets being generated only to be ruled out later on. On the other hand adding facts last may mean that no answer sets are generated at all to start with. The ordering of rules does not only effect how the computational load is distributed been solution requests, it also effects the total amount of work that is done. In light of this and the complexity of the relation between the rule ordering and the behaviour of the algorithm, the ordering of the rules in the benchmarks was kept constant and assigned in a 'natural' way (i.e. a fact is introduced, then the implications that depend on it, then another fact).

Possibly the most important, non obvious[2] variable is the 'degree of interactivity', i.e. how often a solution is requested versus how often rules are added. At one end of the scale all the rules could be added and then a solution requested, and at the other a solution can be requested after every addition. Clearly these are degenerate cases and favour one of other of the algorithms in question. This is an important input as it gives an idea about 'how interactive' a program has to be before this approach is worth while. Thus the benchmarking does not just provide an idea of which algorithm is more responsive, it also gives an idea as to the degree of interactivity which each would be most suited to.

Finally there are questions about which features of the IASAI back end can be used. The implementation has been designed with interactive applications in mind and thus may have advantages over the back end using an external solver if these are replicated in the tests. Individual answer sets can be returned rather than waiting for all of the answer sets to be computed. Dependant on the application this may or may not be required and thus may give an extra edge to the incremental back end. Likewise feeding command into the IDEAS front end as fast as possible is not necessarily representative of an

[1] A maximum of 4 threads was chosen as hardware that can run a more than 4 threads concurrently is comparatively rare.

[2] The preceding variables, especially what solver to use are very important, but are somewhat implicit from the act of benchmarking the solvers

interactive application in which there will be time between calls to the back end as the user / system makes use of what it has been told and gathers information from other sources for the next alteration. If configured to do so the incremental back end can use this time for slack time computation. Which of these to choose is fundamentally application specific and cannot be picked without understanding at least the theoretical context of a particular test program.

7.3.2 Output Variables

Two key measurements should be made for each solver / program / environment combination, the latency of each solve command and how consistent this value is. This in turn will give an idea of how easy it is to interact with the program, the lower the latency and the more consistent a program the easier it is to interact with. Calculating the consistency of the value is a non trivial statistical problem. Simply calculating the standard deviation (or equivalent) values is inappropriate unless the number of rules and the computation time required can shown to be linearly related (which is certainly not true in the theoretical case).

7.4 Choice of Program

Before discussing the nature of suitable problems it is worth noting a requirement on their size. A benchmark program must contain hundreds, if not thousands of choice points (stages in the decision process where a choice between two atom is needed - essentially the branch part of the algorithm) and a correspondingly large number of rules if it's computation time is to be measured accurately. The granularity of any form of timing is based on the granularity of the time slices allocated by an operating system[3], in the case of the Linux kernel this is configurable but defaults to 1×10^{-2}s in 2.4 series and 1×10^{-3}s in 2.6 series. A quick calculation[4] would suggest that a modern, high end processor is capable of executing several million instructions in even the shortest of these slots. In the case a program with only 1000 rules this gives several thousand machine instructions to handle the rule's effect - much more than is needed by existing implementations in most cases. Thus benchmark problems need to be large.

7.4.1 Why Existing Problem Classes are Unsuitable

The initial intention was to convert existing programs from benchmarks into a suitable format for interactive use and then use these as benchmarks to compare the solver algorithms. This cannot be done in a useful fashion for two reasons, some problems do not split into incremental forms in any sensible manner, the rest do not give a particularly useful or interesting variety of increments.

Many of the programs commonly used as benchmarks represent a complete problem that is not divisible by splitting off rules. The prime example of this is the N-Queens problem, it would not be useful to approach this as placing $n-1$ queens, using and $(n-1) \times (n-1)$ board or allowing queens to be on the same diagonal, which correspond to removing some of the rules of the program. What are created by a part of the program are not part of the solution in a useful way. This type of issue vastly reduces the number of suitable problems.

Problems that can be broken down in this fashion have a common structure; rules either specify the semantics of the problem or the structures it works over, essentially they are subclass of constraint satisfaction problems. A prime example are graph colouring programs. These have unground rules specifying that every node must have a colour and it cannot be the same as any node it links to and a series of ground facts specifying the nodes and links used. Increments to these programs can either come in the form of alterations to the data domain, for example a new node and the ground versions of the problem specification rules that apply to that node or they can be further conditions on the structure of a solution. The first case is basically only altering the possible groundings of the existing program, answer sets are extended but there is no real change in the structure of the problem. The later case is effectively adding constraints (implications that further describe the solution to a problem are essentially just removing answer sets in which these do not apply), which only remove answer sets. Incremental extension of constraint satisfaction problems is an interesting topic but does not generate significant or varied changes to the structure of the problem and thus are not suitable as benchmarks for incremental solvers.

[3] This is somewhat of a simplification, the actual process of computing the amount of time used by a process is exceedingly complex. This abstraction suffices for the rough calculation presented.

[4] This is only a rough estimate. Theoretical throughput on some high end processors may be over 12 million instructions per time slot, however allowances for memory latency and pipelining issues reduce this significantly.

7.4.2 Characteristics of a Suitable Problem

Having shown that existing problems are not suitable, the next obvious question is what types of program are suitable. The key requirement is that increments to the program produce different types and different levels of significance of change to the structure of the problem. These are programs in which the links between the key concepts and not just the data is fluid, the nature of the program and the problem it is trying to solve evolves as the program runs. At a more base level these programs will not be composed of large data domains and predicated rules that select from them as these give a very uniform structure to the program which is hard to alter in the required way. A program that genuinely tries to represent the links between concepts and model knowledge rather than being an intelligent approach to abstract data structuring would be one possibility.

7.4.3 Generation of Such Problems

Currently answer set semantics are mostly used as an intelligent way of specifying search algorithms over complex, abstractly structured data domains. Planning programs describe resources and how they relate to each other, the requirements of the plan are added and the solver is run to find a suitable strategy. Automatic diagnostic systems have a model of the components they work with, essentially a structure linking symptoms and causes, the current situation is added and the solver searches for the possible causes. What is needed is an application where the *AnsProlog** program is used to represent what the application 'knows' and the requirements of, and uses for the information are external to the knowledge base.

One possible source of such scenarios is in the construction of autonomous or semi autonomous, intelligent, learning agent systems. Some work[30] has already been done in using answer set solvers in agent systems but as with many agent systems these are research prototypes rather than 'real world' systems. A scenario in which the motivation and direction of the agent is controlled directly within the agent and the logic programming engine is used to represent what it knows about the world (and is used, possibly along with a stochastic system for judgement when it needs to work out if particular actions are feasible or if certain combinations of conditions could be true) would potentially produce the kind of programs needed to benchmark approaches to incremental algorithms, as well as probably being a very good test bed in itself.

Chapter 8

Conclusions

8.1 Appraisal

If evaluated on the basis of it's stated aim this project has been neither a success or a failure. A new algorithm for computing the answer sets of an *AnsProlog** program[1] in an incremental fashion has been discovered and a proof of concept implementation has been created. However this could not be shown to fulfil the criteria of being 'faster' or 'more efficient' in a general way. In particular sub cases, it can be seen to do very little work to produce the new answers sets, and thus could be argued to be faster. However without testing it on some real examples (which of course requires suitable examples to exist), no argument supporting the superiority of this approach could be complete. Thus whether the project is a success or not is undecided.

In the wider context of answer set semantic research however, this project has made some significant progress. The algorithms presented in chapter 4 are at very least of theoretical interest and demonstrate that an incremental approach to computation is not only possible but reasonably practical. The analysis and debugging techniques presented in chapter 6 are believed by the author to be the first formalised discussion of *AnsProlog** in the context of create programmer support tools. Perhaps of most significance is the conclusions of chapter 7. It is the belief of the author that no suitable programs exist because *AnsProlog** is currently only used in applications as a problem solver rather than a knowledge representation and manipulation system. The problems that are being solved are essentially static in nature. By creating the tools that allow such applications to be developed, *AnsProlog** can be applied to new fields where is was not previously considered viable due to the visualisation of the nature of the problem not matching the perception of how *AnsProlog** based systems could work.

8.2 Further Research

This document presents a complete solution for handling the addition and subtraction of single rules from both ground and unground AnsProlog programs. However in terms of the over all topic this is only the beginning. In this section some of the theoretical questions and implementation areas raised by this work are presented.

8.2.1 Theory

As this work treats changes in rule sets as single, atomic operations no consideration of the pattern of rule modification has been made. For example adding an exclusive choice between two new atoms requires at least 2 rules and is potentially quite computationally expensive as the second rule will require a full re-computation. Likewise care must be taken to add constraints before the rules than generate large numbers of options. One area of interest is to look at ways of adding multiple rules simultaneously, allowing a new concept or block of data to be added in one operation with considerable potential savings. An alternative approach to the same problem would be to develop some form of criterion or heuristic for when to handle a series of changes using the presented algorithm and when it is more efficient to recompute completely.

Such issues lead naturally to considering a modular approach to answer set and logic programming. Modular programming is a well accepted technique and a powerful abstraction mechanism, using modified versions of the presented algorithms and techniques for making several simultaneous changes

[1] Or at least a usable subset of these programs

it may well be possible to provide this for answer set programming. This raises more possibilities, from pre-computation of fixed blocks of rules to distributed computation to mixing rule sources (for example using databases as sources of facts) to the possibilities of mixing logical paradigms on a module by module basis (using preference based choice formalisms such as OCLP for human interaction and *AnsProlog* for the actual computation). All of these would be further step towards providing a modern programming environment for answer set computation.

Finally nothing has been presented on the addition of rules in any of the formalisms that extend $AnsDatalog^{\neg,\perp}$. Those that can be mapped or reduced to AnsProlog could be converted quite easily, although the nature of such mappings may significantly reduce the value of such algorithms. However logic systems such as $AnsProlog^{or}$ with a higher computational complexity and logic system which include function symbols are a much more interesting issues. Clearly the presented algorithms provide part of, but definitively not a complete solution.

8.2.2 Implementation

As well as a number of theoretical questions, there are also a large number of implementational issues raised by this work.

Firstly the IDEAS front end is still only in the alpha development stages. There are a large number of features that need to be implemented before it reaches the standard of a finished product. Some of these have been outlined in this document, where as others, such as lexical matching to spot typographical errors, support for a wider variety of input formats and portability to more software platforms are more implementation specific. Support for the query language for answer set semantics would also increase it's functionality as a rapid prototyping tool for knowledge representation. An incomplete todo list is included with the source code.

Although not necessarily of massive theoretical significance, other IASAI front ends would provide interesting applications of answer set solvers in other problem domains. A front end that supported PROLOG syntax (or as much of it as is reasonable) could be a useful teaching tool. Another option is an SQL front end which would provide a more intelligent approach to complex data mining applications while maintaining compatibility with a large array of existing tools.

For comparative and benchmarking purposes it may well be advantageous to implement IASAI back ends that integrate with and / or implement some of the other answer set solver algorithms.

Finally the incremental IASAI back end has considerable scope for future development. The analysis tools provided by could be quite simply extended to handle classification of the program into one of the established classes of sub program, for example definite, stratified and call consistent. This information could then be used to use more efficient techniques for particular classes. AN extension of this idea might be to have support for locating independent sections of a program and splitting it into several smaller sub programs. To support larger input programs the task list and dispatch infrastructure could be extended to distribute not only across threads but between separate machines, allowing the computing power of high performance clusters and distributed systems to be used. Finally the slack time infrastructure could be used to perform speculative execution if there are no tasks remaining. At the most basic level this could be calculating which atoms / answer set combinations give no implications, however as implications are only dependent on the atom in the head of the rule being added rather than the complete rule it may well be possible to handle non trivial implications in a non trivial fashion. Given enough time between rule set alterations it would be possible to compute all possible atom / alteration / answer set combinations so handling alteration would simply require looking up the appropriate results. In computational terms this is a wasteful approach as a large number of the results will never be used and will have to be discarded after the rule set has been changed, however in application areas in which this time would simply be wasted this is not a problem. Further work may develop ways of modifying the data produced by speculative execution rather than simply discarding it. It might also be possible to produce constructive rather than analytic heuristics for what information is to be added; the system could give a list of rules that would fulfil or aid certain criteria on the answer sets (i.e. that there is only one answer set / the program is categorical, the answer sets allow certain queries to be answered definitively). This is envisaged to have significant potential for learning based knowledge representation applications, for example in autonomous, intelligent agent systems.

Bibliography

[1] *The Oxford English Dictionary*, Second Edition 1989, Oxford University Press

[2] K. Gödel : *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I.*, Monatshefte für Mathematik und Physik 38, pp. 173-198, 1931

[3] C. E. Shannon : *A Symbolic Analysis of Relay and Switching Circuits*, MIT Master's Thesis, 1938

[4] A. Newell, J.C. Shaw, and H.A. Simon. : *Empirical explorations of the logic theory machine: a case study in heuristics.*, Proceedings of the Western Joint Computer Conference, pages 218-239, 1956. Also in Computers and Thought (Feigenbaum and Feldman, Eds.), McGraw-Hill, 1963, pages 134-152. An interesting footnote to history is that the logical system described "The Logic Theorist" was credited as a co-author of a paper submitted to the Journal of Symbolic Logic on some of the theories it had been used to develop. The paper was allegedly rejected as the journal was unwilling to publish a paper with a machine author.

[5] S. Costantini, A. Formisano and E. Omodeo : *Mapping Between Domain Models in Answer Set Programming*, ASP03: Answer Set Programming: Advances in Theory and Implementation. Ceur-WS, September 2003, http://CEUR-WS.org/Vol-78/

[6] E. Bertino, A. Provetti and F. Salvetti : *Local Closed-World Assumptions for reasoning about Semantic Web data*, Proceedings of the APPIAGULPPRODE Conference on Declarative Programming (AGP 03), pages 314-323, 2003.

[7] P. Maes : *Agents that reduce work and information overload*, Communications of the ACM, Volume 37, Issue 7

[8] A. Robinson : *A Machine Oriented Logic Based on the Resolution Principle*, JACM 12(1), pages 23-41, 1965

[9] Chitta Baral : *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, 2003

[10] T. S. Geisel : *The Cat in the Hat*, Random House Inc., ISBN 0-394-90001-4, 1957

[11] M. Denecker : *What's in a Model? Epistemological Analysis of Logic Programming*, ASP03: Answer Set Programming: Advances in Theory and Implementation. Ceur-WS, September 2003, http://CEUR-WS.org/Vol-78/

[12] M. Gelfond and V. Lifschitz : *The stable model semantics for logic programming*, Logic Programming: Proc. of the Fifth Int'l Conf. and Symp. Pg 1070-1080, MIT Press, 1988

[13] A. Van Gelder : *The Alternating Fixpoint of Logic Programs with Negation*, Proceedings of the Symposium on Principles of Database Systems, Pg 1-10, 1989

[14] P. Dung : *Negations as Hypothesis: An abductive foundation for logic programming*, Proceedings of the 8th ICLP, MIT Press, Pg 3-17, 1991

[15] J. You and Y. Yaun : *On the Equivalence of Semantics for Normal Logic Programs*, Journal of Logic Programming, 22:212-221, 1995

[16] T. Przymusiński : *Every logic program has a natural Stratification and an iterated least fixed point model*, Proceedings of Principles of Database Systems, 1989.

[17] F. Fages : *Constructive Negation by Pruning*, LIENS tech report 94-14, revised 95-24, 1995.

[18] Smodels main web page, http://www.tcs.hut.fi/Software/smodels/

[19] DLV main web page, http://www.dlvsystem.com/

[20] Tommi Syrjen : *Lparse 1.0 User Manual*, http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz

[21] Patrik Simons : *Extending and Implementing the Stable Model Semantics*, Ph.D. thesis, http://www.tcs.hut.fi/Publications/reports/A58.ps.gz

[22] GNU Licences web page, http://www.gnu.org/licenses/licenses.html

[23] noMoRe main web page, http://www.cs.uni-potsdam.de/ linke/nomore/

[24] cmodels main web page, http://www.cs.utexas.edu/users/tag/cmodels.html

[25] Ilkka Niemel(Ed.) : *WP3 Report: Language Extensions for ASP (draft)*, http://www.tcs.hut.fi/Research/Logic/wasp-wp3-web.html, November 9, 2003

[26] Marina De Vos and D. Vermeir : *A Logic for Modelling Decision Making with Dynamic Preferences*, Proceedings of the Logics in Artificial Intelligence (jelia2000) workshop, pp. 391-406, Springer LNAI 1919, 2000.

[27] Martin Brain and Marina De Vos : *Implementing OCLP as a front-end for Answer Set Solvers: From Theory to Practice*, ASP03: Answer Set Programming: Advances in Theory and Implementation. Ceur-WS, September 2003, http://CEUR-WS.org/Vol-78/asp03-final-brain.ps

[28] NLP main web page, http://www.cs.uni-potsdam.de/ torsten/nlp/

[29] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson and M. Barry : *An A-Prolog decision support system for the Space Shuttle*, AAAI Spring 2001 Symposium, Mar 2001.

[30] S. Costantini and A. Tocciho : *A Logic Programming Language for Multi Agent System*,Proceedings of the Logics in Artificial Intelligence (jelia2002) workshop, 2002.

[31] GNU philosophy web pages,
http://www.gnu.org/philosophy/philosophy.html#AboutFreeSoftware

[32] Debian project web pages, http://www.debian.org/

[33] P. Borchert, C. Anger, T. Schaub, M. Truszczynski : *Towards Systematic Benchmarking in Answer Set Programming: The Dagstuhl Initiative*, Lecture Notes in Computer Science V. 2923, Springer, 2003.

[34] University of Potsdam benchmark generator software, http://www.cs.uni-potsdam.de/ konczak/benchmarks/benchmarks.html

[35] Benchmarks developed for the Logic Programming and Nonmonotonic Reasoning Conference 2001, http://cs.engr.uky.edu/ai/benchmarks.html

[36] Y. Zhao and F. Lin : *Answer Set Programming Phase Transitions: A Study on Randomly Generated Programs*, Proceedings of the Nineteenth International Conference on Logic Programming (ICLP'03), 2003.

[37] A. Aho, R. Sethi and J. Ullman : *Compilers - Principles, Techniques and Tools*, ISBN 0-201-10088-6, Addison Wesley, 1988.

[38] Michael R A Huth and Mark D Ryan : *Logic in Computer Science*, Cambridge University Press, 2002