The University of Edinburgh

Department of Computer Science

On Hereditary Harrop Formulae as a basis for Logic Programming

by

James Harland

CST-81-91

(also published as ECS-LFCS-91-170)

July 1991

James Clerk Maxwell Building,
The King's Buildings,
Mayfield Road,
Edinburgh,
EH9 3JZ.

On Hereditary Harrop Formulae as a Basis for Logic Programming

James Harland

Ph.D. Thesis, University of Edinburgh

July 4, 1991

(Graduation date November 1991)

Abstract

This thesis examines the use of first-order hereditary Harrop formulae, a generalisation of Horn clauses due to Miller, as a foundation for logic programming. As this framework is constructive, this will sometimes dictate an approach which differs slightly from the traditional (classical) one.

We discuss the foundational problems involved in adding negation to the framework of first-order hereditary Harrop formulae, including the role of the Negation as Failure (NAF) rule and the Closed World Assumption (CWA) in a constructive setting, and introduce the notion of a completely defined predicate. This notion may be used to define a notion of NAF for a more general class of goals than literals. We also discuss the possibilities for forms of negation other than NAF, and explore the relationships between NAF and more explicit forms.

Clark's completion of a program is often used in this context, and we show how a more explicit version of the completion may be given in hereditary Harrop formulae. We may think of the completion as specifying a theory in which an atom A fails iff $A \supset \bot$, and hence is an explicit axiomatisation of failure, which in our case is more computationally meaningful than Clark's completion.

The problem of finding answer substitutions for existentially quantified negated goals requires more powerful techniques than unification alone, and so we give an algorithm which is suitable for this purpose, and show how it may be incorporated into the goal reduction process.

A constructive framework necessitates a different approach to model theory, and we give a Kripke-like model for the extended class of programs for which negation is implemented by the Negation as Failure rule. This is based on the model theory developed by Miller for hereditary Harrop formulae. No restriction on the class of programs is used, which requires some departures from the usual T^{ω} process, but the spirit of the construction remains the same.

The Kripke-like model suggests some structural properties of first-order hereditary Harrop formulae which are of semantic interest. One important question is

the precise strength of the class of formulae involved. We consider the redundant features of Miller's language, and show how they may be removed. This leads to a discussion of equivalence for this class of programs, which necessitates the use of an intermediate logic, in which programs which are operationally equivalent are logically equivalent.

Implication in the bodies of clauses also allows a notion of meta-programming within a first-order framework. We explore this possibility to some extent by showing how the application of some of our results allow memoisation to take place, which may be thought of as reflecting meta-level information back into programs by a subtle separation of object and meta- levels. This also demonstrates an elegant connection between removing redundancies from programs and the derivation of a goal in this framework.

Declaration

This thesis has been composed by myself, and the work herein is my own. A preliminary version of some of the results in chapter 5 were presented in [46].

Acknowledgements

"No man is an island, entire of itself ..." – John Donne

There are many people who have helped me on my way in this long and arduous quest. My supervisor, Don Sannella, has been a constant source of advice, encouragement and enthusiasm. His careful reading of various ill-defined and poorly written documents has been a great boon. Dale Miller has been a patient and reliable source of stimulating discussion, often nudging my hastily thought out and sketchy ideas towards enlightenment. Robin Milner and Colin Stirling proved themselves very good at plucking the kernel of wheat out of a mountain of chaff.

Discussions with a number of people have enriched me, and there are far too many to name them all individually, but chief amongst them are Jamie Andrews, Lawrence Cavedon, T.Y. Chen, Amy Felty, Dov Gabbay, John Hannan, Bob Harper, Nevin Heintze, Joxan Jaffar, Ken Kunen, Jean-Louis Lassez, Michael Maher, Thorne McCarty, Spiro Michaylov, Frank Pfenning, Mark Reynolds, and Lincoln Wallen: to you, and to others too numerous to mention, my thanks.

To David Pym, my fellow knight errant and soul-mate, my hearty thanks; it seems your dragon fell before mine.

The Laboratory for the Foundations of Computer Science has been a superb environment in which to conduct my studies, and I am grateful to the Department of Computer Science for allowing me the opportunity so to do. My thanks go to all in the Department for an enthralling time. I was also fortunate to be supported by a Commonwealth Scholarship, administered by the British Council and the Association of Commonwealth Universities.

The Department of Computer Science at the University of Melbourne has been supportive of my efforts and for that I am grateful.

Many others have helped me in various ways: food, shelter, company ... Greg and Amy were kind enough to share their home with me when I was visiting the University of Pennsylvania (and Franç was kind enough to go away!), and Nevin and Tess, as well as the entire Jaffar clan, were generous with their hospitality.

Jordi and Bill have been cheerful and helpful roommates. A healthy body leads to a healthy mind, and I have often dined heartily with Steve and Kay, Mark and Juliet, Mike and Sue, David and Anne, and Philippa. Long may this tradition continue.

My family have been very supportive of my studies and the ensuing distances. Without their support, this enterprise could not have been attempted. In particular, my parents have always been patient, encouraging and helpful when it comes to my studies, and this has been no exception. Thank you all.

No one has had to bear a greater share of the burden than my wife, Lynda, to whose love, care and support no words can do justice.

Ultimately, I thank God, whose goodness knows no bounds, and without whom none of this would be possible.

To my wife, Lynda, who is dedication personified.

Table of Contents

1. Introduction **1**

1.1 Logic and Programming 1

1.2 Computability and Constructivity 3

1.3 Traditional Logic Programming 6

1.4 Constructive Logic Programming 9

2. Hereditary Harrop Formulae and Extensions **16**

2.1 First-order Hereditary Harrop Formulae 17

2.2 Extensions 25

2.2.1 Extensional Universal Quantification 25

2.2.2 Negation in a Constructive Setting 28

2.3 Technicalities 34

2.4 Discussion 55

2.4.1 Terms and Universal Quantification 55

2.4.2 Answer Substitutions 59

2.4.3 Incomplete Definitions and Inconsistency 61

2.4.4 Computational Aspects 64

2.4.5 Stratification 68

2.5 Notation 71

3. Completions and Negation as Failure **76**

3.1 Completions and Provability 76

3.2 The Completion Process 83

3.3 Properties of the Completion 93

4. Answer Substitutions for Negated Goals **110**

4.1 Motivations 110

4.2 Definitions 116

4.3 Preliminaries 120

4.4 The Incremental Algorithm 130

4.5 Answer Substitutions and Induction 139

5. Semantics and Model Theory **145**

5.1 A Kripke-like Model 145

5.2 Worlds and Accessibility 151

5.3 Extending the Framework 158

5.4 The Construction Process 172

5.5 Comparative Results 183

5.6 Problems with Inconsistency 190

6. Semantic Properties of Hereditary Harrop Formulae **195**

6.1 Structural Properties of Programs 195

6.2 Normal Forms and Representation 212

6.3 Minimal Conditions for Operational Equivalence 221

6.4 Notions of Equivalence 224

6.5 Choice of Intermediate Logic 233

6.6 Discussion 250

7. Meta-programming Features of Hereditary Harrop Formulae 254

7.1 Memoisation Properties 254

7.2 Memoisation for Larger Classes of Programs 261

7.3 Programming at the Meta-Level 265

8. Conclusion and Further Work 275

8.1 Completions and Inconsistencies 276

8.2 The Role of Induction 277

8.3 Model Theory 278

8.4 Meta- and Higher-Order Programming 280

Bibliography 281

A. Proof of the Disjunctive Identity 293

List of Figures

5–1 Inverted cone of models 151

5–2 Inverted cone + CWA = diamond 154

5–3 Inverted cone with holes 155

5–4 The append program and its possible extensions 157

6–1 Kripke model in which D_4 is true but D_3 is not 226

6–2 Kripke model in which D_6 is true but D_5 is not 227

A–1 Useful subproof 293

A–2 Proof that $(G_1 \supset A) \wedge (G_2 \supset A) \vdash_I (G_1 \vee G_2) \supset A$ 294

A–3 Proof that $(G_1 \vee G_2) \supset A \vdash_I (G_1 \supset A) \wedge (G_2 \supset A)$ 294

Chapter 1

Introduction

1.1 Logic and Programming

The discipline of mathematical logic arose from a desire for rigour in mathematics during the 1800's. There had been various attempts at systemisation of mathematical thinking before that time, such as the famous 10 axioms of Euclid, but this was the first time that the foundations of mathematics came under intense scrutiny by the community-at-large. This was due to a number of paradoxes and counter-intuitive results which disturbed many mathematicians. Earlier mathematicians often did not feel the need to rigorously define terminology and give formal detailed proofs of every last step in their line of thought. Many felt that the use of infinitesimals needed no justification, whereas the analysts of the 19th century worked to expel them from the vast body of mathematical reasoning. Such a vigorous desire for a minimum standard of rigour led to the evolution of general principles of reasoning, which in turn led to the formal development of the concepts used in mathematics.

The widespread interest in foundational issues led to a variety of important developments – Frege's scheme for the "laws of thought", set theory as conceived by both Zermelo and Fraenkel, and Hilbert and Bernays, Zermelo's Axiom of Choice, Russell and Whitehead's Principia Mathematica, to name but a few. Mathematical logic grew out of this flurry of results into a well-defined discipline, which could serve as a backdrop to any mathematical debate. If a dispute arose over the validity of a supposed proof, then mathematical logic was designed to be the arbiter of the dispute. The claimant would need to demonstrate that his or her proof

was valid with respect to the logical foundations of the theory under discussion, and the doubter would need to demonstrate that the proof was faulty by the same strict standards. Thus, a universal method had been found by which all manner of mathematical disputes may be settled.

There is a vast amount of literature on mathematical logic, and we do not attempt a general introduction here. The interested reader may find further information in [51,71,9], and many other works.

A similar desire for rigour may be detected amongst computer scientists today. There is a growing realisation that programming is a more varied, widespread and complicated task than originally conceived, and that experience has suggested that rigour is needed in the definition of programming tasks, in order to avoid annoying and costly mistakes. Proofs of program correctness are important for large and/or critical systems, and for the comprehension of complex algorithms. Formal and definitive semantics for programming languages is important for the maintenance and alteration of programs, and for portability considerations, so that the language, and hence the application, need not be tied to a particular architecture. The design and analysis of hardware may also be improved by the use of formal techniques. As computer applications become more and more widespread, and the reliability, speed and availability of hardware continues to increase, the need for formalism will not diminish.

It is perhaps not surprising that the same era which produced some fundamental advances in mathematical logic was also the era in which the foundations of modern computing were laid. The 1930's and 40's produced Gödel's Incompleteness theorems as well as the world's first digital computer. Turing's conception of a universal computing machine, the λ-calculus, and recursive function theory all emerged around the same time that Church proved the undecidability of first-order logic. Not surprisingly, there are some obvious similarities between some of these results. The undecidability of the halting problem for Turing machines may be considered as stating that the halting problem is undecidable for any machine model in which the machines have sufficient power. For example, the halting problem is decidable for finite state automata, which are less powerful than Tur-

ing machines. Similarly, Gödel's first incompleteness theorem may be considered as stating that any sufficiently powerful mathematical theory will contain statements which are true but not provable. The proof of this result uses recursive function theory, which in itself may be used as a basis for computation, as well as an encoding of statements of the theory as natural numbers. Such an idea of encoding formal statements of mathematics in a syntactical manner would be instantly familiar to a computer programmer.

One of the best examples of the close relation between logic and programming is given by proof theory. One may consider the work of Gentzen in this area as systematising the notion of proof to such an extent that any idiot can understand any proof, given enough patience. Such a detailed specification is precisely how a programmer instructs a machine.

1.2 Computability and Constructivity

Naturally the availability of an umpire did not stop mathematicians arguing. Whilst the umpire's decisions were irrefutable, there was still a dispute about who the umpire should be. There were three distinct schools of thought: the formalists, whose champion was Hilbert, the logicists, whose champion was Russell, and the intuitionists, with Brouwer as their champion. Briefly, the logicists were Platonists, believing in a divinely beautiful mathematical world, which mankind was free to discover, whereas the formalists believed that mathematics was, in essence, a game of symbol-pushing, correct with respect to itself, but with no external interpretations.

Brouwer could accept neither of these views. He felt that there had to be something intuitive about mathematics, and hence it could not be as the formalists believed, but that mathematics had to be constructive as well, in that one needed to be able to exhibit a given mathematical object in order to discuss it meaningfully. A famous example involves the decimal expansion of π; if we consider a number a whose value is 0 if a sequence of one thousand 5's does not appear

anywhere in the decimal expansion of π, and otherwise is 1 if the sequence begins at an even digit, and -1 if the sequence begins at an odd digit. To an intuitionist, it is meaningless to discuss such a number, as it requires that the decimal expansion of π be written out in full and examined as one entity. If at some stage in the future such a sequence is discovered, then the matter may be settled, but before such an event happens, if ever, it makes no sense to an intuitionist to discuss the value of a. A follower of the other schools may say that a exists, but its value is not currently known, and may never be known. In this way an intuitionist requires that an explicit, unambiguous construction be given for a number before he will accept that it exists.

Brouwer's critique of classical mathematics centred around the treatment of infinite sets. It is clearly acceptable to apply the law of excluded middle to finite sets, so that given a finite set S and a number x, it is true that either $x \in S$ or $x \notin S$. An obvious way to test the truth of this proposition is to list all the elements of S, and check whether or not x occurs in that list. Clearly this process must terminate. However, the same technique cannot be applied to infinite sets, as it is not possible to finitely enumerate all the elements, and so to Brouwer, applying the law of excluded middle to infinite sets was akin to examining an infinite set after all its elements had been enumerated, and hence was unacceptable. Unless there is some way to finitely represent the infinite set, the law of excluded middle could not be true. In this way one may obtain intuitionistic logic from classical logic by removing the law of excluded middle and all its consequences. More on intuitionistic logic may be found in [21,51,42,64].

To the modern mind, Brouwer's arguments are those of a believer in algorithms, so that in order to demonstrate an object's existence, one must produce an algorithm which constructs the object. In this way one may view the intuitionist's thinking as one like that of doubting Thomas; the only way to convince him is to produce the given object in front of his own eyes, rather than try to convince him on the grounds that it couldn't be otherwise.

One of the more compelling properties of intuitionism in this context is the way that proof is identified with truth. For example, in the Tarksi semantics for

classical logic, $A \vee B$ is true iff A is true or B is true. In Heyting's conception of intuitionistic proof, a proof of $A \vee B$ is a pair (i, p) such that if $i = 0$, p is a proof of A, and if $i = 1$, p is a proof of B, and so $A \vee B$ is provable iff A is provable or B is provable. Hence, in order to claim that a statement is true, it is necessary to find an appropriate "proof object" which proves the statement, rather than to perform a model-theoretic construction. Much ink has been spilt over the precise definition of proof objects, but the point here is that there is a direct proof-theoretic interpretation of truth, which seems to be more in keeping with the spirit of computing than the assignment of truth values.

This ideal is often reflected in the way certain mathematical properties are viewed. For example, a proof that an odd perfect number exists would be less satisfying than exhibiting a certain odd number and proving that it is perfect. In a similar vein, it is just about unthinkable that it is possible to prove that there is a polynomial-time algorithm for 3-SAT without explicitly producing such an algorithm, and any such proof would be considerably less rewarding than a constructive one.

The close link between computability and constructivity is illustrated by the trend in computer science in recent years towards constructive logics (i.e. logics in which only constructive conclusions may be reached, of which intuitionistic logic is one). Martin-Löf type theory [70] is one example, as is the Edinburgh LF [47, 97]. Linear logic [42] is another recent development, about which Girard explicitly stated that the idea was to recapture the spirit of intuitionistic logic. It has also recently been shown [20] how a constructive interpretation of recursive function theory may be more appropriate than the classical one. This trend is not surprising given the natural inclination of computer science towards constructive ideas. Just as constructions with a compass and straightedge are natural tools of Euclidean geometry, so are algorithms, i.e. constructive proofs, natural tools of computer science.

1.3 Traditional Logic Programming

There is some confusion about the early years of logic programming, and about when the first logic programming language was implemented. Logic programming was certainly a consequence of Robinson's seminal paper on unification and theorem proving [101], and it now seems that the AbSys system was probably the first implementation of a logic programming language [24,23]. Certainly the most influential implementation was that of the programming language Prolog by Colmerauer and his colleagues at Marseilles [19]. However, the real impetus behind logic programming was given by Kowalski in his seminal paper of 1974 [52]. Since then, Prolog has been found to be a useful and practical programming language for many applications [18,106].

Here we give the essentials; more details and proofs may be found in [61]. We will use $F\theta$ to stand for the application of the substitution θ to the formula F. $\exists(F)$ stands for the existential closure of all free variables in F, and similarly $\forall(F)$ stands for the universal closure of all free variables in F.

Kowalski's key idea was that the Prolog clause

$$A \text{ :- } B_1, \ldots, B_n$$

where A and the B_j are atoms, may be interpreted both *declaratively* and *procedurally.* The declarative interpretation was given by considering the above clause as a shorthand for the formula

$$\forall x_1 \ldots x_m \; A \subset B_1 \wedge \ldots \wedge B_n$$

where x_i are all the variables appearing in A and the B_j. Such a formula is known as a *Horn clause.* A *program* is a finite set of Horn clauses. The procedural interpretation is given by interpreting the above clause as a specification of a procedure, whose "head" is A and hence names the procedure being defined, and

whose "body" specifies calls to further procedures $B_1, \ldots, B_n$. Input and output is to be done through the variables of A and the body. A sequence of procedure calls, then, is a conjunction of atoms, and execution takes place by starting with an initial number of procedure calls specified by

$$G_1, \ldots, G_k$$

where the G_i are atoms, generating more procedure calls by matching each G_i against clauses in the program and proceeding until the subsequent calls are exhausted. The initial sequence of calls is known as the ***goal***, and may be thought of in declarative terms as the formula

$$\exists y_1 \ldots y_r \; G_1 \wedge \ldots \wedge G_k$$

where the y_i are all the variables appearing in the G_j. In this way computation may be interpreted as the manipulation of a sequence of first-order formulae. Not surprisingly, such operations have a direct connection to proofs of the goal, and in particular to a certain method of finding a proof of the goal. This method was related to the resolution methods of automatic theorem proving, and has become known as SLD-resolution (or LUSH-resolution).

This process may be defined as follows: given a goal

$$G_1, \ldots, G_k$$

we use a *computation rule* R to choose G_j for some $1 \leq j \leq k$, which is known as the *selected subgoal.* Next, if there is a clause A :- $B_1, \ldots, B_n$ (where $n \geq 0$) in the program such that $A\theta = G_j\theta$ where θ is the most general unifier (mgu) of A and G_j, then we replace $G_1, \ldots, G_k$ by

$$(G_1, \ldots, G_{j-1}, B_1, \ldots, B_n, G_{j+1}, \ldots, G_k)\theta$$

and continue. Due to the fact that n may be 0, this goal may be smaller in size than the original, and so termination occurs when the goal is empty. As

there may be many choices for clauses with heads which match G_j, for a given computation rule R we define an *R-computed SLD-tree*, whose root is the initial goal, and in which the children of each node are the results of applying the above transformation to the parent, with one choice for each possible matching clause head. The leaves of this possibly infinite tree may be either the empty goal or goals for which there is no clause head which matches the selected subgoal. A branch from the root to a leaf containing the empty goal is known as a *successful SLD-derivation.* A branch from the root to a leaf containing a non-empty goal is known as an *unsuccessful SLD-derivation.* If $\theta_1, \ldots \theta_n$ are the substitutions used in a successful SLD-derivation of the goal G, then the *answer substitution* θ is the composition of $\theta_1, \ldots \theta_n$. This may be thought of as finding a goal $G\theta$ such that $G\theta$ succeeds, and so we refer to $G\theta$ where θ is an answer substitution as an *answer* for G.

Now as the declarative reading of the program and goal are both formulae of first-order logic, we may ask what connection SLD-derivations have with proof theory. It is possible to show that this procedure is sound and complete with respect to provability in classical logic of goals from Horn clauses; that is, a goal G has a successful SLD-derivation from a program P iff $P \vdash_C G$ [61].

Having established a proof-theoretic notion of consequence, it is natural to ask about model theory. Clearly, one such notion may be given by the standard model theory of first-order (classical) logic. It is not hard to see that any model in which P is true should also have G true whenever G has a successful SLD-derivation from P. With this in mind, we say that G is a *logical consequence* of P iff G is true in all models of P. It is clear that this will be satisfied if G is true in a minimal model of P, and such a minimal model may be given by the *least Herbrand model* of P. It should be clear that we expect G to be true in the least Herbrand model iff G has a successful SLD-derivation. This is indeed the case, and is proved by constructing the least Herbrand model as follows. This construction is due to Kowalski and van Emden [25].

Firstly, for a given program P, we define an operator T_P mapping from and into interpretations, as

$$T_P(I) = \{A \mid \text{there is a ground instance } A \subset B_1 \wedge \ldots \wedge B_n \text{ of a clause in } P \text{ such that } B_i \in I \text{ for all } 1 \leq i \leq n\}$$

$T_P(I)$ may be thought of as all things which may be deduced from P and the assumptions I in one step, so that T_P is sometimes referred to as the "immediate consequence" operator. We then define ordinal powers of T_P as follows:

$$\begin{aligned} T_P \uparrow 0 &= T_P(\emptyset) \\ T_P \uparrow n &= T_P(T_P \uparrow (n-1)) \\ T_P \uparrow \omega &= \bigcup_{n=1}^{\infty} T_P \uparrow n \end{aligned}$$

The least fixpoint of T_P turns out to be the least Herbrand model, which has an elegant ring about it, in that one starts from no assumptions and repeatedly applies the T_P operator until no new deductions are made, and that the set of deduced facts is then precisely the consequences of the program. This idea was extended in [5].

In this way Horn clauses were used as a logic programming language because of the two possible interpretations: one as formulae, and the other as specifying sequences of top-down operations.

1.4 Constructive Logic Programming

In the previous section we saw how a goal G has a successful SLD-derivation from P iff $P \vdash_C G$. In the light of the previous discussion, it would seem that it is natural to ask what role intuitionistic logic may play in this scheme. It is not hard to see that classical logic and intuitionistic logic coincide for Horn clauses, i.e. $P \vdash_C G$ iff $P \vdash_I G$ when P is a set of Horn clauses [81]. This may be easily seen from the proof rules which are needed to manipulate Horn clauses and goals. As this class of formulae is not a very large fragment of first-order logic, not many are needed, and all are intuitionistically valid. Certainly the law of excluded middle, i.e. that $F \vee \neg F$ is true for any formula F, is not needed, as there are no negations

in either goals or programs. Thus SLD-resolution may be distinguished from other forms of resolution in that the class of formulae on which it is defined is restrictive enough that only constructive consequences of the program may be derived.

One way to interpret SLD-derivations as proofs is to think of the derivation as a *refutation*. Each step in the SLD-derivation may be considered as a valid deduction in classical logic, and hence the overall derivation may be interpreted as a proof.

Given a program P, and a goal $G = \exists(G_1 \wedge \ldots \wedge G_k)$, we re-write the clauses in P so that they are of the form $\forall(A \vee \neg B_1 \vee \ldots \vee \neg B_n)$. Now assume that G has a successful SLD-derivation from P. We imitate this derivation by a proof in first-order classical logic as follows. First we assume that $\neg G$ is true, i.e. we assume $\forall(\neg G_1 \vee \ldots \vee \neg G_k)$. Let G_j be the selected subgoal, so that $G_j\theta = A\theta$, where $A \vee \neg B_1 \vee \ldots \vee B_n$ is a clause in the program, let $\neg B_1 \vee \ldots \vee \neg B_n$ be C, and let $\neg G_1 \vee \ldots \vee \neg G_{j-1} \vee \neg G_{j+1} \vee \ldots \vee \neg G_k$ be G', so that $\neg G = \forall(G' \vee \neg G_j)$. By our assumption, $\forall(G' \vee \neg G_j)$ is true, and hence so is $\forall(G'\theta \vee \neg G_j\theta)$, and from the program P we get that $\forall(G_j\theta \vee C\theta)$ is true. Now if $G_j\theta$ is true, then $\neg G_j\theta$ is false, and so from the truth of $\forall(G'\theta \vee \neg G_j\theta)$ we get $\forall(G'\theta)$. Otherwise, $G_j\theta$ is false, and so as $\forall(G_j\theta \vee C\theta)$ is true, we have $\forall(C\theta)$ is true. In either case, we may deduce $\forall(G'\theta \vee C\theta)$ is true, which is just the next step in the derivation. Hence, each step matches up with a correct deduction in classical logic.

When the end is reached, the goal is of the form $\forall(\neg B)$ where B is an atom, which matches some atom A in the program such that A has no body. As $A\theta = B\theta$ for some θ, we get $\neg B\theta \wedge B\theta$, a contradiction. Hence, our initial assumption of $\neg G = \neg\exists(G_1 \wedge \ldots \wedge G_k)$ leads to a contradiction, and so the final step is to deduce that $\exists(G_1 \wedge \ldots \wedge G_k)$ is true.

This argument is not valid intuitionistically, as both the final step (deducing G from the fact that $\neg G$ leads to a contradiction) and the conversion of the clauses in the program do not hold in intuitionistic logic. Hence the refutation explanation can only be valid for classical logic.

An alternative explanation of SLD-resolution, which is intuitionistically valid,

is that a successful SLD-derivation represents a search, rather than a proof itself. Each step in the SLD-derivation represents a step in the overall search for a proof, and the search space may be defined by the notion of an *SLD-tree*. From a successful search it is easy to find a proof that $P \vdash G$. From the results of [81] it is clear that this proof is intuitionistically valid. This explanation seems to be more in keeping with Kowalski's original idea of the notion of a clause being interpreted as specifying a number of procedure calls, which has no obvious connection to the refutation interpretation.

The presence of negations in the refutation also seems to be somewhat inappropriate, as Horn clauses do not contain any negations. The proof search explanation does not need negations, and hence seems more aesthetically pleasing. Another criticism of the refutation explanation is that the role of the answer substitution is obscured, when in reality this is an important part of logic programming. As noted earlier, variables are used for input and output in a goal, and so the only way to produce output is to bind a variable in the goal, i.e. produce an answer substitution. It has been noted that "the purpose of a logic programming system is to compute bindings" [61]. Hence, it seems that the answer substitution should be interpreted as an important part of the answer, rather than the by-product of a search to produce yes or no.

This is precisely the case in intuitionistic logic, as in order to prove a goal $\exists x G$, one must exhibit a term t such that $G[t/x]$ is true. Hence the refutation interpretation obscures an essential property of a logic programming system, but in intuitionistic logic, the answer substitution is given an important place. This suggests that the property which makes Horn clauses appropriate for logic programming is not so much the existence of a resolution method for finding proofs as the fact that classical logic and intuitionistic logic coincide on these formulae, thus ensuring that only constructive conclusions may be reached.

An interesting observation is that intuitionistic logic has a similar property for disjunctions; $A \vee B$ is provable iff A is provable or B is provable. This matches up precisely with what happens in Prolog, in that many implementations of Prolog

allow disjunctions, and determine whether there is a proof of the disjunction in precisely the manner specified above.

A recurring feature in the semantics of programming languages is the execution of a program may result in one of three states: termination with success, termination with failure, and non-termination. In logic programming terms, this means that a goal may either succeed, fail or loop. The third case corresponds to the existence of infinite branches in the SLD-tree. Because of this property, Kleene's idea of 3-valued logic [51] has often been cited as a possible way to weaken classical logic in order to give a more appropriate semantics for logic programming and there have been several investigations along these lines [29,30,32,54,55,56,58]. This may be thought of as weakening the mapping of formulae to truth values from a total function varying over 2-valued truth (i.e. each formula is mapped to either true or false) to a total function on 3-valued truth (so that every formula is mapped to true, false or $\perp$). In intuitionistic logic, this mapping may be thought of as a partial, rather than total, mapping from formulae to 2-valued truth, and hence is a somewhat different approach. The mapping is only partial because the proof system is only partial; first-order intuitionistic logic is not decidable. Hence, we use an implicit approach to this problem, rather than specify an explicit third value. Also, the three valued approach does not explicitly address the answer substitution property.

For these reasons it seems that a constructive analysis of logic programming may be useful. Now whilst Horn clauses are sufficient for any computational purpose [108], it is desirable to extend the class of formulae which may be used as a programming language. When considering such extensions, an analysis in terms of the differences between classical logic and intuitionistic logic will be useful, as by using larger and larger classes of formulae of first-order logic we will eventually reach a point at which classical logic and intuitionistic logic differ. As we believe the constructive interpretation is the more natural one, we investigate such extensions from an intuitionistic viewpoint.

Such extensions have been given by Gabbay et al. [37,35,38], McCarty [65,66],

and Miller et al. [81,82]. All are extensions to Horn clauses, and involve using implications in the bodies of clauses.

The QN-Prolog system of Gabbay and his colleagues was motivated by the formalisation of the British Nationality Act [104], and requires some seemingly unnatural restrictions on the variables involved. For example, unlike the other two systems, universally quantified variables may not appear in the body of clauses.

McCarty's extension to Horn clauses is based on clauses of the form

$$P \subset \neg Q$$
$$P \subset (Q \supset R)$$

where the variables of P are assumed to be universally quantified at the front of the clause, and the variables of Q which are not so quantified are universally quantified at the front of the body. It is shown how a proof system may be given in which the converses of the above formulae may also be used as clauses.

The framework of Miller et al. uses a class of formulae known as hereditary Harrop formulae, which, apart from formulae which include negations, is more expressive than both of the other systems. For example, the following is a hereditary Harrop formula, but is not a legal clause in either of the other systems.

$$p \subset \forall x \exists y \forall z \; q(x, y, z)$$

Another way in which this framework differs from the other two is the amount of operational detail that is specified. Both Gabbay et al. and McCarty spend a significant amount of time discussing and describing the ways in which the appropriate proof procedures may be implemented, whereas Miller et al. provide a more abstract operational semantics, which simplifies the presentation somewhat. Clearly it is important to understand how the features of a programming language are to be implemented, but a detailed examination of the issues involved is beyond our scope. As we wish to explore the connection between logic and programming rather than operational issues per se, we adopt the approach of Miller et al. towards extensions to Horn clauses.

Hereditary Harrop formulae may be generalised to higher-order formulae, which may then be used as a higher-order logic programming language, known as λProlog [79,81,86,87]. This language has applications to program transformations [44,45], theorem proving [27], and computational linguistics [80], as well as to modules [77] and lexical scoping [78] in which only the first-order part is needed.

The central notion behind hereditary Harrop formulae is the idea of a *uniform* proof, which may be thought of as a proof which is essentially determined by the structure of the formula proved. It is shown in [81] how uniform proofs are sound and complete for various classes of formulae and notions of provability. In particular, it is known that for first-order hereditary Harrop formulae, uniform proofs are sound and complete with respect to intuitionistic provability [81,82]. There is also a model theory for a large fragment of hereditary Harrop formulae, which seems to be an elegant semantics for this class of programs. This model theory also captures the growth of programs in a natural way, generalises the traditional fixpoint semantics, and, being a possible world semantics, has clear connections to other model theoretic constructions. It is shown in [77] how the operational notion of derivability coincides with the model theoretic one.

We examine this framework as a basis for logic programming, and look at how extensions, such as Negation as Failure [17,100], may be incorporated into it. In chapter 2 we review the preliminary concepts and definitions, and discuss various foundational issues, including the possibilities for negation and the ways to implement universal quantification in goals. We show how a version of Negation as Failure may be incorporated into hereditary Harrop formulae.

In chapter 3 we examine the role of the completion of a program [17], which may be thought of as adding negative information to the program, so that we may derive negative information explicitly, rather than implicitly as in Negation as Failure. We show how the structure of hereditary Harrop formulae make it possible to consider the completion as a program (i.e. a set of clauses) rather than just a formula of first-order logic, and that the completion correctly captures the computational properties of the program.

In the following chapter we discuss how answer substitutions may be computed

for negated goals. The usual unification methods are not sufficient for this task, and so we present and prove correct an algorithm which will be useful in this regard. This algorithm is *incremental*, in the sense that new information may be incorporated without recomputing from scratch.

Chapter 5 deals with model theory, and in particular how to incorporate negation into the model theory for hereditary Harrop formulae as given by Miller [77]. In this case the model in question is similar to a Kripke model, but with a slight difference. We show how the usual T construction [5,25] may be defined in the presence of negation, and that the procedural semantics coincides with the declarative semantics given by this model. No restriction on the class of programs is needed for this process.

In chapter 6 we show how the structural properties of logic programs may be exploited in order to derive a normal form for programs and goals. We may think of a normal form as a formula which is engineered to give a maximal amount of information with a minimal number of constructs. This normal form leads to a discussion of equivalence between programs and goals, and we show that a logic slightly stronger than intuitionistic logic is needed in order to capture the natural notion of equivalence in this context, i.e. that operationally equivalent programs are logically equivalent.

Finally in chapter 7 we give some applications peculiar to hereditary Harrop formulae as opposed to Horn clauses. These include the possibilities for memoisation and a first-order notion of meta-programming, which is made possible by the presence of implications in the bodies of clauses.

Chapter 2

Hereditary Harrop Formulae and Extensions

In this chapter we introduce the foundational issues involved in adding negation to the framework of first-order hereditary Harrop formulae. Section 2.1 deals with the definitions and basic properties of this framework as defined by Miller et al. [82], and Section 2.2 discusses extensions to this framework. Here we also discuss out motivation for interpreting universal quantifiers extensionally. Then we discuss the role of the Negation as Failure (NAF) rule and the Closed World Assumption (CWA) in a constructive setting, and introduce the notion of a completely defined predicate, before presenting the formal extensions necessary to incorporate the above features in Section 2.3. We also show how the notion of completely defined predicates may then be used to define a notion of NAF for a more general class of goals than literals. In Section 2.4 we discuss the technicalities involved in our interpretation of the universal quantifier, as well as the problem of producing answer substitutions for existentially quantified goals. We also discuss the possibilities for forms of negation other than NAF, and explore the relationships between NAF and more explicit forms. Finally we discuss the role of stratification in our approach to the model theory for programs which use the NAF rule.

2.1 First-order Hereditary Harrop Formulae

In this section we present the basic definitions and results pertaining to first-order hereditary Harrop formulae, which may be found in [82,81,87]. Many of our examples are drawn from the same sources.

The class of first-order hereditary Harrop formulae may be defined as follows:

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \forall x G \mid \exists x G \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid D \supset G$$

A program is any set of closed D formulae, and a goal is any closed G formula. We often refer to D formulae as *definite* formulae, and G formulae as *goal* formulae. We denote by $\mathcal{D}$ the set of all D formulae, $\mathcal{P}$ the set of all programs and the set of all G formulae by $\mathcal{G}$. As in the Horn clause case, computation is performed by trying to find a proof that a given goal G follows from a given program P. One significant difference between this approach and the traditional Horn clause methods is that a richer set of search primitives needs to be used. The refutation interpretation of SLD-resolution is not only not intuitionistically valid, it is also difficult to see how it may be extended to hereditary Harrop formulae. Hence the connection between logic and programming in this context can only be given by interpreting a goal formula G as both a formula of first-order logic and a series of instructions to be performed in a search space which is richer than that for Horn clauses. It is also somewhat aesthetically pleasing not to refer to negated formulae in order to explain the behaviour of a class of formulae which do not contain negations. Thus there is no analogue of SLD-refutation, but only a generalisation of the notion of an SLD-derivation.

The relevant search space interpretation is given by defining a consequence relation $\vdash_o$ on $\mathcal{P} \times \mathcal{G}$ as given below.

- $P \vdash_o G_1 \vee G_2$ iff $P \vdash_o G_1$ or $P \vdash_o G_2$

- $P \vdash_o G_1 \wedge G_2$ iff $P \vdash_o G_1$ and $P \vdash_o G_2$
- $P \vdash_o \exists x G$ iff $P \vdash_o G[t/x]$ for some ground term t
- $P \vdash_o \forall x G$ iff $P \vdash_o G[c/x]$ where c is a new constant
- $P \vdash_o D \supset G$ iff $P \cup \{D\} \vdash_o G$

This gives us a top-down definition of the computational process, so that given the goal $p \wedge \exists x\, q(x)$ we proceed by finding a proof of p and a proof of $\exists x\, q(x)$, each of which we may further reduce according to the rules above. Note that there is no explicit reference to the details of the operational processes, such as unification or how to choose between alternatives in the search space. This allows us to consider the above definitions as a general prescription for the behaviour of logic programming languages, rather than as an explicit definition of a specific language.

Note that there is no specification above about how to establish $P \vdash_o A$. We give below a definition of this in accordance with the above principles of generality.

Let P be a set of definite formulae. We define $[P]$ as the least set of definite formulae which satisfies the following conditions:

1. $P \subseteq [P]$
2. If $\forall x D \in [P]$ then $D[t/x] \in [P]$ for all terms t
3. If $D_1 \wedge D_2 \in [P]$ then $D_1 \in [P]$ and $D_2 \in [P]$

We can now define $\vdash_o$ as the smallest relation satisfying the above conditions and

$$P \vdash_o A \text{ iff } A \in [P] \text{ or there is a formula } G \supset A \in [P] \text{ such that } P \vdash_o G$$

The definition of $[P]$ indicates an important point: we are interested in validity with respect to a fixed set of terms, rather than validity with respect to all possible

sets of terms. We denote the set of all ground terms as $\mathcal{U}$, which will be referred to as the *Herbrand Universe.* Given this set of terms, we refer to the set of all ground atoms as $\mathcal{H}$, known as the *Herbrand base.*

A proof system for these formulae may be given by the standard sequent calculus of cut-free proofs for intuitionistic logic, so that $P \vdash_o G$ iff there is a proof in this calculus of the sequent $P \longrightarrow G$. Initial sequents are of the form $P \longrightarrow A$ where $A \in [P]$. We use the usual notational convention that the set on the left hand side of $\vdash$ is written as a sequence of formulae. The rules for this calculus are the standard ones for intuitionistic logic (which may be found, amongst other places, in [81]). The important property of this class of formulae is that the proofs are *uniform*; that is, in a proof of $P \longrightarrow G$, the top-level connective of G is introduced in the last step of the proof. Thus a proof of $P \longrightarrow G_1 \wedge G_2$ must have as its immediate predecessors the sequents $P \longrightarrow G_1$ and $P \longrightarrow G_2$. In this way uniform proofs capture the search space properties discussed above.

The advantage of this class of proofs is that the search space rules are sound and complete with respect to uniform proofs, i.e. it is clear that $P \vdash_o G$ iff $P \longrightarrow G$ has a uniform proof. Moreover, uniform proofs are sound and complete with respect to intuitionistic proofs of sequents of the form $P \longrightarrow G$, as is stated in the following theorem (Theorem 4 in [81]):

Theorem 2.1.1 *Let P be a program and G be a goal. Then*

$$P \vdash_I G \Leftrightarrow P \vdash_o G$$

The proof may be found in [81]. The importance of this result is that we only need to consider uniform proofs in order to determine whether a given goal follows from a program. For example, consider the goal $G_1 \wedge G_2$. We know that if we can show that G_1 and G_2 both follow from the program, then $G_1 \wedge G_2$ does, but the uniform proof property enables us to reach the stronger conclusion that if we do not find a proof by this method, then there is no proof of $G_1 \wedge G_2$ from the program. So the proof search procedure described above for a goal G is the *only* way to find a proof of G, and it is this lack of ambiguity which allows us to give

a computational interpretation of formulae, and thus make a direct connection between mathematical logic and programming. In this way we may explain the computational process in terms of proof in that a successful search corresponds to the discovery of a uniform proof of a goal, and that a failure in the search corresponds to the discovery that the goal has no uniform proof.

Uniform proofs are often referred to as O-proofs, both here and in the literature, due to this operational interpretation of provability.

Given the consequence relation $\vdash_o$, which is a representation of the desired properties of a logic programming system, we may ask whether this consequence relation is equivalent to some well-known consequence relation that has been studied previously. In addition to the theorem quoted above, the following result is shown in [81] (an immediate consequence of Lemma 10):

Proposition 2.1.2 *Let P be a program and G be a goal. Then*

$$P \vdash_I G \Leftrightarrow P \vdash_M G$$

where $\vdash_M$ is the standard consequence relation of minimal logic.

It follows immediately from this proposition and the theorem above that $P \vdash_o G$ iff $P \vdash_I G$ iff $P \vdash_M G$. It is also pointed out in [81] that $\vdash_C$, the standard consequence relation of first-order classical logic, is too strong to be useful in this context, as there are proofs in classical logic for which there are no uniform proofs. This is expressed in the following proposition.

Proposition 2.1.3 *Let P be a program and G be a goal. Then*

$$P \vdash_C G \not\Rightarrow P \vdash_o G$$

A counterexample to $P \vdash_C G \Rightarrow P \vdash_o G$ is the program $(p(a) \wedge p(b)) \supset q$ and the goal $\exists x(p(x) \supset q)$. In [77] it is shown that

$$(p(a) \wedge p(b)) \supset q \vdash_C \exists x(p(x) \supset q)$$

but it is clear that there is no uniform proof of this goal from this program, and so it is not the case that

$$(p(a) \wedge p(b)) \supset q \vdash_o \exists x(p(x) \supset q)$$

A similar problem is encountered with the goal $p \vee (p \supset q)$, as $\vdash_C p \vee (p \supset q)$, but it is not true that $\vdash_o p \vee (p \supset q)$. Thus classical logic is too strong to capture the search space interpretation discussed above, which seems fundamental to the concept of logic programming. Perhaps the best example of this phenomenon is given by the classical equivalence

$$G_1 \vee (D \supset G_2) \equiv_C (D \supset G_1) \vee G_2$$

As both of these formulae are equivalent in classical logic to $\neg D \vee G_1 \vee G_2$, interpreting the goal $G_1 \vee (D \supset G_2)$ as a formula of first-order classical logic destroys the direct correspondence between proofs of the goal and the search space interpretation. The search interpretation suggests that we may think of the implication $D \supset G_2$ as an instruction to load in the code stored in a module D, and evaluate the goal G_2 in this larger environment, whereas G_1 is to be evaluated without this extra code (this idea is discussed and developed in [77]). This seems to contradict the above classical equivalence, which suggests that it does not matter which goal is evaluated in the extended environment. Hence, classical logic derives too many equivalences to allow the direct and natural association between proof and search spaces that is possible in some weaker logics, such as intuitionistic logic or minimal logic. In this way the search space characterisation of a logic programming system excludes the possibility of using $\vdash_C$ to analyse first-order hereditary Harrop formulae.

Note that first-order hereditary Harrop formulae preserve the existential property, i.e. that if $P \longrightarrow \exists x G$ has a uniform proof, then there is a term t such that $P \longrightarrow G[t/x]$ has a uniform proof. As in the Horn clause case, we will often refer to $G[t/x]$ as an *answer* for the goal $\exists x G$.

Note also that, just as in the Horn clause case, variables which appear in the head of a clause may be considered existentially quantified when the body of the

clause is used as a new goal. This is due to the fact that we are searching in a top-down fashion. For example, consider the goal $\exists x\, p(x)$ and the program

$$q(a)$$
$$\forall x\; q(x) \supset p(x)$$

We first match the goal against the second clause, and produce the next goal $\exists x\, q(x)$, which is clearly a valid step as if $\exists x\, q(x)$ and $\forall x\, q(x) \supset p(x)$ are true, then $\exists x\, p(x)$ is true. Now $\exists x\, q(x)$ succeeds with the answer $q(a)$, and so the original goal succeeds with the answer $p(a)$. Hence, once the head of the clause has been "passed", we may consider all free variables of the body to be existentially quantified.

One interesting structural property of first-order hereditary Harrop formulae is that we may define the D formulae as above or in the equivalent fashion

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset D$$

which conveys the essential symmetry between implications which occur in programs and those which occur in goals. That this is equivalent to the previous definition may be seen by the equivalences

$$G \supset (D_1 \wedge D_2) \equiv_I (G \supset D_1) \wedge (G \supset D_2)$$
$$G_1 \supset (G_2 \supset D) \equiv_I (G_1 \wedge G_2) \supset D$$
$$G \supset \forall x D \equiv_I \forall x\, G \supset D$$

where x is not free in G. The first form of the definition shows the connection to Horn clauses more concretely, and allows easier formal manipulation and so it is generally preferred. The second form is more useful when considering connections between first-order hereditary Harrop formulae and full first-order logic.

Another useful application of the second form occurs when we consider the class of formulae which are both D and G formulae. These are denoted as M formulae, and are often referred to as *core* formulae. They may be defined as follows:

$$M := A \mid \forall x M \mid M_1 \wedge M_2 \mid M_1 \supset M_2$$

By the above equivalences, these are the same as the class of formulae defined by replacing $M_1 \supset M_2$ in the above definition by $M \supset A$. As these formulae may be used both as programs and goals, if an M formula is asked as a goal and succeeds, we may "store" this result by adding it to the program. Note that this class of formulae includes Horn clauses. We will have more to say about this class of formulae later.

Another observation that may be made about the structure of first-order hereditary Harrop formulae is that we may think of programs as sets of clauses, where a clause C is any closed formula satisfying

$$C := A \mid \forall x C \mid G \supset A$$

with G formulae defined as above. This allows us to consider a program as a set of clauses without sacrificing any expressive power, as $\forall x\,(D_1 \wedge D_2)$ is equivalent to $(\forall x D_1) \wedge (\forall x D_2)$, and so we may push conjunctions outwards until we arrive at a conjunction of clauses. We may think of this conjunction as a set of clauses, which is often more convenient. The definition below makes formal the notion of the head of a clause.

Definition 2.1.1 *Let C be a clause, A be an atom, G be a goal, and D be a definite formula. Then*

$$\begin{aligned} head(A) &= A \\ head(\forall x D) &= head(D) \\ head(G \supset A) &= A \end{aligned}$$

$$heads(D) = \{A \mid \exists \text{ a clause } C \in [D] \text{ with } head(C) = A\}$$

We may then define the set of clauses which correspond to a definite formula D as follows:

Definition 2.1.2 *Let D be a definite formula. Then we define*

$$\begin{aligned}
\text{clausal}(A) &= \{A\} \\
\text{clausal}(D_1 \wedge D_2) &= \text{clausal}(D_1) \cup \text{clausal}(D_2) \\
\text{clausal}(\forall x D) &= \{\forall x D' \mid D' \in \text{clausal}(D)\} \\
\text{clausal}(G \supset A) &= \{G \supset A\}
\end{aligned}$$

Let $P = \{D_1, \ldots D_n\}$ be a finite set of definite formulae. Then

$$\text{clausal}(P) = \bigcup_{i=1}^{n} \text{clausal}(D_i)$$

Note that $[D]$ contains all instances of all elements of clausal(D).

As the above definition makes clear, we may think of a definite formula as a set of clauses, and hence we may think of a program as a set of clauses. We will often omit clausal(D) and clausal(P) when it is clear from the context what is meant.

Given that we may think of programs as sets of clauses in this way, it is then clear that Horn clauses may be defined as follows:

$$\begin{aligned}
H &:= A \mid \forall x H \mid H_1 \wedge H_2 \mid Q \supset A \\
Q &:= A \mid \exists x Q \mid Q_1 \wedge Q_2
\end{aligned}$$

By moving the conjunctions outwards in definite formulae, as mentioned above, it is clear that a set of closed H formulae is equivalent to a set of Horn clauses.

In fact if we allow disjunctions in the bodies of clauses, we arrive at a class of programs which is no more powerful than Horn clauses, as for any progam in the class of programs defined below, there is an equivalent Horn clause program.

$$\begin{aligned}
H &:= A \mid \forall x H \mid H_1 \wedge H_2 \mid Q \supset A \\
Q &:= A \mid \exists x Q \mid Q_1 \wedge Q_2 \mid Q_1 \vee Q_2
\end{aligned}$$

The equivalence of $(Q_1 \vee Q_2) \supset A$ and $(Q_1 \supset A) \wedge (Q_2 \supset A)$ ensures that we may rewrite any program in the larger class as a Horn clause program. Thus there is some innate redundancy in the definition of the larger class of programs. We will explore such issues in chapter 6.

2.2 Extensions

2.2.1 Extensional Universal Quantification

One feature of logic programming is that we usually consider the set of all closed terms, called the Herbrand Universe and which is here denoted as $\mathcal{U}$, as a set which is fixed prior to the writing of a program, and hence is constant throughout the computation process. For simplicity, we assume that the Herbrand Universe is not empty. We usually think of this set of terms as being generated by a finite number of symbols, i.e. a signature, and so it seems natural to associate a signature with a particular program. Often the signature of a program is taken to contain exactly the function and constant symbols which appear in the program. In our case we will assume that the signature must contain such symbols, but need not be limited to them. For example, consider the program below.

$$\forall x \ \text{non\_zero}(s(x))$$

It is clear that non_zero($s^n(0)$) succeeds for all $n \geq 1$, and that non_zero(0) fails. Clearly we wish the latter goal to fail, rather than produce a type error or something similar. Hence we need the external knowledge provided by the signature in order to have the correct "view" of the information in the program. This also makes it clear what to do with a goal such as non_zero($s(a)$), which would otherwise succeed.

We will assume that the number of symbols is in the signature is finite, so that the Herbrand Universe is recursively enumerable. It seems difficult to see how infinite signatures can be useful in this context, and so this does not seem to be a particularly restrictive assumption.

Definition 2.2.1 *A* signature Σ *is a set of pairs* f/n *where* f *is a constant or function symbol of arity* $n \geq 0$.

Note that this definition allows the same symbol to appear more than once in the signature with different arities.

The idea of restricting attention to a given set of terms seems natural from programming considerations. For any given predicate, there are some terms which will be applicable and some which will not. For example, it does not seem very sensible to discuss whether Mickey Mouse is a natural number, and so the natural number predicate should be restricted to terms constructed from 0 and the successor function. We may use a similar idea to further decompose the signature, so that each predicate may use a particular subset of the signature. For example, in a program which contains information about products, suppliers and customers, there may be some predicates which refer to products and suppliers but not to customers, and so we may think of these predicates as using a subset of the overall signature. In this sense, given the Herbrand Universe $\mathcal{U}$, we may derive the corresponding signature Σ, and for any given predicate p there is a signature $\Sigma_p \subseteq \Sigma$ such that p is only applicable to terms constructed from the symbols in Σ_p . These considerations will be useful in what follows, as we will not have to worry about Domain Closure Axioms and so forth to ensure that a given atom has a sensible interpretation.

It is possible to extend the notion of signature into a rudimentary notion of typing. For example, an n-ary function symbol f from a signature Σ may use any term from Σ^* as an argument, and the resulting term itself is an element of Σ^*. This may be thought of as ascribing the type $\Sigma^* \times \Sigma^* \times \ldots \times \Sigma^* \rightarrow \Sigma^*$ to f. However, it is conceivable that a more useful notion is to ascribe the type $\Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_n^* \rightarrow \Sigma_{n+1}^*$ to f, where the Σ_i are distinct signatures. We have not pursued this approach here, for the sake of simplicity, but as it would only require a change in the way terms are generated, it should not be too hard to incorporate into what follows.

We will only consider terms which may be built from the signature. This means that we cannot use "polymorphic" predicates such as append and member, for which the natural signature is $\{[]/0, ./2\}$, but for which terms such as [1,2] are perfectly valid. This is somewhat restrictive limitation, but it greatly simplifies the discussion. There is no reason in principle why the following remarks should not apply to predicates such as append and member, but as the technical details involved are peripheral to our scope (and somewhat overwhelming), we will not pursue such issues here.

Given that we think of the Herbrand Universe in this way, the rule given for deriving the success of universal quantification of goals in the definition of $\vdash_o$ above is not quite adequate for our purposes. We may think of the above rule as requiring that there is an explicit rule stating the desired conclusion, whereas we wish to allow reasoning by cases. For example, let the Herbrand Universe be $\{a, f(a), f(f(a)), \ldots\}$ and consider the program P below:

$$p(a)$$
$$\forall x\ p(f(x))$$

According to the definition above, it is not the case that $P \vdash_o \forall x\ p(x)$, and yet it is clear that for every term t in the Herbrand Universe $P \vdash_o p(t)$. We may think of this as requiring that universally quantified conclusions be independent of the language of the program. In our case we want the success of universally quantified goals to reflect the fact that we are dealing with a known Herbrand Universe, and so we will require something slightly different. The details are given in the next section; essentially we want a universally quantified goal to succeed precisely when all of its instances succeed. However we will still retain the "compactness" of the previous version, in that success of a universally quantified goal will only depend on the success of a finite number of instances, and hence describes a feasible search operation. This point is discussed in more detail in section 2.4.

2.2.2 Negation in a Constructive Setting

It has already been mentioned how intuitionistic logic and classical logic may be expected to differ for various extensions to Horn clauses, and that intuitionistic logic often seems better suited to computation than classical logic. One such strength of intuitionistic logic may be shown by one of the most common extensions to Horn clauses: to allow the bodies of clauses to be conjunctions of literals, rather than conjunctions of atoms alone. The usual extension to the proof theory is to introduce the *Negation as Failure* (NAF) rule: for a ground atom A, we say that $\neg A$ succeeds precisely when A fails [17,33,49,100,105]. Now classically, A and $\neg A$ are symmetric in the sense that if one is false then the other is true, and vice-versa, and a proof that one leads to a contradiction is a proof of the other. This is due to the fact that in classical logic we must have that at least one of A and $\neg A$ is true. The NAF rule would thus suggest that this symmetry should be observed by the computational behaviour of the two goals A and $\neg A$, i.e. that either A succeeds or A fails, and that from the failure of A we can prove $\neg A$ and vice-versa. However, it is not clear that defining $\neg A$ in terms of the failure of A preserves this symmetry, as A may loop. Thus the complementational nature of the NAF rule may introduce an asymmetry between A and $\neg A$, which does not sit well with their symmetry in classical proof theory.

There is an asymmetry between the two in intuitionistic logic, as there is no rule which identifies the truth of $\neg\neg A$ with that of A. In fact, $\neg A$ is generally harder to prove than A, as in order to prove $\neg A$ intuitionistically, we must show that A can never be true. In this way we expect that $\neg A$ will generally be much harder to prove than A.

Note that NAF is an implicit form of negation in that the negative consequences are defined as those which fail to be positive consequences. The logical justification for this extra-logical rule is known as the *Closed World Assumption* (CWA), which may be thought of as stating that anything which does not follow from the program is false. Thus the law of excluded middle holds for every predicate, and so it is difficult to see how the CWA may be reconciled with a constructivist philosophy.

Computability considerations also affect NAF, and so it really only coincides with the usual conception of negation in mathematical logic for the class of programs for which any goal either succeeds or fails, (i.e. there are no loops) and where the program's knowledge is complete, so that the CWA is satisfied. Such special cases may be found in certain deductive databases, such as a databases of student records which lists all the courses in which a student is enrolled. Obviously it is decidable whether or not a student is enrolled in a given course, and the knowledge in the database is complete, so that if we find that Computer Science 1 is not on the list of all courses in which the student is enrolled, we may conclude that the student is not enrolled in Computer Science 1.

Unfortunately, not all programs satisfy both of the above conditions, i.e. are loop-free and omniscient, and for such programs, NAF becomes incomplete with respect to the consequences of the CWA, in that it does not follow that if $\neg A$ is true according to the CWA then $\neg A$ is computed by the NAF rule. The main difficulty is that NAF may be thought of as inferring that an atom A which fails is false, whereas from the CWA we infer that any atom A which does not succeed is false. A general slogan which seems to be applicable to all formal programming languages is that any language of sufficient power to be interesting and useful will have some sort of undecidability or incompleteness property. One obvious example is that anything of equivalent power to Turing machines will inherit the undecidability of the halting problem. In order to use some notion of NAF in such a system, we need to give more justification than was given above in order to reconcile this approach with its formal basis in mathematical logic.

It may be argued that the reason that NAF is only an approximation in such cases (i.e. that NAF is sound but not complete with respect to the consequences of the CWA) is that NAF is an inherently computable form of negation, but that the consequence of the CWA are inherently non-computable, as the above considerations indicate. Thus *any* computable form of negation can only be an approximation to the CWA, and thus cannot significantly improve on NAF. This is to miss the point of the argument; we are trying to incorporate negation into the class of "programmable" formulae, not to encode the CWA. We see the role

of the CWA as a justification, rather than a desirable end in itself, and so if the CWA does not provide an adequate justification for the NAF rule, then it does not necessarily follow that we must abandon NAF, but that in order to use it, we need to find some other logical[1] justification for it.

As noted earlier, constructive truth is "inherently" three-valued, as the lack of the law of excluded middle means that we do not require every formula to be either true or false, and so in a constructive setting there are formulae which are neither true nor false, just as in the programming setting there are goals which neither succeed nor fail. This suggests that a constructive interpretation of NAF together with a modification of the CWA so that it is consistent with a constructive approach seems appropriate.

An interesting property of intuitionistic logic compared with classical logic is that finer distinctions are made between programs. For example, consider the programs P_1 and P_2 where P_1 is just p and P_2 is $(q \supset p) \wedge (\neg q \supset p)$. These two programs are equivalent in classical logic, whereas intuitionistically they are not. This is due to the fact that in intuitionistic logic, for the second program one must either derive q or derive $\neg q$ in order to derive p, which seems more in keeping with the operational nature of the program than the approach of classical logic, in which one may take a global view and deduce that p must hold on the grounds that it could not be otherwise. Now if we extend both programs by adding q, then there is no change in the behaviour of the goal p for the extension of the first program, but there is a significant change in the behaviour of the goal p from the extension of the second program, as q now succeeds rather than fails. This change is reflected in intuitionistic logic as $P_2, q \vdash_I p$ but $P_2 \nvdash_I p$, whereas $P_2 \vdash_C p$ and $P_2, q \vdash_C p$. Thus classical logic is too strong to precisely capture the nature of computation in this context, as there are too many classical equivalences to allow an unambiguous association between derivability and proof.

---

[1]Here we use the term *logical* in the narrow mathematical sense.

One approach suggested by the above argument is that in order to recast the CWA, we need to remove the insistence that there are only two possibilities, being that a goal G must be either true or false. A way to do this is to shift the emphasis of the CWA from being a global notion to a local one, in the same way that in P_2 above intuitionistic logic may be thought of as being more locally orientated than classical logic. This will remove the conflict between the two-valued approach of the CWA and the three-valued nature required by NAF. An important consideration here is that in shifting from the global view to the local one, we lose our global perspective. This manifests itself in the consideration that there are some situations in which NAF is not really applicable. In some ways, NAF is an attractive way to implement negation; there are completeness results which state that A is true iff A succeeds, i.e. that $P \models A \Leftrightarrow P \vdash A$ [61], and so a natural dual to this principle would be that A is false iff A fails, or $P \models \neg A \Leftrightarrow P \not\vdash A$. However, there is an underlying assumption here that the definition of every predicate in the program is complete, so that it is sensible to consider as false everything that is not explicitly stated to be true. This assumption of "universal completeness" will hold for some programs, but there are many others for which it will not, and for these the NAF rule does not make much sense. For example, the append predicate given below is complete in the sense that there is no additional clause we can insert which would correctly extend the append relation; all the information we ever want to consider about appending lists together is given, and so it is correct to apply NAF. Thus we may think of the append predicate as given below as *completely defined.*

$$\forall x \ \text{append}([], x, x)$$
$$\forall x \forall y \forall z \forall w \ \text{append}(y, z, w) \supset \text{append}(x.y, z, x.w)$$

On the other hand, not every predicate will have such a complete definition. For example, a predicate containing information about carcinogens we would wish to consider *incompletely defined*, as it is possible that our list of carcinogens is not complete. Thus whilst we wish to be able to prove $\neg\text{append}([], [1, 2], [3])$ from the failure of the goal $\text{append}([], [1, 2], [3])$, we may be undecided whether to conclude

$\neg$carcinogen(chocolate) from the failure of the goal carcinogen(chocolate). Hence the NAF approach is inappropriate here, as whilst $\exists x \neg$carcinogen(x) may be true, it is not necessarily true that everything not known to be a carcinogen is known not to be a carcinogen. In this way we may classify each definition in the program as completely or incompletely defined, i.e. suitable for NAF or not. Thus for any program P, we say that a predicate p is *completely defined in* P if any extension to the definition of p given in P is either wrong or equivalent to the original definition. Otherwise, p is *incompletely defined in* P. This idea is explored further in section 2.3.

Note that we cannot give a formal definition in the narrow mathematical sense. This is due to the fact that the property in question is inherently semantic (i.e. a matter of judgement); only the programmer can know whether a given predicate is completely defined or not, and then only on the basis of the relation between the formal definition (i.e. the program) and the specification of what the program is supposed to do, which may or may not be given formally. With a formal specification it is possible to consider the question of whether the definition of a predicate is complete with respect to the formal specification or not. As issues of specification and formal correctness are beyond our scope, we will use the informal definition above for the sake of simplicity and generality, and so we require the programmer to indicate which predicates are completely defined in any given program, similar to an idea expressed in [43].

We need to impose some restrictions on the way that completely defined predicates may depend on other completely defined predicates and incompletely defined predicates in order to guarantee sensible behaviour. The notion of completely defined predicates will not make a great deal of sense if there is a clause $q \supset p$ in the program where p is completely defined but q is not, as if we are able to increase our knowledge about q, we would be able to increase our knowledge about p. Thus we need to restrict the definition of completely defined predicates so that the only place that incompletely defined predicates can appear in the definition of a completely defined predicate is in the assumption part of an implication in

the body of a clause. This property will be useful later on, and is formalised in section 2.3.

Note that the presence of completely defined predicates will have an effect on the way that implications in goals are treated. As it will not make sense to add to the definition of a completely defined predicate, we will not be able to execute goals which require the addition to the program of a clause whose head is a completely defined predicate. This point is discussed more fully in section 2.3.

For these reasons we distinguish between completely defined predicates, i.e. those for which we may apply NAF, and incompletely defined predicates, for which some other form of negation will be necessary. This gives us our form of localisation of the CWA, in that only when a predicate p is completely defined will we be able to identify the failure of $p(t)$ with the truth of $\neg p(t)$. Thus we may think of the notion of completely or incompletely defined predicates as an indication of whether the CWA is true or not for smaller localised worlds, rather than viewing the CWA as a global condition on the entire program.

Another useful property of the distinction between completely and incompletely defined predicates will allow us to consider different forms of negation within the same framework. For completely defined predicates, it is clear that NAF is appropriate. For incompletely defined predicates, we need some other rule. This may be thought of as specifying what we may deduce from a failure to prove a goal. If the goal is an atom $p(t)$ where p is completely defined, then the NAF rule says that we may deduce $\neg p(t)$ from the failure of $p(t)$, as we know that we can never have $p(t)$ being true. In this way we may think of NAF as a form of consistency test, in that if $p(t)$ fails, then it is not inconsistent to assume $\neg p(t)$, i.e. $\neg p(t)$ is consistent with the program, but not necessarily true. It is the fact that the predicate is completely defined which leads us to the stronger conclusion that $\neg p(t)$ is true. In other cases, i.e. for goals other than completely defined atoms, some other action may be appropriate. For example, for incompletely defined predicates, we might look for an explicit statement that $\neg p(t)$ is true, or some indication that the assumption of $p(t)$ leads to an inconsistency.

One such form of computation is the Negation as Inconsistency (NAI) principle

introduced by Gabbay and Sergot [39], which is an explicit form of negation in that formulae such as $\neg p(t)$ form part of the program. Thus whilst our information may not be complete, we may know a particular piece of negative information. For example, although the list of carcinogens may be incomplete, it may be known that bananas are not carcinogenic, and so we would wish to state ¬carcinogen(bananas). We may thus build up our knowledge of incompletely defined predicates in both a positive and a negative fashion. The NAI rule allows us to determine the truth of $\neg A$ by showing that A leads to an inconsistency. A thorough discussion of this idea is given in [39]; here we note that such a rule is appropriate for incompletely defined predicates.

2.3 Technicalities

In this section we define the necessary extensions to the earlier framework in order to incorporate negation.

The definitions of D and G formulae given in [82] and in section 2.1 are as follows:

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \forall x G \mid \exists x G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G$$

The addition of negated atoms to goals requires that we extend the definition of a G formula to include the case $\neg A$. The definition of the G formulae which reflects this is given below.

Definition 2.3.1 A definite formula D *and* a goal formula G *are defined via:*

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid \forall x G \mid \exists x G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G$$

We denote by $\mathcal{D}$ the set of all D formulae, and the set of all G formulae by $\mathcal{G}$. Such an extension requires that programs consist of more than just a set of closed

definite clauses, as we need to know which predicates are completely defined. In fact, we will include a pair of disjoint sets of predicates names as part of the program, where the first set of names are the predicates which are incompletely defined, and the second are the completely defined predicates. This allows us to make some significant technical simplifications. Whilst we insist that the two sets be disjoint, it is not necessary that the two sets cover all predicate names, as there may be some predicates whose status is somewhat unclear (i.e. it is possible that the definition is complete, but we do not know that it is complete). Thus the first set of names may be thought of as those predicates for which we know our information is incomplete, and hence it is reasonable to extend the definition of such predicates during execution of the program, but not to apply the NAF rule to them. On the other hand, the completely defined predicates may use the NAF rule, but their definitions may not be extended.

This leads us to the definition of a program which appears below.

Definition 2.3.2 *Given an atom $A = p(t_1, \ldots t_n)$, we define $name(A) = p$, and for any formula F, $names(F) = \{name(A) \mid A \text{ appears in } F\}$.*

We say an atom A appears positively (negatively) in a formula F as follows:

- *A appears positively in A*
- *A appears positively (negatively) in $F_1 \vee F_2$ iff A appears positively (negatively) in either F_1 or F_2*
- *A appears positively (negatively) in $F_1 \wedge F_2$ iff A appears positively (negatively) in either F_1 or F_2*
- *A appears positively (negatively) in $\exists x F$ iff A appears positively (negatively) in F*
- *A appears positively (negatively) in $\forall x F$ iff A appears positively (negatively) in F*
- *A appears positively (negatively) in $F_1 \supset F_2$ iff A appears positively (negatively) in F_2 or A appears negatively (positively) in F_1*

- *A appears positively (negatively) in $\neg F$ iff A appears positively (negatively) in F*

The above definition is used to determine the "parity" of an atom A in regard to implications. For example, $p(a)$ occurs positively in $p(a) \wedge q(b)$, but negatively in $p(a) \supset q(b)$.

Definition 2.3.3 *A* derivation state *is a pair $\langle D, N\rangle$ where D is a set of definite formulae and N is a pair $\langle N_1, N_2\rangle$ where $N_i \subseteq names(\mathcal{H})$ and $N_1 \cap N_2 = \emptyset$, and which satisfies:*

For all atoms A and B such that name$(A) \in N_2$, *$G \supset A$ is a closed formula in $[D]$, and B occurs in G, then*

- *If B occurs positively in G, then* name$(B) \in N_2$.
- *If B occurs negatively in G, then* name$(B) \in N_1$.

If $N = \langle N_1, N_2\rangle$, we say ass$(N) = N_1$, den$(N) = N_2$.

A program *is a derivation state $\langle D, N\rangle$ in which D is a set of closed definite formulae.*

We denote the set of all programs by $\mathcal{P}$. When N is the pair $\langle$names$(D), \emptyset\rangle$, we often write the program as just D. As mentioned above, we may think of D as either a set of closed definite formulae or as a set of clauses.

The restrictions on the occurrences of atoms in the bodies of the clauses of completely defined predicates ensure that completely defined predicates may only depend on the success of other completely defined predicates and the assumption of incompletely defined predicates. For example, given the clause

$$(r \supset q) \supset p$$

then if den(N) contains p, then it must contain q and ass(N) must contain r.

Clearly this property of programs will be maintained throughout execution provided that additions to the program only extend incompletely defined predicates.

We also need to extend the notion of operational provability. We will do so by introducing two relations $\vdash_s$ and $\vdash_f$, where the former is used to indicate success and hence will be similar to $\vdash_o$, and the latter is used to indicate failure. There will be some interplay between the two relations, as we wish to identify $P \vdash_s \neg A$ with $P \vdash_f A$ when name(A) is a completely defined predicate, i.e. for completely defined predicates we identify negation ($P \vdash_s \neg A$) with failure ($P \vdash_f A$).

As mentioned above, we are interested in validity with respect to a given set of ground terms $\mathcal{U}$. In the presence of the Negation as Failure rule, this raises some compactness problems. For example, it seems natural to state that the goal $\exists x\, p(x)$ fails iff $p(t)$ fails for each $t \in \mathcal{U}$. However, this can lead to some technical complications, and is somewhat at variance with what happens in Prolog. Consider the program

$$\forall x\; p(x) \supset p(s(x))$$
$$(\exists x\; p(x)) \supset q$$

where the Herbrand Universe is $\{0, s(0), s(s(0)), \ldots\}$. According to the above rule, q fails. However, a Prolog system will not return an answer for the goal q. The problem is that we need more than ω steps in order to show that q fails. Whilst this in itself is not an insurmountable problem, it seems more appropriate (and more elegant) to alter the definition of failure so that q neither fails nor succeeds. Hence we will need an extra condition, in that not only must we have that every ground instance $p(t)$ of $p(x)$ fails, but also that they do so *compactly*, i.e. that there is a finite set of instances which fail, and the failure of this finite set of instances implies the failure of all ground instances. For this reason we will need to consider arbitrary terms, and not just ground terms, in the definition of failure for existentially quantified goals. A formal definition is given below.

We will assume that the number of symbols in the Herbrand universe is finite, and so we may associate a signature with $\mathcal{U}$. This signature will be denoted as Σ.

We also assume the existence of a countably infinite set of variables disjoint from the set of constants and function symbols of all signatures.

Definition 2.3.4 *Let Σ be a signature containing at least one constant symbol.*

Terms are defined as follows:

- *A variable or a constant in Σ is a term*
- *If f is an n-ary function symbol in Σ and $t_1, \ldots t_n$ are terms, then $f(t_1, \ldots t_n)$ is a term.*
- *Nothing else is a term.*

The Herbrand universe *$\mathcal{U}$ is the set of all ground terms which may be formed from the symbols in Σ.*

We denote by $\mathcal{T}$ the set of all terms which may be formed from the symbols in Σ.

Note that $\mathcal{U}$ is the set of all ground terms, whereas $\mathcal{T}$ is the set of all terms. We think of each of the variables appearing in a term in $\mathcal{T}$ as ranging over elements of $\mathcal{U}$, so that we think of $\mathcal{T}$ as a more sophisticated representation of the same set of terms. Thus the non-ground terms in $\mathcal{T}$ do not have any deep meaning; they merely act as place holders.

As mentioned above, we wish to define the failure of existentially quantified goals (and also the success of universally quantified goals) by way of a finite set of "representative" instances, rather than by way of all ground instances (which is generally infinite). In order so to do, below we introduce the notion of a *representation.*

Definition 2.3.5 *A* covering set *of $\mathcal{U}$ is a set of terms T such that $t \in \mathcal{U}$ iff t is a ground instance of a term $t' \in T$.*

A minimal *covering set is a covering set of which no proper subset is a covering set.*

A representation *of $\mathcal{U}$ is a finite minimal covering set of $\mathcal{U}$.*

We refer to the set of all representations of $\mathcal{U}$ as $\mathcal{R}(\mathcal{U})$.

We may now state that $\exists x G$ goal fails if there is a representation R such that $G[t/x]$ fails for all $t \in R$. This essentially guarantees a continuity property, in that $\exists x G$ only fails when there is a finite set of instances of G that fails. Consider again the above example. As any representation must contain a term of the form $s^n(y)$ for some n, for $\exists x p(x)$ to fail we must have that $p(y)$ fails. However it seems that any reasonable definition of failure would not allow $p(y)$ to fail in this instance, as it "matches" a clause in the program which generates the same goal. Hence as $p(y)$ does not fail (even though every ground instance of it does), $\exists x p(x)$ does not fail.

We will define the success of universally quantified goals in a similar way, i.e. that $\forall x G$ succeeds iff there is a representation R such that $G[t/x]$ succeeds for all $t \in R$. Note that as a term in a representation may contain variables, we will have to consider the possibility that variables may occur in atoms, and hence take this into account in the definition of success and failure for atoms. As we desire the failure of $G[t/x]$ for all $t \in R$ to be at least as strong a condition as the failure of $G[t/x]$ for all $t \in \mathcal{U}$, it seems natural to expect that the former property implies the latter. Similarly it seems natural to expect that if $G[t/x]$ succeeds for all $t \in R$ then $G[t/x]$ succeeds for all $t \in \mathcal{U}$. Thus the success or failure of the instances of an atom shall be our guiding intuition in the relevant definitions of success and failure, and so it seems natural to adopt the policy that an atom succeeds if every instance of it succeeds, and an atom fails if every instance of it fails. However this is not quite sufficient for our purposes. Ultimately, we are interested in the validity of sentences, i.e. whether a given set of closed definite formulae implies a given closed goal formula. Free variables and the like are merely tools used in the derivation process. Hence we are not interested in the validity of formulae containing free variables per se, but only in using the success or failure of such formulae to determine the validity of sentences. Thus we know that any free variable in a derivation must be introduced by a quantifier, which allows us

to simplify the definition of success for an atom. For example, let Σ be $\{a/0, f/1\}$ (so that the Herbrand Universe consists of $a, f(a), f^2(a), \ldots$), and consider the program

$$p(a)$$
$$\forall x\ p(f(x))$$

As $p(a)$ and $p(f(y))$ succeed, we have that $\forall x\ p(x)$ succeeds. On the other hand, it is less clear what we should expect for $p(z)$. It is clear that every (proper) instance of $p(z)$ succeeds (i.e. that $p(a)$ and $p(f(y))$ succeed for any y), which suggests that $p(z)$ should succeed. However this means that the definition of success for an atom may be somewhat complicated, as we may have to "split" $p(z)$ into a number of instances. This also means that for a goal such as $\forall x\ p(x)$ there are two "layers" of universal quantification – one being the explicit quantifier and the other being the implicit quantification given by the occurrence of free variables in terms such as $p(f(y))$. Hence we shall define the success of an atom as above and in [77], i.e. in terms of the atom itself, rather than its instances. Thus in the above example, $\forall x\ p(x)$ succeeds, but $p(z)$ does not. It should be noted that if an atom succeeds according to the definition of $\vdash_o$ in Section 2.1, then all its instances succeed. In this way the definition of success and failure for atoms may appear to be somewhat asymmetric, but as we are ultimately interested only in closed formulae, this will not be of great concern.

Note that the definition of $\vdash_o$ for $\forall x G$ may be thought of as utilizing only the representation $\{y\}$ of $\mathcal{U}$. Thus our definition of success for $\forall x G$ seems a natural extension of $\vdash_o$ when considering validity with respect to a given Herbrand Universe.

An important point to note is that the definition of $\vdash_o$ in Section 2.1 is inductive. Hence it is tempting to define $P \vdash_s \neg A$ via $P \vdash_f A$ and vice-versa. However it is not clear that this will lead to the desired definition of $\vdash_s$ and $\vdash_f$. Note that according to the rules for $\vdash_o$ in Section 2.1, $p \supset p \vdash_o p$ iff $p \supset p \vdash_o p$, and so the minimality requirement is necessary, i.e. that $\vdash_o$ be the least relation satisfying the above rules. Now if we were to define $P \vdash_s \neg A$ as $P \vdash_f A$ and then impose a

similar minimality requirement, it is not clear that the resulting relations are well-defined. For these reasons it seems better to give an iterative (i.e. non-inductive) definition of the relations $\vdash_s$ and $\vdash_f$ which is obviously well-defined, and then show that the relations satisfy the appropriate inductive properties.

As our notions of success and failure are dependent on instances, it will be convenient to use the instances of a program P rather than the program itself in some circumstances. To this end we define below a mapping (), which is similar to the mapping [] above. The former is more convenient for stating some later results, as well as somewhat more intuitive. Essentially the difference is that (D) consists purely of instances, whereas $[D]$ contains D itself.

The definitions of $\vdash_s$ and $\vdash_f$ follow. For the reasons mentioned above, these are not defined (only) over $\mathcal{P} \times \mathcal{G}$, but over the pairs $\langle\langle D, N\rangle, G\rangle$, where $\langle D, N\rangle$ is a derivation state and G is a goal formula. Note that neither D nor G need be closed here. We will refer to such pairs as *derivation pairs.*

Definition 2.3.6 *Let D be a definite formula. We define (D) by cases as follows:*

$$\begin{aligned}
(A) &= \{A\} \\
(\forall x D) &= \bigcup_{t \in \mathcal{T}} (D[t/x]) \\
(D_1 \wedge D_2) &= (D_1) \cup (D_2) \\
(G \supset A) &= \{G \supset A\}
\end{aligned}$$

Let D' be a set of definite formulae. Then we define

$$(D') = \bigcup_{D \in D'} (D)$$

We denote by $A \propto B$ the statement that A is an instance of B.

Definition 2.3.7 *Let $\langle\langle D, N\rangle, G\rangle$ be a derivation pair. We define the relations $\vdash_s^k$ and $\vdash_f^k$ on derivation pairs for any $k \geq 0$ by cases on G as follows:*

- $\langle D, N\rangle \vdash_s^0 A$ *iff* $A \in (D)$

- $\langle D, N\rangle \vdash_f^0 A$ *iff* $\forall B \in (D)$, $B \not\propto A$ *and* $\forall G \supset B \in (D)$, $B \not\propto A$

- $\langle D, N\rangle \vdash_s^{k+1} A$ *iff* $\langle D, N\rangle \vdash_s^k A$ *or* $\exists G \supset A \in (D)$ *such that* $\langle D, N\rangle \vdash_s^k G$

- $\langle D, N\rangle \vdash_f^{k+1} A$ *iff either* $\langle D, N\rangle \vdash_f^k A$ *or* $\forall B \in (D)$, $B \not\propto A$ *and* $\forall G \supset B \in (D)$ *such that* $B \propto A$, $\langle D, N\rangle \vdash_f^k G$

- $\langle D, N\rangle \vdash_s^{k+1} \neg A$ *iff* $\langle D, N\rangle \vdash_f^k A$ *and* $name(A) \in \mathrm{den}(N)$

- $\langle D, N\rangle \vdash_f^{k+1} \neg A$ *iff* $\langle D, N\rangle \vdash_s^k A$

- $\langle D, N\rangle \vdash_s^k G_1 \vee G_2$ *iff* $\langle D, N\rangle \vdash_s^k G_1$ *or* $\langle D, N\rangle \vdash_s^k G_2$

- $\langle D, N\rangle \vdash_f^k G_1 \vee G_2$ *iff* $\langle D, N\rangle \vdash_f^k G_1$ *and* $\langle D, N\rangle \vdash_f^k G_2$

- $\langle D, N\rangle \vdash_s^k G_1 \wedge G_2$ *iff* $\langle D, N\rangle \vdash_s^k G_1$ *and* $\langle D, N\rangle \vdash_s^k G_2$

- $\langle D, N\rangle \vdash_f^k G_1 \wedge G_2$ *iff* $\langle D, N\rangle \vdash_f^k G_1$ *or* $\langle D, N\rangle \vdash_f^k G_2$

- $\langle D, N\rangle \vdash_s^k \exists x G$ *iff* $\langle D, N\rangle \vdash_s^k G[t/x]$ *for some* $t \in \mathcal{U}$

- $\langle D, N\rangle \vdash_f^k \exists x G$ *iff* $\exists R \in \mathcal{R}(\mathcal{U})$ *such that* $\langle D, N\rangle \vdash_f^k G[t/x]$ *for all* $t \in R$ *where the variables in* R *do not appear free in* D *or* G

- $\langle D, N\rangle \vdash_s^k \forall x G$ *iff* $\exists R \in \mathcal{R}(\mathcal{U})$ *such that* $\langle D, N\rangle \vdash_s^k G[t/x]$ *for all* $t \in R$ *where the variables in* R *do not appear free in* D *or* G

- $\langle D, N\rangle \vdash_f^k \forall x G$ *iff* $\langle D, N\rangle \vdash_f^k G[t/x]$ *for some* $t \in \mathcal{U}$

- $\langle D, N\rangle \vdash_s^k D' \supset G$ *iff* $\langle D \cup \{D'\}, N\rangle \vdash_s^k G$ *and* $names(heads(D')) \subseteq \mathrm{ass}(N)$

- $\langle D, N\rangle \vdash_f^k D' \supset G$ *iff* $\langle D \cup \{D'\}, N\rangle \vdash_f^k G$ *and* $names(heads(D')) \subseteq \mathrm{ass}(N)$

The relations $\vdash_s$ and $\vdash_f$ are defined as

$$\vdash_s = \bigcup_{k \geq 0} \vdash_s^k$$

$$\vdash_f = \bigcup_{k \geq 0} \vdash_f^k$$

Note the case for negation, which ensures that the NAF rule is only applied to completely defined predicates of P, i.e. those whose names appear in den(N). We do not insist on the same restriction for the failure of $\neg A$ as it seems reasonable for $\neg A$ to fail whenever A succeeds, regardless of whether name(A) is a completely defined predicate or not. As negation is only applied to literals, all this does is allow more things to fail than would be the case otherwise. For example, given the program $P = \langle p, \langle \{p\}, \emptyset \rangle \rangle$, we have that $P \vdash_s p$, and so it seems reasonable that we have that $P \vdash_f \neg p$, even though p is not completely defined. Thus the success of an atom implies the failure of its negation, but the failure of an atom doesn't necessarily imply the success of its negation.

Note also that the implication rule has to be slightly modified so that only predicates in ass(N) may be extended. There may be less restrictive ways of dealing with this problem; this way ensures that only assumptions known to be consistent with the program are allowed to be made, and that a goal of the form $D \supset G$ for which D is an extension of the definition of a predicate not in ass(N) is computationally indeterminate. Without this restriction, computation of the goal append([], [], [1, 2, 3]) $\supset G$ from the standard append program (i.e. the two standard clauses with append being completely defined) involves a program which extends the definition of append, and so it is not obvious what the computational behaviour should be. This is a form of the consistency problem: which goals should be provable from an inconsistent program? This question is taken up in a later section; here we note that the present way of dealing with the problem is "safe", in that inconsistencies are avoided.

It should also be noted that this form of the implication rule is not a conservative extension of the implication rule for $\vdash_o$, in that it is not the case that

$\langle p \supset p, \langle \emptyset, \{p\} \rangle \rangle \vdash_s p \supset p$, due to the fact that p is completely defined, and hence the antecedent of the goal cannot be added to the program. However, it should be clear that for a program $P = \langle D, N \rangle$ and a goal G in which all predicates which occur negatively in G are in ass(N) that $D \vdash_o G$ implies that $P \vdash_s G$. Hence ass(N) may be used to identify formulae for which $\vdash_s$ conservatively extends $\vdash_o$. In particular, if G is a goal in which all negatively occurring predicates are in ass(N) and positive occurrences of a universal quantifier are not allowed in goals, it should be clear that $D \vdash_o G$ iff $P \vdash_s G$. We shall see how this device is useful in chapter 5. Clearly a conservative extension is desirable and would simplify the definitions of $\vdash_s$ and $\vdash_f$, but raises some difficult problems for the model theory. Since it seems problematic for the model theory to cope with extensions to completely defined predicates, we place this restriction here to avoid considering cases which are semantically meaningless. We may think of this restriction (i.e. that predicates occurring in a negative position in a goal must appear in ass(N) for the goal to succeed or fail) as insisting that additions to the program must be known to be consistent with the program, just as we insist that to use NAF we must know that the predicate involved cannot be consistently extended. In this way this restriction, whilst somewhat undesirable, does seem to be in keeping with our approach.

Note that there are programs and goals for which neither $P \vdash_s G$ nor $P \vdash_f G$. For example, if $P = \langle p \supset p, \langle \emptyset, \emptyset \rangle \rangle$ and $G = p$, then it is clear that $P \nvdash_s p$ and $P \nvdash_f p$.

The above definitions of $\vdash_s$ and $\vdash_f$ may be used to derive a proof system by interpreting the left hand side of each iff as the derived sequent and the right hand side as the previous sequent or sequents. We may think of proofs in this system as trees whose nodes are sequents, where each sub-tree is classified as either a "success" sub-tree or a "fail" sub-tree, and so this is a generalisation of the concept of an SLDNF-tree [61].

A formal definition is given below.

Definition 2.3.8 *Let $\langle P, G\rangle$ be a derivation pair where $P = \langle D, N\rangle$.*

Let $match(A) = \{G \supset B \in (D) \mid B \propto A\}$.

An O-derivation *is a tree built using the following rules:*

$$\frac{P \longrightarrow^{+} G}{P \longrightarrow^{+} A} \text{ where } G \supset A \in (D) \qquad \frac{\forall B \in (D), B \not\propto A \quad \forall G \supset B \in \text{match}(A)\ P \longrightarrow^{-} G}{P \longrightarrow^{-} A}$$

$$\frac{P \longrightarrow^{-} A}{P \longrightarrow^{+} \neg A} \text{ where } name(A) \in \text{den}(N) \qquad \frac{P \longrightarrow^{+} A}{P \longrightarrow^{-} \neg A}$$

$$\frac{P \longrightarrow^{+} G_1 \quad P \longrightarrow^{+} G_2}{P \longrightarrow^{+} G_1 \wedge G_2} \qquad \frac{P \longrightarrow^{-} G_i}{P \longrightarrow^{-} G_1 \wedge G_2}\ i = 1,2$$

$$\frac{P \longrightarrow^{+} G_i}{P \longrightarrow^{+} G_1 \vee G_2}\ i = 1,2 \qquad \frac{P \longrightarrow^{-} G_1 \quad P \longrightarrow^{-} G_2}{P \longrightarrow^{-} G_1 \vee G_2}$$

$$\frac{P \longrightarrow^{+} G[t/x]}{P \longrightarrow^{+} \exists x G} \text{ for some } t \in \mathcal{U} \qquad \frac{P \longrightarrow^{-} G[t/x]\ \forall t \in R}{P \longrightarrow^{-} \exists x G} \text{ for some } R \in \mathcal{R}(\mathcal{U})$$

$$\frac{P \longrightarrow^{+} G[t/x]\ \forall t \in R}{P \longrightarrow^{+} \forall x G} \text{ for some } R \in \mathcal{R}(\mathcal{U}) \qquad \frac{P \longrightarrow^{-} G[t/x]}{P \longrightarrow^{-} \forall x G} \text{ for some } t \in \mathcal{U}$$

$$\frac{P, D \longrightarrow^{+} G}{P \longrightarrow^{+} D \supset G} \qquad \frac{P, D \longrightarrow^{-} G}{P \longrightarrow^{-} D \supset G}$$

where the cases $P \longrightarrow^{+} \forall x G$ and $P \longrightarrow^{-} \exists x G$ have the side condition that no variable in R occurs free in P or G and the last two rules have the side condition that $\text{names}(\text{heads}(D)) \subseteq \text{ass}(N)$.

A sequent $P \longrightarrow^{+} G$ is called a positive *sequent, and a sequent $P \longrightarrow^{-} G$ is called a* negative *sequent.*

A positive sequent $P \longrightarrow^{+} G$ is initial *if G is an atom A and $A \in (D)$. A negative sequent $P \longrightarrow^{-} G$ is* initial *if G is an atom A and we have $\forall B \in (D)$, $B \not\propto A$ and $\forall G \supset B \in (D)$, $B \not\propto A$.*

An O-proof *is an O-derivation whose root is positive and whose leaves are initial.*

An O-denial *is an O-derivation whose root is negative and whose leaves are initial.*

We may think of an O-proof or O-denial as exploiting the iterative nature of the definition of $\vdash_s$ and $\vdash_f$. This will allow us to use O-proofs and O-denials to derive results about the relations $\vdash_s$ and $\vdash_f$ more easily.

The following proposition shows how $\vdash_s$ and $\vdash_f$ may be thought of in a more inductive style which is closely related to the definition of $\vdash_o$ in Section 2.1.

Proposition 2.3.1 *Let $\langle D, N\rangle$ be a derivation state. Then*

- $\langle D, N\rangle \vdash_s A$ *iff* $A \in (D)$ *or* $\exists G \supset A \in (D)$ *such that* $\langle D, N\rangle \vdash_s G$
- $\langle D, N\rangle \vdash_s \neg A$ *iff* $\langle D, N\rangle \vdash_f A$ *and* $name(A) \in \mathrm{den}(N)$
- $\langle D, N\rangle \vdash_s G_1 \vee G_2$ *iff* $\langle D, N\rangle \vdash_s G_1$ *or* $\langle D, N\rangle \vdash_s G_2$
- $\langle D, N\rangle \vdash_s G_1 \wedge G_2$ *iff* $\langle D, N\rangle \vdash_s G_1$ *and* $\langle D, N\rangle \vdash_s G_2$
- $\langle D, N\rangle \vdash_s \exists x G$ *iff* $\langle D, N\rangle \vdash_s G[t/x]$ *for some* $t \in \mathcal{U}$
- $\langle D, N\rangle \vdash_s \forall x G$ *iff* $\exists R \in \mathcal{R}(\mathcal{U})$ *such that* $\langle D, N\rangle \vdash_s G[t/x]$ *for all* $t \in R$ *where no variable in* R *appears free in* D *or* G
- $\langle D, N\rangle \vdash_s D' \supset G$ *iff* $\langle D \cup \{D'\}, N\rangle \vdash_s G$ *and* $names(heads(D')) \subseteq \mathrm{ass}(N)$

- $\langle D, N\rangle \vdash_f A$ *iff* $\forall B \in (D)\ B \not\propto A$ *and* $\forall G \supset B \in (D)$ *such that* $B \propto A$, $\langle D, N\rangle \vdash_f G$
- $\langle D, N\rangle \vdash_f \neg A$ *iff* $\langle D, N\rangle \vdash_s A$

- $\langle D, N\rangle \vdash_f G_1 \vee G_2$ *iff* $\langle D, N\rangle \vdash_f G_1$ *and* $\langle D, N\rangle \vdash_f G_2$
- $\langle D, N\rangle \vdash_f G_1 \wedge G_2$ *iff* $\langle D, N\rangle \vdash_f G_1$ *or* $\langle D, N\rangle \vdash_f G_2$
- $\langle D, N\rangle \vdash_f \exists x G$ *iff* $\exists R \in \mathcal{R}(\mathcal{U})$ *such that* $\langle D, N\rangle \vdash_f G[t/x]$ *for all* $t \in R$ *where no variable in* R *appears free in* D *or* G
- $\langle D, N\rangle \vdash_f \forall x G$ *iff* $\langle D, N\rangle \vdash_f G[t/x]$ *for some* $t \in \mathcal{U}$
- $\langle D, N\rangle \vdash_f D' \supset G$ *iff* $\langle D \cup \{D'\}, N\rangle \vdash_f G$ *and* $names(heads(D')) \subseteq \mathrm{ass}(N)$

Proof: Obvious.

□

It is not hard to see that the two notions of derivability coincide, as stated in the proposition below.

Proposition 2.3.2 *Let* $\langle P, G\rangle$ *be a derivation pair where* $P = \langle D, N\rangle$. *Then*

1. $P \vdash_s G$ *iff* $P \longrightarrow^+ G$ *is provable.*
2. $P \vdash_f G$ *iff* $P \longrightarrow^- G$ *is provable.*

Proof: As the rules are derived directly from the definitions of $\vdash_s$ and $\vdash_f$, the proof is immediate. □

It is also easy to see that the success (failure) of a goal implies the success (failure) of each of its ground instances. This is formally stated in following proposition.

Proposition 2.3.3 *Let* $\langle P, G\rangle$ *be a derivation pair where* $P = \langle D, N\rangle$. *Then*

1. $P \vdash_s G \Rightarrow P[t/x] \vdash_s G[t/x]$ *for any* $t \in \mathcal{U}$
2. $P \vdash_f G \Rightarrow P[t/x] \vdash_f G[t/x]$ *for any* $t \in \mathcal{U}$

Proof: We proceed by induction on the depth of the relevant **O**-derivation.

In the base case, G is an atom A.

1. As the sequent is initial, $A \in (D)$, and hence $A[t/x] \in (D[t/x])$ for any t, i.e. $P[t/x] \vdash_s A[t/x]$ for any $t \in \mathcal{U}$.
2. As the sequent is initial, $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$, $B \not\propto A$, and hence $\forall B \in (D[t/x])\ B \not\propto A[t/x]$ and $\forall G \supset B \in (D[t/x])$, $B \not\propto A[t/x]$ for any t, i.e. $P[t/x] \vdash_f A[t/x]$ for any $t \in \mathcal{U}$.

Hence the induction hypothesis is that the proposition holds when the relevant **O**-derivation is of no more than a given depth. There are six cases:

A:
1. If the base case does not hold, then we have $\exists G \supset A \in (D)$ such that $P \vdash_s G$, and hence $G[t/x] \supset A[t/x] \in (D[t/x])$, and by the hypothesis $P[t/x] \vdash_s G[t/x]$, and so $P[t/x] \vdash_s A[t/x]$.
2. If the base case does not hold, then we have $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$, $P \vdash_f G$, and by the hypothesis, $P[t/x] \vdash_f G[t/x]$ for any $t \in \mathcal{U}$. Hence $\forall G' \supset B' \in (D[t/x])$ such that $B' \propto A[t/x]$ we have $P[t/x] \vdash_f G'$, and as above $\forall B \in (D[t/x])$ $B \not\propto A[t/x]$, and so $P[t/x] \vdash_f A[t/x]$ for any $t \in \mathcal{U}$.

$\neg A$:
1. $P \vdash_s \neg A$ iff $P \vdash_f A$ and name$(A) \in$ den(N), and by the hypothesis this implies that $P[t/x] \vdash_f A[t/x]$ and name$(A) \in$ den(N) for any $t \in \mathcal{U}$, i.e. $P[t/x] \vdash_s \neg A[t/x]$.
2. $P \vdash_f \neg A$ iff $P \vdash_s A$, and by the hypothesis this implies that $P[t/x] \vdash_s A[t/x]$ for any $t \in \mathcal{U}$, i.e. $P[t/x] \vdash_f \neg A[t/x]$.

$G_1 \vee G_2$:
1. $P \vdash_s G_1 \vee G_2$ iff $P \vdash_s G_1$ or $P \vdash_s G_2$, and by the hypothesis, $P[t/x] \vdash_s G_1[t/x]$ or $P[t/x] \vdash_s G_2[t/x]$ for any $t \in \mathcal{U}$, and so $P[t/x] \vdash_s G_1[t/x] \vee G_2[t/x]$, i.e. $P[t/x] \vdash_s (G_1 \vee G_2)[t/x]$ for any $t \in \mathcal{U}$.
2. $P \vdash_f G_1 \vee G_2$ iff $P \vdash_f G_1$ and $P \vdash_f G_2$, and by the hypothesis, $P[t/x] \vdash_f G_1[t/x]$ and $P[t/x] \vdash_f G_2[t/x]$ for any $t \in \mathcal{U}$, and so

$P[t/x] \vdash_f G_1[t/x] \vee G_2[t/x]$, i.e. $P[t/x] \vdash_f (G_1 \vee G_2)[t/x]$ for any $t \in \mathcal{U}$.

$G_1 \wedge G_2$: 1. $P \vdash_s G_1 \wedge G_2$ iff $P \vdash_s G_1$ and $P \vdash_s G_2$, and by the hypothesis, $P[t/x] \vdash_s G_1[t/x]$ and $P[t/x] \vdash_s G_2[t/x]$ for any $t \in \mathcal{U}$, and so $P[t/x] \vdash_s G_1[t/x] \wedge G_2[t/x]$, i.e. $P[t/x] \vdash_s (G_1 \wedge G_2)[t/x]$ for any $t \in \mathcal{U}$.

2. $P \vdash_f G_1 \wedge G_2$ iff $P \vdash_f G_1$ or $P \vdash_f G_2$, and by the hypothesis, $P[t/x] \vdash_f G_1[t/x]$ or $P[t/x] \vdash_s G_2[t/x]$ for any $t \in \mathcal{U}$, and so $P[t/x] \vdash_f G_1[t/x] \wedge G_2[t/x]$, i.e. $P[t/x] \vdash_f (G_1 \wedge G_2)[t/x]$ for any $t \in \mathcal{U}$.

$\exists yG$: 1. $P \vdash_s \exists yG$ iff $P \vdash_s G[t'/y]$ for some $t' \in \mathcal{U}$ and by the hypothesis, $P[t/x] \vdash_s G[t'/y][t/x]$ for any $t \in \mathcal{U}$, and so $P[t/x] \vdash_s \exists yG[t/x]$ for any $t \in \mathcal{U}$.

2. $P \vdash_f \exists yG$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_f G[t'/y]$ for all $t' \in R$ and by the hypothesis, $P[t/x] \vdash_f G[t'/y][t/x]$ for any $t \in \mathcal{U}$, and as no variables in t' can contain x, we have $P \vdash_f \exists yG[t/x]$ for any $t \in \mathcal{U}$.

$\forall yG$: 1. $P \vdash_s \forall yG$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_f G[t'/y]$ for all $t' \in R$ and by the hypothesis, $P[t/x] \vdash_s G[t'/y][t/x]$ for any $t \in \mathcal{U}$, and as no variables in t' can contain x, we have $P[t/x] \vdash_s \forall yG[t/x]$ for any $t \in \mathcal{U}$.

2. $P \vdash_f \forall yG$ iff $P \vdash_f G[t'/y]$ for some $t' \in \mathcal{U}$ and by the hypothesis, $P[t/x] \vdash_f G[t'/y][t/x]$ for any $t \in \mathcal{U}$, and so $P[t/x] \vdash_f \forall yG[t/x]$ for any $t \in \mathcal{U}$.

$D' \supset G'$: 1. $\langle D, N\rangle \vdash_s D' \supset G'$ iff names(heads(D')) $\subseteq$ ass(N) and $\langle D \cup \{D'\}, N\rangle \vdash_s G'$, and so by the hypothesis we have $\langle (D \cup \{D'\})[t/x], N\rangle \vdash_s G'[t/x]$ for any $t \in \mathcal{U}$. Hence we have that $\langle D[t/x], N\rangle \vdash_s D'[t/x] \supset G[t/x]$, i.e. $\langle D[t/x], N\rangle \vdash_s (D' \supset G')[t/x]$ for any $t \in \mathcal{U}$.

2. $\langle D, N\rangle \vdash_f D' \supset G'$ iff names(heads(D')) $\subseteq$ ass(N) and $\langle D \cup \{D'\}, N\rangle \vdash_f G'$, and so by the hypothesis we have $\langle (D \cup \{D'\})[t/x], N\rangle \vdash_f G'[t/x]$ for any $t \in \mathcal{U}$. Hence we have that $\langle D[t/x], N\rangle \vdash_f D'[t/x] \supset G[t/x]$, i.e. $\langle D[t/x], N\rangle \vdash_f (D' \supset G')[t/x]$ for any $t \in \mathcal{U}$.

□

A possibly surprising result is that there is a weak dual to Proposition 2.3.3, in that for D_{HHF} programs, if a representative set of instances of a goal fails, then so do all sets of representative instances, including the goal itself. This result is proved below.

Proposition 2.3.4 *Let $\langle P, G\rangle$ be a D_{HHF} derivation pair where $P = \langle D, N\rangle$. Let $R \in \mathcal{R}(\mathcal{U})$ be such that the variables in R do not contain any variables which occur free or bound in P or G, and $t' \in \mathcal{T}$ be such that no variable in t' occurs bound in P or G. Then*

$$P[t/x] \vdash_f G[t/x] \textit{ for all } t \in R \Rightarrow P[t'/x] \vdash_f G[t'/x]$$

Proof: We proceed by induction on the depth of the **O**-derivation of $P[t/x] \vdash_f G[t/x]$.

Let $R = \{t_1, \ldots t_n\}$.

In the base case, G is an atom A.

As $P[t_i/x] \vdash_f A[t_i/x]$ for each i, we have $\forall B \in (D[t_i/x])$, $B \not\propto A[t_i/x]$ and $\forall G \supset B \in (D[t_i/x])$, $B \not\propto A[t_i/x]$ for each i. Let $B \in (D[t'/x])$, and so $B = B'[t'/x]$ for some B', i.e. $B'[t'/x] \in (D[t'/x])$, and so $B'[t_i/x] \in (D[t_i/x])$. If $B'[t'/x] \propto A[t'/x]$, then $B'[t_i/x] \propto A[t_i/x]$ for any i, which is a contradiction, and so $B'[t'/x] \not\propto A[t'/x]$. Now let $G \supset B \in (D[t'/x])$, and so $G \supset B = G'[t'/x] \supset B'[t'/x]$ for some $G' \supset B'$, i.e. $G'[t'/x] \supset B'[t'/x] \in (D[t'/x])$, and so $G'[t_i/x] \supset B'[t_i/x] \in (D[t_i/x])$. If $B'[t'/x] \propto A[t'/x]$, then $B'[t_i/x] \propto A[t_i/x]$ for any i, which is a contradiction, and so $B'[t'/x] \not\propto A[t'/x]$. Hence $\forall B \in (D[t'/x])$, $B \not\propto A[t'/x]$, and $\forall G \supset B \in (D[t'/x])$, $B \not\propto A[t'/x]$, and so $P[t'/x] \vdash_f G[t'/x]$.

Hence we assume that the proposition is true for all **O**-derivations of no more than a given depth. There are six cases:

A: If the base case does not hold, then $P[t_i/x] \vdash_f A[t_i/x]$ iff $\forall B \in (D[t_i/x])$ $B \not\propto A[t_i/x]$ and $\forall G \supset B \in (D[t_i/x])$ such that $B \propto A[t_i/x]$, $P[t_i/x] \vdash_f G$. As above, if $B \in (D[t'/x])$, we cannot have $B \propto A[t'/x]$, and so we have that $B \not\propto A[t'/x]$ $\forall B \in (D[t'/x])$. Let $G \supset B \in (D[t'/x])$, and as above $G \supset B = G'[t'/x] \supset B'[t'/x]$ for some $G' \supset B'$, i.e. $G'[t'/x] \supset B'[t'/x] \in (D[t'/x])$. If $B'[t'/x] \propto A[t'/x]$, then $G'[t_i/x] \supset B'[t_i/x] \in (D[t_i/x])$, and $B'[t_i/x] \propto A[t_i/x]$ for all i. Hence we have that $P[t_i/x] \vdash_f G'[t_i/x]$ for all i, and by the hypothesis we have $P[t'/x] \vdash_f G'[t'/x]$. Hence $\forall B \in (D[t'/x])$ $B \not\propto A[t'/x]$ and $\forall G \supset B \in (D[t'/x])$ such that $B \propto A[t'/x]$, $P[t'/x] \vdash_f G$, i.e. $P[t'/x] \vdash_f A[t'/x]$.

$G_1 \vee G_2$: $P[t/x] \vdash_f (G_1 \vee G_2)[t/x]$ iff $P[t/x] \vdash_f G_1[t/x]$ and $P[t/x] \vdash_f G_2[t/x]$, and so by the hypothesis $P[t'/x] \vdash_f G_1[t'/x]$ and $P[t'/x] \vdash_f G_2[t'/x]$, i.e. $P[t'/x] \vdash_f (G_1 \vee G_2)[t'/x]$.

$G_1 \wedge G_2$: $P[t/x] \vdash_f (G_1 \wedge G_2)[t/x]$ iff $P[t/x] \vdash_f G_1[t/x]$ or $P[t/x] \vdash_f G_2[t/x]$, and so by the hypothesis $P[t'/x] \vdash_f G_1[t'/x]$ or $P[t'/x] \vdash_f G_2[t'/x]$, i.e. $P[t'/x] \vdash_f (G_1 \wedge G_2)[t'/x]$.

$\exists yG$: $P[t/x] \vdash_f \exists yG[t/x]$ iff $\exists R' \in \mathcal{R}(\mathcal{U})$ such that $P[t/x] \vdash_f G[t/x][t''/y]$ for all $t'' \in R$, and so as t'' does not contain x and t does not contain y, by the hypothesis $P[t'/x] \vdash_f G[t''/y][t'/x]$ for all $t'' \in R$, i.e. $P[t'/x] \vdash_f \exists yG[t'/x]$.

$\forall yG$: $P[t/x] \vdash_f \forall yG[t/x]$ iff $P[t/x] \vdash_f G[t/x][t''/y]$ for some $t'' \in \mathcal{U}$, and as t does not contain y, by the hypothesis $P[t'/x] \vdash_f G[t''/y][t'/x]$ for some $t'' \in \mathcal{U}$, i.e. $P[t'/x] \vdash_f \forall yG[t'/x]$.

$D' \supset G$: $\langle D[t/x], N\rangle \vdash_f (D' \supset G)[t/x]$ iff $\text{names}(D') \subseteq \text{ass}(N)$ and $\langle (D \cup \{D'\})[t/x], N\rangle \vdash_f G[t/x]$ and so by the hypothesis $\langle (D \cup \{D'\})[t'/x], N\rangle \vdash_f G[t'/x]$, i.e. $\langle D[t'/x], N\rangle \vdash_f (D' \supset G)[t'/x]$.

□

Note that a corresponding result does *not* hold for $\vdash_s$; consider the program below.

$$p(a)$$
$$\forall x\, p(f(x))$$

Now $\{a, f(y)\}$ is a representation, and $p(a)$ and $p(f(y))$ both succeed, but $p(y)$ does not succeed. This result cannot be extended to D_{HHF-} formulae either, as for the above program, $\neg p(y)$ does not fail, although $\neg p(a)$ and $\neg p(f(y))$ both do. The significance of this point will be seen in Section 6.1.

The intuitive reading of $P \vdash_s G$ and $P \vdash_f G$ may be given as "G succeeds" and "G fails" respectively. The validity of this interpretation is shown by the proposition below.

Proposition 2.3.5 *Let $\langle P, G\rangle$ be a derivation pair where $P = \langle D, N\rangle$. Then*

1. $P \vdash_s G \Rightarrow P \not\vdash_f G$
2. $P \vdash_f G \Rightarrow P \not\vdash_s G$

Proof: We proceed by simultaneous induction on the depth of the **O**-proof and **O**-denial for $P \vdash_s G$ and $P \vdash_f G$.

In the base case, the sequent is initial, and hence G is just an atom A.

1. As $P \longrightarrow^+ A$ is initial, we have that $A \in (D)$, and so as $A \propto A$, it is not the case that $\forall B \in (D)\ B \not\propto A$, and by Proposition 2.3.1 it is not the case that $P \vdash_f A$.
2. As $P \longrightarrow^- A$ is initial, we have that $\forall B \in (D)$, $B \not\propto A$ and $\forall G \supset B \in (D)$, $B \not\propto A$, and so as $A \propto A$, we must have $A \notin (D)$ and $\nexists G \supset A \in (D)$, and so by Proposition 2.3.1 it is not the case that $P \vdash_s A$.

Hence the induction hypothesis is that the proposition holds for all derivation states whose **O**-proof or **O**-denial is no more than a given depth.

There are seven cases:

A: 1. $P \vdash_s A \Rightarrow \exists G \supset A \in (D)$ such that $P \vdash_s G$, and by the hypothesis it is not the case that $P \vdash_f G$. Hence, it is not the case that $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$ we have $P \vdash_f G$, and by Proposition 2.3.1 it is not the case that $P \vdash_f A$.

2. $P \vdash_f A \Rightarrow \forall B \in (D), B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$ we have $P \vdash_f G$. Now as $A \propto A$, we must have that $A \notin (D)$. Also, by the hypothesis it is impossible that $\exists G \supset A \in (D)$ such that $P \vdash_s G$, and so it is impossible that $P \vdash_s A$.

$\neg A$: 1. $P \vdash_s \neg A \Rightarrow \text{name}(A) \in \text{den}(N)$ and $P \vdash_f A$, and by the hypothesis it is impossible that $P \vdash_s A$, and so it is impossible that $P \vdash_f \neg A$.

2. $P \vdash_f \neg A \Rightarrow P \vdash_s A$, and by the hypothesis it is impossible that $P \vdash_f A$, and so it is impossible that $P \vdash_s \neg A$.

$G_1 \vee G_2$: 1. $P \vdash_s G_1 \vee G_2$ iff $P \vdash_s G_1$ or $P \vdash_s G_2$ and by the hypothesis, this implies that either it is not the case that $P \vdash_f G_1$ or it is not the case that $P \vdash_f G_2$, i.e. it is not the case that $P \vdash_f G_1$ and $P \vdash_f G_2$, and so it is not true that $P \vdash_f G_1 \vee G_2$.

2. $P \vdash_f G_1 \vee G_2$ iff $P \vdash_f G_1$ and $P \vdash_f G_2$ and by the hypothesis, this implies that it is not the case that $P \vdash_s G_1$ and it is not the case that $P \vdash_s G_2$, i.e. it is not the case that either $P \vdash_s G_1$ or $P \vdash_s G_2$, and so it is not true that $P \vdash_s G_1 \vee G_2$.

$G_1 \wedge G_2$: 1. $P \vdash_s G_1 \wedge G_2$ iff $P \vdash_s G_1$ and $P \vdash_s G_2$ and by the hypothesis, this implies that it is not the case that $P \vdash_f G_1$ and it is not the case that $P \vdash_f G_2$, i.e. it is not the case that either $P \vdash_f G_1$ or $P \vdash_f G_2$, and so it is not true that $P \vdash_f G_1 \wedge G_2$.

2. $P \vdash_f G_1 \wedge G_2$ iff $P \vdash_f G_1$ or $P \vdash_f G_2$ and by the hypothesis, this implies that either it is not the case that $P \vdash_s G_1$ or it is not the case that $P \vdash_s G_2$, i.e. it is not the case that $P \vdash_s G_1$ and $P \vdash_s G_2$, and so it is not true that $P \vdash_s G_1 \wedge G_2$.

$\exists x G$: 1. $P \vdash_s \exists x G$ iff $P \vdash_s G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis, this implies that it is not the case that $P \vdash_f G[t/x]$. Now if $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_f G[t'/x]$ for all $t' \in R$, then by Proposition

2.3.3 $P \vdash_f G[t/x]$ for all $t \in \mathcal{U}$, as no variables of R appear free in P. Hence it is impossible that $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_f G[t/x]$ for all $t' \in R$, i.e. it is not true that $P \vdash_f \exists x G$.

2. $P \vdash_f \exists x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_f G[t/x]$ for all $t \in R$, and by Proposition 2.3.3, $P \vdash_f G[t/x]$ for all $t \in \mathcal{U}$, as no variables of R appear free in P. By the hypothesis this implies that it is not the case that $P \vdash_s G[t/x]$ for any $t \in \mathcal{U}$, and so it is impossible that $P \vdash_s G[t/x]$ for some $t \in \mathcal{U}$, i.e. it is not true that $P \vdash_s \exists x G$.

$\forall x G$: 1. $P \vdash_s \forall x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_s G[t/x]$ for all $t \in R$, and by Proposition 2.3.3, $P \vdash_s G[t/x]$ for all $t \in \mathcal{U}$, as no variables of R appear free in P. By the hypothesis this implies that it is not the case that $P \vdash_f G[t/x]$ for any $t \in \mathcal{U}$, and so it is impossible that $P \vdash_f G[t/x]$ for some $t \in \mathcal{U}$, i.e. it is not true that $P \vdash_f \forall x G$.

2. $P \vdash_f \forall x G$ iff $P \vdash_f G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis, this implies that it is not the case that $P \vdash_s G[t/x]$ for some $t \in \mathcal{U}$. Now if $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_s G[t'/x]$ for all $t' \in R$, then by Proposition 2.3.3 $P \vdash_s G[t/x]$ for all $t \in \mathcal{U}$, as no variables of R appear free in P. Hence it is impossible that $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_s G[t/x]$ for all $t' \in R$, i.e. it is not true that $P \vdash_s \forall x G$.

$D' \supset G'$: 1. $\langle D, N\rangle \vdash_s D' \supset G'$ iff names(heads((D'))) $\subseteq$ ass(N) and $\langle D \cup \{D'\}, N\rangle \vdash_s G'$, and by the hypothesis, this implies that it is not the case that $\langle D \cup \{D'\}, N\rangle \vdash_f G'$, and so it is not true that $\langle D, N\rangle \vdash_f D' \supset G'$.

2. $\langle D, N\rangle \vdash_f D' \supset G'$ iff names(heads((D'))) $\subseteq$ ass(N) and $\langle D \cup \{D'\}, N\rangle \vdash_f G'$, and by the hypothesis, this implies that it is not the case that $\langle D \cup \{D'\}, N\rangle \vdash_s G'$, and so it is not true that $\langle D, N\rangle \vdash_s D' \supset G'$.

□

Thus the above definitions are consistent. Note that a similar result will hold

for the case when universal quantification is interpreted intensionally (i.e. using the new constant rule).

This result suggests that it is consistent to define a more general form of negation. If we let G be a goal in which all predicates which occur in positive positions are in den(N) and ll those in negative positions are in ass(N), then we may define an extension of the relations $\vdash_s$ and $\vdash_f$ as follows:

- $P \vdash_s \neg G$ iff $P \vdash_f G$
- $P \vdash_f \neg G$ iff $P \vdash_s G$

From the above result we see that $P \vdash_f G$ implies that $P \vdash_s G$ does not hold, and so $P \vdash_s \neg G$ is a consistent conclusion. A similar argument holds for the other case, and so this indicates how we may implement negation for a wider class of formulae than just atoms. This theme is taken up in section 3.2.

2.4 Discussion

2.4.1 Terms and Universal Quantification

A distinctive feature of the definition of $\vdash_s$ compared to that of $\vdash_o$ is that the rule for universally quantified goals is defined in terms of representations, of which there may be infinitely many. However, we know that if an atom succeeds, then every instance of it succeeds, and so a successful search for a proof of a universally quantified goal will only require a finite number of instances to be tested. For example, if Σ is $\{a/0, f/1\}$, consider the program below.

$$p(a)$$
$$\forall x\ p(f(x))$$

It is clear that $p(a)$ and $p(f(y))$ succeed, and so $\forall x\ p(x)$ succeeds. Clearly it is not necessary to consider a representation such as $\{a, f(a), f(f(x))\}$. On the

other hand, an implementation will not need to consider inductive methods. For example, consider the program

$$p(a)$$
$$\forall x\ p(x) \supset p(f(x))$$

In this case $p(t)$ succeeds for every ground term t, but every non-ground term neither succeeds nor fails, and so there is no representation R such that $p(t)$ succeeds for all $t \in R$. Hence it is unnecessary for us to consider inductive methods of proof.

There are some cases in which an inductive method of proof would indeed be useful, however. For example, given the program

$$\forall x\ \text{less}(0, s(x))$$
$$\forall x \forall y\ \text{less}(x, y) \supset \text{less}(s(x), s(y))$$

it seems reasonable to conclude that $\forall x\, \text{less}(x, s(x))$ is true, but as the definition of less is recursive, we need an inductive method of proof in order to reach this conclusion. One such way would be to replace the quantified variable with a new constant, and "expand" the constant where necessary, using our knowledge of the Herbrand Universe. For example, the success of the above goal would be found by the following derivation.

$$P \vdash_s \forall x\, \text{less}(x, s(x))$$
$$P \vdash_s \text{less}(c, s(c))$$
$$P \vdash_s \text{less}(0, s(0)) \wedge \forall x\, \text{less}(x, s(x)) \supset \text{less}(s(x), s(s(x)))$$
$$P \vdash_s \text{less}(c, s(c)) \supset \text{less}(s(c), s(s(c)))$$
$$\text{less}(c, s(c)), P \vdash_s \text{less}(s(c), s(s(c)))$$
$$\text{less}(c, s(c)), P \vdash_s \text{less}(c, s(c))$$

and so we get that $P \vdash_s \forall x\, \text{less}(x, s(x))$.

The key step is the replacement of $\text{less}(c, s(c))$ by

$$\text{less}(0, s(0)) \wedge \forall x\, \text{less}(x, s(x)) \supset \text{less}(s(x), s(s(x)))$$

Clearly it is not hard to specify an induction scheme for goals such as $\forall x\, p(x)$ where the signature of p is $\{a_1/0, \ldots a_n/0, f_1/1, \ldots f_m/1\}$. The problem is more complicated when considering a general goal $\forall x G$, and signatures which contain functions of a number of different arities. One way to proceed is to substitute a new constant for each universally quantified variable in the goal, and then deal with each of the new constants as they are encountered when the atomic parts of the goal are reached. In this way we "delay" the decision of what to do for each variable until we are forced to decide, similar to the way that the choice of substitution for existentially quantified variables may be delayed by the use of unification. The techniques of Bundy et al. [12,13] may be useful in this context.

Such an inductive method will presumably be somewhat intricate and computationally expensive, and so we do not pursue it here. It should be noted that whilst our approach makes use of a known set of terms, it does so in a way that is compact, as noted above, and so no inductive properties are needed.

An observation which is relevant to this discussion is that we may think of our method of success for universal quantification as similar to that of an intermediate logic. One such logic is the *logic of constant domains* [34,21]. This logic is characterised by the property that in order to establish $\forall x F$ it is not necessary to establish that $\forall x F$ must always hold, no matter what extra assumptions are made, but only to show that $F[t/x]$ is true for all terms t. A proof theoretic characterisation is given by adding the following inference rule to those of intuitionistic logic:

$$\forall x\, (\phi \vee \psi(x)) \supset (\phi \vee \forall x\, \psi(x))$$

where x is not free in ϕ.

Such an inference rule is precisely what we would expect from the considerations discussed above, as both sides of the implication become $\phi \vee \psi(c)$ when we replace the universal quantifier with a new constant. In our case we go one step further, in that not only is the domain constant, but it is known.

Hence our interpretation of the universal quantifier is a natural one when the Herbrand Universe is fixed in advance. This theme is taken up in section 6.5.

One example for which the intensional interpretation (i.e. the "new constant" version) is perhaps more appropriate is in an example due to Miller [73]. If we wish to deduce a rule from the constitution of the USA such as the one below

$$\forall x \text{ president}(x) \supset \text{US\_citizen}(x)$$

then the intensional interpretation is perhaps more appropriate than the extensional one. The reason is that it is not now known who all the U.S. Presidents will be, although we do know all the U.S. Presidents up to date. Hence, there is a difference between the extensional interpretation (i.e. all presidents up to the present have had this property) and the intensional interpretation (i.e. there is a rule in the constitution which states that the U.S. President must be a U.S. citizen). The difference is more striking for the goal

$$\forall x \text{ president}(x) \supset \text{white\_male}(x)$$

This is true for all U.S. Presidents so far, but there is no corresponding rule in the U.S. constitution.

One way to develop a unified framework for these two possibilities is to consider some signatures as closed and some as open. For example, the signature of all Catholic monarchs of Great Britain would be considered closed, as the Act of Settlement of 1701 ensures that no Catholic may occupy the throne. On the other hand, the signature of U.S. Presidents (or that of all British monarchs) would be considered open, as there are still future Presidents (and monarchs) whose identity is unknown. Hence for closed signatures, the extensional interpretation is appropriate, whereas for open signatures, the intensional one is probably more appropriate. In the latter case a modal interpretation is probably better still, so that we can distinguish between what must be true (for example, the U.S. citizen rule) from what happens to be true now, but need not be so in the future (e.g. the white male rule). Such notions of necessity and possibility coincide when the signature is closed.

As mentioned above, we only consider the case where the terms of the Herbrand Universe are generated only from the symbols in the signature. In order to allow for polymorphic predicates, we need to relax this restriction by specifying in more detail how the relevant terms may be built. For example, the arguments to append must be lists, but the elements of the list may be of any kind whatsoever. This may be thought of as adopting a type system for programs and goals, so that each term must have a type. The adoption of a type system, such as those of [111,85, 90,84], would provide the framework in which we may use induction over a much wider class of programs; however the issues involved will be essentially the same as those discussed above.

2.4.2 Answer Substitutions

A well-known limitation of the NAF rule is that it is only defined for ground atoms. A naive approach to extending it to include non-ground atoms is for $\neg p(x)$ to succeed iff $p(x)$ fails. This approach is implemented in many Prolog systems, but may lead to counterintuitive behaviour. For example, given the program

$$p(a)$$
$$q \supset p(b)$$

the goal $\neg p(x)$ fails as $p(x)$ succeeds. However, it is clear that $\neg p(b)$ succeeds because $p(b)$ fails.

One way to interpret this approach is to consider $\neg p(x)$ to be universally quantified, so that the goal is actually $\forall x \neg p(x)$, which is equivalent (both intuitionistically and classically) to $\neg \exists x\, p(x)$. This may be seen as a direct counterpart to the goal $\exists x\, p(x)$ in that $\forall x \neg p(x)$ succeeds precisely when $\exists x\, p(x)$ fails.

This is not the only possible extension of NAF to non-ground atoms (although it is clearly easy to implement). An obvious alternative is to consider the quantification as $\exists x \neg p(x)$, so that $\neg p(x)$ succeeds iff there is an instance of $p(x)$ which fails. In this case, for the program above, we expect the answer substitution $x \leftarrow b$ to be returned for the goal $\exists x \neg p(x)$. Note that the previous case does not require

an answer substitution as it involves a universally quantified formula. This case has a more subtle symmetry with $\exists x\, p(x)$ in that $\exists x\, p(x)$ succeeds iff there is an instance $p(t)$ of $p(x)$ which succeeds, whereas $\exists x \neg p(x)$ succeeds iff there is an instance $p(t)$ of $p(x)$ which fails.

This interpretation of NAF requires a more powerful way to generate answer substitutions than the unification methods employed in SLD-resolution. The standard method by which answer substitutions are generated is by unifying an atomic goal with the head of a clause in the program, applying the resulting substitution to the body, and then applying the answer substitution for this instance of the body to the variables of the original goal. Clearly we cannot generate any answer substitutions for negated goals in this way, as the answer substitutions are only generated for successful goals. Thus a more general method which searches for instances of $p(x)$ which fail rather than for instances of $p(x)$ which succeed is needed in order to use formulae such as $\exists x \neg p(x)$ as goals.

A limited implementation of this idea is present in some Prolog systems, such as Mu-Prolog [88], which uses a delay mechanism for non-ground negated goals. This technique delays the processing of the goal $\neg p(x)$ until some other goal produces a substitution under which x becomes a ground term. For example, given the goal $\exists x(\neg p(x) \wedge q(x))$, the processing of $\neg p(x)$ is delayed until after the processing of $q(x)$, and if this succeeds with the answer substitution being $x \leftarrow t$ where t is a ground term, then the NAF rule is directly applied by checking that $p(t)$ fails. In this way answer substitutions can only be generated by non-negated goals, and these are then filtered by the negated goals. Naturally a ground substitution may not occur, in which case no more processing of negated goals may occur. When such an impasse is reached, the goal is said to *flounder* [63]. As there are many goals which flounder, such as $\exists x \neg p(x)$, it is clear that this technique can be nothing more than approximation to the desired process.

In chapter 4 we give an algorithm for constructing answer substitutions for non-ground negated goals. The existence of such an algorithm allows us to define the notion of success for any existentially quantified goal in a uniform way, i.e. that $\exists x G$ succeeds iff there is a term $t \in \mathcal{U}$ such that $G[t/x]$ succeeds. In this way we

may think of this algorithm as a justification for this definition. One interesting possible connection between such an algorithm and an inductive algorithm for universally quantified goals may be that if the induction mechanism used for the universal quantification will always produce a counterexample when the goal fails, then we may implement the generation of answer substitutions for $\exists x \neg p(x)$ by searching for a proof of $\forall x p(x)$. If this search is successful, then $\exists x \neg p(x)$ fails. If the search fails, producing a counterexample $p(t)$, then $x \leftarrow t$ is an answer substitution for $\exists x \neg p(x)$. In this way any such induction mechanism may be useful in this context as well. This point is discussed more fully in section 4.5.

2.4.3 Incomplete Definitions and Inconsistency

The fact that NAF is an implicit form of negation means that we only need to extend goals to include negated atoms to capture it, as programs do not need to state explicitly which formulae are false. In contrast, any form of negation for incompletely defined predicates will need to explicitly state negative information, and so one way of incorporating such a form of negation is to allow negated atoms to play the same role as atoms, so that we build up programs and goals from literals, rather than atoms alone. The definition of D and G formula may now be given as

$$L := A \mid \neg A$$
$$D := L \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset L$$
$$G := L \mid \forall x G \mid \exists x G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G$$

Here we use L to stand for a literal. Note that the formula $G \supset \neg A$ is a definite formula. In this way we may think of the extension as allowing our basic conclusions to consist of both positive and negative pieces of information (literals), rather than positive information alone (atoms). Hence we may use clauses such as

$$\forall x \neg \text{carcinogen(x)} \subset \text{unfertilised}(x) \wedge \text{unprocessed}(x)$$

meaning that anything which has been grown without fertiliser and has not been processed is not a carcinogen.

Note also that this merely extends the conclusions that may be drawn to include negative information, as well as positive information. Hence this extension adds some extra colour to the class of first-order hereditary Harrop formulae, but does not change their fundamental computational properties, as we only allow negation to be applied to atoms.

We need to extend the definition of (D), which is done in the obvious way, i.e. that $(\neg A) = \{\neg A\}$.

Another needed extension is to the definitions of $\vdash_s$ and $\vdash_f$ to cope with this extended class of programs and goals. For example, we want $\vdash_s$ to have the property that

$$\langle D, N\rangle \vdash_s \neg A \text{ iff } \neg A \in (D) \text{ or } \exists G \supset \neg A \in (D) \text{ such that } \langle D, N\rangle \vdash_s G \text{ or } \langle D, N\rangle \vdash_f A \text{ and name}(A) \in \text{den}(N)$$

This ensures that the NAF rule is only applied to completely defined predicates of P, i.e. those whose names appear in den(N), and that negation for incompletely defined predicates is computed in a similar manner to that for positive literals.

An important question which arises for such programs is the question of consistency. For example, it is obvious that the program $\neg A \wedge A$ is inconsistent. We may define consistency as follows:

Definition 2.4.1 *A program P is* consistent *if there is no atom A such that $P \vdash_s A \wedge \neg A$. Otherwise P is* inconsistent.

As discussed in [77], the question of what is to be done with inconsistent programs depends on the context in which the inconsistency occurs. In some cases, the approach of full intuitionistic logic may be appropriate, i.e. that if a contradiction is found, then all goals are provable. Various mathematical examples suggest themselves, such as if we find that both even(0) and ¬even(0), then we

would think it reasonable to disregard all the knowledge contained in the program, and so deduce that every goal is provable. On the other hand, knowledge bases and the like often have to deal with inconsistent information in a less mathematical way. For example, in a parent-children database, we may have a relation named mother. If we find that both mother(Mary, Jane) and $\neg$ mother(Mary, Jane) are true, then we have an obvious inconsistency, but it seems difficult to see how this information should effect the truth of mother(Gladys, Fred) or father(Alan, Jenny). In this way the inconsistency would be dealt with locally, and without forcing all other formulae, such as carcinogen(chocolate) $\wedge\neg$carcinogen(chocolate), to be true. Thus the inconsistency may be thought of as being inherently local, in that the number of formulae affected by the contradiction is comparatively small. In a more mathematical setting, it may be considered that the interdependence is much greater, and so it is more reasonable to expect that most, if not all, formulae become trivially true. In [77] it was shown that $\vdash_o$ may be trivially extended to capture the intuitionistic notion of inconsistency. We merely note that there is a similar notion to that of completely and incompletely defined predicates, in that we may assume that the default solution is to use the minimal version, and that in special cases which warrant it, we may use the stronger intuitionistic version. We may then adopt a similar solution by allowing the programmer to specify a dependency relation between predicates, so that if a contradiction is found in one predicate, then we can immediately tell which predicates may also contain suspect information and which are independent of the contradiction.

Such problems with inconsistency can also arise from consistent programs. For example, we would expect the behaviour of

$$\neg A \wedge A \vdash_s G$$

to be the same as that of

$$\neg A \vdash_s A \supset G$$

and so anything which applies to the first case should also apply to the second. Note that this problem does not arise when negation could only appear in G

formulae. In this case it may be arbitrarily difficult to determine whether we may "safely" assume A, as checking the consistency of P extended by A will involve determining whether $P \vdash_s \neg A$ is true or not.

A similar problem is encountered if we allow formulae of the form $\neg A$ where name$(A) \in$ den(N) as definite formulae in order to memoise successful goals of the form $\forall(\neg A)$. This involves adding the successful goals to the program, so that other goals may use the memoised ones as a shortcut, rather than recomputing known results. The class of programs so obtained is consistent, as for $\neg A$ to succeed we must have that A fails. However, the operational provability relation given above does not allow completely defined predicates to be extended. The precise nature of this difficulty is discussed in section 7.1; for now we note that a method for dealing with inconsistent programs will be useful for this task as well. Thus a minimal approach to inconsistency may be a way of allowing more forms of negation than NAF.

2.4.4 Computational Aspects

One question that comes to mind when dealing with negation in a constructive context is the relation between rules such as NAF and the intuitionistic interpretation of $\neg A$ as $A \supset \bot$. In [77] it was shown how we may implement this form of negation by allowing $\bot$ as a distinguished atom, so that $\bot$ may be the head of a clause. We may then compute $\langle D, N\rangle \vdash_s A \supset \bot$ via $\langle D \cup \{A\}, N\rangle \vdash_s \bot$. Thus this form of negation seems natural for incompletely defined predicates. Now in order to implement negation in a uniform way, as well as to directly justify NAF in terms of intuitionistic logic, we may wish to implement negation for completely defined predicates in the same way. This will involve the addition of negative information to the program in order to make this identification, as otherwise there is no way to derive a contradiction from the program extended with the assumption A. The standard way to add such information is by the completion of the program. The details of this process are discussed in chapter 3, but for now we may think of this process as adding negative information to the program in such a way that if

$P \vdash_s \neg A$, then from the completion of the program we may derive $A \supset \bot$. In this way we can see the completion of a program as making NAF explicit, in that there is a given theory in which $A \supset \bot$ is true iff A fails.

One problem with deriving $\neg A$ via $A \supset \bot$ is that the latter requires that a completely defined predicate be extended, and the rules for $\vdash_s$ as described above prevent this. However it seems natural to make an exception in this case, as we are trying to use the negative information in the completion to show that the assumption of a given atom leads to an inconsistency. One way around this problem is to exclude N from consideration, so that we consider the completion of a program $P = \langle D, N\rangle$ as just a set of clauses P', which we may think of as a shorthand for $\langle P', \langle \text{names}(\mathcal{H}), \emptyset\rangle\rangle$.

This will lead to problems with goals of the form $D \supset G$ where G is not $\bot$ and the head of D is a completely defined predicate, in that such goals may succeed from the completion but do not succeed from the original program. It may be argued that this is reasonable; we think of the completion as an explicit statement of what is known, and as a result, it is inappropriate to use implicit forms of information. This also means that it will not be true that $p \supset \bot \vdash_s p \supset G$ where G is any goal, and hence this method cannot be expected to be complete for intuitionistic logic. This in itself is not a great problem, as we are seeking to represent the implicit inferences made from the program in an explicit form, and not to implement a given logical system. As mentioned above, the intuitionistic approach to inconsistency is probably not the most appropriate one in this context anyway, and so it seems that this approach is defensible. However, we prefer to make an exception to the rule for implication, so that goals of the form $A \supset \bot$ are always computed via $\langle D, N\rangle \vdash_s A \supset \bot$ iff $\langle D \cup \{A\}, N\rangle \vdash_s \bot$, but all other implications must use the rule given in section 2.3. This will make it easier to state and prove results, as well as reinforcing the perception that the completion is used to capture the operational properties of the program, rather than a semantic device per se. In this way the completion will only alter the computation of negated atoms, and not any other goals, which seems more appropriate than altering an established and understood method.

Now if we imagine trying to write the completion as a set of clauses, rather than a meta-level formula in the manner of Clark [17], then we will need to be able to state negative conclusions. The natural way to do so is to write a clause of the form $G \supset \neg A$ as $(G \wedge A) \supset \bot$, from which it is easy to see that $A \supset \bot$ succeeds if G succeeds. In this way the completion allows us to implement negation in a uniform and explicit way, provided we can make the $A \supset \bot$ mechanism work. In such an implementation there is little distinction between completely and incompletely defined predicates, as all the information is given explicitly.

One problem with this way of implementing negation is that we may prove undesired results. For example, the clause $p \supset \neg p$ would be written as $(p \wedge p) \supset \bot$, which is obviously the same as $p \supset \bot$, i.e. $\neg p$, which may not be what was intended. As we shall see, the choice of logic is crucial for issues such as these.

Another problem is that we may get loops where they are not expected. For example, consider the program $\neg p \supset \neg q$ and the goal $\neg q$. We would expect this goal to fail, as it is perfectly consistent to add q to the program. Clearly $(p \supset \bot) \supset (q \supset \bot)$ is equivalent to $((p \supset \bot) \wedge q) \supset \bot$. We then get the following (infinite) derivation sequence

$$
\begin{aligned}
((p \supset \bot) \wedge q) \supset \bot &\vdash_s q \supset \bot \\
q, ((p \supset \bot) \wedge q) \supset \bot &\vdash_s \bot \\
q, ((p \supset \bot) \wedge q) \supset \bot &\vdash_s (p \supset \bot) \wedge q \\
q, ((p \supset \bot) \wedge q) \supset \bot &\vdash_s p \supset \bot \\
p, q, ((p \supset \bot) \wedge q) \supset \bot &\vdash_s \bot \\
p, q, ((p \supset \bot) \wedge q) \supset \bot &\vdash_s (p \supset \bot) \wedge q \\
p, q, ((p \supset \bot) \wedge q) \supset \bot &\vdash_s \bot \\
&\ldots
\end{aligned}
$$

The problem is that we are always able to match $\bot$ with the clause

$$((p \supset \bot) \wedge q) \supset \bot$$

and so we loop. However, it is clear that we are, in a sense, matching the goal $\neg p$ with the "head" $\neg q$, and so it is the ambiguity behind the use of $\bot$ that is the

problem. In this way the translation of $(p \supset \bot) \supset (q \supset \bot)$ to $((p \supset \bot) \wedge q) \supset \bot$ loses some important computational information, i.e. that it is $\neg q$ which is the real "head" of the clause. If we were only to match the goal $p \supset \bot$ with itself, rather than with $q \supset \bot$ as above, then it is possible to avoid this looping process. This seems desirable, as the assumption of q does not make the above program inconsistent. This may be thought of as binding the $\bot$ more closely to q than to the other atom, and so we need to be aware of the context in which we are searching for a contradiction. Hence it may be more enlightening to write such clauses as $G \supset (A \supset \bot)$ with the head of the clause considered as $\neg A$, so that we may consider a literal $\neg B = B \supset \bot$ to match the head of this clause iff A matches B. This may be thought of as a delaying process; for purposes of unification we consider a negative literal to be in its "closed" or compact form $\neg A$, but after this, the deduction component uses the "open" form $A \supset \bot$.

However, this technique will not work on arbitrary programs. The problem is that there may be other ways to prove $\bot$ from the program, especially when it is extended by an assumption. For example, consider the program below.

$$q \wedge (p \supset \neg q)$$

It is clear that if p is assumed, then we have a contradiction, and so we desire $p \supset \bot$ to succeed. However, if we use the strategy described above, $p \supset \bot$ will fail, as there is no clause head which matches either p or $\neg p$. The more general rule will indeed find that $p \supset \bot$ succeeds, as shown by the derivation sequence below.

$$\begin{aligned} q, (p \wedge q) \supset \bot &\vdash_s p \supset \bot \\ p, q, (p \wedge q) \supset \bot &\vdash_s \bot \\ p, q, (p \wedge q) \supset \bot &\vdash_s p \wedge q \end{aligned}$$

Thus the simplified procedure will only work if we know that the only way for $\neg A$ to succeed is for A to be an instance of an atom B such that there is a clause head $(B \supset \bot)$. We shall see in section 3.2 that our version of the completion is such a program. We may think of any class of programs which satisfies this property

as one in which we use the closed form of a negative literal for deduction as well as unification. A consequence of this is that the two clauses $\neg A$ and $A \supset \neg A$ are now no longer operationally equivalent. This may been seen by the fact that the first clause ($\neg A$) is just a negative fact, and so $\neg A \vdash_s \neg A$, as the computation terminates with success after the unification step, whereas $A \supset \neg A \not\vdash_s \neg A$, as the next goal produced is A, which does not match anything, and so fails. This corresponds to the approach taken to inconsistency in minimal logic, which may be thought of as intuitionistic logic without the rule that any formula may be derived from a contradiction. Hence, the difference between intuitionistic and minimal deduction in this context may be characterised by the strength of the binding of $A \supset \bot$, and that any such class of programs will use a minimal rather than intuitionistic approach to inconsistency.

In this way the computational nature of $A \supset \bot$ seems to indicate that the full intuitionistic approach to inconsistency is less natural in this context than the minimal one. We take up this theme in section 5.6.

2.4.5 Stratification

For model-theoretic reasons, it is common to restrict the use of negation in the bodies of clauses. One such restriction is to insist that programs be *stratified* [4,54, 55,62], i.e. that it be possible to divide the predicates of the program into layers so that each predicate p may depend positively on predicates in its own layer or lower down, but may only depend negatively on predicates in lower layers. This means that negated predicates may only be used by other predicates once the negated predicates are fully defined.

For example, program P_1 is stratified, but programs P_2 and P_3 are not.

P_1	P_2	P_3
$\text{even}(0)$	$\text{even}(0)$	$\text{even}(0)$
$\text{even}(x) \supset \text{even}(s^2(x))$	$\text{odd}(x) \supset \text{even}(s(x))$	$\neg\text{even}(x) \supset \text{even}(s(x))$
$\text{even}(x) \supset \neg\text{odd}(x)$	$\neg\text{even}(x) \supset \text{odd}(x)$	

The restriction to stratified programs allows the model theory of such programs to be given in a manner known as the *iterated fixpoint semantics* [4]. This is done by dividing the program up into the strata suggested by the above definition, so that we know that if p depends on $\neg q$, then q is defined in a lower stratum than p. We may then apply the standard fixpoint semantics to the lowest stratum, as it can contain no negations, and then use this interpretation to construct a similar fixpoint for the two lowest strata by noticing that any negated atom $\neg q$ is such that q is defined in the lowest stratum, and so we may determine the truth of $\neg q$ from the model of the lowest stratum. Otherwise we may proceed as normal, thus getting a model for the second lowest stratum, as well as for the lowest. We then continue this process for all remaining strata.

One problem with this approach is that there are non-stratified programs which seem useful, and so this restriction is stronger than we would like. Whilst P_2 and P_3 above are not stratified, their operational behaviour is known; for example it is clear that the goal even($s^2(0)$) succeeds for both programs (assuming that negation is computed via the NAF rule). We may think of stratified programs as defining a well-ordered hierarchy of predicates. There is a similar hierarchy defined over ground atoms by both P_2 and P_3, and so it seems that a refinement of the idea of stratification will lead to a weaker restriction. A weaker form of stratification based on this idea has been proposed by Przymusinski [94,95], in which the only restriction is that a ground atom A may not depend upon its negation, and so programs such as $\neg p \supset p$ are not allowed. The reasons for having any restriction at all are semantic rather than operational; for the program $\neg p \supset p$ it is obvious that both the goals p and $\neg p$ loop, and so the computational nature of this program is known, although it is not very enlightening. The programs which obey this weaker restriction are known as *locally stratified.* Similar weaker forms of stratification have also been studied [32,41,14,93].

Whilst this is a reasonable restriction, in that the programs outside this class do not seem very useful, we feel that no restriction should be placed on the class of programs. This is because all programs have some clear, if perhaps eccentric, operational behaviour, and so this should be captured in the semantics of such

programs, rather than ignored. A constructive approach can help in this regard, as is it natural for formulae to exist which are neither true nor false, and so we may consider programs such as $\neg p \supset p$ as both operationally and model-theoretically indeterminate, in that p neither succeeds nor fails, and so p should be neither true nor false. This cannot be done in classical logic, as we must have that p is either true or false, and so we can only capture such operational behaviour in a three-valued logic or something similar, rather than the direct approach which is possible in intuitionistic logic. Perhaps there is little philosophical difference between the alternatives of defining troublesome programs out of existence or allowing them to exist but ignoring them, but it seems important from a computer scientist's point of view that all programs be given an interpretation. Naturally there are some programs for which there is no sensible interpretation, but it seems preferable that programs of little practical use be given a corresponding interpretation, rather than being ignored.

One interesting example of this was given by Przymusinska and Przymusinski [93], in which the authors show how the program P_1 below, which is not locally stratified, may be converted to program P_2, which is locally stratified, and preserves the operational properties of P_1.

P_1	P_2
$p(1,2)$	$p(1,2)$
$p(1,2) \wedge \neg q(2) \supset q(1)$	$p(1,2) \wedge \neg q(2) \supset q(1)$
$p(1,1) \wedge \neg q(1) \supset q(1)$	
$p(2,2) \wedge \neg q(2) \supset q(2)$	
$p(2,1) \wedge \neg q(1) \supset q(2)$	

The reason that these two programs are operationally equivalent is that as $p(1,1)$, $p(2,2)$ and $p(2,1)$ all fail, nothing can ever succeed from the last three clauses, and so they cannot make any contribution to the set of goals which succeed. However, it is the presence of these three clauses which prevents P_1 from being locally stratified. This suggests that it is really the operational behaviour of the program that should "drive" the model theory, in that two programs which

are equivalent operationally should have the same model. We will see in chapter 5 how this concept may be used to derive a model theory which is valid for all programs, so that no restriction needs to be placed on the class of programs.

A similar problem exists with the completion of programs. Clark's completion was restricted to Horn clause programs, so that no negations could appear in the bodies of clause. This restriction has the property of ensuring that the completion is never inconsistent, as there can never be a program P such that comp(P) contains the formula $A \leftrightarrow \neg A$. However, the weaker restrictions discussed above, such as local stratification, may also be used to ensure the consistency of the completion [14,103]. Now even though the completion is to be used for operational reasons rather than model theory, it seems one useful approach is to define the completion for locally stratified programs only. As the clause $\neg A \supset A$ may be interpreted under NAF as meaning A succeeds if A fails, it is difficult to see how such clauses can convey any information relevant to the computation of the negation of completely defined predicates, as there is no clear definition of derivability from such a clause. Hence it seems reasonable to exclude such clauses from any completion process in which we wish to allow negations to appear in the body of clauses.

This is related to the earlier discussion on the difference between NAF and the (global) CWA. As the goal A loops forever for the program $\neg A \supset A$, the CWA would imply $\neg A$, as A does not succeed, whereas NAF does not infer $\neg A$ as A does not fail. Hence if we view the completion as making explicit the operational behaviour of the program, we do not lose very much by restricting the class of programs in this way.

2.5 Notation

Here we list the various classes of formulae which will be used in later chapters. As can be seen from the definitions below, the main difference between all the classes is the formulae which may be used as goals. In most cases we wish to consider

the addition of negation in goals, and so we will give first the "positive only" form, followed by the more general one. Generally the largest class of programs, i.e. first-order hereditary Harrop formulae, is the most desirable for programming, but we often use a slightly smaller class, in which universal quantifiers are not allowed in goals, for technical convenience. This smaller class may be normalised, in that there is a smaller class of formulae which have the same expressive power, and so we need to study at least three languages other than Horn clauses. These are introduced below.

In all cases, only closed D formulae may be used in programs, and only closed G formulae may be used as goals.

Horn clauses may be defined as follows:

Definition 2.5.1 *D_{Horn} and G_{Horn} formulae have the following form*

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid G_1 \wedge G_2 \mid \exists x G$$

D_{Horn-} and G_{Horn-} formulae have the following form

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid G_1 \wedge G_2 \mid \exists x G$$

As discussed in [77] and section 2.1, this class of formulae is no more powerful than the usual definition of Horn clauses. As also noted in [77] and section 2.1, Horn clause goals may include disjuncts without increasing the power of the language, and hence Horn clauses may be equivalently defined as follows.

Definition 2.5.2 *$D_{Horn\vee}$ and $G_{Horn\vee}$ formulae have the following form*

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G$$

$D_{Horn\vee-}$ and $G_{Horn\vee-}$ formulae have the following form

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G$$

A more general (and more powerful) class of formulae is first-order hereditary Harrop formulae.

Definition 2.5.3 *D_{HHF} and G_{HHF} formulae have the following form*

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid \forall x G \mid D \supset G$$

D_{HHF-} and G_{HHF-} formulae have the following form

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid \forall x G \mid D \supset G$$

For technical reasons, we will often wish to omit universally quantified goals from consideration, and hence use the following class of formulae, which we will refer to as *module* formulae.

Definition 2.5.4 *D_{mod} and G_{mod} formulae have the following form*

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid D \supset G$$

D_{mod-} and G_{mod-} formulae have the following form

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid D \supset G$$

As discussed in section 6.1, we will have cause to consider the following languages as well.

Definition 2.5.5 *D_{object} and G_{object} formulae have the following form*

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid G_1 \wedge G_2 \mid D \supset G$$

$D_{object-}$ and $G_{object-}$ formulae have the following form

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid G_1 \wedge G_2 \mid D \supset G$$

Definition 2.5.6 *D_{meta} and G_{meta} formulae have the following form*

$$D := A \mid \neg A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A \mid G \supset \neg A$$
$$G := A \mid \neg A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid \forall x G \mid D \supset G$$

Note that we may equivalently define D_{meta} and G_{meta} formulae as

$$L := A \mid \neg A$$
$$D := L \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset L$$
$$G := L \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid \forall x G \mid D \supset G$$

It should be clear that the definition of D formulae does not change very much in from case to case, and so we may think of the various alternatives above as variations on a standard theme. Some possible variations are given below:

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset D$$
$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$D := A \mid \neg A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$D := A \mid \neg A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A \mid G \supset \neg A$$

We have seen how the first variation is no more powerful than the second, and hence we usually use the second. However, whenever we do wish to use the first variation, we refer to it as the *extended form*, and the second variation as the *clausal form*. For example, the extended form of D_{object} and G_{object} formulae are given below.

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset D$$
$$G := A \mid \neg A \mid G_1 \wedge G_2 \mid D \supset G$$

The third and fourth variations are used to construct explicit negative information as well as positive information, and hence may lead to an inconsistency, which means that they must be used with care. Thus the most common of these variations will be the second one, when negation is not considered, or the third one, when negation is considered.

In all of the above languages, it is interesting to consider which formulae are both programs and goals. Such formulae will be referred to as *core* formulae, and labelled as M formulae. For example, M_{Horn} consists of conjunctions of closed (or ground) atoms. Clearly hereditary Harrop formulae will have the most interesting core, which may be given as follows:

$$M := A \mid \forall x M \mid M_1 \wedge M_2 \mid M \supset A$$

The core of the module formulae is similar, except that no universal quantifiers may occur, and so M_{mod} is the set of all ground D_{obj} programs, which may be given as follows:

$$M := A \mid M_1 \wedge M_2 \mid M \supset A$$

Core formulae are of interest as they represent the formulae which may be both asserted (programs) and tested for truth (goals). Hence, a core formulae, once it has been proved to be true, may be added to the program, i.e. *memoised*. We will have more to say about memoisation in chapter 7.

Chapter 3

Completions and Negation as Failure

In this chapter we examine the relationship between the Closed World Assumption (CWA) and the Negation as Failure (NAF) rule. A common method of attack is to consider the completion of a program [17], which may be thought of as adding negative information to the program in such a manner that an atom fails iff its negation is derivable from the completion of the program. We give a form of the completion which is more explicit than that of Clark [17], as well as being adapted to first-order hereditary Harrop formulae. This requires that we place a restriction on the class of programs, so that the completion is never inconsistent. With this restriction, we show that the completion behaves as expected, in that the computational properties of the program and its completion correspond precisely.

3.1 Completions and Provability

As discussed above, NAF is an implicit form of negation, which may be seen from the definition of $P \vdash_s \neg A$. Now as intuitionistic negation identifies $\neg A$ and $A \supset \bot$, we may ask whether such an interpretation can be given in this case, i.e. is there a formula P' such that $P \vdash_s \neg A$ iff $P' \vdash_I A \supset \bot$? One way to achieve this is through the *completion* of the program, first proposed by Clark [17]. We may think of this as adding extra information to the program, so that for any atom A for which name$(A) \in N$, we have that $P \vdash_s \neg A$ iff $P', A \vdash_s \bot$, where P' is an extension of

P. Note that the classes of programs defined above (i.e. sets of closed D formulae) can never be inconsistent, and so there is no way that $P \cup \{A\}$ can prove $\perp$. However, we envisage the larger program P' as adding negative information to P, and hence enabling the assumption of A to lead to an inconsistency.

We may think of the larger program as a theory in which $\neg A$ holds iff $A \supset \perp$ holds, and so the two may be considered interchangeable. In our case we will often wish to think of $\neg A$ as a shorthand for $A \supset \perp$, and so whilst our formal results will use the latter version, we shall see how the former version is sometimes more intuitive. Hence, we will often use $\neg A$ rather than $A \supset \perp$ in discussion.

We may add such negative information to the program by allowing $\perp$ as a distinguished atom, as is done in [77]. In Clark's case, the extra information was also added implicitly, by making the "if"'s of the program into "if and only if"'s.

Clark's completion is given in the definition below.

Definition 3.1.1 *Let P be a Horn clause program. For each predicate p, of arity n, which is defined by k clauses*

$$p(t_{11}, t_{12}, \ldots t_{1n}) \subset B_1$$
$$\ldots$$
$$p(t_{k1}, t_{k2}, \ldots t_{kn}) \subset B_k$$

where B_i is a conjunction of atoms (or body), the completion $comp(P)$ of P is given by

$$\forall x_1 \ldots x_n \; p(x_1, \ldots x_n) \leftrightarrow \exists(E_1 \wedge B_1) \vee \ldots \vee \exists(E_k \wedge B_k)$$

where E_i is $x_1 = t_{i1} \wedge \ldots \wedge x_n = t_{in}$, $1 \leq i \leq k$, $\exists(E_i \wedge B_i)$ stands for the existential closure of $E_i \wedge B_i$ for all variables of $E_i \wedge B_i$ other than $x_1, \ldots, x_n$, and the variables $\{x_1, \ldots x_n\}$ are new variables, i.e. they do not appear anywhere in the k clauses above.

Note the importance of the equations, which may be thought of as encapsulating the unification process.

In addition, Clark's scheme involves adding the following formula for each predicate p which appears in the body of some clause in the program but not as the head of any clause:

$$\forall x_1, \ldots x_n \neg p(x_1, \ldots x_n)$$

This is just the case when a predicate has no clauses defining it in the program, and so for all terms $t_1 \ldots t_n$ we have that $p(t_1, \ldots t_n)$ is false.

For a Horn clause program P, we refer to the Clark completion as comp(P). Note that this completion is defined for Horn clause programs, so that no negation appears in the body of the clauses in the program. This means that comp(P) is only meaningful when negation is only allowed in queries, rather than in both the body of a clause and in a query. As remarked in section 2.4.5, this ensures that the completion is consistent, but there are less restrictive assumptions which also ensure the consistency of the completion. For this reason we will assume that programs are locally stratified, and it is probable that any restriction which leads to the consistency of the completion will suffice.

From our point of view, it will only make sense to define the completion for predicates which are completely defined, as the completion gives an explicit representation of information which is left implicit in the program. Thus in the completion of a program $P = \langle D, N \rangle$, we restrict our attention to predicates whose names appear in den(N).

One important thing to note is that the completion is really carried out as a meta-level process rather than as an object level transformation. The equations and the $\leftrightarrow$ ensure that comp(P) is not a Horn clause program, and so cannot be used directly for the computation of the negation of completely defined predicates. We will be able to use the completion directly for computation, rather than just as a semantic device, if we can write the completion in the form of an extension to the program, rather than as a meta-level formula. The particular form we have in mind is one that adds clauses whose heads are negative literals. For example, consider the even predicate defined below.

$$\begin{aligned} &\text{even}(0) \\ &\forall x\, \text{even}(x) \supset \text{even}(s^2(x)) \end{aligned}$$

The Clark completion comp(P) is

$$\forall x\, \text{even}(x) \leftrightarrow x = 0 \vee \exists y\; x = s^2(y) \wedge \text{even}(y)$$

From this we may infer the additional information that

$$\forall x\, \neg\text{even}(x) \leftrightarrow x \neq 0 \wedge \forall y\; x \neq s^2(y) \vee (x = s^2(y) \wedge \neg\text{even}(y))$$

Hence, a more explicit form of the completion may be given as

$$\begin{aligned} &\text{even}(0) \\ \forall x\quad &\text{even}(x) \supset \text{even}(s^2(x)) \\ &\neg\text{even}(s(0)) \\ \forall x\; &\neg\text{even}(x) \supset \neg\text{even}(s^2(x)) \end{aligned}$$

Note that if we replace ¬even by odd, this process has added the usual definition of the odd predicate to the usual definition of the even predicate, which is just what would be expected.

It is this latter form that we will define as our completion P^c, which is then an extension to the program. This seems a more natural way to view the added information.

As there are negations in the head of these formulae as written above, we will need to find some way of writing such formulae as definite formulae. One possibility is to allow negated atoms to be heads of clauses, and thus extend the class of programs. This extension raises some technical issues with regard to inconsistency, as discussed above. Another possibility is to allow $\perp$ as a distinguished atom, and compute $P^c \vdash_s \neg A$ via $P^c, A \vdash_s \perp$. Using this technique the last two clauses of the completion P^c of the even predicate defined above may be written as

$$\text{even}(s(0)) \supset \bot$$
$$\forall x\,((\text{even}(x) \supset \bot) \wedge \text{even}(s^2(x))) \supset \bot$$

Note that the formula below, which is equivalent to the second clause, is not a clause in the usual sense, as the "head" of the clause is not an atom.

$$\forall x\ (\text{even}(x) \supset \bot) \supset (\text{even}(s^2(x) \supset \bot)$$

The operational features of this approach were discussed in an earlier section. It is not hard to see that this approach to the completion will allow us to use the technique discussed in the previous chapter in which $\text{even}(s^2(x)) \supset \bot$ would be considered the head of the clause. The derivation of $\neg\text{even}(s^3(0))$ from this program is given below.

$$P^c \vdash_s \text{even}(s^3(0)) \supset \bot$$
$$\text{even}(s^3(0)), P^c \vdash_s \bot$$
$$\text{even}(s^3(0)), P^c \vdash_s \exists x\,(\text{even}(s^2(x)) \wedge (\text{even}(x) \supset \bot))$$
$$\text{even}(s^3(0)), P^c \vdash_s \text{even}(s(0)) \supset \bot$$
$$\text{even}(s(0)), \text{even}(s^3(0)), P^c \vdash_s \bot$$
$$\text{even}(s(0)), \text{even}(s^3(0)), P^c \vdash_s \text{even}(s(0))$$

This derivation hinges on the fact that we made three "right" choices along the way: firstly that the first time that we encounter $\bot$ we match $\bot$ against (the second version of) the fourth clause in P^c, next that we match $\text{even}(s^2(x))$ against the assumption $\text{even}(s^3(0))$ rather than the original clauses of the program, and finally that we use the third clause in P^c to match against the second occurrence of $\bot$. These seemingly arbitrary choices may be understood by considering that in order to derive a negated atom $\neg A$, we must use information added to the program by the completion. Hence we can only derive $\neg A$ if there is an instance of a clause in the completion corresponding to this formula. In this way we may think of the clause

$$\forall x((\text{even}(x) \supset \bot) \wedge \text{even}(s^2(x))) \supset \bot$$

as equivalent to the "clause"

$$\forall x\ (\text{even}(x) \supset \bot) \supset (\text{even}(s^2(x)) \supset \bot)$$

so that we can think of $\text{even}(s^2(x)) \supset \bot$, or $\neg\text{even}(s^2(x))$ as the "head" of the clause. Thus in the above derivation, the goal $\text{even}(s^3(0)) \supset \bot$ matches the fourth clause in P^c but not the third, and so there is only one possible choice, rather than two. The goal $\text{even}(s(0)) \supset \bot$ works in a similar manner, this time matching the third clause but not the fourth.

The choice of which clause to match $\text{even}(s^2(x))$ with may also be explained by this concept of the head of a clause. Consider the program given below.

$$D \wedge (G \supset (A \supset \bot))$$

The clause $G \supset (A \supset \bot)$ represents a part of the completion of some smaller program, in particular a clause dealing with the definition of A. Now for a goal $B \supset \bot$ where $B\theta = A\theta$, we have the following derivation sequence

$$\begin{aligned}
&P, G \supset (A \supset \bot) \vdash_s B \supset \bot \\
&B, P, G \supset (A \supset \bot) \vdash_s \bot \\
&B, P, G \supset (A \supset \bot) \vdash_s A \wedge G \\
&B, P, G \supset (A \supset \bot) \vdash_s G\theta
\end{aligned}$$

The last step in the above derivation may be derived directly from the first step by matching $B \supset \bot$ with $A \supset \bot$, provided that B is not needed in the computation. We will show later that this is indeed the case for a large class of programs. Now if B and A do not unify, then the derivation above would be different, as we need to find some other way for A to succeed. However, we know that the completion has the property that if A and B do not unify, then this clause cannot tell us anything about the negation of B, and so some other clause will be needed for $B \supset \bot$ to succeed. In other words, either B and A are unifiable or this clause cannot tell us anything about the negation of B. Hence, the technique of matching against the extended clause head is a correct short-cut, in that no

correct derivations of $B \supset \bot$ are missed out. A formal proof of this property is given in section 3.3.

This enables us to write the above completion of the even predicate in the original form, i.e.

$$\begin{array}{rl} & \text{even}(0) \\ \forall x & \text{even}(x) \supset \text{even}(s^2(x)) \\ & \neg\text{even}(s(0)) \\ \forall x & \neg\text{even}(x) \supset \neg\text{even}(s^2(x)) \end{array}$$

with the above convention about the equivalence of $\neg A$ and $A \supset \bot$ and the concept of the extended clause head understood. This may be thought of as a compromise between the logician's idea of negation, i.e. $A \supset \bot$ and the programmer's idea of negation, i.e. A fails.

Note that this technique may be thought of as ensuring that in deriving $A \supset \bot$, the assumption of A must be used in the computation. This is justified by the perception that we define the completion in such a way that we know it is consistent, and so $P^c \not\vdash_s \bot$, and so for $P^c, A \vdash_s \bot$ to hold, we must have that the assumption is used in the derivation. It is this property that allows us to use the more specialised computation rule; we are not trying to find whether the program is in any way inconsistent, which is the natural interpretation of $P \vdash_s \bot$; we are merely checking whether a given extension to a consistent program makes it inconsistent. Thus we can consider this case as a specialised form of a consistency check.

3.2 The Completion Process

It is interesting to note that although the CWA is defined on a program, Clark's completion is defined for a predicate, and the completion of the program is given by the conjunction of the completions of each predicate. In this section we give a similar completion procedure for a first-order hereditary Harrop formula program, and our completion will have the added advantage of being defined in an executable language, and so it directly specifies a computational method for NAF. This is done by adding the extra information as clauses, rather than converting clauses into stronger statements, as is done in [17].

The basic idea is the observation that if $G \supset A$ is the only clause whose head matches A and name(A) is a completely defined predicate, then we know that if G fails, then A fails, and as name(A) is a completely defined predicate, we have $\neg A$. Hence, we identify the failure of A with the negation of A, and so the negation of A with the failure of G. Recall that a completely defined predicate can only depend on completely defined predicates, and so this identifies the negation of A with the negation of G. The main problem is then to express the negation of G in our form of clauses.

For example, consider the program below, where p, q and r are completely defined, and s and t are not.

$$\forall x\ (\exists y \neg q(y) \wedge r(y)) \supset p(x)$$
$$\forall x\ s(x) \supset t(x)$$

We wish the completion of this program to be

$$\forall x\ (\exists y \neg q(y) \wedge r(y)) \supset p(x)$$
$$\forall x\ s(x) \supset t(x)$$
$$\forall x\ (\forall y q(y) \vee \neg r(y)) \supset \neg p(x)$$
$$\forall x\ \neg q(x)$$
$$\forall x\ \neg r(x)$$

Note that the third clause in the completion requires the use of a universal quantifier in the body, and that the completion is a set of D_{HHF} formulae.

We may write the definition of a completely defined predicate p in the following manner:

Let the clauses of p be the universal closure of

$$p(t_{11}, t_{12}, \ldots t_{1n}) \subset G_1$$

$$\ldots$$

$$p(t_{k1}, t_{k2}, \ldots t_{kn}) \subset G_k$$

where G_i is a goal. We use the same idea as above for the encapsulation of unification by expressing the clauses in the equivalent form

$$\forall x_1 \ldots x_n \; p(x_1, \ldots x_n) \subset \exists(E_1 \wedge G_1) \vee \ldots \vee \exists(E_k \wedge G_k)$$

where E_i is $x_1 = t_{i1} \wedge \ldots \wedge x_n = t_{in}$, $\exists(E_i \wedge G_i)$ is the existential closure of all free variables of $E_i \wedge G_i$ other than $\{x_1, \ldots x_n\}$, $1 \leq i \leq k$, and the x_j are new variables, and so they do not appear in any G_i.

Now as p is a completely defined predicate, we may then conclude that

$$\forall x_1 \ldots x_n \; \neg p(x_1, \ldots x_n) \subset \neg[\exists(E_1 \wedge G_1) \vee \ldots \vee \exists(E_k \wedge G_k)]$$

We then proceed to transform the body into a G formula, so that the above formula becomes a clause defining $\neg p$. The above observation that a completely defined predicate can only depend on completely defined predicates will be useful in the following transformation process. We wish to push the $\neg$ inwards, and when this connective is only applied to atoms, we may consider the clauses so given as executable, giving us the desired explicit form of the completion. In classical logic, the justification of this process is immediate due to the larger number of equivalences between formulae than in intuitionistic logic, but in our case we need to do more work to justify it. In order to do this, we introduce below the operator *fails*, which takes a goal and returns another goal such that fails(G) succeeds if the original goal G fails. This is akin to the transformation process of Barbuti et al. [7], and may be thought of as a way of expressing $\neg G$ as a goal.

Definition 3.2.1 *Let G be a G_{HHF-} goal formula. We say G is* negatable *iff all predicates which occur positively in G are completely defined, and all predicates which occur negatively in G are incompletely defined.*

It should be clear that if $G \supset A \in (D)$ and name(A) is completely defined, then G is a negatable goal formula. This is due to the way that completely defined predicates may depend on other predicates, and so all the bodies of the clauses of the program in which we will be interested will be negatable goal formulae.

The reason that negatable goals are interesting is that we may identify the failure of a negatable goal with the truth of its negation. This may be thought of as an extension of the way that we infer $\neg A$ from the failure of A when name(A) is a completely defined predicate. The reason that all the negatively occurring predicates must be incompletely defined is that we must be able to add the necessary assumptions to the program in order to show that the goal fails, and hence that its negation succeeds. The operator fails may now be defined as follows:

Definition 3.2.2 *Let G be a negatable goal formula.*

We define another G_{HHF-} goal formula fails(G) *as follows:*

$$\begin{aligned}
\text{fails}(A) &= \neg A \\
\text{fails}(\neg A) &= A \\
\text{fails}(G_1 \vee G_2) &= \text{fails}(G_1) \wedge \text{fails}(G_2) \\
\text{fails}(G_1 \wedge G_2) &= \text{fails}(G_1) \vee \text{fails}(G_2) \\
\text{fails}(D \supset G) &= D \supset \text{fails}(G) \\
\text{fails}(\exists x G) &= \forall x\, \text{fails}(G) \\
\text{fails}(\forall x G) &= \exists x\, \text{fails}(G)
\end{aligned}$$

It is obvious that for any negatable G_{HHF-} goal G, fails(G) is also a G_{HHF-} goal. As we are dealing only with completely defined predicates, we know that the failure of A and the success of $\neg A$ are equivalent, as are the success of A and the failure of $\neg A$. For the other goal formulae, the definition of fails(G) is very similar

to that of $\vdash_f$; we know that if $G_1 \vee G_2$ fails then we must have that G_1 fails and G_2 fails, and vice-versa, and so on.

We may think of this transformation as exploiting the properties of failure and success. For example, the goal $p \wedge q$ fails iff either p fails or q fails, and under Negation as Failure, this is the same as either $\neg p$ succeeds or $\neg q$ succeeds, i.e. $\neg p \vee \neg q$ succeeds.

Note also that $\text{fails}(\text{fails}(G)) = G$. Below we show that the expected behaviour of $\text{fails}(G)$ indeed occurs.

Lemma 3.2.1 *Let $\langle P, G\rangle$ be a derivation pair where $P = \langle D, \dot{N}\rangle$ and G is negatable. Then*

$$P \vdash_s G \Leftrightarrow P \vdash_f \text{fails}(G)$$

Proof: We proceed by induction on the depth of the relevant **O**-derivation. The base case occurs when G is a literal, i.e. either A or $\neg A$, where A is an atom.

By Proposition 2.3.1, we have that

$$P \vdash_s A \Leftrightarrow P \vdash_f \neg A$$
$$P \vdash_s \neg A \Leftrightarrow P \vdash_f A$$

and so it is clear that the base case holds.

Hence the induction hypothesis is that the statement holds for all **O**-derivations of no more than a given depth. There are seven cases:

A, $\neg A$: As above, it is clear that these two cases follow immediately from Proposition 2.3.1.

$G_1 \vee G_2$: $P \vdash_s G_1 \vee G_2$ iff $P \vdash_s G_1$ or $P \vdash_s G_2$ and by the hypothesis this is equivalent to $P \vdash_f \text{fails}(G_1)$ or $P \vdash_f \text{fails}(G_2)$, which in turn is equivalent to $P \vdash_f \text{fails}(G_1) \wedge \text{fails}(G_2)$, i.e. $P \vdash_f \text{fails}(G_1 \vee G_2)$.

$G_1 \wedge G_2$: $P \vdash_s G_1 \wedge G_2$ iff $P \vdash_s G_1$ and $P \vdash_s G_2$ and by the hypothesis this is equivalent to $P \vdash_f \text{fails}(G_1)$ and $P \vdash_f \text{fails}(G_2)$, which in turn is equivalent to $P \vdash_f \text{fails}(G_1) \vee \text{fails}(G_2)$, i.e. $P \vdash_f \text{fails}(G_1 \wedge G_2)$.

$\exists x G$: $P \vdash_s \exists x G$ iff $P \vdash_s G[t/x]$ for some $t \in \mathcal{U}$ and by the hypothesis this is equivalent to $P \vdash_f \text{fails}(G[t/x])$ for some $t \in \mathcal{U}$, which in turn is equivalent to $P \vdash_f \forall x\, \text{fails}(G)$, i.e. $P \vdash_f \text{fails}(\exists x G)$.

$\forall x G$: $P \vdash_s \forall x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_s G[t/x]$ for all $t \in R$ and by the hypothesis this is equivalent to $P \vdash_f \text{fails}(G[t/x])$ for all $t \in R$, which in turn is equivalent to $P \vdash_f \exists x\, \text{fails}(G)$, i.e. $P \vdash_f \text{fails}(\forall x G)$.

$D' \supset G$: $\langle D, N\rangle \vdash_s D' \supset G$ iff $\text{names}(\text{heads}(D')) \cap \text{den}(N) = \emptyset$ and $\langle D \cup \{D'\}, N\rangle \vdash_s G$ and by the hypothesis this is equivalent to $\text{names}(\text{heads}(D')) \cap \text{den}(N) = \emptyset$ and $\langle D \cup \{D'\}, N\rangle \vdash_f \text{fails}(G)$, which in turn is equivalent to $\langle D, N\rangle \vdash_f D' \supset \text{fails}(G)$, i.e. $\langle D, N\rangle \vdash_f \text{fails}(D' \supset G)$.

□

The dual result to the above is an immediate corollary, which gives us the proposition below.

Proposition 3.2.2 *Let $\langle P, G\rangle$ be a derivation pair where G is negatable. Then*

1. $P \vdash_s G \Leftrightarrow P \vdash_f \text{fails}(G)$

2. $P \vdash_f G \Leftrightarrow P \vdash_s \text{fails}(G)$

Proof:

1. This was proved in lemma 3.2.1 above.

2. Let $G' = \text{fails}(G)$, and so $\text{fails}(G') = G$. By lemma 3.2.1, $P \vdash_s G' \Leftrightarrow P \vdash_f \text{fails}(G')$, which is just $P \vdash_s \text{fails}(G) \Leftrightarrow P \vdash_f G$.

□

As mentioned in section 2.3, we may interpret this result as defining a notion of NAF for a larger class of formulae than just atoms. If $P \vdash_s \neg G$ iff $P \vdash_f G$, then from the above proposition we have that $P \vdash_s \neg G$ iff $P \vdash_s \text{fails}(G)$, and so we can think of fails(G) as a way of writing $\neg G$ as a goal. In this way once we have implemented $\neg A$ as a goal, we may derive a more widespread notion of NAF, i.e. one that is applicable to any negatable goal, not just to atoms.

We need another transformation before we can define our completion, so that each occurrence of $\neg A$ is replaced by $A \supset \bot$. This is done by the transformation below.

Definition 3.2.3 *Let D be a D_{HHF-} definite formula and G be a G_{HHF-} goal formula.*

We define contr(G) *and* contrd(D) *as follows:*

$$\begin{aligned}
\text{contr}(A) &= A \\
\text{contr}(\neg A) &= A \supset \bot \\
\text{contr}(G_1 \vee G_2) &= \text{contr}(G_1) \vee \text{contr}(G_2) \\
\text{contr}(G_1 \wedge G_2) &= \text{contr}(G_1) \wedge \text{contr}(G_2) \\
\text{contr}(\exists x G) &= \exists x\, \text{contr}(G) \\
\text{contr}(\forall x G) &= \forall x\, \text{contr}(G) \\
\text{contr}(D \supset G) &= \text{contrd}(D) \supset \text{contr}(G)
\end{aligned}$$

$$\begin{aligned}
\text{contrd}(A) &= A \\
\text{contrd}(D_1 \wedge D_2) &= \text{contr}(D_1) \wedge \text{contr}(D_2) \\
\text{contrd}(\forall x D) &= \forall x\, \text{contr}(D) \\
\text{contrd}(G \supset A) &= \text{contr}(G) \supset A
\end{aligned}$$

Note that neither contrd(D) nor contr(G) can contain any occurrence of $\neg A$.

We use these two constructs to construct the completion as follows: if A is an instance $p(s_1, \ldots s_n)$ of $p(x_1, \ldots x_n)$, then from the program we have that

$$A \subset \bigvee_{i=1}^{k} \exists (E_i \wedge G_i)$$

where E_i is $s_1 = t_{i1} \wedge \ldots \wedge s_n = t_{in}$.

Now if name(A) is a completely defined predicate, we may then deduce that $\neg A$ holds if the body of the clause fails, i.e.

$$\neg A \subset \neg \bigvee_{i=1}^{k} \exists (E_i \wedge G_i)$$

Now as all predicates in all the G_i must be completely defined, we get

$$\neg A \subset \bigwedge_{i=1}^{k} \forall (\neg E_i \vee \text{fails}(G_i))$$

It is obvious that $(E \wedge G)$ fails iff E fails or G fails, and so as E is just a set of equations and we may think of equality over terms as being completely defined, we get that $\neg(E \wedge G) \equiv \neg E \vee \neg G$. A similar argument may by given by the intuitionistic equivalence of $E \wedge G$ and $E \wedge (E \supset G)$, and so $\neg(E \wedge G)$ is the same as $\neg(E \wedge (E \supset G))$, and so we get $\neg E \vee \neg(E \supset G)$ which in turn is just $\neg E \vee (E \supset \neg G)$. This equivalence will hold even if G contains an incompletely defined predicate, as the equations may be thought of as expressing "$p(x_1, \ldots x_n)$ unifies with $p(t_{i1}, \ldots t_{in})$", and as first-order unification is decidable, we know that the E_i obey the law of excluded middle, so that either E_i is true or E_i is false, and we can easily determine which via unification.

As noted above, this process will not work for a program such as

$$\neg p \supset p$$

as we first derive

$$\neg p \supset p$$
$$p \supset \neg p$$

which is then transformed to

$$(p \supset \bot) \supset p$$
$$(p \wedge p) \supset \bot$$

Thus from the completion we get that $p \supset \bot$, which should not happen as in the original program both p and $\neg p$ loop. This is the reason that we need to restrict our attention to locally stratified programs. This is not necessarily the only restriction that will work, but it is a convenient one.

In our case, we need to consider the possibility that the program will grow during execution, in that goals may contain implications. Hence, we need not only that the original program is locally stratified, but also that all extensions of the program which occur during execution are also locally stratified. This leads us to the definitions below.

Definition 3.2.4 *Let $P = \langle D, N \rangle$ be a derivation state.*

We define the P-reliant relation on literals as follows:

1. *A is P-reliant on L iff $\exists G \supset A \in (D)$ such that L appears in G*
2. *$\neg A$ is P-reliant on L if A is P-reliant on L*

We define the P-dependent relation on literals as follows:

1. *A is P-dependent on L iff A is P-reliant on L or there is an L' such that A is P-reliant on L' and L' is P-dependent on L*
2. *$\neg A$ is P-dependent on L if A is P-dependent on L*

We say A is P-self-dependent *if A is P-dependent on A.*

It is easily seen that the P-dependent relation is the transitive closure of the P-reliant relation. Thus in the program below we have that p is P-dependent on p but not on $\neg p$, and that q is P-dependent on $\neg r$ and s.

$$p \supset p$$
$$\neg r \supset q$$
$$s \supset r$$

These relations are useful in order to formally define the property of local stratification, which is done below.

Definition 3.2.5 *Let $\langle P, G\rangle$ be a derivation pair.*

P is locally stratified *iff there is no atom A such that A is P-dependent on $\neg A$.*

P and G are a locally stratified derivation pair *if every derivation state which occurs in the computation of G is locally stratified.*

Thus the program given above is locally stratified, whereas the program consisting of the clause $\neg p \supset p$ is clearly not locally stratified.

Note that if $P = \langle D, N\rangle$ is locally stratified, then so is $\langle D \cup \{A\}, N\rangle$, so that the addition of an atom to a locally stratified program results in a locally stratified program, i.e. the assumption of an atom preserves local stratification.

We need the notion of a locally stratified derivation pair P and G rather than just a locally stratified program because the program may increase during execution, and so we need to ensure that all programs which occur during execution preserve local stratification.

We are now in a position to give a formal definition of our completion.

Definition 3.2.6 *Let $P = \langle D, N\rangle$ be a locally stratified derivation state. For each predicate letter p appearing in P, we define p^+ and p^- as follows:*

Let the clauses defining p be the universal closure of

$$p(t_{11}, t_{12}, \ldots t_{1n}) \subset G_1$$
$$\ldots$$
$$p(t_{k1}, t_{k2}, \ldots t_{kn}) \subset G_k$$

We denote by p^+ the clause

$$\forall x_1 \ldots x_n \, p(x_1, \ldots x_n) \subset \bigvee_{i=1}^{k} \exists (E_i \wedge \mathrm{contr}(G_i))$$

where E_i is $x_1 = t_{i1} \wedge \ldots \wedge x_n = t_{in}$, $\exists(E_i \wedge G_i)$ is the existential closure of all free variables of $E_i \wedge G_i$, $1 \leq i \leq k$, and $x_1, \ldots x_n$ are new variables which do not appear in any G_i.

If $p \in \mathrm{den}(N)$, then p^- is the clause

$$\forall x_1 \ldots x_n \ \perp \subset p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

If $p \notin \mathrm{den}(N)$, then p^- is empty.

If there is no clause in P whose head's name is p, then p^+ is empty. If $p \notin \mathrm{den}(N)$, then p^- is also empty. Otherwise, p^- is the clause

$$\forall x_1, \ldots x_n \ \perp \subset p(x_1, \ldots x_n)$$

The completion of a predicate p is $\{p^+, p^-\}$.

The completion P^c of P is $\langle D^c, N \rangle$, where D^c is the union of the completions of each predicate appearing in P.

We will write often D^c as $D^+ \cup D^-$ to emphasise the nature of our definition of the completion, i.e. the original clauses of the program (in a slightly modified form) together with a set of extra clauses containing the negative information.

Note also that P^c contains the clause

$$\forall x_1, \ldots x_n \perp \subset p(x_1, \ldots x_n)$$

for all completely defined predicates which occur in the body of some clause in the program, but for which there is no clause head whose name is p. This is the same as in Clark's case.

The only thing that prevents us from using the above two clauses directly to compute negated atoms (as distinct from the usually semantic use of $\mathrm{comp}(P)$) is the inequations. We show how we may incorporate these by giving an algorithm to solve such inequations as explicitly as possible (i.e. producing an explicit answer whenever possible) in the next chapter.

It should be noted that solving the equations and inequations may be done independently of the idealised interpreter, as the equations and inequations are relations between terms rather than predicates. Hence, we may call the equation/inequation solver at any point in a derivation, and so we shall write $P \vdash_o E$ to denote that the set E of equations and inequations has a solution. This will come in handy when mixing equations, inequations and goals.

Apart from the inequations, the clauses above are definite formulae according to our definition, and so we may think of this form of the completion as some sort of meta-program for the computation of literals. The reason that we consider it a meta-level program is that we consider the programmer as writing the "positive" clauses for the completely defined predicates, and the "negative" clauses as being implicitly understood due to the fact that p is completely defined. Thus this executable form of the completion makes explicit the complement of the definitions written by the programmer.

In the next section we show the expected results about the completion, i.e. that the completion of a locally stratified program behaves as expected, and precisely captures the NAF rule for the program.

3.3 Properties of the Completion

An interesting property to note is that if P is locally stratified and $G \supset A \in (D)$, then we know that A is not P-dependent on $\neg A$, which means that no instance of any atom which appears in G is P-dependent on $\neg A$. Hence, if P' is the program $\{\text{fails}(G) \supset A \mid G \supset A \in \text{clausal}(P)\}$, then no instance of any atom which appears in fails(G) is P'–dependent on A, and so if the clause

$$\bot \subset A \wedge \text{contr}(\text{fails}(G))$$

appears in P^c, then

$$\langle D^c \cup \{A\}, N\rangle \vdash_s \text{contr}(\text{fails}((G)) \text{ iff}$$
$$\langle D^c, N\rangle \vdash_s \text{contr}(\text{fails}(G))$$

This is the property mentioned in section 2.4.4, i.e. that for a large class of programs, when searching for a proof of $A \supset \bot$, we only need A to determine which clause to use, and not as an assumption. For example, consider the completion (below) of the program $p \supset q$, where p and q are completely defined:

$$p \supset q$$
$$((p \supset \bot) \wedge q) \supset \bot$$
$$p \supset \bot$$

It is clear that to find a proof of $q \supset \bot$ it is sufficient to determine whether $p \supset \bot$ from the completion of the program, rather than from the completion extended by the assumption of q.

This property is formalised in the lemma below. We write the substitution $[x_1 \leftarrow s_1, \ldots x_n \leftarrow s_n]$ as $[x_i \leftarrow s_i]_{i=1}^n$.

Lemma 3.3.1 *Let $P = \langle D, N \rangle$ be a locally stratified derivation state, and $p(s_1, \ldots s_n)$ be an atom such that there is a clause*

$$\forall x_1, \ldots x_n \;\; \bot \subset p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

in D^c. Then

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N \rangle \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^n)$$

iff

$$\langle D^c, N \rangle \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^n)$$

Proof: The $\Leftarrow$ direction is clear.

For the other direction, consider an **O**-proof of $P \longrightarrow A$ for some atom A. It is clear that A is P-dependent on any other atom which appears as a consequent in this proof. Hence, if A is not P-dependent on A', then A' does not appear as a consequent in any **O**-proof of $P \longrightarrow A$.

Assume that $\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^n)$. Now as P is locally stratified, we know that $\text{fails}(G_i)[x_i \leftarrow s_i]_{i=1}^n$ is not $\text{fails}(P)$-dependent on $p(s_1, \ldots s_n)$, where $\text{fails}(P) = \{\text{fails}(C) \mid C \in P\}$. Hence, $p(s_1, \ldots s_n)$ does not appear as a consequent in any uniform proof of $D^c \longrightarrow \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^n$, and so we may omit $p(s_1, \ldots s_n)$ from the antecedent of every sequent in the proof, and so $\langle D^c, N\rangle \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^n)$.

□

Next we show a useful lemma about P^c.

Lemma 3.3.2 *Let $\langle P, G\rangle$ be a locally stratified derivation pair where $P = \langle D, N\rangle$.*

Then it is not the case that

$$P^c \vdash_s \text{contr}(G) \wedge \text{contr}(\text{fails}(G)).$$

Note that this is a weaker statement than

$$P^c \vdash_f \text{contr}(G) \wedge \text{contr}(\text{fails}(G)).$$

This stronger statement is not true, as it requires that either $\text{contr}(G)$ or $\text{contr}(\text{fails}(G))$ fails for any G and any program P, and thus for no loops to occur. However, the weaker statement above is sufficient for our purposes, as whilst it does not guarantee that $\text{contr}(G) \wedge \text{contr}(\text{fails}(G))$ always fails, it does guarantee that it never succeeds.

Proof: As $D' \supset G$ is a goal formula for any definite formula D' and goal formula G, the above is equivalent to showing that is it impossible that

$$\langle D^c \cup D'^c, N\rangle \vdash_s \text{contr}(G) \wedge \text{contr}(\text{fails}(G)).$$

for any definite formula D' and any goal formula G. This is due to the fact that if $\langle D, N\rangle \vdash_s D' \supset G$, then we must have that D' does not extend any completely defined predicates of D, and so $D'^c = D'^+$.

We proceed by induction on the depth of the **O**-derivation .

The base case occurs when the purported proof has the smallest depth, in which case G must be an atom A, and the problem reduces to showing that it is impossible that

$$\langle D^c \cup D'^c, N\rangle \vdash_s A \wedge (A \supset \bot)$$

This is true iff both the following are true:

$$\langle D^c \cup D'^c, N\rangle \vdash_s A$$
$$\langle D^c \cup D'^c, N\rangle \vdash_s A \supset \bot$$

Let A be $p(s_1, \ldots s_n)$. As the overall **O**-proof is of the smallest possible height, for the first case to be true we must have that $A \in (D^+ \cup D'^+)$, so there is a clause

$$\forall x_1 \ldots x_n\, p(x_1, \ldots x_n) \subset \bigvee_{i=1}^{k} \exists (E_i \wedge \mathrm{contr}(G_i))$$

in $D^+ \cup D'^+$ such that $E_j[x_i \leftarrow s_i]_{i=1}^n$ is true and G_j is empty for some $1 \leq j \leq k$.

For the second case, we must have that

$$\langle D^c \cup D'^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \bot$$

and so we must have

$$\langle D^c \cup D'^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s$$
$$\exists x_1, \ldots x_m\, q(x_1, \ldots x_m) \wedge \bigwedge_{i=1}^{k} \forall (\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

for some predicate q.

Now as $\langle D^c \cup D'^c, N\rangle \vdash_s p(s_1, \ldots s_n)$, this is equivalent to

$$\langle D^c \cup D'^c, N\rangle \vdash_s \exists x_1, \ldots x_m\, q(x_1, \ldots x_m) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))$$

In this case the **O**-proof will be shortest if the answer for the first conjunct is $q(t_1, \ldots t_m)$ where $q(t_1, \ldots t_m) \in (D^+ \cup D'^+)$. This means that $E_j[x_i \leftarrow t_i]_{i=1}^m$ is true for some j and that G_j is empty. But as

$$\langle D^c \cup D'^c, N\rangle \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))$$

we have that both $E_j[x_i \leftarrow t_i]_{i=1}^n$ and $\neg E_j[x_i \leftarrow t_i]_{i=1}^n$, which is a contradiction.

Hence, the base case holds.

So the inductive hypothesis is that there is no **O**-proof of height less than n such that

$$\langle D^c \cup D'^c, N\rangle \vdash_s \text{contr}(G) \wedge \text{contr}(\text{fails}(G)).$$

The cases for $G = A$ and $G = \neg A$ coincide, and so there are six cases to consider:

A: Let A be $p(s_1, \ldots s_n)$. For there to be an **O**-proof in this case, we must have that both the following hold:

$$\langle D^c \cup D'^c, N\rangle \vdash_s A$$
$$\langle D^c \cup D'^c, N\rangle \vdash_s A \supset \bot$$

In the first case, we must have that $\exists G \supset A \in (D^+ \cup D'^+)$ such that

$$\langle D^c \cup D'^c, N\rangle \vdash_s G$$

In the second case, we must have that

$$\langle D^c \cup D'^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \bot$$

and so we must have

$$\langle D^c \cup D'^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s$$
$$\exists x_1, \ldots x_m\, q(x_1, \ldots x_m) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))$$

for some predicate q.

As above, as $\langle D^c \cup D'^c, N\rangle \vdash_s A$, this is equivalent to

$$\langle D^c \cup D'^c, N\rangle \vdash_s \exists x_1, \ldots x_m\, q(x_1, \ldots x_m) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

Let the answer for the first conjunct be $q(t_1, \ldots t_m)$. As above, if $q(t_1, \ldots t_m) \in (D^+ \cup D'^+)$ then the second conjunct cannot have an **O**-proof. Otherwise, we have that $\exists G \supset q(t_1, \ldots t_m) \in (D^+ \cup D'^+)$ such that

$$\langle D^c \cup D'^c, N\rangle \vdash_s G$$

Now as $q(t_1, \ldots t_m)$ matches a clause in the program, we must have that $E_j[x_i \leftarrow t_i]_{i=1}^m$ is true for some $1 \leq j \leq k$ and that

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(G_j)$$

But we must also have that the second conjunct succeeds, and so

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(\mathrm{fails}(G_j))$$

which contradicts the induction hypothesis.

Hence, there is no **O**-proof of length n such that

$$\langle D^c \cup D'^c, N\rangle \vdash_s A \wedge (A \supset \bot)$$

$G_1 \vee G_2$: Note that $\mathrm{contr}(G_1 \vee G_2) = \mathrm{contr}(G_1) \vee \mathrm{contr}(G_2)$, and that

$$\begin{aligned} &\mathrm{contr}(\mathrm{fails}(G_1 \vee G_2)) \\ &= \mathrm{contr}(\mathrm{fails}(G_1) \wedge \mathrm{fails}(G_2)) \\ &= \mathrm{contr}(\mathrm{fails}(G_1)) \wedge \mathrm{contr}(\mathrm{fails}(G_2)) \end{aligned}$$

and so

$$\begin{aligned} &\mathrm{contr}(G_1 \vee G_2) \wedge \mathrm{contr}(\mathrm{fails}(G_1 \vee G_2)) \\ \equiv_I\ &(\mathrm{contr}(G_1) \wedge \mathrm{contr}(\mathrm{fails}(G_1)) \wedge \mathrm{contr}(\mathrm{fails}(G_2))) \vee \\ &\quad (\mathrm{contr}(G_2) \wedge \mathrm{contr}(\mathrm{fails}(G_1)) \wedge \mathrm{contr}(\mathrm{fails}(G_2))) \end{aligned}$$

Hence,

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(G_1 \vee G_2) \wedge \mathrm{contr}(\mathrm{fails}(G_1 \vee G_2))$$

implies that

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(G_i) \wedge \mathrm{contr}(\mathrm{fails}(G_i))$$

for some $i = 1, 2$, which contradicts the induction hypothesis.

$G_1 \wedge G_2$: Note that $\mathrm{contr}(G_1 \wedge G_2) = \mathrm{contr}(G_1) \wedge \mathrm{contr}(G_2)$, and that

$$\begin{aligned} &\mathrm{contr}(\mathrm{fails}(G_1 \wedge G_2)) \\ &= \mathrm{contr}(\mathrm{fails}(G_1) \vee \mathrm{fails}(G_2)) \\ &= \mathrm{contr}(\mathrm{fails}(G_1)) \vee \mathrm{contr}(\mathrm{fails}(G_2)) \end{aligned}$$

and so

$$\begin{aligned} &\mathrm{contr}(G_1 \wedge G_2) \wedge \mathrm{contr}(\mathrm{fails}(G_1 \wedge G_2)) \\ \equiv_I\ &(\mathrm{contr}(G_1) \wedge \mathrm{contr}(G_2) \wedge \mathrm{contr}(\mathrm{fails}(G_1)) \vee \\ &(\mathrm{contr}(G_1) \wedge \mathrm{contr}(G_2) \wedge \mathrm{contr}(\mathrm{fails}(G_2)) \end{aligned}$$

Hence,

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(G_1 \wedge G_2) \wedge \mathrm{contr}(\mathrm{fails}(G_1 \wedge G_2))$$

implies that

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(G_i) \wedge \mathrm{contr}(\mathrm{fails}(G_i))$$

for some $i = 1, 2$, which contradicts the induction hypothesis.

$\exists x G$: Note that

$$\begin{aligned} &\mathrm{contr}(\exists x G) \wedge \mathrm{contr}(\mathrm{fails}(\exists x G)) \\ &= \exists x\, \mathrm{contr}(G) \wedge \mathrm{contr}(\forall x\, \mathrm{fails}(G)) \\ &= \exists x\, \mathrm{contr}(G) \wedge \forall x\, \mathrm{contr}(\mathrm{fails}(G)) \end{aligned}$$

Hence,

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(\exists x G) \wedge \mathrm{contr}(\mathrm{fails}(\exists x G))$$

iff

$$\langle D^c \cup D'^c, N\rangle \vdash_s \exists x\, \mathrm{contr}(G) \wedge \forall x\, \mathrm{contr}(\mathrm{fails}(G))$$

which implies that

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(G[t/x]) \wedge \mathrm{contr}(\mathrm{fails}(G[t/x]))$$

for some $t \in \mathcal{U}$, which contradicts the induction hypothesis.

$\forall x G$: Note that

$$\begin{aligned} &\mathrm{contr}(\forall x G) \wedge \mathrm{contr}(\mathrm{fails}(\forall x G)) \\ =\ &\forall x\, \mathrm{contr}(G) \wedge \mathrm{contr}(\exists x\, \mathrm{fails}(G)) \\ =\ &\forall x\, \mathrm{contr}(G) \wedge \exists x\, \mathrm{contr}(\mathrm{fails}(G)) \end{aligned}$$

Hence,

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(\forall x G) \wedge \mathrm{contr}(\mathrm{fails}(\forall x G))$$

iff

$$\langle D^c \cup D'^c, N\rangle \vdash_s \forall x\, \mathrm{contr}(G) \wedge \exists x\, \mathrm{contr}(\mathrm{fails}(G))$$

which implies that

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(G[t/x]) \wedge \mathrm{contr}(\mathrm{fails}(G[t/x]))$$

for some $t \in \mathcal{U}$, which contradicts the induction hypothesis.

$D'' \supset G$: Note that

$$\begin{aligned} &\mathrm{contr}(D'' \supset G) \wedge \mathrm{contr}(\mathrm{fails}(D'' \supset G)) \\ =\ &(\mathrm{contrd}(D'') \supset \mathrm{contr}(G)) \wedge (\mathrm{contrd}(D'') \supset \mathrm{contr}(\mathrm{fails}(G))) \\ \equiv_I\ &\mathrm{contrd}(D'') \supset (\mathrm{contr}(G) \wedge \mathrm{contr}(\mathrm{fails}(G))) \end{aligned}$$

Hence,

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contr}(D'' \supset G) \wedge \mathrm{contr}(\mathrm{fails}(D'' \supset G))$$

iff

$$\langle D^c \cup D'^c, N\rangle \vdash_s \mathrm{contrd}(D'') \supset (\mathrm{contr}(G) \wedge \mathrm{contr}(\mathrm{fails}(G)))$$

which implies that

$$\langle D^c \cup D'^c \cup D''^c, N\rangle \vdash_s \mathrm{contr}(G) \wedge \mathrm{contr}(\mathrm{fails}(G))$$

which contradicts the induction hypothesis.

Thus by induction, we get the result.

□

This in itself is not a particularly important result, but it is useful to prove further results, such as the corollary below.

Corollary 3.3.3 *Let $P = \langle D, N \rangle$ be a locally stratified derivation state. Then it is not the case that*

$$P^c \vdash_s \bot$$

Proof: For this to occur, we must have that there is a clause

$$\bot \subset \exists\, x_1 \ldots x_n\; p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

in D^- such that

$$\langle D^c, N \rangle \vdash_s \exists x_1, \ldots x_n\; p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

Let the answer for the first conjunct be $p(s_1, \ldots s_n)$. Thus we must have that

$$\langle D^c, N \rangle \vdash_s (\bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))[x_i \leftarrow s_i]_{i=1}^{n})$$

Now for $p(s_1, \ldots s_n)$ to succeed we must have that

$$\langle D^c, N \rangle \vdash_s \exists(\mathrm{contr}(G_j)[x_i \leftarrow s_i]_{i=1}^{n})$$

for some j. Then we have that

$$\langle D^c, N \rangle \vdash_s \mathrm{contr}(G') \wedge \mathrm{contr}(\mathrm{fails}(G'))$$

for the goal $G' = \exists(G_j[x_i \leftarrow s_i]_{i=1}^{n})$, which contradicts lemma 3.3.2.

□

Next we give a formal proof of the fact that the "short-cut" rule referred to above is indeed correct.

Proposition 3.3.4 *Let $P = \langle D, N\rangle$ be a locally stratified derivation state, and $p(s_1, \ldots s_n)$ be an atom such that there is a clause*

$$\bot \subset \exists x_1, \ldots x_n\, p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

in P^c. Then

1. $\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \bot$ *iff* $\langle D^c, N\rangle \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^{n})$

2. *If* $\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_f \bot$ *then* $\langle D^c, N\rangle \vdash_f \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^{n})$

3. *If* $\langle D^c, N\rangle \vdash_f \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^{n})$ *then* $\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \not\vdash_s \bot$.

Proof:

1. The $\Leftarrow$ direction is clear. For the other direction, consider

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s$$
$$\exists x_1, \ldots x_n\, q(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

where $p \neq q$. If $\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s q(t_1, \ldots t_m)$, then $E_j[x_i \leftarrow t_i]_{i=1}^{m}$ for some j and

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \exists((E_j \wedge \mathrm{contr}(G_j))[x_i \leftarrow t_i]_{i=1}^{m})$$

But we must also have that

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \forall((\neg E_j \vee \mathrm{contr}(\mathrm{fails}(G_j)))[x_i \leftarrow t_i]_{i=1}^{m})$$

This is impossible by lemma 3.3.2, and so we must have $p = q$ and $n = m$. Now if we have

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s p(t_1, \ldots t_n)$$

where $s_i \neq t_i$, then, as above, there is a goal G' such that

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \mathrm{contr}(G') \wedge \mathrm{contr}(\mathrm{fails}(G'))$$

Hence, we must have $s_i = t_i$ for all $1 \leq i \leq n$, and so

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^{n})$$

Now as P is locally stratified, we have that $\text{fails}(G_i)[x_i \leftarrow s_i]_{i=1}^{n}$ is independent of $p(s_1, \ldots s_n)$, and by lemma 3.3.1, we have that

$$\langle D^c, N\rangle \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^{n})$$

2. Obvious.

3. Consider

$$\langle D^c, N\rangle \vdash_f \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^{n})$$

By a similar argument to 1 above, we know that it cannot be the case that

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s$$
$$\exists x_1, \ldots x_n\, q(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))$$

unless we have

$$\langle D^c, N\rangle \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^{n})$$

This is a contradiction by proposition 2.3.5, and so we cannot have that

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \bot$$

□

We may think of the above proposition as the formal justification of the "short cut" described above, in that when searching for an **O**-proof of

$$\langle D^c \cup \{p(s_1, \ldots s_n)\}, N\rangle \vdash_s \bot$$

it is only necessary to consider the clause

$$\bot \subset \exists x_1, \ldots x_n\, p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))$$

rather than all clauses in D^-, as all that can happen by ignoring the other clauses is that some loops are avoided.

The reason that we use 3 above, rather than the converse to 2, which is stronger, is that there may be a clause in D^- which leads to a loop, even though the relevant clause for $p(s_1, \ldots s_n) \supset \bot$ does not. For example, let P be the program $\langle D, \{p, q, r\}\rangle$ where D is

$$\neg q \supset p$$
$$r \supset r$$

P^c is then

$$(q \supset \bot) \supset p$$
$$r \supset r$$

$$q \wedge p \supset \bot$$
$$(r \wedge (r \supset \bot)) \supset \bot$$
$$q \supset \bot$$

It is clear that

$$\langle D^c \cup \{p\}, N\rangle \vdash_f q \wedge p$$

but that an attempt to find an **O**-proof such that

$$\langle D^c \cup \{p\}, N\rangle \vdash_s r \wedge (r \supset \bot)$$

leads to a loop. Hence, $\langle D^c \cup \{p\}, N\rangle \nvdash_f \bot$, but $\langle D^c \cup \{p\}, N\rangle \vdash_f q \wedge p$. This means that $\vdash_f$ is not quite strong enough for our purposes, and so we need a slightly stronger relation in order to avoid some unnecessary loops.

Hence we define the relations $\vdash_{sc}$ and $\vdash_{fc}$, which are to be used on the completion of derivation states.

Definition 3.3.1 *Let $\langle P, G\rangle$ be a locally stratified derivation pair where $P = \langle D, N\rangle$. We define the relations $\vdash_{sc}, \vdash_{fc} \subseteq \mathcal{P}^c \times \mathcal{G}^c$, where $\mathcal{P}^c$ is the set of all possible completions of derivation states, and $\mathcal{G}^c$ is the set of all goal formulae in "contradiction" form (i.e. contr(G)), as the smallest relations which satisfy*

1. $\vdash_{sc} = \vdash_s$

2. $\vdash_f \subseteq \vdash_{fc}$

3. $P^c \vdash_{fc} p(s_1, \ldots s_n) \supset \bot$ *iff*

$$P^c \vdash_{fc} (\bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))[x_i \leftarrow s_i]_{i=1}^{n})$$

where there is a clause

$$\bot \subset \exists x_1, \ldots x_n\, p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))$$

It is clear that $P^c \vdash_s G$ iff $P^c \vdash_{sc} G$, and by proposition 3.3.4 we get $P^c \vdash_{sc} p(s_1, \ldots s_n) \supset \bot$ iff

$$P^c \vdash_{sc} \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^{k}$$

It is also clear that $P^c \vdash_f G$ implies $P^c \vdash_{fc} G$, but the converse does not hold. As noted above, it is possible that $\langle D^c \cup \{p\}, N\rangle$ may lead to a loop when trying to prove $\bot$ without the short-cut rule, and so it is not the case that $P^c \not\vdash_s p \supset \bot$, but that $P^c \vdash_{fc} p \supset \bot$.

We are now in a position to prove the central result, i.e. that the completion has the expected operational behaviour.

Proposition 3.3.5 *Let $P = \langle D, N\rangle, G$ be locally stratified derivation pair. Then*

1. $P \vdash_s G \Leftrightarrow P^c \vdash_{sc} \text{contr}(G)$

2. $P \vdash_f G \Leftrightarrow P^c \vdash_{fc} \text{contr}(G)$

Proof: We proceed by induction on the height of the **O**-derivation for G.

Clearly 1 and 2 above are equivalent to the corresponding statements when replaced by $D \cup \{D'\}$. This is due to the fact that for any goal formula G, $D' \supset G$ is also a goal formula and that if we have $\langle D, N\rangle \vdash_s D' \supset G$ or we have $\langle D, N\rangle \vdash_f D' \supset G'$, then we must have that D' does not extend any completely defined predicate of D, and so $D'^c = D'^+$.

In the base case G is an atom A.

A: Let A be $p(s_1, \ldots s_n)$.

1. As the **O**-proof is of the smallest possible height, we have $P \vdash_s A \Leftrightarrow A \in (D \cup \{D'\})$, which is clearly equivalent to $P^c \vdash_{sc} A$.
2. Similarly, $P \vdash_f A \Leftrightarrow \forall B \in (D \cup \{D'\})\ B \not\propto A$ and $\forall G \supset B \in (D \cup \{D'\})\ B \not\propto A$, and so $p(s_1, \ldots s_n)$ does not match any clause head $p(t_{i1}, \ldots t_{in})$, which is equivalent to $\bigwedge_{i=1}^{k} \forall(\neg E_i)$, i.e. $\neg \bigvee_{i=1}^{k} \exists(E_i)$ where $E_i = s_1 = t_{i1} \wedge \ldots \wedge s_n = t_{in}$, and so $P^c \vdash_{fc} A$.

Hence the inductive hypothesis is that 1 and 2 hold for all D and D' and for all goals G whose **O**-derivation is less than a given depth. There are seven cases:

A: Let A be $p(s_1, \ldots s_n)$.

1. $P \vdash_s A$ iff $\exists G \supset A \in (D \cup \{D'\})$ such that $P \vdash_s G$, i.e. $P \vdash_s \exists((E_j \wedge G_j)[x_i \leftarrow s_i]_{i=1}^n)$, and by the hypothesis this is equivalent to $P^c \vdash_{sc} \exists((E_j \wedge \mathrm{contr}(G_j))[x_i \leftarrow s_i]_{i=1}^n)$, which in turn is just $P^c \vdash_{sc} A$.
2. $P \vdash_f A$ iff $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D \cup \{D'\})$ such that $B \propto A$ we have $P \vdash_f G$, i.e. $P \vdash_f \exists((E_i \wedge G_i)[x_i \leftarrow s_i]_{i=1}^n)$ for all $1 \leq i \leq n$, and by the hypothesis this is equivalent to $P^c \vdash_{fc} \exists((E_i \wedge \mathrm{contr}(G_i))[x_i \leftarrow s_i]_{i=1}^n)$, which in turn is just $P^c \vdash_{fc} A$.

$\neg A$: Let A be $p(s_1, \ldots s_n)$.

1. $P \vdash_s \neg A$ iff name$(A) \in$ den(N) and $P \vdash_f A$, which is equivalent to $\forall B \in (D \cup \{D'\})\ B \not\propto A$ and $\forall G \supset B \in (D \cup \{D'\})$ such that $B \propto A$, we have $P \vdash_f G$, i.e. $P \vdash_f \exists((E_i \wedge G_i)[x_i \leftarrow s_i]_{i=1}^n)$ for all $1 \leq i \leq n$. This in turn is equivalent to $P \vdash_s \forall(\neg E_i \vee \text{fails}(G_i)[x_i \leftarrow s_i]_{i=1}^n)$ by proposition 3.2.1, and by the hypothesis this is equivalent to $P^c \vdash_{sc} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^n)$ for all $1 \leq i \leq n$, which is just $P^c \vdash_{sc} p(s_1, \ldots s_n) \supset \bot$.

2. $P \vdash_f \neg A$ iff $P \vdash_s A$, which is equivalent to $\exists G \supset A \in (D \cup \{D'\})$ such that $P \vdash_s G$, i.e. $P \vdash_s \exists((E_j \wedge G_j)[x_i \leftarrow s_i]_{i=1}^n)$ for some $1 \leq j \leq k$. This in turn is equivalent to $P \vdash_f \forall((\neg E_j \vee \text{fails}(G_j)[x_i \leftarrow s_i]_{i=1}^n)$ by proposition 3.2.1, and by the hypothesis this is equivalent to $P^c \vdash_{fc} \forall((\neg E_j \vee \text{fails}(\text{contr}(G_j)))[x_i \leftarrow s_i]_{i=1}^n)$, which is just $P^c \vdash_{fc} p(s_1, \ldots s_n) \supset \bot$.

$G_1 \vee G_2$:

1. $P \vdash_s G_1 \vee G_2$ iff $P \vdash_s G_1$ or $P \vdash_s G_2$, and by the hypothesis this is equivalent to $P^c \vdash_{sc} \text{contr}(G_1)$ or $P^c \vdash_{sc} \text{contr}(G_2)$, which in turn is just $P^c \vdash_{sc} \text{contr}(G_1) \vee \text{contr}(G_2)$, i.e. $P^c \vdash_{sc} \text{contr}(G_1 \vee G_2)$.

2. $P \vdash_f G_1 \vee G_2$ iff $P \vdash_f G_1$ and $P \vdash_f G_2$, and by the hypothesis this is equivalent to $P^c \vdash_{fc} \text{contr}(G_1)$ and $P^c \vdash_{fc} \text{contr}(G_2)$, which in turn is just $P^c \vdash_{fc} \text{contr}(G_1) \vee \text{contr}(G_2)$, i.e. $P^c \vdash_{fc} \text{contr}(G_1 \vee G_2)$.

$G_1 \wedge G_2$:

1. $P \vdash_s G_1 \wedge G_2$ iff $P \vdash_s G_1$ and $P \vdash_s G_2$, and by the hypothesis this is equivalent to $P^c \vdash_{sc} \text{contr}(G_1)$ and $P^c \vdash_{sc} \text{contr}(G_2)$, which in turn is just $P^c \vdash_{sc} \text{contr}(G_1) \wedge \text{contr}(G_2)$, i.e. $P^c \vdash_{sc} \text{contr}(G_1 \wedge G_2)$.

2. $P \vdash_f G_1 \wedge G_2$ iff $P \vdash_f G_1$ or $P \vdash_f G_2$, and by the hypothesis this is equivalent to $P^c \vdash_{fc} \text{contr}(G_1)$ or $P^c \vdash_{fc} \text{contr}(G_2)$, which in turn is just $P^c \vdash_{fc} \text{contr}(G_1) \wedge \text{contr}(G_2)$, i.e. $P^c \vdash_{fc} \text{contr}(G_1 \wedge G_2)$.

$\exists x G$:

1. $P \vdash_s \exists x G$ iff $P \vdash_s G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis this is equivalent to $P^c \vdash_{sc} \text{contr}(G[t/x])$, which in turn is just $P^c \vdash_{sc} \exists x\ \text{contr}(G)$, i.e. $P^c \vdash_{sc} \text{contr}(\exists x G)$.

2. $P \vdash_f \exists x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_f G[t/x]$ for all $t \in R$, and by the hypothesis this is equivalent to $P^c \vdash_{fc} \text{contr}(G[t/x])$

for all $t \in R$, which in turn is just $P^c \vdash_{fc} \exists x \text{ contr}(G)$, i.e. $P^c \vdash_{fc} \text{contr}(\exists x G)$.

$\forall x G$: 1. $P \vdash_s \forall x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_s G[t/x]$ for all $t \in R$, and by the hypothesis this is equivalent to $P^c \vdash_{sc} \text{contr}(G[t/x])$ for all $t \in R$, which in turn is just $P^c \vdash_{sc} \forall x \text{ contr}(G)$, i.e. $P^c \vdash_{sc} \text{contr}(\forall x G)$.

2. $P \vdash_f \forall x G$ iff $P \vdash_f G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis this is equivalent to $P^c \vdash_{fc} \text{contr}(G[t/x])$, which in turn is just $P^c \vdash_{fc} \forall x \text{ contr}(G)$, i.e. $P^c \vdash_{fc} \text{contr}(\forall x G)$.

$D'' \supset G$: 1. $\langle D \cup \{D'\}, N\rangle \vdash_s D'' \supset G$ iff $\text{names}(\text{heads}(D'')) \subseteq \text{ass}(N)$ and $\langle D \cup \{D'\} \cup \{D''\}\rangle \vdash_s G$ and by the hypothesis this is equivalent to $\langle D^c \cup \{D''\}, N\rangle \vdash_{sc} \text{contr}(G)$, i.e. $P^c \vdash_{sc} D'' \supset \text{contr}(G)$, and as $\text{names}(\text{heads}(D'')) \subseteq \text{ass}(N)$, $\text{contrd}(D'') = D''$, and so this is just $P^c \vdash_{sc} \text{contr}(D'' \supset G)$.

2. $\langle D \cup \{D'\}, N\rangle \vdash_f D'' \supset G$ iff $\text{names}(\text{heads}(D'')) \subseteq \text{ass}(N)$ and $\langle D \cup \{D'\} \cup \{D''\}\rangle \vdash_f G$ and by the hypothesis this is equivalent to $\langle D^c \cup \{D''\}, N\rangle \vdash_{fc} \text{contr}(G)$, i.e. $P^c \nvdash_{sc} D'' \supset \text{contr}(G)$ and as $\text{names}(\text{heads}(D'')) \subseteq \text{ass}(N)$, $\text{contrd}(D'') = D''$, and so this is just $P^c \vdash_{fc} \text{contr}(D'' \supset G)$.

□

In this way we see that our completion preserves the computational behaviour of the program, and gives an explicit form of negation, rather than the implicit way NAF is defined. Our completion also has the advantage of being executable, in the sense that it is a first-order hereditary Harrop formula program, and so, provided we can find some way of solving the inequations, we may compute directly from the completion, rather than only use it as a semantic device. This is due to the fact that we may view the completion as a program for which A fails iff $A \supset \bot$ succeeds (where A is completely defined). This property may also be seen as a way of reconciling NAF with the interpretation of negation in intuitionistic logic, in that an implicit form of negation (NAF) is given an explicit form (P^c). It is this explicit representation of an implicit definition which requires that the

class of programs be restricted. We shall see in chapter 5 how we may give a form of semantics to programs which contain negations for which no restriction on the class of programs is necessary. This suggests that we should perhaps use the completion as some kind of guide to the success and failure of goals, rather than a semantic device per se, and use some kind of model which does not use such an explicit construction for the semantics.

Chapter 4

Answer Substitutions for Negated Goals

In this chapter we consider the problem of finding answer substitutions for existentially quantified negated goals. This requires an extension to the usual process, as we need more than just unification. We give an algorithm which is suitable for this purpose, which is incremental, and hence able to make use of new information without starting again from scratch. We also show the correctness of the algorithm, and discuss some possible extensions of it.

4.1 Motivations

As mentioned in the previous chapter, we wish to find answer substitutions for existentially quantified negated goals. In order to generate the required answer substitutions, we need to do more work than is typically done by the resolution process. For example, consider the program

$$p(a)$$
$$q(b) \supset p(b)$$
$$q(a)$$

and the goal $\exists x \neg p(x)$. The goal $\exists x p(x)$ succeeds with the answer substitution being $x \leftarrow a$. A direct application of the NAF rule will then give us that as $p(x)$ succeeds, $\neg p(x)$ fails. Now whilst $\forall x \neg p(x)$ is false, $\neg p(b)$ follows from the CWA,

and so one correct answer substitution for the goal $\exists x \neg p(x)$ is $x \leftarrow b$. Thus our process needs to find failures rather than successes, which generally takes more effort, as we may need to consider every clause in the program whose head matches $p(x)$, rather than looking for a matching clause whose body succeeds, which we may find at the first attempt to match the goal with a clause in the program.

It also is possible that there are more answer substitutions for the above goal. For example, if there is another constant c in the language, then $\neg p(c)$ is true, and so $x \leftarrow c$ is another correct answer substitution. Hence, we may need to consider more information than is given in the program in order to generate answer substitutions. This is a consequence of the fact that NAF is an implicit form of negation, i.e. that we define what is false by the complement of what is true, and so if $p(c)$ is false, then we do not add $\neg p(c)$ to the program, but merely leave it out. Thus the signature used may include constant and function symbols which do not appear anywhere in the program. This is not a problem for SLD-resolution, as it only generates answer substitutions by unification, and so all symbols used in such answer substitutions must appear somewhere in the program. For this reason, we assume that each predicate has a signature, in the manner discussed earlier, so that we can tell what language may be used to construct answer substitutions.

In this way the problem reduces to an exhaustiveness check in the sense that given a goal G, we wish to find, if possible, all instances of G which fail. This is the direct complement of the SLD-resolution process, as we may think of that as finding all instances of G which succeed. The signature is used to determine exactly what all the instances of G are. In what follows we assume that the goal is just an atom A, as we only allow negation to be applied to atoms, and that we wish to find instances of A which fail, i.e. we wish to find the instances of A which are not in the set of successful instances of A.

Now if we are given the completion of the program, as described above, then our task is made easier, as we already have a computational description of the goals which fail. The only difficulty is the solution of inequations. This may be addressed by looking at it as an instance of the *relative complement* problem, which may be stated as: given a term t and a set T of instances of t, find all instances of

t which are not instances of any term in T. As the formula $x_1 \neq t_{i1} \vee \ldots \vee x_n \neq t_{in}$ is equivalent to the set of all instances of $p(x_1, \ldots, x_n)$ which are not instances of $p(t_{i1}, \ldots t_{in})$, we may use an algorithm for the relative complement problem to solve the inequations.

The relative complement problem was addressed by Lassez and Marriott [59], and it was shown that an explicit finite representation of the relative complement may not exist. However, a criterion was given for determining when it is possible to give a finite representation of the relative complement, as well as an algorithm for finding the finite representation when it does exist. A parallel algorithm for this problem was given in [57].

This algorithm is not quite suitable for our purposes. In [59] the emphasis was on finding a representation of the entire relative complement; here we are interested in finding correct answer substitutions. Thus we may only need to produce one substitution, although subsequent failures in the search process may mean we need to backtrack over $\neg p(x)$ and so search for another correct answer substitution. Our search for a correct answer substitution may not need to examine all of the relative complement, and so even if there is no finite representation, we may still be able to find what we want. Rarely will we need to enumerate all of the relative complement, and so we approach the problem in a slightly different way than was done in [59].

Another property important for inequation solving algorithms for logic programming is that the algorithm be *incremental*, in that if the set of constraints increases, we wish to be able to use previous answers to solve the new problem, rather than recompute from scratch. In our case this means that if the relative complement of a given term t with respect to a set of terms T is computed, then we wish to be able to use this answer to reduce the amount of work needed to find the relative complement of t with respect to $T \cup t'$, where t' is another instance of t. This will make our algorithm useful for wider applications, such as an inequation solver for a constraint logic programming language [48].

Even if the completion of the program is not available to us (for example, the principle of information hiding may mean that we do not have access to the code

for a certain program, but only to the answers it generates) we will still be able to use the incremental algorithm to generate answer substitutions by computing the relative complement of A with respect to the set of successful instances of A. The set of most general instances of A which succeed may be infinite, and this set will be generated incrementally, i.e. one such instance will be found, then another, and so on. If the set is infinite then the process will not terminate, and so we cannot merely sit and wait for this set to be produced. An incremental algorithm will allow us to produce successive approximations to the relative complement, so that if the enumeration of the successful instances of A terminates, then the most recent approximation becomes exact.

An important observation is that in order to find an answer substitution we may need to consider a significant part of an infinite set of terms, as the relative complement of the success set may not be finitely representable, even if the set of successful instances of A is finitely representable. However, the desired answer may occur in a finitely representable subset of the relative complement, and so whilst the lack of a finite representation is a nuisance, it may suit our purpose to attempt the enumeration of an infinite representation.

For example, given the program

$$\begin{gathered}\text{even}(0)\\ \forall x\, \text{even}(x) \supset \text{even}(s^2(x))\end{gathered}$$

the set of successful instances is $\{\text{even}(s^n(0)) \mid n \text{ is even }\}$. It seems natural given the goal $\exists x \neg \text{even}(x)$ to produce the answer substitution $x \leftarrow s(0)$, and then if another is required, $x \leftarrow s^3(0)$ etc. These substitutions may be "used" by other goals. Naturally, there is still some room for the delaying technique of Mu-Prolog, in that we may wish to compute answer substitutions from other goals and then check them using NAF. However, this technique can only be supplementary to the process described herein.

In this way our algorithm is incremental and attempts to enumerate possible answers even when it is known that there is no explicit finite representation of the instances of A which fail.

There remains a problem of finite representation even when the success set is finite, or we have the completion of the program. As noted above, there may not be a finite set of answer substitutions which is complete, i.e. includes all correct answers. The reason that there may not be a finite representation of such instances is due to the possibility that different variables in A may be mapped in the success set to terms containing the same variable. For example, given the program $\forall z\, p(z,z)$, the goal $\exists x \exists y \neg p(x,y)$ and the signature $\{a/0, f/1\}$, then whilst the set of successful instances of $p(x,y)$ is just $\{p(z,z)\}$, the relative complement of $p(x,y)$ with respect to $p(z,z)$ is

$$\{p(a, f(\_)), p(f(\_), a), p(f(a), f(f(\_))), p(f(f(\_)), f(a)) \ldots\}$$

where _ is used to denote an arbitrary term.

The problem is that we have to go deeper and deeper into the term in order to finitely specify the complement. A formalisation of this idea is given in [59]. The instances which cause this behaviour are called *restricted* in [59].

As noted above, we are interested in finding an answer substitution and hence one particular term of this set, one of these rather than a finite representation of all of them. However, due to the fact that we may need to backtrack over this substitution, we wish our algorithm to produce as many of these instances as desired. The way this is done is to enumerate all such terms of depth d, then all of depth $d+1$, and so forth until the desired term is found, thus enumerating the relative complement of a term with respect to a restricted instance in a stratified way.

In the light of this result, it is possible to argue that explicit substitutions are not desirable, and that some form of implicit representation of the answers should be used. The most prominent example of such a representation is constraints [48]. This is not quite the point at issue; whilst constraints do capture all possible answers in a finite representation, we may still want to look at some explicit answers. Indeed, for the formula $\exists x \neg p(x)$ to be intuitionistically provable we need to provide a witness, i.e. an explicit answer. As mentioned above, we do not necessarily want all answers, as we are after a particular correct answer substitution

for existentially quantified goals, and so whilst we desire each substitution to be maximally general[1] (so that it is not an instance of some other correct answer substitution), we do not necessarily need to find all most general correct answer substitutions for the goal. Also, any constraint mechanism will still need some criterion and algorithm for giving an explicit answer where possible. For example, given the inequations $x \neq 0$ and $x \neq s(s(y))$ we would expect to get the answer $x = s(0)$. The algorithm below shows how this may be smoothly integrated with a general method of finding finitely representable subsets of a relative complement, which will be required by a constraint solver. Hence, any representation, be it substitutions or constraints, will need to address the issues raised above.

Some other approaches to this problem have concentrated on representation of the solutions [15,16,53,110,96] or on a transformation approach [7,6]. We feel that whilst these are relevant issues, there is still a need to provide a method to generate explicit answers where possible, and for this method not to depend on restrictions to the class of programs.

For example, given the program

$$\forall x\ le(0, s(x))$$
$$\forall x \forall y\ le(x, y) \supset le(s(x), s(y))$$

and the goal $\exists x \neg le(x, s(0))$, an answer of $x \neq 0 \wedge x \neq s(0)$ is certainly correct, but conveys less information than the answer substitution $x \leftarrow s^2(y)$. In this way an answer of the form $x \neq t$ to a query may not be as informative as an explicit example of a term t' such that $t' \neq t$. Also, it seems that such a method of generating explicit answers should not depend upon structural properties of the program, but purely in terms of what succeeds and fails. Thus all that is important is to give an algorithm that produces answer substitutions for non-ground negated atoms.

---

[1]We wish to return the substitution $x \leftarrow s(y)$, rather than $x \leftarrow s(0)$, $x \leftarrow s^2(0) \ldots$

In this way our philosophy is similar to that of Maluszyński and Näslund [68], in that we need only consider the operational behaviour of the program. However, as will be seen, we do not use an explicit constraint, but rather stratify the production of the answers until all the required answers have been found.

4.2 Definitions

We assume that there are a countable number of distinct new variables available. We denote variables by $v, w, x, y, z, \ldots$, constants by $a, b, \ldots$, and function symbols by $f, g, \ldots$. As in Prolog, we use _ to denote distinct variables whose names are of no interest. Hence, $f(x, y)$ and $f(\_,\_)$ denote the same term. We say that a term t is a function term if t is neither a variable nor a constant, so that $t = f(t_1, \ldots t_n)$ with $n > 0$.

As noted in [59], we often think of a term as a finite representation of the set of all its instances, so that a term t may represent either the syntactic term t or $\{t\ \theta \mid \theta$ is a substitution$\}$. This may be thought of as some sort of implicit universal quantification of the variables appearing in t. It will always be clear from the context whether we mean the term itself or the set of its instances.

We use the notation t/T to denote the set of all instances of t which do not have any instance in common with any term in T, where T is a set of terms. Thus t/T represents an implicit generalisation in the terminology of [59]. We will often refer to t/T as a *relative complement problem.* When T is just a singleton set $\{t'\}$ we write t/t'. We also use the obvious generalisation T_1/T_2 to denote the set of all instances of any term in T_1 which do not have any instance in common with any term in T_2. Hence $t \in T_1/T_2$ means that t is an instance of some term in T_1, and t is not an instance of any term in T_2. We use $term(t/T)$ to denote t and $rest(t/T)$ to denote T, and for convenience we define $term(t) = t$, and $rest(t) = \emptyset$.

It is obvious that if t and t' do not unify then $t/t' = \{t\}$.

As we are dealing with an implicit form of negation, we need a notion of signature to explain precisely which terms are under consideration.

Definition 4.2.1 *Let Σ be a signature. For any $f/n \in \Sigma$ we denote by $(f/n)^+$ the term $f(x_1, \ldots x_n)$, where the x_i are distinct new variables.*

We denote by Σ^* the set of all ground terms which may be constructed from Σ. Thus if Σ is the signature of all function and constant symbols used in the program, Σ^* is the Herbrand Universe $\mathcal{U}$. When the value of n is obvious from the context (e.g. for a constant) we will often write f/n as just f. We denote $\{(f/n)^+ \mid f/n \in \Sigma\}$ by Σ^+. For a term t which is not a variable, we denote by $symbol(t)$ the outermost symbol of t. For example, $symbol(f(a, g(b))) = f$.

As pointed out in [59], the relative complement of $f(t)$ with respect to some set of instances T is always impossible to represent explicitly when there are an infinite number of symbols in the signature, and is trivial to compute when Σ contains only constants, and so the only interesting case is when Σ is finite and contains constants and a function symbol of non-zero arity (thus ensuring, in the absence of typing, that the Herbrand Universe is infinite). For this reason the algorithm given below assumes that Σ is finite. It may be possible to somehow extend the algorithm below to cope with infinite signatures, but it is difficult to see how such an extension would be useful.

We use the definition of a restricted instance from [59], given below. We denote by $vars(t)$ the set of all variables which appear in t.

Definition 4.2.2 *An instance $t\theta$ of t is a* restricted instance *of t if any variable appears more than once in the sequence $v_1\theta, \ldots v_n\theta$ where $vars(t) = \{v_1, \ldots v_n\}$. Otherwise $t\theta$ is an* unrestricted *instance of t.*

Note that a restricted instance t' of t imposes more dependencies between the variables of t than does t itself. For example, $f(z, z, z)$ is a restricted instance of $f(x, y, y)$, but $f(a, g(z), g(z))$ is an unrestricted instance of $f(x, y, y)$.

The predicate $restricted(t, t')$ is true iff t' is a restricted instance of t. Similarly, the predicate $unify(t, t')$ is true iff t and t' are unifiable. Note that t and t' have a common instance iff t and t' unify, and that if θ is the most general unifier of t and t' then the *most general instance* $mgi(t, t')$ is given by $t\theta = t'\theta$.

If t' is a restricted (unrestricted) instance of t, then t/t' is a restricted (unrestricted) relative complement.

We will find it convenient to use the following extended form of substitution.

Definition 4.2.3 *Given two terms t' and t'', we define the substitution $t' \leftarrow t''$ by*

- *If t' is a variable then $t[t' \leftarrow t'']$ is t with all occurrences of t' replaced simultaneously by t''*
- *If t' is a constant then $t[t' \leftarrow t''] = t$*
- *If t' is a function term and t' and t'' are not unifiable then $t[t' \leftarrow t''] = t$*
- *If t' is a function term and t' and t'' are unifiable with $t' = f(t_1 \ldots t_n)$ and $t'' = f(s_1, \ldots s_n)$ then $t[t' \leftarrow t'']$ is the simultaneous application of the substitutions $[t_1 \leftarrow s_1] \ldots [t_n \leftarrow s_n]$ to t*

For example, $g(f(x,x),y)[f(a,x) \leftarrow f(y,b)] = g(f(b,b),y)$.

This may be thought of as using the two terms to induce a substitution so that t' imitates t'' whilst remaining an instance of t'. This imitation can be exact when t' is a variable, but is less so when t' is a function term. We will see how this is used later.

We denote the empty list by $[]$, and concatenation by $A.B$, so that $A.B$ is a list with head A and tail B. We use *head*(List) and *tail*(List) to denote the usual list destructor functions, so that $head(A.B) = A$, $tail(A.B) = B$.

We denote by *refresh*(t) the same term t except that each variable occurrence in t is replaced by a new distinct variable, i.e. $refresh(f(x,x)) = f(y,z)$.

Definition 4.2.4 *If t and refresh(t') are unifiable, then we define* $\mathrm{ref}(t,t')$ *as $\{s\}$, where s is the most general instance of t and refresh(t'). Otherwise,* $\mathrm{ref}(t,t')$ *is the empty set.*

We also define

$$\text{ref}(t,T) = \bigcup_{t' \in T} ref(t,t')$$
$$\text{ref}(S,T) = \bigcup_{s \in S} ref(s,T)$$

We will often refer to $ref(t,t')$ as just s, rather than the set whose only element is s.

Note that $t/ref(t,t')$ is an unrestricted relative complement. This is due to the fact that the variables in $refresh(t')$ are only made identical in $ref(t,t')$ in the way that the variables in $refresh(t)$ are made identical in $mgi(t, refresh(t))$. Thus no new dependencies are introduced, and hence $ref(t,t')$ is an unrestricted instance of t.

This allows use to break the problem down somewhat. For example, for the relative complement $f(x,y)/f(g(z),g(z))$ with $\Sigma = \{a/0, g/1\}$, we wish to produce

$$\{f(a,\_), f(\_,a), f(g(x),g(y))/f(g(z),g(z))\}$$

Note that

$$ref(f(x,y), f(g(z),g(z)) = f(g(x),g(y))$$

and so

$$f(x,y)/ref(f(x,y), f(g(z),g(z))) = \{f(a,\_), f(\_,a)\}$$

and that

$$f(x,y)/f(g(z),g(z)) = f(x,y)/f(g(x'),g(y')) \cup f(g(x'),g(y'))/f(g(z),g(z))$$

It is this reduction which allows us to stratify the production of the relative complement t/t' when t' is a restricted instance of t. This idea is formalised in lemma 4.3.6.

For an atom A, we denote by $succeeds(A)$ the set $\{A\theta \mid A$ succeeds with answer substitution $\theta\}$. This may be an infinite set, as in the case of the program given

above for the even predicate. We denote by $match(A)$ the set of all atoms which are the head of some clause in the program and unify with A.

The following definition will be useful when we consider restricted instances.

Definition 4.2.5 *Let t be a term over the signature Σ. If t is non-ground, then* strata(t) $= \{t[x_i \leftarrow t_i]_{i=1}^{n} \mid t_i \in \Sigma^+\}$, *where $vars(t) = \{x_1, \ldots x_n\}$. Otherwise, $strata(t) = \{t\}$.*

The reason for the name strata is that if t is non-ground (i.e. contains a variable) then each term in $strata(t)$ has depth one greater than that of t, as each variable is replaced by a term from Σ^+. Thus we penetrate one level deeper into the term t, and so $strata(t)$ contains all instances of t of no more than a given depth, and so we can stratify the instances of t in this way. This allows us to give some answer substitutions when it is impossible to enumerate all of them, as discussed below.

4.3 Preliminaries

Before we present the algorithm itself, we show some useful lemmas.

The first is an easy result about the relative complement.

Lemma 4.3.1 *Let t and t' be two terms. Then either $t/t' = \{t\}$ or t and t' are unifiable with $t/t' = t/mgi(t, t')$.*

Proof: If t and t' are not unifiable, then t and t' have no instances in common, and so there is no instance of t which is an instance of t', i.e. $t/t' = \{t\}$.

Otherwise, t and t' are unifiable, and hence have a most general common instance, $mgi(t, t')$. As $mgi(t, t')$ is an instance of t', we have that $t/t' \subseteq t/mgi(t, t')$. Now let $t'' \in t/mgi(t, t')$. Then t'' is an instance of t, but t'' is not an instance of $mgi(t, t')$, and as any common instance of t and t' is an instance of $mgi(t, t')$, we must have t'' is not an instance of t', i.e. $t'' \in t/t'$. Thus $t/t' = t/mgi(t, t')$.

□

The next lemma, although trivial to prove, is important for computational purposes.

Lemma 4.3.2 *Let T_1 and T_2 be sets of terms, and let t be a term. Then*

$$t/(T_1 \cup T_2) = (t/T_1)/T_2 = (t/T_2)/T_1.$$

This ensures that we can compute t/T via a loop like

```
Inst := t;
for each t' ∈ T do
    Inst := Inst/t'
```

This also ensures that our algorithm will be incremental, as once we have $t/\{t_1, \ldots, t_n\}$, we know that $t/\{t_1, \ldots, t_n, t_{n+1}\} = (t/\{t_1, \ldots, t_n\})/t_{n+1}$.

Another result which will be useful later is the following one, also trivial to prove.

Lemma 4.3.3 *Let t, t_1 and t_2 be terms. Then*

$$\{t_1, t_2\}/t = (t_1/t) \cup (t_2/t).$$

A less trivial result is the following one, which gives a structural result for unrestricted instances, which means that we may use a localised procedure to calculate an unrestricted relative complement.

Lemma 4.3.4 *Let t' be an unrestricted instance of t, where $t = f(t_1, \ldots t_n)$ and $t' = f(s_1, \ldots s_n)$. Then for any $1 \leq k \leq n$ we have*

$$t/t' = \{t[t_k \leftarrow t''] \mid t'' \in t_k/s_k\} \cup t[t_k \leftarrow s_k]/t'.$$

Proof: We proceed by first showing that for any $1 \leq k \leq n$, $\{t[t_k \leftarrow t''] \mid t'' \in t_k/s_k\} \cup t[t_k \leftarrow s_k]/t' \subseteq t/t'$, and then the reverse inclusion.

$\subseteq$: If $u \in t[t_k \leftarrow s_k]/t'$, then u is an instance of t and not an instance of t', and so $u \in t/t'$.

If $u \in \{t[t_k \leftarrow t''] \mid t'' \in t_k/s_k\}$, then it is obvious that u is an instance of t. Let u_k be the kth argument of u. Now for some $t'' \in t_k/s_k$ we have $u = t[t_k \leftarrow t'']$. Thus u_k is not an instance of s_k, which means that u is not an instance of t', i.e. $u \in t/t'$.

$\supseteq$: Let $u \in t/t'$, and let the ith argument of u be u_i. As u is an instance of t, we have $t\theta = u$ for some substitution θ. Thus we have $t_k\theta = u_k$ for any $1 \leq k \leq n$, and that if θ_k is the restriction of θ to the variables which appear in t_k, then $\theta_k = [t_k \leftarrow u_k]$, and so $u = t\theta$ is an instance of $t\theta_k = t[t_k \leftarrow u_k]$.

Now if u_k is not an instance of s_k, then as u_k must be an instance of t_k (as u is an instance of t), we have that $u_k \in t_k/s_k$, and so $u \in \{t[t_k \leftarrow t''] \mid t'' \in t_k/s_k\}$.

If u_k is an instance of s_k, then as t' is an unrestricted instance of t, we know that the variables of t' which appear in s_i are only introduced to t' by the substitution $[t_i \leftarrow s_i]$, and so u is an instance of $t[t_k \leftarrow s_k]$.

□

To see why this result is important computationally, consider the case when we wish to find the unrestricted relative complement t/s, where $t = f(t_1, \ldots t_n)$ and $s = f(s_1, \ldots s_n)$. The above result allows us to search locally, so that we first look for instances of t_1 which are not instances of s_1, and having found such a term t', we then produce the term $t[t_1 \leftarrow t']$ as an answer to the original problem. Once all such terms t' have been found, we have then found all of the relative complement such that $t_1 \neq s_1$, and so we apply the substitution $[t_1 \leftarrow s_1]$ to t before proceeding to do the same for t_2 and s_2 and so on.

Now let us examine the possible cases for t_i and s_i.

1. If t_i is a constant c, then there are no instances of t_i other than c, and so we can get no information from this case (note that s_i must then be c, as s is an instance of t).

2. If s_i is a variable, then we have a similar case, as t_i must also be a variable, and so again no information may be deduced.

3. If t_i is a variable and s_i is not a variable, we may produce an answer $t[t_i \leftarrow f^+]$, for each $f/n \in \Sigma$ such that $f \neq symbol(s_i)$. Hence, it remains only to consider the relative complement $symbol(s_i)^+/s_i$, which is the same as the case when both t_i and s_i are function terms, which is dealt with below.

4. If t_i is a function term, then so is s_i, and $symbol(t_i) = symbol(s_i)$. This case corresponds exactly to the original case of t/s, and so we use the algorithm recursively at this point. Note that as s is an unrestricted instance of t, s_i is an unrestricted instance of t_i, as if there are no repeated variables in $v_1\theta, \ldots v_n\theta$ where $vars(t) = \{v_1, \ldots v_n\}$ and θ is the mgu of t and s, then there can be no repeated variables in $v'_1\theta, \ldots v'_m\theta$ where $vars(t_i) = \{v'_1, \ldots v'_m\}$. Thus the restricted condition may be thought of as global; if the initial problem is unrestricted, then so are any sub-problems generated from it.

Thus for the unrestricted problem we may use this localised procedure to find relative complement. The reason that s must be an unrestricted instance for this procedure to work is that we know that there are no extra dependencies between the variables of s when compared with those of t, and so the structure of the two terms is similar. For example, let Σ be $\{a/0, g/1\}$ and consider the two relative complements $f(x,x)/f(g(z),g(z))$ and $f(x,y)/f(g(z),g(z))$. Note that the first case involves an unrestricted instance and the second a restricted instance.

For the first case, we find all instances of x which are not instances of $g(z)$, which is just a, giving the partial answer $\{f(a,a)\}$, and then having found all instances of x which differ from $g(z)$ the problem may be easily reduced to to $f(g(z),g(z))/f(g(z),g(z)) = \emptyset$ and we are done. Thus the similarity of structure between $f(x,x)$ and $f(g(z),g(z))$ ensures that the localisation produces the right

answer, as the problem reduces to finding all instances of x which are not instances of $g(z)$.

In the second case, the localised procedure alone is not sufficient. Firstly, (ignoring the unrestrictedness requirement), we again find all instances of x which are not instances of $g(z)$, i.e. a, and so we get the partial answer $\{f(a,y)\}$. The procedure above suggests that we then proceed to the relative complement $f(g(z),y)/f(g(z),g(z))$, and so look for instances of y which are not instances of $g(z)$, i.e. a, giving another partial answer $\{f(a,y), f(g(z),a)\}$. We have now reached the end, as we are left with $f(g(z),g(z))/f(g(z),g(z))$. However, we do not have $f(x,y)/f(g(z),g(z)) = \{f(a,y), f(g(z),a)\}$, as $f(g(a),g(g(a)))$ is an instance of $f(x,y)$ which is not an instance of $f(g(z),g(z))$, and so $f(g(a),g(g(a))) \in f(x,y)/f(g(z),g(z))$, but it is not an instance of either $f(a,y)$ or $f(g(z),a)$. Hence, this simple procedure will only work for unrestricted instances, although it may be used as a method of generating some but not all of the relative complement of a restricted instance, as the answers returned will be correct but not complete.

The problem is that a restricted instance produces more dependencies between variables than exist in the original term. In the above example, the most general unifier of the two terms binds x and y to terms containing a shared variable, and so there is a global connection between the two, which is ignored by the localised procedure. As hinted above, we may use the localised procedure to produce partial answers for the restricted case, but we need to do more work in order to produce all possible answers. This idea is formalised in the following lemmas, which give structural results which are important for computational purposes.

Lemma 4.3.5 *Let t' be an instance of t. Then*

1. *$ref(t,t')$ is not the empty set*
2. *s is an unrestricted instance of t, where $\{s\} = ref(t,t')$*
3. *t' is a instance of s, where $\{s\} = ref(t,t')$*

Proof:

1. It is clear that t' is an instance of $refresh(t')$, and as t' is an instance of t, t and $refresh(t')$ have a common instance, and so $mgi(t, refresh(t, t'))$ exists, i.e. $ref(t, t')$ is not the empty set.

2. Let $ref(t, t') = \{s\}$. As $s = mgi(t, refresh(t'))$, s must be an instance of t. As there are no repeated variables in $refresh(t')$, the most general unifier θ of $refresh(t')$ and t can only bind the variables of t to terms containing different variables, and hence there are no repeated variables in $v_1\theta, \ldots v_n\theta$ where $vars(t) = \{v_1, \ldots v_n\}$, i.e. s is an unrestricted instance of t.

3. As t' is an instance of $refresh(t')$ and of t, t' is an instance of s.

□

Lemma 4.3.6 *Let t' be an instance of t. Then*

$$t/t' = t/ref(t, t') \cup ref(t, t')/t'.$$

Proof: From lemma 4.3.5 it follows that $ref(t, t')$ is not the empty set. Let $ref(t, t')$ be $\{s\}$.

$\supseteq$: If $t'' \in t/ref(t, t')$, then t'' is an instance of t but not of s, and from lemma 4.3.5 we have that t' is an instance of s, and so t'' is not an instance of t'.

If $t'' \in ref(t, t')/t'$, then t'' is an instance of s but not of t', and from lemma 4.3.5 we have that s is an instance of t, and so t'' is an instance of t.

In either case we have that $t'' \in t/t'$.

$\subseteq$: Assume $t'' \in t/t'$, and so t'' is an instance of t but not of t'. If t'' is an instance of s, then $t'' \in ref(t, t')/t'$. Otherwise, t'' is not an instance of s, and hence $t'' \in t/ref(t, t')$.

In either case we have $t'' \in t/ref(t, t') \cup ref(t, t')/t'$.

□

A generalisation of this result is given below.

Lemma 4.3.7 *Let $\{t_1, \ldots t_n\}$ be a set of instances of t. Then*

$$t/\{t_1, \ldots t_n\} = t/ref(t, \{t_1, \ldots t_n\}) \cup ref(t, \{t_1, \ldots t_n\})/\{t_1, \ldots t_n\}$$

Proof: We proceed by induction on n. The base case follows immediately from lemma 4.3.6. Hence, we assume that the lemma holds for all values less than a given size. We will write $ref(t, \{t_1, \ldots t_k\})$ as r_k. Now

$$\begin{aligned}
t/\{t_1, \ldots t_n\} &= (t/\{t_1, \ldots t_{n-1}\})/t_n \text{ by lemma 4.3.2} \\
&= ((t/r_{n-1} \cup r_{n-1}/\{t_1, \ldots t_{n-1}\})/t_n \text{ by the hypothesis} \\
&= (t/t_n)/r_{n-1} \cup r_{n-1}/\{t_1 \ldots t_n\} \text{ by lemmas 4.3.2 and 4.3.3} \\
&= ((t/ref(t, t_n) \cup ref(t, t_n)/t_n)/r_{n-1} \cup r_{n-1}/\{t_1 \ldots t_n\} \text{ by lemma 4.3.6} \\
&= t/r_n \cup ref(t, t_n)/\{t_n, r_{n-1}\} \cup r_{n-1}/\{t_1, \ldots t_n\} \text{ by lemma 4.3.2}
\end{aligned}$$

The result will then follow if we can show that

$$\begin{gathered}
ref(t, t_n)/\{t_n, r_{n-1}\} \cup r_{n-1}/\{t_1, \ldots t_n\} = \\
ref(t, t_n)/\{t_1, \ldots t_n\} \cup r_{n-1}/\{t_1, \ldots t_n\}
\end{gathered}$$

as the latter is just

$$r_n/\{t_1, \ldots t_n\}$$

from lemma 4.3.3.

Now from lemma 4.3.5, t_i is an instance of s where $\{s\} = ref(t, t_i)$, and so

$$ref(t, t_n)/\{t_n, r_{n-1}\} \subseteq ref(t, t_n)/\{t_1, \ldots t_n\}$$

which establishes one direction of the desired equality.

Let $\{s_i\} = ref(t, t_i)$.

For the other direction, consider $t' \in ref(t, t_n)/\{t_1, \ldots t_n\}$, so that t' is not an instance of any t_i. If t' is an instance of s_j for some $1 \leq j \leq n-1$, then as t' is not an instance of any t_i, $t' \in r_{n-1}/\{t_1, \ldots t_n\}$. Otherwise, t' is not an instance of any t_j, $1 \leq j \leq n-1$, and so as t' is an instance of s_n, we have that $t' \in ref(t, t_n)/\{t_n, r_{n-1}\}$. Thus we have that

$$ref(t, t_n)/\{t_1, \ldots t_n\} \subseteq ref(t, t_n)/\{t_n, r_{n-1}\} \cup r_{n-1}/\{t_1 \ldots t_n\}.$$

Hence

$$r_n/\{t_1, \ldots t_n\} = ref(t, t_n)/\{t_n, r_{n-1}\} \cup r_{n-1}/\{t_1 \ldots t_n\}$$

which in turn shows that

$$t/\{t_1, \ldots t_n\} = t/r_n \cup r_n/\{t_1, \ldots t_n\}$$

and so the lemma is true for n.

Thus by induction we get the result.

□

The next lemma is also important computationally.

Lemma 4.3.8 *Let t' be an instance of t. Then $t/t' = t/strata(t')$.*

Proof: Clearly every ground instance of t' is an instance of an element of $strata(t')$, and vice-versa, and hence $t/t' = t/strata(t')$. □

These results suggest that when dealing with the case when s is a restricted instance of t, we should produce some possible answers, and then leave the problem in an intermediate state so that we may resume computation later if more answers are desired. For a term $t = f(t_1 \ldots t_n)$ and a restricted instance $s = f(s_1, \ldots s_n)$, the first step is to break the problem up as suggested by lemma 4.3.6, i.e. we

reduce t/s to $t/ref(t,s) \cup ref(t,s)/s$. By lemma 4.3.5, $t/ref(t,s)$ is an unrestricted relative complement, and so we may use the procedure outlined above to produce some answers. By lemma 4.3.6, $t/s = t/ref(t,s) \cup ref(t,s)/s$, and as t/s is restricted and $t/ref(t,s)$ is unrestricted, we must have that $ref(t,s)/s$ is a restricted relative complement, as otherwise the restricted relative complement t/s would be finitely representable, contradicting the quoted result in [59]. Thus $ref(t,s)/s$ is a restricted relative complement, and so the next step is to reduce $ref(t,s)/s$ to $ref(t,s)/strata(s)$, as suggested by lemma 4.3.8. Hence, we need to generate the terms $strata(s) = \{f(s_1 \ldots s_n)[x_i \leftarrow g^+]_{i=1}^m \mid g \in \Sigma\}$ where $vars(s) = \{x_1, \ldots x_m\}$. Next we divide $strata(s)$ into two disjoint sets U and R such that for each $t' \in U, ref(t,s)/t'$ is an unrestricted relative complement, and that for each $t' \in R, ref(t,s)/t'$ is a restricted relative complement. In this way the problem is reduced to T/R, where $T = ref(t,s)/U$. Now we may consider that this has reduced the problem enough, and that we may leave T/R as a reasonable continuation of the problem which may be resumed later. However, we go a little further that this, as there may be terms in T which do not unify with anything in R, and so these terms lead to answers which will not take much effort to produce. Hence, $T/R = T_1 \cup (T_2/R)$ where T_1 is the set of all terms in T which do not unify with any term in R. The final step before we leave this problem is to reduce T_2/R to $T_2/ref(T_2,R) \cup ref(T_2,R)/R$, which ensures that all answers of the same depth are found at once. We then leave $ref(T_2,R)/R$ as the continuation of the problem which may be resumed later if more answers are needed.

In this way, we can use the twofold technique of first considering s as an unrestricted instance of t, producing the solutions so generated, and then substituting for all the variables in s, finding what solutions there may be, and then leaving another restricted relative complement problem behind, which may be attacked later to provide further answers. Thus we work our way through the infinite number of possibilities by first finding all answers such that variables first appear at depth d (the depth of the term s), then those in which they first appear at depth $d+1$, then at depth $d+2$ and so forth. Thus the level at which variables appear in the answers is always increasing.

For example, to find $f(x,y)/f(z,z)$ where $\Sigma = \{a/0, g/1\}$, we proceed as follows:

First, we generate $refresh(f(z,z)) = f(\_,\_)$, and find $f(x,y)/f(\_,\_) = \emptyset$. Next, we find $strata(f(z,z)) = \{f(a,a), f(g(w),g(w))\}$, and so we proceed to $(f(x,y)/f(a,a))/f(g(w),g(w))$. Now $f(x,y)/f(a,a)$ is an unrestricted problem, and so we get that this is just $\{f(g(\_),\_), f(a,g(\_))\}$, and so the overall problem is now $\{f(g(\_),\_), f(a,g(\_))\}/f(g(w),g(w))$. This is the stage referred to above as T/R. Now as $f(a,g(\_))$ does not unify with $f(g(w),g(w))$ we may reduce this to $\{f(a,g(\_)\} \cup f(g(\_),\_)/f(g(w),g(w))$. It is now that we perform the final "refresh" step, in that we reduce $f(g(\_),\_)/f(g(w),g(w))$ to $f(g(\_),\_)/f(g(\_),g(\_)) \cup f(g(\_),g(\_))/f(g(w),g(w))$. The first relative complement is easily seen to be just $\{f(g(\_),a)\}$, and so we finally arrive at $f(x,y)/f(z,z) = \{f(a,g(\_),f(g(\_),a)\} \cup f(g(\_),g(\_))/f(g(w),g(w))\}$. Note that any term in the remaining relative complement has variables appearing at level 4 or more. Hence, the known structure of the answers is always increasing.

Note that the sets of terms $\{f(a,\_), f(g(\_),a)\}$ and $\{f(a,\_), f(\_,a)\}$ "cover" the same set of instances, in that for the signature Σ, the set of all term which are an instance of either term in the first set is the same as the set of all terms which are an instance of either term in the second set. However, the first set has the useful property that the two representative terms do not unify, and so have no instances in common. Thus we have a more specific representation than in the second case.

We can ensure that the computational process has a similar partition property, i.e. that no two terms in the explicit representation are unifiable. For the unrestricted relative complement t/t', consider $\{t[t_i \leftarrow t''] \mid t'' \in t_i/s_i\}$. We may think of this as the set of all instances of t for which the kth subterm is not an instance of s_k. Certainly no instance of $t[t_k \leftarrow s_k]/t'$ is unifiable with any instance of any term in $\{t[t_i[k \leftarrow t''] \mid t'' \in t_k/s_k\}$, and so we only need to establish the partition property for this latter set.

If t_k is a variable and s_k is a function or constant, then the finite representation of t_k/s_k will include any term $t'' \in \Sigma^+$ such that $symbol(t'') \neq symbol(s_k)$. As

the partition property holds for Σ^+, then it will hold for any terms of the form $t[t_k \leftarrow t'']$ where $t'' \in \Sigma^+$.

Hence, if we only generate the representation of t/s in the above manner, i.e. substituting terms from Σ^+ for t_k when t_k is a variable and s_k is not a variable, then the above argument shows that the partition property will hold for the relative complement. Note that we only generate explicit representations from unrestricted complements, and so this establishes the partition property for any relative complement. That this is indeed the case here may be seen from the algorithm below.

Thus the process partitions the instances, and so ensures that none of the terms representing the relative complement "overlap".

4.4 The Incremental Algorithm

Firstly we present the incremental algorithm for the relative complement problem. We then give a formal proof of the algorithm's correctness.

The code for the incremental solution of the relative complement problem is given below. We present the algorithm in the style of a producer/consumer environment, in that we assume that there is some consumer process waiting for the output, and that when enough output has been generated, the consumer process will kill this producer. Thus we imagine that the extension of the SLD-resolution process will use the substitutions generated in the same way as any other, i.e. when a substitution is found, it is applied to the rest of the goal and the next sub-goal is attempted. If subsequently a failure occurs, then on backtracking to this goal we wish for another substitution to be produced, if possible, and then to proceed as before. Backtracking may also be asked for by the user, as it may be desirable to see some of the possible answers. In either case, the resolution process will control the action of the process which generates the substitutions, killing it when no more are needed.

The procedure complement(Term, Restr, Inst, Cont) below calculates the relative complement Term/Restr, giving the explicit answer Inst, and implicit answer Cont, so that Cont is another relative complement problem which may be attempted later in order to produce more solutions to the original problem.

When Term/Restr is an unrestricted relative complement, then Cont is the empty set and Inst is a finite representation of the relative complement.

```
procedure complement(Term,Restr,Inst,Cont)
  Cont := ∅; Inst := Term;
  for each t' ∈ Restr do
    Approx := ∅;
    for each t ∈ Inst do
      if not unify(t,t') then
        Approx := Approx ∪ {t});
      else
        t' := mgi(t,t');
        if restricted(t,t') then
          Ref := ref(t,t');
          call complement({t}, Ref, Ans1, _);
          Approx := Approx ∪ Ans1;
          U := {s ∈ strata(t') | unrestricted(Ref,s)};
          R := {s ∈ strata(t') | restricted(Ref,s)};
          call complement(Ref, U, T, _);
          T2 := { t ∈ T | ∃r ∈ R such that t unifies with r };
          T1 := T\T2;
          Approx := Approx ∪ T1);
          call complement(T2, ref(T2,R), Ans2, _);
          Approx := Approx ∪ Ans2;
          Cont := ref(T2/R)/R;
        else
          if t' is not a variable and t is not a constant then
            if t is a variable then
```

```
                    for each t'' ∈ (Σ\{symbol(t')/arity(t')})^+ do
                        Approx := Approx ∪ {t''};
                    t := (symbol(t')/arity(t'))^+;
                fi
                for each i ∈ {1...arity(t')} do
                    t_i := ith argument of t; s_i := ith argument of t';
                    call complement({t_i}, {s_i}, Ans, _);
                    for each t'' ∈ Ans do
                        Approx := Approx ∪ {t[t_i ← t'']};
                    t := t[t_i ← s_i];
                rof
            fi
        fi
    fi
  rof
  Inst := Approx;
rof
erudecorp
```

We now show that the procedure *complement* is correct. First we show that it always terminates, and then we prove that all the correct answers are found.

Lemma 4.4.1 *The procedure* complement

1. *always terminates.*

2. *always returns finite sets in the variables Inst and Cont*

Proof: The termination of the two outer loops is immediate, as Inst is not altered within the inner loop, and Restr is not altered in the outer loop, and both lists are finite. Thus termination will follow if we can show that the procedure halts for the two terms t and t'.

The proof proceeds in two phases. In the first we show by induction on the depth of t' that the procedure halts when t' is an unrestricted instance of t. In the second phase we show that this is also the case when t' is a restricted instance of t. Notice that the predicates $unify(t, t')$ and $restricted(t, t')$ are both decidable and so cause no termination problems, and also that $mgi(t, t')$ may be easily computed from the most general unifier.

It is also obvious that $ref(t, t')$ and $strata(t')$ cause no termination problems, as both may be easily computed.

The second part of the statement will be shown if we can ensure that only finite sets are added to Approx.

Firstly, assume that t' is an unrestricted instance of t. The if section obviously terminates and only adds a finite set to Approx, and so we concentrate on the for loop. The base case occurs when t' is a constant or a variable. If t' is a variable then termination is obvious. If t' is a constant, then we need only show that the for loop terminates, which is obvious as arity$(t') = 0$, and so the for loop will not be entered. Obviously, the finite set condition is met.

For the inductive case, the only difference is when t' is a function. As arity(t') is finite, we need only show that the recursive call to *complement* terminates. As the depth of s_i is less than that of t', by the inductive hypothesis we have that it terminates.

The subsequent for loop must terminate, as Ans must be finite by the inductive hypothesis, and so the finite set condition holds for the inductive case as well.

Thus we have that the procedure must halt for any t, t' where t' is an unrestricted instance of t, and that the variables Inst and Cont are assigned values which are finite sets.

Secondly, assume that t' is a restricted instance of t. The recursive calls to complement must terminate, as each of the three is made with an unrestricted relative complement problem. As noted above, the functions strata

and ref and the predicates restricted and unrestricted cause no termination problems, and so the procedure complement defined above always terminates for restricted relative complement problems.

The finite set condition must hold also, as we know that it holds for the unrestricted case, and so the sets Ans1, T and Ans2 must all be finite, and so must T_1, being a subset of a finite set, and so only finite sets are ever appended to Approx.

Thus the procedure halts for any t, t' where t' is a restricted instance of t, and so from this and the above result, we get the termination of complement in all cases, as well as the finite set condition. □

We now turn to the more difficult task of proving the correctness of the above procedure. Consider first the code fragment

```
procedure complement(Term,Restr,Inst,Cont)
    Contlist := ∅; Inst := Term;
    for each t' ∈ Restr do
        Approx := ∅;
        for each t ∈ Inst do
            update Approx and Cont as appropriate
        Inst := Approx;
    rof
erudecorp
```

This code is precisely the code from procedure complement above with most of the body replaced by the line beginning "update ...". This code will be correct provided that Approx and Cont are updated correctly, as given some t', we calculate the value of t/t' for each element $t \in$ Inst, adding it to Approx each time, and when Inst is exhausted, we may use Approx rather than go back to Term by the fact that $t/\{t_1, t_2\} = \{t/t_1\}/t_2$. Thus we only need to prove correct the code that updates Approx and Cont.

Proposition 4.4.2 *The procedure* complement *always returns all correct answers.*

Proof: From the above discussion and lemma 4.4.1, we need only consider the correctness for two terms t and t'.

If t and t' are not unifiable, then t and t' have no instances in common, and so there is no instance of t which is an instance of t'. Hence, $t/t' = \{t\}$, and so we add $\{t\}$ to Approx. Otherwise, t and t' are unifiable, and from lemma 4.3.1 we have that $t/t' = t/mgi(t, t')$.

Thus the assignment $t' := mgi(t, t')$ is correct.

As before, the proof now divides into two phases, one for the unrestricted case and the second for the restricted case.

First, we assume that t' is an unrestricted instance of t.

We proceed by induction on the depth of t'. The base case occurs when t' is a constant or a variable. If t' is a variable then nothing happens, which is correct as t must then be a variable too, and so $t/t' = \emptyset$. If t' is a constant and so is t, again nothing happens, and nothing should, as again $t/t' = \emptyset$. Otherwise, t is a variable, and so $t/t' = \Sigma^+ \backslash t'$, which is the same as $(\Sigma \backslash \{symbol(t')\})^+$, as t' is a constant. Now arity$(t') = 0$, and so nothing is done by the for loop and so $\{t[t \leftarrow t''] \mid t'' \in t/t'\}$ is added to Approx, which is correct.

For the inductive case, assume that *complement* is correct for all terms less than depth m, and let t' have depth m. As t' is an instance of t, we note that t must have depth no more than m. The only interesting case is when t' is a function term.

If t is a variable, then $(\Sigma \backslash \{symbol(t')\})^+$ is added to Approx. This addition is the same as $\{t[t \leftarrow t''] \mid t'' \in t/t'\}$. Next t is updated to $symbol(t')^+$, and so if the case when t is a function is correct, this will ensure that $symbol(t')^+/t'$ is added to Approx, and as

$$(\Sigma \backslash \{symbol(t')\})^+ \cup symbol(t')^+/t' = \Sigma^+/t' = t/t'$$

we know that Approx is updated correctly.

If t_1 is a function term, then by lemma 4.3.4 we know that the operation of the for loop is correct, i.e. that it is correct to compute t_1/s_1 and then apply the substitutions $[t_1 \leftarrow t'']$ to t, where $t'' \in t_1/s_1$, and so forth for all the other arguments of t. Hence, we need only show that each iteration performs correctly.

Now Approx is updated by $\{t[t_i \leftarrow t''] \mid t'' \in t_i/s_i\}$, which is correct. t becomes $t[t_i \leftarrow s_i]$, which, by the above discussion, is also correct.

Hence, by induction we get that the procedure *complement* is correct when t' is an unrestricted instance of t.

Next, we assume that t' is a restricted instance of t. Now from lemma 4.3.6 we know that $t/t' = t/ref(t,t') \cup ref(t,t')/t'$, and by lemma 4.3.8 this is just $t/ref(t,t') \cup ref(t,t')/strata(t')$. Hence, the first update to Approx is correct as it adds the former of these two relative complements. Now if $strata(t') = U \cup R$ where unrestricted$(ref(t,t'),u)$ holds for each $u \in U$ and restricted$(ref(t,t'),r)$ holds for each $r \in R$, then by lemma 4.3.2 we have $ref(t,t')/strata(t') = (ref(t,t')/U)/R$. Thus the second call to complement is correct, as it calculates $T = ref(t,t')/U$, which is an unrestricted relative complement. It remains to compute T/R. Now by lemma 4.3.3 we know that

$$T/R = \bigcup_{t'' \in T} t''/R$$

If t'' does not unify with any element of R, then $t''/R = \{t''\}$, and so $T/R = T_1 \cup (T_2/R)$, where T_1 is the set of such terms t''. Hence the second update to Approx is correct. Finally, from lemma 4.3.7 we get that $T_2/R = T_2/ref(T_2,R) \cup ref(T_2,R)/R$. As the first is an unrestricted relative complement, the third call to complement and update to Approx is correct. Thus $T_2/ref(T_2,R)$ may be given an explicit representation, and so the only remaining relative complement problem is $ref(T_2,R)/R$. Hence we get

$$t/t' = t/ref(t,t') \cup T_1 \cup T_2/ref(T_2,R) \cup ref(T_2,R)/R$$

where $strata(t') = U \cup R$, and $T_1 \cup T_2 = ref(t, t')/U$.

Thus all the updates to Approx are correct, and Cont is updated correctly.
□

Thus we see that our algorithm performs as claimed.

Now in order to apply the above algorithm to the generation of answer substitutions, we proceed by finding the relative complement of the goal with respect to the set of successful instances, i.e. the set of answers. Now if the set of instances of $p(t_1, \ldots t_n)$ which succeed is computable and finite, then it is clear that any element of $p(t_1, \ldots t_n)/\text{succeeds}(p(t_1, \ldots t_n))$ is an instance of $p(t_1, \ldots t_n)$ which fails, and so we may use the above algorithm to find answer substitutions for existentially quantified negated goals. In the case of the completion of a program, we may consider the above process as solving the inequations in an incremental fashion, so that the solution to previous inequalities may be used to solve later inequalities. To see this, consider that the negative part of each predicate definition is of the form

$$\forall x_1, \ldots x_n \neg p(x_1, \ldots x_n) \subset \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))$$

which may be re-written as

$$\forall x_1, \ldots x_n \neg p(x_1, \ldots x_n) \subset \bigwedge_{i=1}^{k} \forall(\neg E_i \vee (E_i \wedge \text{contr}(\text{fails}(G_i))))$$

Hence we use the algorithm to solve the inequations, and the usual process for the rest of the goal. Now one possible correct answer for the goal $\neg p(t_1, \ldots, t_n)$ is given by the relative complement

$$p(t_1, \ldots, t_n)/\{p(t_{11}, \ldots, t_{1n}), p(t_{21}, \ldots, t_{2n}), \ldots, p(t_{k1}, \ldots, t_{kn})\}$$

where $E_i = x_1 = t_{i1} \wedge \ldots \wedge x_n = t_{in}$. As we have to find an answer substitution which is valid for each conjunct, one way to proceed is to find a substitution for the first conjunct, and then use the next conjunct to refine this substitution in such a way that the refined substitution is a correct answer substitution for both

conjuncts, and so on. This process of refinement is what happens in the normal derivation process for conjuncts; an answer substitution for B_1 is found, say θ_1, and then we search for an answer substitution for $B_2\theta_1$, and if one is found, say θ_2, we continue the process for $B_3\theta_1\theta_2$ and so forth until all the conjuncts are exhausted or the refinement fails, in which case there is no answer substitution for the conjunct. The incremental property of our algorithm allows the process of generating answer substitutions from the inequations to be smoothly integrated into the derivation process; when processing each conjunct, we may choose either of two refinement techniques - one for the relevant relative complement problem, the other for the substitution generated by the success of the goal B_i. A non-incremental algorithm would not be able to use this simple procedure, as it would need to know in advance which conjuncts will be chosen to solve the inequations, and hence would need to operate "globally", rather than "locally".

Another point to note is that it may be necessary to use universal quantification in goals to ensure that the correct answer is obtained. For example, given the program

$$\forall x \forall y\ q(x,y) \supset p(x)$$
$$\forall x\ q(x,a)$$

and the goal $\exists x \neg p(x)$, then the next goal is $\exists x \forall y \neg q(x,y)$ which obviously fails, as $q(x,a)$ succeeds. Hence, we need to ensure that all such universally quantified variables are correctly handled. Techniques to handle such variables were discussed in section 2.4, and may be used here in addition to the relative complement procedure. This is due to the fact that answer substitutions are only returned through existentially quantified variables. For example, given the program

$$\forall x\ q(x,a)$$

over the signature $\{a/0, b/0, f/1\}$ and the goal $\exists y \forall x \neg q(f(x),y)$, it is clear that $q(f(c),y)/q(f(c),a)$ is $\{q(f(c),b), q(f(c),f(\_))\}$, and hence $y \leftarrow b$ is a correct answer. In this way universal quantifications of variables in negated goals reduce

the number of variables which may be instantiated by the relative complement process.

In the case when the success set is either infinite or not computable (i.e. some instance of the goal loops), then the search for answer substitutions may not terminate. A heuristic which may be useful in this context is to compute $A/\text{succeeds}(A)$ via $A/\text{ref}(A, \text{match}(A)) \cup \text{ref}(A, \text{match}(A))/\text{succeeds}(A)$. This initial computation avoids dependence on the search for successful instances, and so may be thought of as insurance against the possibility that the success set for a given atom may be infinite, or that the search for successful instances loops. This approach ensures that the maximal number of answer substitutions is found.

For example, let Σ be $\{a/0, b/0, f/1, g/2\}$. Consider the goal $\exists y \neg p(y)$ and the program

$$\forall x\ p(x) \supset p(f(x))$$

As the goal will loop, the enumeration of the success set will not terminate, and so we will not find the correct answer substitutions $y \leftarrow a$, $y \leftarrow b$ and $y \leftarrow g(\_,\_)$ unless we do so before attempting the enumeration of the success set. We consider what may be done about infinite success sets and the like in the next section.

4.5 Answer Substitutions and Induction

We have given an incremental algorithm for the relative complement problem tailored to the production of answer substitutions for a constructive form of NAF. This process allows us to consider existentially quantified negated atoms in the same computational fashion as existentially quantified non-negated atoms in that both have the existential property, i.e. that a proof of $\exists x G$ will always yield a term t such that $G[t/x]$ is true. As mentioned above, it is possible that the relative complement may not be finitely representable by terms alone. This incompleteness will remain even with some form of constraint system; an explicit answer cannot always be given, but whenever it can, this process will find one. Also, when no

complete explicit representation of all the answers is possible, we may produce any finite subset of it, by enumerating all answers of no more than a given depth.

There is another incompleteness in that the set of positive answers may be infinite, and so we may produce no negative answers at all, but just wait for the termination of an infinite process. However, the successive approximations contain information which may be used to produce exact answers under certain circumstances.

For example, consider the program for the even predicate given above. The goal $\exists x\, \mathrm{even}(x)$ will generate the success set

$$\{\mathrm{even}(0),\ \mathrm{even}(s^2(0)),\ \mathrm{even}(s^4(0)),\ \ldots\}$$

The successive values of Approx are

$$\begin{aligned}
&\mathrm{even}(x)/\mathrm{even}(0) = \{\mathrm{even}(s(x))\}\\
&\mathrm{even}(s(x))/\mathrm{even}(s^2(0)) = \{\mathrm{even}(s(0)), \mathrm{even}(s^3(x))\}\\
&\{\mathrm{even}(s(0)), \mathrm{even}(s^3(x))\}/\mathrm{even}(s^4(0)) = \{\mathrm{even}(s(0)), \mathrm{even}(s^3(0)), \mathrm{even}(s^5(x))\}
\end{aligned}$$

and so forth. It is obvious that $\neg\mathrm{even}(s(0))$ and $\neg\mathrm{even}(s^3(0))$ are correct answers for the query $\exists x\, \neg\mathrm{even}(x)$, and so, provided that we can somehow ensure that the approximate answers can never succeed, some parts of the approximation may be made exact. In this way we may produce some correct answer substitutions without waiting for the entire success set to be enumerated.

It is instructive to examine which programs produce an infinite set of (distinct) answers. The answer seems to be that such programs use inductive definitions. For example, consider the two programs P_1 and P_2 below and the goal $\exists x\, p(x)$.

P_1	P_2
$p(a)$	$p(a)$
$\forall x\, p(x) \supset p(x)$	$\forall x\, p(x) \supset p(f(x))$

In P_1, we initially match against the fact to get the answer $p(a)$. Next the rule is used to produce the same subgoal, and so we produce the answer $p(a)$ infinitely

often. In P_2, we again initially produce the answer $p(a)$, and then match $p(x)$ against the second clause to produce the subgoal $\exists x' p(x')$, which gives answer $p(a)$, and so a second answer is $p(f(a))$.

Now for an infinite number of answers (distinct or otherwise) to exist, there must be an infinite number of **O**-proofs of the goal. We may think of these **O**-proofs as a tree (in the manner of the SLD-tree [61]) in which there are an infinite number of finite branches. There may be infinite branches interspersed between the finite ones, and so different ways of searching the tree may produce different results. Hence, we assume the use of a fair search strategy, so that any of the infinite number of answers may be eventually discovered. For this to occur, the search strategy must include a fair clause selection strategy, i.e. one which ensures that each matching clause is eventually reached by the proof search process. A particularly suitable one is given by the "fact first" rule: the facts, or unit clauses, are to be used before any other in the derivation. Once all facts are exhausted, we may use rules. In this way we may generate the set of answers in a breadth-first manner, thus giving us a rough measure of the progress of the search for answers.

The idea behind this is to ensure that at every step in the generation of answers, the depth of the answers will increase. For example, let P be the program

$$\text{even}(0)$$
$$\forall x\ \text{even}(x) \supset \text{even}(s^2(x))$$

For the goal $\exists y\, \text{even}(y)$, we first get the answer even(0), note the substitution $y \leftarrow s^2(x)$ and then the next goal is $\exists x\, \text{even}(x)$. Assuming we match against the unit clause first, we then get another answer $\text{even}(s^2(0))$, and then using the rule generate another goal, and so forth.

If we now turn our attention to the program itself, it is clear that there are only a finite number of clauses in the program and that each (distinct) answer is an instance of the head of some clause. Hence by the pigeonhole principle, there must be a clause head which is used an infinite number of times in the derivation process, and so there must be a predicate whose definition depends on itself, either

directly or indirectly. In this way the only class of programs which can possibly produce an infinite number of answers are those which contain a self-dependent predicate.

We may define the class of inductive programs as follows:

Definition 4.5.1 *An atom* $p(t_1, \ldots t_n)$ *is* inductively defined in P *if*

1. $p(t_1, \ldots t_n)$ *is not P-self-dependent*

2. $p(t_1, \ldots t_n)$ *is either a unit clause or P-dependent on an atom* $q(s_1, \ldots s_m)$ *which is inductively defined*

The definition of a predicate p in a program P is inductive *if every term* $p(t_1, \ldots t_n)$ *is inductively defined.*

A program P is inductive if it contains only inductive predicate definitions.

The first condition ensures that the inductive definition is not cyclic, and the second ensures that the induction must "bottom out" somewhere. For example, the even predicate as defined above is inductive.

Clearly, not all programs are inductive. However, we feel that a number of useful programs are inductive, and that many programmers write programs in this inductive style. Many list processing predicates are written in the form of one clause, often just a unit clause, for the empty list case, and another for the case of an element with a list appended to it. For example, the standard definition of append below is inductive.

$$\forall x \text{ append}([], x, x)$$
$$\forall x \forall y \forall z \forall w \text{ append}(y, z, w) \supset \text{append}(x.y, z, x.w)$$

There are many similar predicates involving lists, generally of the following form, which may not be strictly inductive, but are similar in spirit:

base_case(Terms) $\supset$ process_list([], Terms)

process_element(x,Terms) $\wedge$ process_list(y, Terms) $\supset$ process_list(x.y, Terms)

where Terms is a list of terms and both base_case and process_element are independent of process_list.

There are many programs written by a similar process of "signature exhaustion", i.e. writing a clause for each symbol in the signature, which will typically depend on a subterm, and so make the definition inductive.

Now for such programs we may devise some measure of inductive depth, so that at each stage in the calculation of an answer substitution, we are guaranteed to increase the depth of the answer produced in a measurable way. Thus we can produce a lower bound on the depth of the answers in terms of the number of times we iterate through a particular clause, and so if we come across an atom A in Approx whose depth is less than this lower bound, we know that we can never find that A succeeds, and so we know that A is in fact an exact answer.

For example, consider the even program above and the goal $\exists x\, \text{even}(x)$. We note that at each iteration through the second clause, the depth of the answer will increase by 2, and so after one such iteration, we get that Approx is $\{\text{even}(s(0)), \text{even}(s^3(x))\}$, and so any further answers must have depth greater than than of $\text{even}(s(0))$. Hence $\text{even}(s(0))$ is in fact an exact answer.

An interesting observation that may be made at this point is that many of the programs exhibited here were discussed in a different context in section 2.4, i.e. in terms of universally quantified goals. There is clearly a duality between $\forall x\, p(x)$ and $\exists x\, \neg p(x)$ in that both cannot be true simultaneously. Any algorithm used to establish the truth or falsity of $\forall x\, p(x)$ may then be used to establish the truth or falsity of $\exists x\, \neg p(x)$, particularly if the algorithm constructs counterexamples, so that if $\forall x\, p(x)$ fails, then the algorithm finds a term t such that $p(t)$ fails, i.e. $\neg p(t)$ succeeds. Such an algorithm will then be untroubled by inductive programs, as the search to find a term t such that $p(t)$ fails will not loop needlessly. Thus an algorithm for computation of universally quantified goals which demonstrates

failure by finding counterexamples may be used to find answer substitutions for negated goals, making such an algorithm doubly useful.

Chapter 5

Semantics and Model Theory

In this chapter we give a Kripke-like model for programs which may contain negations in the bodies of clauses. This is inspired by the work of Miller on the semantics of first-order hereditary Harrop formulae [77]. We concentrate on NAF, although some other forms of negation may be incorporated. No restriction on the class of programs is needed in this approach; our method allows for programs which are not locally stratified [94]. This necessitates a slight departure from the standard methods, but the important properties of the construction still hold.

5.1 A Kripke-like Model

In [77] it was shown how a Kripke-like model may be constructed for D_{mod} formulae. This uses techniques inspired by the possible worlds approach of Kripke [107]. This construction was then shown to precisely model the computational behaviour of D_{mod} and G_{mod} formulae, just as the previous construction of Kowalski and van Emden [25] did for Horn clauses. We now look at how to extend the construction of [77] to cater for the inclusion of negation, and the inclusion of our notion of universally quantified goals. We will first extend the model theory to cope with negated atoms as goals, and afterwards we shall consider some aspects of a further extension to include negated atoms as the heads of clauses.

Before plunging into the details of our extension, we review some details about the original model theory, and so for the time being we consider D_{mod} programs and G_{mod} goals, which are given as follows:

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid D \supset G$$

Note the absence of universally quantified goals in this class of formulae.

The aim of our investigation is to construct a model of the program which precisely matches up with the operational behaviour of the program. In this way we expect $P \vdash_o G$ to be equivalent to the statement that G is true in the model associated with the program P. The model theory of [77] uses a consequence relation $\Vdash$ similar to the relation $\models$ defined over Kripke models of first-order intuitionistic logic. The worlds in this model are identified with programs. If we let $\mathcal{U}$ be the set of all closed terms, $\mathcal{H}$ be the set of all closed atomic formulae and $\mathcal{P}$ be the set of all programs, then an *interpretation* is defined as an function $I : \mathcal{P} \rightarrow \text{powerset}(\mathcal{H})$ such that $\forall P_1, P_2 \in \mathcal{P}$ with $P_1 \subseteq P_2, I(P_1) \subseteq I(P_2)$. Thus, interpretations are "internally monotonic". Given this notion the consequence relation $\Vdash$ was defined in [77] as follows:

- $I, P \Vdash A$ iff $A \in I(P)$
- $I, P \Vdash G_1 \vee G_2$ iff $I, P \Vdash G_1$ or $I, P \Vdash G_2$
- $I, P \Vdash G_1 \wedge G_2$ iff $I, P \Vdash G_1$ and $I, P \Vdash G_2$
- $I, P \Vdash \exists x G$ iff $I, P \Vdash G[t/x]$ for some $t \in \mathcal{U}$
- $I, P \Vdash D \supset G$ iff $I, P \cup \{D\} \Vdash G$

Next we define the operators $\sqcap$ and $\sqcup$ for interpretations.

Definition 5.1.1 *Let I_1 and I_2 be interpretations. Then*

$$I_1 \sqsubseteq I_2 \text{ iff } \forall P \in \mathcal{P},\ I_1(P) \subseteq I_2(w)$$
$$(I_1 \sqcup I_2)(P) = I_1(P) \cup I_2(P)$$
$$(I_1 \sqcap I_2)(P) = I_1(P) \cap I_2(P)$$

We may think of this model theory as a large collection of models indexed by programs, so that $I, P \models\!\!\!\models G$ iff G is true in the model located in I at program P. A least fixed point method is given in [77] from which we get a single interpretation J such that $P \vdash_o G$ iff $J, P \models\!\!\!\models G$. Thus for any program we can find an interpretation in this collection which precisely describes the program's behaviour.

This construction is carried out by an operator on interpretations named T in the spirit of [25], and defined as follows:

$$T(I)(P) = \{A \mid A \in [P] \text{ or there is a closed clause } G \supset A \in [P] \text{ such that } I, P \models\!\!\!\models G\}$$

This operator is shown to be continuous, and so the least fixed point is

$$T^{\omega}(I_{\perp}) = \bigsqcup_{i=1}^{\infty} T^i(I_{\perp})$$

where $I_{\perp}$ is the null interpretation, i.e. $I_{\perp}(P) = \emptyset$ for any P. It is then shown how $P \vdash_o G$ iff $T^{\omega}(I_{\perp}), P \models\!\!\!\models G$, so that $T^{\omega}(I_{\perp})(P)$ may be thought of as a model for the program P.

It is interesting to examine the difference between the model constructed and the standard Kripke model. The definition in [107] is reproduced below.

Definition 5.1.2 *A* Kripke model *is a quadruple $K = \langle \mathcal{W}, \leq, D, \models \rangle$ such that $\mathcal{W}$ is partially ordered by $\leq$, D is a non-decreasing function mapping elements of $\mathcal{W}$ to non-empty sets, and $\models$ is a binary relation on worlds and formulae defined via:*

- *For $t_1 \ldots t_n \in D(w)$, if $w \models p(t_1, \ldots t_n)$ then $\forall w' \geq w$, $w' \models p(t_1, \ldots t_n)$*
- *$w \models \phi \wedge \psi$ iff $w \models \phi$ and $w \models \psi$*

- $w \models \phi \vee \psi$ *iff* $w \models \phi$ *or* $w \models \psi$
- $w \models \exists x\phi$ *iff* $w \models \phi[t/x]$ *for some* $t \in D(w)$
- $w \models \forall x\phi$ *iff* $\forall w' \geq w$, $w' \models \phi[t/x]$ *for all* $t \in D(w')$
- $w \models \phi \supset \psi$ *iff* $\forall w' \geq w$, *if* $w' \models \phi$ *then* $w' \models \psi$
- $w \models \neg\phi$ *iff* $\forall w' \geq w$, $w' \not\models \phi$

The partial order $\leq$ may be thought of as an "information" ordering; $w_1 \leq w_2$ is interpreted as stating that w_2 has no less information than w_1. We often refer to $\leq$ as the *access* relation, or the *reachability* relation between worlds. The function D may be thought of as determining the objects of interest for a given world. As knowledge is increased, and hence we progress to worlds which are "higher" in the partial order, we may construct new objects of interest, and so there may be objects in w_2 which do not exist in w_1 where $w_1 \leq w_2$. Hence the rule for $\forall$ in the definition of $\models$ must take this possibility into account, and so we must show that for all worlds $w' \geq w$, $w' \models \phi[t/x]$ for all $t \in D(w')$ rather than just $w \models \phi[t/x]$ for all $t \in D(w)$. A similar remark applies to the cases for $\supset$ and $\neg$.

We may interpret the statement $w \models \phi$ as ϕ is true at (or in) world w. We write $\forall w \in \mathcal{W}$, $w \models \phi$ as $\models \phi$. It is well known that intuitionistic provability is sound and complete with respect to Kripke models. More on Kripke models may be found in [21,107].

In our case, it is easy to see that a natural choice for $\mathcal{W}$ and $\leq$ is that the worlds are derivation states with the reachability relationship between worlds being set inclusion. We will consider two worlds (i.e. two derivation states) equal iff $(w_1) = (w_2)$. An obvious difference between the two consequence relations is that $\models\!\!\models$ is a relation between interpretations, worlds and formulae, whereas $\models$ is a binary relation between worlds and formulae. This may be thought of as another specialisation in our case, as the definition of a Kripke model does not specify how the atomic formulae which are true at a given world are to be determined, whereas in our case we will always do so by an interpretation. As the formulae

true at a given world in the Kripke-like model will clearly depend on the chosen interpretation, it seems natural to include the interpretation as above, so that $I, w \models\!\!\!\models G$ may be interpretated as stating that under the interpretation I, G is true at world w. In a similar way, we will sometimes write $I, w \models G$, to mean that if the atomic formulae true at w are specified by the interpretation I, then $w \models G$. Thus we may interpret the Kripke-like model directly as a Kripke model, using the same worlds structure and reachability relation as in the Kripke-like case, so that the only difference between the two is the difference between the the consequence relations $\models$ and $\models\!\!\!\models$.

One such difference is given by the rules for implication. Using the above syntax and convention, the rule for implication in a Kripke model reads

$$I, P \models D \supset G \text{ iff for every world } w' \supseteq P, \text{ we have } I, w' \models D \Rightarrow I, w' \models G$$

As the relation $\models\!\!\!\models$ is only defined when the antecedent is a D formula and the consequent a G formula, this condition cannot be used directly for $\models\!\!\!\models$. A more serious objection is that the T operator resulting from the $\models$ definition (i.e. the definition of $T(I)(P)$ with $I, P \models\!\!\!\models G$ replaced by $I, P \models G$) is not monotonic, and so the usual fixed point method will not work. This may be seen from the following example, due to Dale Miller [73]:

Consider the two programs P_1 and P_2 below.

P_1	P_2
$(r \supset p) \supset q$	$(r \supset p) \supset q$
	r

Note that $P_1 \subseteq P_2$. Now as $I_\perp(P_1) = I_\perp(P_2) = \emptyset$ and there are no atoms in P_1, we get

$$T(I_\perp)(P_1) = \{q \mid I_\perp, P_1 \models r \supset p\}$$

Now $I_\perp, P_1 \models r \supset p$ iff $\forall w' \supseteq P_1$ we have $I_\perp, w' \models r \Rightarrow I_\perp, w' \models p$. This is vacuously true, as there is no world w' such that $I_\perp, w' \models r$. Hence, $T(I_\perp)(P_1) = \{q\}$. In a similar way we get that $T(I_\perp)(P_2) = \{q, r\}$.

However, it is obvious that $T(I_\perp), P_1 \not\models r \supset p$, as we know that $P_2 \supseteq P_1$, and $T(I_\perp), P_2 \models r$ but $T(I_\perp), P_2 \not\models p$. Hence, we get that $T^2(I_\perp)(P_1) = \emptyset$, and by a similar argument that $T^2(I_\perp)(P_2) = \{r\}$. Thus the version of the T operator defined by $\models$ is not monotonic, and so we need to use a different relation.

However, it can be shown that there is a Kripke model such that $\models G$ iff $T^\omega(I_\perp), P \Vdash G$, so that the two relations $\models$ and $\Vdash$ coincide "at the fixed point" [73]. The worlds in this model are the worlds w of the Kripke-like model such that $w \geq P$, so that P becomes the bottom world. This may be seen to be intuitively reasonable by the fact that the lack of monotonicity of the $\models$ version of the T operator is due to its behaviour on $I_\perp$, but as the iterations increase, this idiosyncracy disappears.

Note that a notion of universality similar to that of the $\models$ definition is captured in the $\Vdash$ definition by the fact that the reachability relation between worlds is just set inclusion. As $w \subseteq w \cup \{D\}$ for any D, we have that $w \cup \{D\}$ is always reachable from w, i.e. $w \leq w \cup \{D\}$, and that any $w' \supseteq w$ that contains D also contains $w \cup \{D\}$. Hence, the $\Vdash$ version may be read as "for any world containing w in which D is *assumed*", rather than as "for any world containing w in which D is *provable*", as is the case for the $\models$ version. Thus $\Vdash$ circumvents the "all worlds reachable from w" condition, does not involve definite formulae as consequents and leads to a monotonic operator, and hence we may derive our desired interpretation by the calculation of a least fixed point.

In the light of the equivalence result, the $\Vdash$ relation may be thought of as a computational version of $\models$ for this class of formulae, in that $\Vdash$ behaves differently and perhaps more intuitively on the internal construction, but leads to the same final result.

We may gain a geometric insight into the structure of the collection of models by viewing it as an inverted cone, with the empty program at the bottom. The cone extends infinitely upwards in an ever-widening way, as every possible extension (of which there are infinitely many) of a program P is reachable from P. A representation of this idea is given in Figure 5–1. There are $\aleph_1$ worlds here, as any program may be extended in an infinite number of ways, as may each extension.

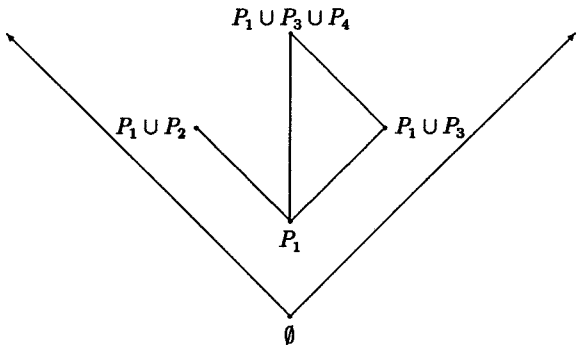

Figure 5–1: Inverted cone of models

The lines between worlds in Figure 5–1 represent reachability, so that P_1 may reach $P_1 \cup P_2$, $P_1 \cup P_3$ or $P_1 \cup P_3 \cup P_4$, but $P_1 \cup P_2$ may not reach $P_1 \cup P_3 \cup P_4$.

5.2 Worlds and Accessibility

We wish to extend the programs and goals covered by this model theory to D_{HHF-} and G_{HHF-} formulae, i.e. where D and G formulae are defined as

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid \exists x G \mid \forall x G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G$$

The definition of $\Vdash$ for a universally quantified formula may be stated as follows for a goal G:

$$I, P \Vdash \forall x G \text{ iff } \forall w' \geq P \text{ we have } I, w' \Vdash G[t/x] \text{ for each } t \in \mathcal{U}$$

This is just the condition for intuitionistic universal quantification given in [107] translated into our syntax and with $\Vdash$ substituted for $\models$. A feature of

the Kripke-like model which is not a general feature of Kripke models is that the function D does not change from world to world, but is constant, in that for all $w \in \mathcal{W}$, $D(w) = \mathcal{U}$. This is a definitive property of Beth models [21], which use a notion of derivability which differs from both of those discussed above. Hence, we prefer to think of the Kripke-like model as one in which there are no new objects constructed in the process of increasing our knowledge. It is this possibility that necessitates the side conditions on the definitions of truth in a Kripke model for the connectives $\forall, \supset$ and $\neg$ which state that the formula must not only be true in the current world but also true in every future world, as these are the ones which will be affected if new objects are constructed at a later stage. The worlds "below" the world in which the new object is first constructed can have no knowledge about formulae involving the new object, and so in order to preserve the property that whatever is true at a given world will always be true in all future worlds, we need to restrict the formulae which are considered true in the current world.

In our case, we saw above that due to the fact that the reachability relation is just set inclusion, we may circumvent this side condition for $\supset$, and analogously, since we will never construct new objects, it should be possible to do the same for $\forall$. Now as we think in terms of a Herbrand universe, which is fixed before the program is written, we may simply neglect the side condition for $\forall$, so that we may replace the above definition by the following:

$$I, P \models \forall x G \text{ iff } I, P \models G[t/x] \text{ for all } t \in \mathcal{U}$$

We do not need to consider all worlds $w' \geq P$ here as we know that no new objects can be constructed, and so the Herbrand universe $\mathcal{U}$ is never increased. Now from the definition of an interpretation it is easy to see that if $I, P \models G$ and $w' \geq P$ then $I, w' \models G$, so if $I, P \models G[t/x]$ then $I, w' \models G[t/x]$, and so $I, P \models G[t/x]$ for all $t \in \mathcal{U}$ implies that $I, w' \models G[t/x]$ for all $t \in \mathcal{U}$. Clearly the reverse holds, i.e. that if $I, w' \models G[t/x]$ for all $t \in \mathcal{U}$ and for all $w' \geq P$, then $I, P \models G[t/x]$ for all $t \in \mathcal{U}$.

A relevant observation at this point is that there are intermediate logics (i.e. strictly between intuitionistic logic and classical logic) which have model-theoretic

properties very similar to that of the Kripke-like model. The best known example is called the *logic of constant domains*, whose models are characterised by Kripke models in which the domain is constant, i.e. the mapping D is the same for all worlds. Clearly the Kripke-like model is one such model, as $\mathcal{U}$ is fixed for all worlds. However, the logic of constant domains is not quite right in our case, as we are interested in one particular domain, rather than a class of domains. Nevertheless, the natural place to study the semantics seems to be an intermediate logic, rather than intuitionistic logic. This point is taken up in section 6.5.

Note that we also want the definition of $\models$ to behave similarly to $\vdash_s$, and so we will also use representations in the relevant formal definition. However the above remarks will still apply.

The reconciliation between the NAF rule and the condition for truth of a negated atom in a Kripke model is more problematic. The desired definition of $\models$ for negated formulae, using the analogy of Kripke models, would be

$$I, P \models \neg A \text{ iff } \forall w' \geq P \text{ we have } I, w' \not\models A.$$

If the relation $\leq$ between worlds is set inclusion ($\subseteq$), then there is no w and A such that $T^{\omega}(I_{\perp}), w \models \neg A$, as the world $w \cup \{A\}$ is always reachable from w, and we know from the properties of interpretations that $A \in T^{\omega}(I_{\perp})(P \cup \{A\})$, i.e. $T^{\omega}(I_{\perp}), P \cup \{A\} \models A$ for any A. Thus in order to incorporate negated atoms into this model, we need to restrict the reachability relation between worlds, so that there are less worlds "above" a given world w. We may think of this as a form of pruning of the inverted cone.

It may be informative to consider what sort of pruning occurs if we apply the CWA to a program P. The CWA may be understood as identifying $P \vdash_s \neg A$ with $P \vdash_f A$. So $P \vdash_f A$ means that $P \cup \{A\}$ is not reachable from P, and so the only worlds reachable from P are those of the form $P \cup \{A\}$ where $P \vdash_s A$, in which case $P \cup \{A\} \vdash_s G \Leftrightarrow P \vdash_s G$. Hence, we understand the CWA as saying that there are no significant worlds reachable from P, so that P is maximal in the sense that the only worlds reachable from P are those in which the information added to P is a consequence of P.

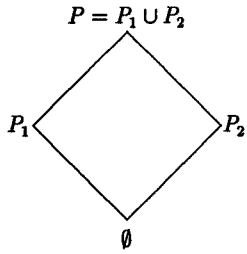

Figure 5–2: Inverted cone + CWA = diamond

The effect of this pruning of the inverted cone above P is shown in Figure 5–2.

Note how the maximality of P makes the cone finite, as all elements of the cone must be subsets of P. Thus this drastic pruning may be interpreted as stating that all true formulae follow from P and P alone.

This form of pruning is too extreme for our purposes. What we desire is some form of selective pruning, whereby we may make some worlds unreachable, but allow access to others. Thus we desire some form of inverted cone with "holes" appearing here and there, so that the geometric interpretation is in the form of Figure 5–3.

The shaded triangles are the parts pruned from the initial diagram of Figure 5–1. Here there are several worlds that P can reach, but some that P can not reach. The non-reachable ones are intended to include all extensions to the completely defined predicates of P. Thus P can only access "reasonable" worlds.

In order to find the correct form of pruning, let us examine the growth of programs in this context. One feature of the Kripke-like model is that it captures the growth of programs very nicely. The only programs above a given world w are those which extend w. Thus, this approach induces the view that the process of programming begins at the tip of the inverted cone and proceeds upwards to a point where the finished program is reached. At this point, we may consider

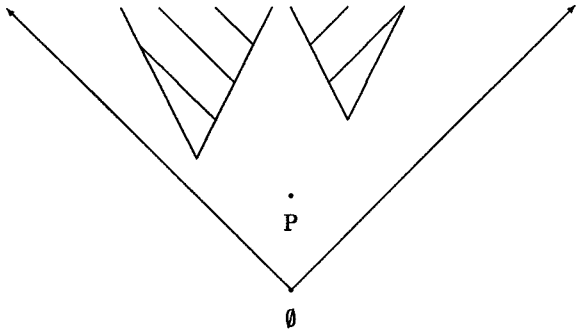

Figure 5–3: Inverted cone with holes

that the programmer has said "That is all that I know to be true", and then the machine makes deductions based on the information supplied. We may further contemplate that there are some predicates for which the programmer knows his or her knowledge to be complete, and so there is more information known to the programmer than he is able to express in the program, i.e. the negative parts of the information known about the predicates of the program. Such predicates we may consider completely defined, i.e. that no other programming process will lead to more information. This is certainly not true in the model theory of [77], as if w is a program containing clauses for the predicate p, then the program $w \cup \{p(t)\}$ is reachable from w for any t, and so it is always possible to extend the definition of p given in w. Thus no predicate can be completely defined. Hence, we wish the reachability relation between worlds to reflect the following:

$w \cup \{D\}$ is reachable from w iff D does not contain any more information than w about the completely defined predicates of w.

The notion of completely and incompletely defined predicates is used to determine which worlds are reachable and which are not. If P contains an incompletely

defined predicate p, then we wish any world which extends the definition of p to be reachable from P. On the other hand, any world which extends the definition of a completely defined predicate of P should not be reachable from P.

For example, let P_1 be the program which defines the append predicate in the usual way, given below:

$$\forall x \text{ append}([], x, x)$$
$$\forall x \forall y \forall z \forall w \text{ append}(y, z, w) \supset \text{append}(x.y, z, x.w)$$

We wish that no program which extends this definition of append be reachable from P_1. On the other hand, the program P_2 which defines all known carcinogens we would wish to be able to extend in any fashion, as we know that our knowledge is not complete, and so we wish to be able to extend it in any way possible. Thus the world $P_1 \wedge \text{append}([], [1, 2], [3])$ is not reachable from P_1, as $\text{append}([], [1, 2], [3])$ is false, whereas carcinogen(chocolate) is unknown, and so $P_2 \wedge$carcinogen(chocolate) is reachable from P_2, as we may find that chocolate is a carcinogen at some time in the future.

Now in [77] worlds and sets of definite clauses tended to be interchangeable. As noted above, in our case programs need to be more than just definite clauses, as we need to know which predicates are completely defined. Thus whilst in our case the worlds will again be just the same as programs, we have a more sophisticated notion of worlds as we have a more sophisticated notion of programs.

As discussed above, the partial order on these worlds will need to be something more restrictive than set inclusion. The natural partial order between worlds is given below.

Definition 5.2.1 *Let P_1 and P_2 be sets of definite formulae, and $N_i \subseteq$ names(P_i) $\times$ names(P_i), i = 1,2. Then $\langle P_1, N_1 \rangle \leq \langle P_2, N_2 \rangle$ iff*

1. $P_1 \subseteq P_2$

2. $\text{ass}(N_1) \subseteq \text{ass}(N_2)$

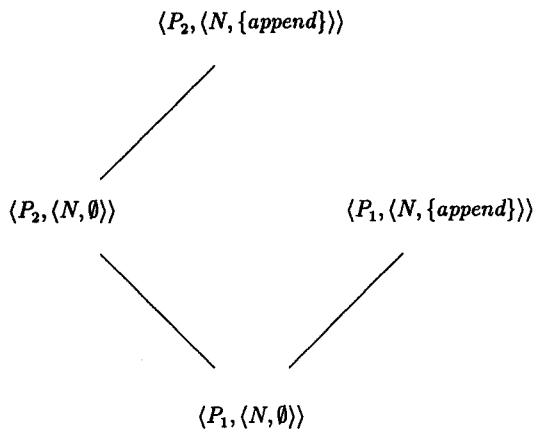

Figure 5–4: The append program and its possible extensions

3. $\mathrm{den}(N_1) \subseteq \mathrm{den}(N_2)$

4. *for each* $C \in P_2 \backslash P_1$, *name(head(C))* $\in \mathrm{ass}(N_1)$

The fourth condition in the definition of $\leq$ is the interesting one, as it ensures that no completely defined predicate of P_1 is extended by P_2. Recall that $\mathrm{ass}(N_1) \cap \mathrm{den}(N_1) = \emptyset$, and so the only permitted extensions are those which extend the definitions of predicates which are known to be incompletely defined.

For example, let P_1 be the two clauses for append given above, and let $P_2 = P_1 \cup \{\mathrm{append}(\mathrm{nil}, \mathrm{nil}, [1,2])\}$. The partial order for $\langle P_i, \langle N, \emptyset\rangle\rangle$ and $\langle P_i, \langle N, \{\mathrm{append}\}\rangle\rangle, i = 1, 2$ is given in Figure 5–4.

Note that $\langle P_1, \langle N, \emptyset\rangle\rangle \leq \langle P_2, \langle N, \{\mathrm{append}\}\rangle\rangle$ but that $\langle P_1, \langle N, \{\mathrm{append}\}\rangle\rangle \not\leq \langle P_2, \langle N, \{\mathrm{append}\}\rangle\rangle$. Thus our partial order restricts the reachable worlds to those which do not extend the completely defined predicates, and in which incompletely defined predicates remain incompletely defined.

Given this partial order, we know that all worlds above a given world w are those which consistently extend w, and so we know that if $\neg A$ is true at world w, then $\neg A$ will be true for all worlds w' above w. It is this property that allows us to extend the $\models\!\!\!\models$ relation, as described in the next section.

5.3 Extending the Framework

We saw above that an interpretation was defined as any function I mapping sets of definite formulae to sets of closed atoms such that whenever $P_1 \subseteq P_2$ then $I(P_1) \subseteq I(P_2)$. Thus we may think of an interpretation as providing an indication of which atoms are true for each definite formula. However, in order to serve as an indication of which atoms are false, we will need more information. As argued in chapter 2, the computational behaviour of programs fits naturally into a constructive setting. As it is possible for neither $P \vdash_s A$ nor $P \vdash_s \neg A$ to be true, it seems natural to allow an interpretation I to be such that neither $I, P \models\!\!\!\models A$ nor $I, P \models\!\!\!\models \neg A$, and so we wish for a more general notion of interpretation than that given in [77]. Due to the possible occurrence of free variables in programs and goals, we also need to consider maps which range over sets of atoms which are not necessarily ground. In this way our construction will resemble that of [26]. As mentioned in [26], it seems unreasonable for $p(x)$ to be true and $p(a)$ to be false. We may circumvent this difficulty by thinking of the ground instances of the atoms in the interpretation as the "real" items of interest, and the non-ground atoms as place-holders. This leads us to the following definition of an interpretation. Let $\mathcal{H}'$ be the set of all atomic formulae. Let $\mathcal{W}$ be the set of all derivation states.

Definition 5.3.1 *Let X be a set of atoms. We refer to the set of all instances of all elements of X as* $\mathrm{inst}(X)$. *Note that* $X \subseteq \mathrm{inst}(X)$.

We define $X_1 \prec X_2$ as $\mathrm{inst}(X_1) \subseteq \mathrm{inst}(X_2)$. *When X_1 is a singleton set $\{A\}$ we will often write $X_1 \prec X_2$ as $A \prec X_2$.*

Let A be an atom. We define A^+ to be the set of instances of A in which each variable in A is instantiated to an element of Σ^+.

Let P be a derivation state. An interpretation *is any function $I : \mathcal{W} \to \mathcal{H}' \times \mathcal{H}'$ satisfying the following conditions, where $I(P) = \langle S, F \rangle$:*

1. $\text{inst}(S) \cap \text{inst}(F) = \emptyset$

2. *for all worlds w_1, w_2 such that $w_1 \leq w_2$ where $I(w_1) = \langle S_1, F_1 \rangle$ and $I(w_2) = \langle S_2, F_2 \rangle$, we have $S_1 \prec S_2$ and $F_1 \prec F_2$*

Let $I(w) = \langle S, F \rangle$. We define $pos(I)(w) = S$ and $neg(I)(w) = F$. We define $I_\perp(w) = \langle \emptyset, \emptyset \rangle \forall w \in \mathcal{W}$.

We refer to $I(w)$ as an interpreted world.

We think of $\text{pos}(I)$ as specifying which atoms are true, and of $\text{neg}(I)$ as specifying which atoms are false. Thus, we may think of the definition of an interpretation given in [77] as the special case of our definition obtained when $\text{neg}(I)(w) = \emptyset$ for any world w. The first side condition ensures that no atom is specified as being both true and false, and so this condition ensures that interpretations are internally consistent. The second condition is a generalisation of the previous condition of internal monotonicity. This is justified by the perception that as programs increase, the knowledge contained in the program cannot decrease, and so no extension to a program is allowed to decrease either the set of atoms known to be true or the set of atoms known to be false. Thus we preserve the principle of monotonicity of information.

We use the relation $\prec$ merely as a shorthand; this is a device which allows us to handle the non-ground atoms more easily.

Note that we do not explicitly require that $\text{inst}(S \cup F) = \mathcal{H}'$.

This is possible, of course; indeed, the (global) CWA may be thought of as requiring this to be the case for any interpretation, i.e. that if $\text{inst}(\text{pos}(I)) = S$, then $\text{inst}(neg(I)) = \mathcal{H}' \backslash S$. However, there are many programs for which such an interpretation will be inappropriate. For example, consider the simple loop program

$$p(a) \supset p(a)$$

There are no successful goals, but the set of goals which fail does not include $p(a)$, and so if we wish for some form of interpretation which precisely matches up with computational behaviour, we cannot insist that $\text{inst}(S \cup F) = \mathcal{H}'$.

The partial order $\sqsubseteq$ on interpretations is extended in the obvious way, as is the operator $\sqcap$. The dual operator $\sqcup$ provides a slight difficulty as there is now no longer one maximal interpretation.

This means that the obvious definition of $\sqcup$ may not lead to an interpretation, as if $I_i(w) = \langle S_i, F_i \rangle$ where $i = 1, 2$, then if $\text{inst}(S_2) \cap \text{inst}(F_1) \neq \emptyset$ or $\text{inst}(S_1) \cap \text{inst}(F_2) \neq \emptyset$, the mapping $(I_1 \sqcup I_2)(w) = \langle S_1 \cup S_2, F_1 \cup F_2 \rangle$ is not an interpretation as $\text{inst}(\text{pos}(I_1 \sqcup I_2)(w)) \cap \text{inst}(\text{neg}(I_1 \sqcup I_2)(w)) \neq \emptyset$. However, we may consider two interpretations I_1 and I_2 with this property as mutually inconsistent, and so we never wish to consider the mapping $I_1 \sqcup I_2$ as an interpretation. This consideration motivates the definitions below.

Definition 5.3.2 *Let I_1, I_2 be interpretations.*

I_1 and I_2 are mutually consistent interpretations *if for all worlds w we have* $\text{inst}(\text{pos}(I_1)(w)) \cap \text{inst}(\text{neg}(I_2)(w)) = \emptyset$ *and* $\text{inst}(\text{neg}(I_1)(w)) \cap \text{inst}(\text{pos}(I_2)(w)) = \emptyset$. *Otherwise, I_1 and I_2 are* mutually inconsistent.

We define the relations $\leq$ and $\sqsubset$ and the operator $\sqcap$ as follows:

$$\begin{aligned}
&I_1(w) \leq I_2(w) \textit{ iff } \text{pos}(I_1)(w) \prec \text{pos}(I_2)(w) \textit{ and } \text{neg}(I_1)(w) \prec \text{neg}(I_2)(w) \\
&I_1 \sqsubseteq I_2 \textit{ iff } \forall w \in \mathcal{W} \textit{ we have } I_1(w) \leq I_2(w) \\
&(I_1 \sqcap I_2)(w) = \langle \text{inst}(\text{pos}(I_1)(w)) \cap \text{inst}(\text{pos}(I_2)(w)), \\
&\qquad\qquad \text{inst}(\text{neg}(I_1)(w)) \cap \text{inst}(\text{neg}(I_2)(w)) \rangle
\end{aligned}$$

If I_1 and I_2 are mutually consistent then we define the operator $\sqcup$ as follows:

$$(I_1 \sqcup I_2)(w) = \langle \text{pos}(I_1)(w) \cup \text{pos}(I_2)(w), \text{neg}(I_1)(w) \cup \text{neg}(I_2)(w) \rangle$$

In this case the interpretations do not form a lattice under the operations $\sqcup$ and $\sqcap$, as there are an infinite number of maximal interpretations. For any two such maximal interpretations I_1 and I_2 with $I_1(P) = \langle S_1, \mathcal{H}'\backslash S_1\rangle$ and $I_2(P) = \langle S_2, \mathcal{H}'\backslash S_2\rangle$ such that $S_1 \neq S_2$, then $I_1 \sqcup I_2$ is not an interpretation. However, it will be seen below that the formal results do not depend upon the interpretations forming a lattice, and so this will not be a problem.

There is a third cone structure too, apart from that of worlds and of interpretations. In this structure, each node is a pair of sets of atoms (i.e. each node is an interpreted world), with the partial order being componentwise set inclusion. It is this structure on which the construction process described below is carried out. As the relation $\models$ relates interpretations, worlds (i.e. derivation states) and goals, it may sometimes be helpful to think of the relation $\models$ as a relation between nodes in this third structure (i.e. pairs of sets of atoms) and goals.

Now we come to the generalisation of the relation $\models$ defined in [77]. As there is both positive and negative information explicitly given in an interpretation, it seems natural to define two relations $\models^+$ and $\models^-$ such that $\models^+$ is used for the positive information and $\models^-$ for the negative information. These are defined below.

Definition 5.3.3 *Let $\langle P, G\rangle$ be a D_{HHF-} derivation pair where $P = \langle D, N\rangle$ and I be an interpretation. Then*

- $I, P \models^+ A$ *iff* $A \prec \mathrm{pos}(I)(P)$
- $I, P \models^+ \neg A$ *iff* $A \prec \mathrm{neg}(I)(P)$
- $I, P \models^+ G_1 \vee G_2$ *iff* $I, P \models^+ G_1$ *or* $I, P \models^+ G_2$
- $I, P \models^+ G_1 \wedge G_2$ *iff* $I, P \models^+ G_1$ *and* $I, P \models^+ G_2$
- $I, P \models^+ \exists x G$ *iff* $I, P \models^+ G[t/x]$ *for some* $t \in \mathcal{U}$
- $I, P \models^+ \forall x G$ *iff* $\exists R \in \mathcal{R}(\mathcal{U})$ *such that* $I, P \models^+ G[t/x]$ *for all* $t \in R$ *where the variables in* R *do not appear free in* P *or* G

- $I, \langle D, N\rangle \models^{+} D' \supset G$ *iff* $I, \langle D \cup \{D'\}, N\rangle \models^{+} G$ *and* $\langle D \cup \{D'\}, N\rangle \geq \langle D, N\rangle$

- $I, P \models^{-} A$ *iff* $A \prec \mathrm{neg}(I)(P)$

- $I, P \models^{-} \neg A$ *iff* $A \prec \mathrm{pos}(I)(P)$

- $I, P \models^{-} G_1 \vee G_2$ *iff* $I, P \models^{-} G_1$ *and* $I, P \models^{-} G_2$

- $I, P \models^{-} G_1 \wedge G_2$ *iff* $I, P \models^{-} G_1$ *or* $I, P \models^{-} G_2$

- $I, P \models^{-} \exists x G$ *iff* $\exists R \in \mathcal{R}(\mathcal{U})$ *such that* $I, P \models^{-} G[t/x]$ *for all* $t \in R$ *where the variables in* R *do not appear free in* P *or* G

- $I, P \models^{-} \forall x G$ *iff* $I, P \models^{-} G[t/x]$ *for some* $t \in \mathcal{U}$

- $I, \langle D, N\rangle \models^{-} D' \supset G$ *iff* $I, \langle D \cup \{D'\}, N\rangle \models^{-} G$ *and* $\langle D \cup \{D'\}, N\rangle \geq \langle D, N\rangle$

It should be clear that these definitions are similar to those of [77], with the main differences being the cases for universal quantification, implication and negation. The motivation for the universal quantification case is clear from the discussion in Chapter 2 on the corresponding operational definition. In the case of [77] the side condition on implication is vacuously true, as $D \cup \{D'\} \supseteq D$. Here we explicitly require that $\langle D \cup \{D'\}, N\rangle$ be reachable from $\langle D, N\rangle$, i.e. that the new world is reachable from the first. This may be thought of as ensuring that the assumption makes sense. This restriction is not strictly necessary, in that there may be weaker restrictions that work. However, this is a safe choice, and in our opinion a natural one.

One interesting thing to note is that given two interpretations I_1 and I_2, we have that $I_1 \sqsubseteq I_2$ iff for all P and A we have $I_1, P \models^{+} A \Rightarrow I_2, P \models^{+} A$ and $I_1, P \models^{+} \neg A \Rightarrow I_2, P \models^{+} \neg A$. This property will be useful in some subsequent proofs, as it allows us to deduce that if $I_1, P \models^{+} G \Rightarrow I_2, P \models^{+} G$ for any P and G, then $I_1 \sqsubseteq I_2$.

Note that this definition of an interpretation may be used for the negation of incompletely defined predicates, in that if we know $\neg p(b)$, then we may represent this information in an interpretation I by ensuring that $p(b) \in \text{neg}(I)(w)$. Thus, Negation as Inconsistency [39] may be captured this way, and so our model theoretic framework may be used for more than one kind of negation. Naturally the construction process described below would need to be modified, but the notion of interpretation would need no extension. We take up this theme in section 5.6.

The following lemma establishes that interpretations respect the reachability relation between worlds.

Lemma 5.3.1 *Let P_1 and P_2 be D_{HHF-} derivation states, G be a G_{HHF-} goal formula, and let I be an interpretation. If $P_1 \leq P_2$ then*

1. $I, P_1 \models^+ G \Rightarrow I, P_2 \models^+ G$
2. $I, P_1 \models^- G \Rightarrow I, P_2 \models^- G$

Proof: We proceed by induction on the size of G. The base case occurs when G is a literal.

A: 1. $I, P_1 \models^+ A$ implies that $A \prec \text{pos}(I)(P_1)$, and as $\text{inst}(\text{pos}(I(P_1))) \subseteq \text{inst}(\text{pos}(I(P_2)))$, we have $I, P_2 \models^+ A$.

2. $I, P_1 \models^- A$ implies that $A \prec \text{neg}(I)(P_1)$, and as $\text{inst}(\text{neg}(I(P_1))) \subseteq \text{inst}(\text{neg}(I(P_2)))$, we have $I, P_2 \models^- A$.

$\neg A$: As $I, P \models^+ \neg A$ iff $I, P \models^- A$ and $I, P \models^+ A$ iff $I, P \models^- \neg A$, this case follows directly from the one above.

Hence we assume that the lemma is true for all goals of no more than a given size. There are five cases:

$G_1 \vee G_2$: 1. $I, P_1 \models^+ G_1 \vee G_2$ iff $I, P_1 \models^+ G_1$ or $I, P_1 \models^+ G_2$ and by the hypothesis this implies that $I, P_2 \models^+ G_1$ or $I, P_2 \models^+ G_2$, i.e. $I, P_2 \models^+ G_1 \vee G_2$.

2. $I, P_1 \models^- G_1 \vee G_2$ iff $I, P_1 \models^- G_1$ and $I, P_1 \models^- G_2$ and by the hypothesis this implies that $I, P_2 \models^- G_1$ and $I, P_2 \models^- G_2$, i.e. $I, P_2 \models^- G_1 \vee G_2$.

$G_1 \wedge G_2$: 1. $I, P_1 \models^+ G_1 \wedge G_2$ iff $I, P_1 \models^+ G_1$ and $I, P_1 \models^+ G_2$ and by the hypothesis this implies that $I, P_2 \models^+ G_1$ and $I, P_2 \models^+ G_2$, i.e. $I, P_2 \models^+ G_1 \wedge G_2$.

2. $I, P_1 \models^- G_1 \wedge G_2$ iff $I, P_1 \models^- G_1$ or $I, P_1 \models^- G_2$ and by the hypothesis this implies that $I, P_2 \models^- G_1$ or $I, P_2 \models^- G_2$, i.e. $I, P_2 \models^- G_1 \wedge G_2$.

$\exists x G$: 1. $I, P_1 \models^+ \exists x G$ iff $I, P_1 \models^+ G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis this implies that $I, P_2 \models^+ G[t/x]$ for some $t \in \mathcal{U}$, i.e. $I, P_2 \models^+ \exists x G$.

2. $I, P_1 \models^- \exists x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $I, P_1 \models^+ G[t/x]$ for all $t \in R$, and by the hypothesis this implies that $I, P_2 \models^+ G[t/x]$ for all $t \in R$, i.e. $I, P_2 \models^- \exists x G$.

$\forall x G$: 1. $I, P_1 \models^+ \forall x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $I, P_1 \models^+ G[t/x]$ for all $t \in R$, and by the hypothesis this implies that $I, P_2 \models^+ G[t/x]$ for all $t \in R$, i.e. $I, P_2 \models^+ \forall x G$.

2. $I, P_1 \models^- \forall x G$ iff $I, P_1 \models^+ G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis this implies that $I, P_2 \models^+ G[t/x]$ for some $t \in \mathcal{U}$, i.e. $I, P_2 \models^- \forall x G$.

$D' \supset G$: 1. $I, \langle D_1, N_1 \rangle \models^+ D' \supset G$ iff $I, \langle D_1 \cup \{D'\}, N_1 \rangle \models^+ G$ and $\langle D_1 \cup \{D'\}, N_1 \rangle \geq \langle D_1, N_1 \rangle$, and so names(heads($D'$)) $\subseteq$ ass(N_1). Now as $\langle D_1, N_1 \rangle \leq \langle D_2, N_2 \rangle$, this implies that names(heads(D')) $\subseteq$ ass(N_2), and hence names(heads(D')) $\cap$ den(N_2) $= \emptyset$, i.e. $\langle D_1 \cup \{D'\}, N_1 \rangle \leq \langle D_2 \cup \{D'\}, N_2 \rangle$. Hence by the hypothesis we have $I, \langle D_2 \cup \{D'\}, N_2 \rangle \models^+ G$, and names(heads($D'$)) $\subseteq$ ass(N_2), and so $I, \langle D_2, N_2 \rangle \models^+ D' \supset G$.

2. $I, \langle D_1, N_1 \rangle \models^- D' \supset G$ iff $I, \langle D_1 \cup \{D'\}, N_1 \rangle \models^- G$ and $\langle D_1 \cup \{D'\}, N_1 \rangle \geq \langle D_1, N_1 \rangle$, and so names(heads($D'$)) $\subseteq$ ass(N_1). Now as $\langle D_1, N_1 \rangle \leq \langle D_2, N_2 \rangle$, this implies that names(heads(D')) $\subseteq$

ass(N_2), and hence names(heads(D')) $\cap$ den(N_2) $= \emptyset$, i.e. $\langle D_1 \cup \{D'\}, N_1\rangle \leq \langle D_2 \cup \{D'\}, N_2\rangle$. Hence by the hypothesis we have $I, \langle D_2 \cup \{D'\}, N_2\rangle \models^- G$, and names(heads($D'$)) $\subseteq$ ass(N_2), and so $I, \langle D_2, N_2\rangle \models^- D' \supset G$.

□

Note that this result depends critically on the fact that if $P_1 \leq P_2$, then ass(N_1) $\subseteq$ ass(N_2). It is difficult to see how such a result could hold in the absence of this property.

It is easy to prove a lemma analogous to lemma 2 of [77].

Lemma 5.3.2 *Let $\langle P, G\rangle$ be a D_{HHF-} derivation pair where $P = \langle D, N\rangle$ and I_1 and I_2 be two interpretations. Then*

1. $I_1 \sqsubseteq I_2$ *iff* $\forall P\, \forall G\; I_1, P \models^+ G \Rightarrow I_2, P \models^+ G$.

2. $I_1 \sqsubseteq I_2$ *iff* $\forall P\, \forall G\; I_1, P \models^- G \Rightarrow I_2, P \models^- G$.

Proof: For the $\Leftarrow$ direction of 1, consider the cases $G = A$ and $G = \neg A$. $A \prec$ pos(I_1)(P) iff $I_1, P \models^+ A$ which implies that $I_2, P \models^+ A$ which is equivalent to $A \prec$ pos(I_2)(P). Similarly, $A \prec$ neg(I_1)(P) iff $I_1, P \models^+ \neg A$ which implies that $I_2, P \models^+ \neg A$ which is equivalent to $A \prec$ neg(I_2)(P). A similar argument establishes 2.

For the other direction, we proceed by induction on the structure of G. For the base case, if G is an atom A and $I_1, P \models^+ A$, then $A \prec$ pos(I_1)(P), and so $A \prec$ pos(I_2)(P), as $I_1(P) \leq I_2(P)$, and so $I_2, P \models^+ A$. If G is $\neg A$ for some A and $I_1, P \models^+ \neg A$, then $A \prec$ neg(I_1)(P), so we have $A \prec$ neg(I_2)(P), as $I_1(P) \leq I_2(P)$ and so $I_2, P \models^+ \neg A$.

Similarly, if $I_1, P \models^- A$ then $A \prec$ neg(I_1)(P) which implies that $A \prec$ neg(I_2)(P), as $I_1(P) \leq I_2(P)$, and so $I_2, P \models^- A$. If G is $\neg A$ for some A and $I_1, P \models^- \neg A$, then $A \prec$ pos(I_1)(P), so we have $A \prec$ pos(I_2)(P), as $I_1(P) \leq I_2(P)$ and so $I_2, P \models^- \neg A$.

Hence the inductive hypothesis is that the lemma holds for all goals of no more than a given size. There are five cases:

$G_1 \vee G_2$:

1. As $I_1, P \models^+ G_1$ or $I_1, P \models^+ G_2$, by the inductive hypothesis $I_2, P \models^+ G_1$ or $I_2, P \models^+ G_2$, and so we have $I_2, P \models^+ G_1 \vee G_2$.
2. As $I_1, P \models^- G_1$ and $I_1, P \models^- G_2$, by the inductive hypothesis $I_2, P \models^- G_1$ and $I_2, P \models^- G_2$, and so we have $I_2, P \models^- G_1 \vee G_2$.

$G_1 \wedge G_2$:

1. As $I_1, P \models^+ G_1$ and $I_1, P \models^+ G_2$, by the inductive hypothesis $I_2, P \models^+ G_1$ and $I_2, P \models^+ G_2$, and so we have $I_2, P \models^+ G_1 \wedge G_2$.
2. As $I_1, P \models^- G_1$ or $I_1, P \models^- G_2$, by the inductive hypothesis $I_2, P \models^- G_1$ or $I_2, P \models^- G_2$, and so we have $I_2, P \models^- G_1 \wedge G_2$.

$\exists x G$:

1. As $I_1, P \models^+ G[t/x]$ for some $t \in \mathcal{U}$, by the inductive hypothesis $I_2, P \models^+ G[t/x]$ for some $t \in \mathcal{U}$, and so we have $I_2, P \models^+ \exists x G$.
2. As $\exists R \in \mathcal{R}(\mathcal{U})$ such that $I_1, P \models^- G[t/x]$ for all $t \in R$, by the inductive hypothesis $I_2, P \models^- G[t/x]$ for all $t \in R$, and so we have $I_2, P \models^- \exists x G$.

$\forall x G$:

1. As $\exists R \in \mathcal{R}(\mathcal{U})$ such that $I_1, P \models^+ G[t/x]$ for all $t \in R$, by the inductive hypothesis $I_2, P \models^+ G[t/x]$ for all $t \in R$, and so we have $I_2, P \models^+ \forall x G$.
2. As $I_1, P \models^- G[t/x]$ for some $t \in \mathcal{U}$, by the inductive hypothesis $I_2, P \models^- G[t/x]$ for some $t \in \mathcal{U}$, and so we have $I_2, P \models^- \forall x G$.

$D' \supset G'$:

1. As $I_1, \langle D \cup \{D'\}, N\rangle \models^+ G'$, by the inductive hypothesis $I_2, \langle D \cup \{D'\}\rangle \models^+ G'$, and so we have $I_2, P \models^+ D' \supset G'$.
2. As $I_1, \langle D \cup \{D'\}, N\rangle \models^- G'$, by the inductive hypothesis $I_2, \langle D \cup \{D'\}\rangle \models^- G'$, and so we have $I_2, P \models^- D' \supset G'$.

□

In the light of Proposition 2.3.3, it should not be surprising that the following Lemma holds.

Lemma 5.3.3 *Let $\langle P, G\rangle$ be a derivation state, and let I be an interpretation. Then*

1. $I, P \models^{+} G \Rightarrow I, P \models^{+} G[t/x]$ *for any* $t \in \mathcal{U}$
2. $I, P \models^{-} G \Rightarrow I, P \models^{-} G[t/x]$ *for any* $t \in \mathcal{U}$

Proof: Obvious. □

Next we show that interpretations conserve the consistency of $\models^{+}$ and $\models^{-}$.

Lemma 5.3.4 *Let I be an interpretation. Then there is no derivation pair $\langle P, G\rangle$ such that*

$$I, P \models^{+} G \text{ and } I, P \models^{-} G$$

Proof: We proceed by induction on the size of G.

The base case occurs when G is an atom A. Now

$$I, P \models^{+} A \text{ iff } A \prec \mathrm{pos}(I)(P)$$
$$I, P \models^{-} A \text{ iff } A \prec \mathrm{neg}(I)(P)$$

and as I is an interpretation, $\mathrm{inst}(\mathrm{pos}(I)(P)) \cap \mathrm{inst}(\mathrm{neg}(I)(P)) = \emptyset$, and so there can be no atom A such that $I, P \models^{+} A$ and $I, P \models^{-} A$.

Hence the induction hypothesis is that the lemma is true for all goals of no more than a given size. There are five cases:

$G_1 \vee G_2$: $I, P \models^{+} G_1 \vee G_2$ iff $I, P \models^{+} G_1$ or $I, P \models^{+} G_2$ and by the hypothesis this implies that it is impossible that $I, P \models^{-} G_1$ or it is impossible that $I, P \models^{-} G_2$, and in either case it is impossible that $I, P \models^{-} G_1$ and $I, P \models^{-} G_2$, i.e. it is impossible that $I, P \models^{-} G_1 \vee G_2$.

$G_1 \wedge G_2$: $I, P \models^+ G_1 \wedge G_2$ iff $I, P \models^+ G_1$ and $I, P \models^+ G_2$ and by the hypothesis this implies that it is impossible that $I, P \models^- G_1$ and it is impossible that $I, P \models^- G_2$, and so it is impossible that $I, P \models^- G_1$ or $I, P \models^- G_2$, i.e. it is impossible that $I, P \models^- G_1 \wedge G_2$.

$\exists x G$: $I, P \models^+ \exists x G$ iff $I, P \models^+ G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis this implies that it is impossible that $I, P \models^- G[t/x]$ for some $t \in \mathcal{U}$, and so by Lemma 5.3.3 it is impossible that $\exists R \in \mathcal{R}(\mathcal{U})$ such that $I, P \models^- G[t/x]$ for all $t \in R$, i.e. it is impossible that $I, P \models^- \exists x G$.

$\forall x G$: $I, P \models^+ \forall x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $I, P \models^+ G[t/x]$ for all $t \in R$, and so $I, P \models^+ G[t/x]$ for all $t \in \mathcal{U}$, and by the hypothesis this implies that it is impossible that $I, P \models^- G[t/x]$ for all $t \in \mathcal{U}$, and so it is impossible that $I, P \models^- \forall x G$.

$D' \supset G$: $I, \langle D, N \rangle \models^+ D' \supset G$ iff $I, \langle D \cup \{D'\}, N \rangle \models^+ G$ and $\langle D \cup \{D'\}, N \rangle \geq \langle D, N \rangle$, and by the hypothesis this implies that it is impossible that $I, \langle D \cup \{D'\}, N \rangle \models^- G$, and so it is impossible that $I, \langle D, N \rangle \models^- D' \supset G$.

□

The next three lemmas are important for the construction process.

Lemma 5.3.5 *Let I_1 and I_2 be interpretations.*

If $I_1 \sqsubseteq I_2$, then I_1 and I_2 are mutually consistent.

Proof: As $I_1 \sqsubseteq I_2$, for any P we have $I_1(P) \leq I_2(P)$, and so

$$\mathrm{pos}(I_1)(P) \prec \mathrm{pos}(I_2)(P)$$
$$\mathrm{neg}(I_1)(P) \prec \mathrm{neg}(I_2)(P)$$

As I_2 is an interpretation, $\mathrm{inst}(\mathrm{pos}(I_2)(P)) \cap \mathrm{inst}(\mathrm{neg}(I_2)(P)) = \emptyset$, and so $\mathrm{inst}(\mathrm{pos}(I_2)(P)) \cap \mathrm{inst}(\mathrm{neg}(I_1)(P)) = \emptyset$, and $\mathrm{inst}(\mathrm{neg}(I_2)(P)) \cap \mathrm{inst}(\mathrm{pos}(I_1)(P)) = \emptyset$, i.e. I_1 and I_2 are mutually consistent.

□

Lemma 5.3.6 *Let I_1 and I_2 be interpretations. If I_1 and I_2 are mutually consistent, then $I_1 \sqcup I_2$ is an interpretation.*

Proof: As I_1 and I_2 are mutually consistent, we have that for any program P

$$\mathrm{inst}(\mathrm{pos}(I_1)(P)) \cap \mathrm{inst}(\mathrm{neg}(I_2)(P)) = \emptyset$$
$$\mathrm{inst}(\mathrm{neg}(I_1)(P)) \cap \mathrm{inst}(\mathrm{pos}(I_2)(P)) = \emptyset$$

and so as both I_1 and I_2 are interpretations, it is clear that

$$\mathrm{inst}(\mathrm{pos}(I_1)(P) \cup \mathrm{pos}(I_2)(P)) \cap \mathrm{inst}(\mathrm{neg}(I_1)(P) \cup \mathrm{neg}(I_2)(P)) = \emptyset$$

Thus $I_1 \sqcup I_2$ is internally consistent.

Now as I_i is an interpretation for $i = 1, 2$, for any programs P_1 and P_2 such that $P_1 \leq P_2$, we have

$$\mathrm{pos}(I_i)(P_1) \prec \mathrm{pos}(I_i)(P_2)$$
$$\mathrm{neg}(I_i)(P_1) \prec \mathrm{neg}(I_i)(P_2)$$

for $i = 1, 2$, and hence we have

$$\mathrm{pos}(I_1)(P_1) \cup \mathrm{pos}(I_2)(P_1) \prec \mathrm{pos}(I_1)(P_2) \cup \mathrm{pos}(I_2)(P_2)$$
$$\mathrm{neg}(I_1)(P_1) \cup \mathrm{neg}(I_2)(P_1) \prec \mathrm{neg}(I_1)(P_2) \cup \mathrm{neg}(I_2)(P_2)$$

Hence, $I_1 \sqcup I_2$ is an interpretation. □

Corollary 5.3.7 *Let I_1 and I_2 be interpretations. If $I_1 \sqsubseteq I_2$, then $I_1 \sqcup I_2$ is an interpretation.*

Proof: Follows immediately from lemmas 5.3.5 and 5.3.6. □

Lemma 5.3.8 *Let I_1 and I_2 be mutually consistent interpretations. Then $I_1 \sqsubseteq I_1 \sqcup I_2$ and $I_2 \sqsubseteq I_1 \sqcup I_2$.*

Proof: Obvious. □

Another important lemma, again similar to one in [77] is given below.

Lemma 5.3.9 *Let $I_1 \sqsubseteq I_2 \sqsubseteq I_3 \sqsubseteq \ldots$ be an increasing sequence of interpretations, and let $\langle P, G\rangle$ be a D_{HHF-} derivation pair where $P = \langle D, N\rangle$.*

1. *If $\bigsqcup_{i=1}^{\infty} I_i, P \models^{+} G$, then $\exists k \geq 1$ such that $I_k, P \models^{+} G$.*

2. *If $\bigsqcup_{i=1}^{\infty} I_i, P \models^{-} G$, then $\exists k \geq 1$ such that $I_k, P \models^{-} G$.*

Proof: Note that $\bigsqcup_{i=1}^{\infty} I_i$ is an interpretation by corollary 5.3.7. Let $I_i(P) = \langle S_i, F_i\rangle$. We proceed by induction on the structure of G.

1. If G is an atom A and $\bigsqcup_{i=1}^{\infty} I_i, P \models^{+} A$, then $A \prec \bigcup_{i=1}^{\infty} S_i$, and as A is an instance of itself, $A \in \text{inst}(\bigcup_{i=1}^{\infty} S_i) = \bigcup_{i=1}^{\infty} \text{inst}(S_i)$. Hence $\exists k$ such that $A \in \text{inst}(S_k)$, and so $\text{inst}(A) \subseteq \text{inst}(S_k)$, which implies that $A \prec S_k$, i.e. $I_k, P \models^{+} A$.

   If G is $\neg A$ and $\bigsqcup_{i=1}^{\infty} I_i, P \models^{+} \neg A$, then $A \prec \bigcup_{i=1}^{\infty} F_i$, and as A is an instance of itself, $A \in \text{inst}(\bigcup_{i=1}^{\infty} F_i) = \bigcup_{i=1}^{\infty} \text{inst}(F_i)$. Hence $\exists k$ such that $A \in \text{inst}(F_k)$, and so $\text{inst}(A) \subseteq \text{inst}(F_k)$, which implies that $A \prec F_k$, i.e. $I_k, P \models^{+} \neg A$.

2. As $I, P \models^{-} A$ iff $I, P \models^{+} \neg A$ and $I, P \models^{-} \neg A$ iff $I, P \models^{+} A$, this follows from the above argument.

Hence the inductive hypothesis is that the lemma holds for all goals of no more than a given size. There are five cases:

$G_1 \vee G_2$:

1. As $\bigsqcup_{i=1}^{\infty} I_i, P \models^{+} G_1 \vee G_2$, we have $\bigsqcup_{i=1}^{\infty} I_i, P \models^{+} G_1$ or $\bigsqcup_{i=1}^{\infty} I_i, P \models^{+} G_2$. By the hypothesis, we have $I_i, P \models^{+} G_1$ or $I_j, P \models^{+} G_2$ for some $i, j \geq 1$. Let k be the maximum of i and j. By lemma 5.3.2, we have $I_k, P \models^{+} G_1$ or $I_k, P \models^{+} G_2$, as $I_i \sqsubseteq I_k$ and $I_j \sqsubseteq I_k$, and so $I_k, P \models^{+} G_1 \vee G_2$.

2. As $\bigsqcup_{i=1}^{\infty} I_i, P \models^- G_1 \vee G_2$, we have $\bigsqcup_{i=1}^{\infty} I_i, P \models^- G_1$ and $\bigsqcup_{i=1}^{\infty} I_i, P \models^- G_2$. By the hypothesis, we have $I_i, P \models^- G_1$ and $I_j, P \models^- G_2$ for some $i, j \geq 1$. Let k be the maximum of i and j. By lemma 5.3.2, we have $I_k, P \models^+ G_1$ and $I_k, P \models^+ G_2$, as $I_i \sqsubseteq I_k$ and $I_j \sqsubseteq I_k$, and so $I_k, P \models^- G_1 \vee G_2$.

$G_1 \wedge G_2$:

1. As $\bigsqcup_{i=1}^{\infty} I_i, P \models^+ G_1 \wedge G_2$, we have $\bigsqcup_{i=1}^{\infty} I_i, P \models^+ G_1$ and $\bigsqcup_{i=1}^{\infty} I_i, P \models^+ G_2$. By the hypothesis, we have $I_i, P \models^+ G_1$ and $I_j, P \models^+ G_2$ for some $i, j \geq 1$. Let k be the maximum of i and j. By lemma 5.3.2, we have $I_k, P \models^+ G_1$ and $I_k, P \models^+ G_2$, as $I_i \sqsubseteq I_k$ and $I_j \sqsubseteq I_k$, and so $I_k, P \models^+ G_1 \wedge G_2$.
2. As $\bigsqcup_{i=1}^{\infty} I_i, P \models^- G_1 \wedge G_2$, we have $\bigsqcup_{i=1}^{\infty} I_i, P \models^+ G_1$ or $\bigsqcup_{i=1}^{\infty} I_i, P \models^+ G_2$. By the hypothesis, we have $I_i, P \models^+ G_1$ or $I_j, P \models^+ G_2$ for some $i, j \geq 1$. Let k be the maximum of i and j. By lemma 5.3.2, we have $I_k, P \models^+ G_1$ or $I_k, P \models^+ G_2$, as $I_i \sqsubseteq I_k$ and $I_j \sqsubseteq I_k$, and so $I_k \models^- G_1 \wedge G_2$.

$\exists x G'$:

1. As $\bigsqcup_{i=1}^{\infty} I_i, P \models^+ \exists x G'$, we have that $\bigsqcup_{i=1}^{\infty} I_i, P \models^+ G'[x/t]$ for some $t \in \mathcal{U}$. By the hypothesis, $I_k, P \models^+ G'[x/t]$ for some $k \geq 1$, and so we have $I_k, P \models^+ \exists x G'$.
2. As $\bigsqcup_{i=1}^{\infty} I_i, P \models^- \exists x G'$, we have that $\exists R \in \mathcal{R}(\mathcal{U})$ such that $\bigsqcup_{i=1}^{\infty} I_i, P \models^- G'[x/t]$ for all $t \in R$, and so by the hypothesis, for each $t \in R$ there is a k_t such that $I_{k_t}, P \models^- G'[t/x]$. Let k be the maximum of all the k_t. Hence $I_k, P \models^- G'[t/x]$ for all $t \in R$, and so we have $I_k, P \models^- \exists x G'$.

$\forall x G'$:

1. As $\bigsqcup_{i=1}^{\infty} I_i, P \models^+ \forall x G'$, we have that $\exists R \in \mathcal{R}(\mathcal{U})$ such that $\bigsqcup_{i=1}^{\infty} I_i, P \models^+ G'[x/t]$ for all $t \in R$, and so by the hypothesis, for each $t \in R$ there is a k_t such that $I_{k_t}, P \models^+ G'[t/x]$. Let k be the maximum of all the k_t. Hence $I_k, P \models^+ G'[t/x]$ for all $t \in R$, and so we have $I_k, P \models^+ \forall x G'$.

2. As $\bigsqcup_{i=1}^{\infty} I_i, P \models^- \forall x G'$, we have that $\bigsqcup_{i=1}^{\infty} I_i, P \models^- G'[x/t]$ for some $t \in \mathcal{U}$. By the hypothesis, $I_k, P \models^- G'[x/t]$ for some $k \geq 1$, and so we have $I_k, P \models^- \forall x G'$.

$D' \supset G'$:

1. As $\bigsqcup_{i=1}^{\infty} I_i, \langle D, N\rangle \models^+ D' \supset G'$, we have that $\bigsqcup_{i=1}^{\infty} I_i, \langle D \cup \{D'\}, N\rangle \models^+ G'$. By the hypothesis, $I_k, \langle D \cup \{D'\}, N\rangle \models^+ G'$ for some $k \geq 1$, and so we have $I_k, \langle D, N\rangle \models^+ D' \supset G'$.
2. As $\bigsqcup_{i=1}^{\infty} I_i, \langle D, N\rangle \models^- D' \supset G'$, we have that $\bigsqcup_{i=1}^{\infty} I_i, \langle D \cup \{D'\}, N\rangle \models^- G'$. By the hypothesis, $I_k, \langle D \cup \{D'\}, N\rangle \models^- G'$ for some $k \geq 1$, and so we have $I_k, \langle D, N\rangle \models^- D' \supset G'$.

□

Note that this result depends critically on the compactness properties of goals containing quantifiers.

Thus our extended notion of interpretation preserves important semantic properties. We show in the next section how the important properties of the $T^{\omega}(I_{\perp})$ construction are preserved as well.

5.4 The Construction Process

We wish to find a single interpretation J such that $P \vdash_s G$ iff $J, P \models^+ G$ and $P \vdash_f G$ iff $J, P \models^- G$. The construction of this interpretation may be thought of as mirroring the computational process, and is performed by a machine which has an unbounded amount of time and space and never makes a mistake. It is this step which justifies all the above definitions etc., as we may interpret this as defining a model for the program P. The interpretation J is traditionally constructed in logic programming semantics as the least fixed point of a monotonic operator T which maps interpretations to interpretations. The monotonicity of T guarantees that the least fixed point of T is $T^{\omega}(I_{\perp})$ for suitably defined powers of T, where $I_{\perp}$ is the empty interpretation.

We proceed in a similar manner to that in [77], i.e. we build ordinal powers of a T operator, and use the union of all such powers to produce the desired interpretation. Before we do so, let us consider the nature of the construction process. The desired process is one that builds upwards, so that as the process goes on, the knowledge we have is always increasing, but never in an inconsistent fashion.

We may think of this process as ascending a cone similar to the worlds described above, except that each node is an ordered pair $\langle S, F\rangle$ where S and F are sets of atoms with the accessibility relation being componentwise set inclusion. The process begins at the base node and continues upward monotonically, in that when going from node $\langle S_1, F_1\rangle$ to $\langle S_2, F_2\rangle$, we have $S_1 \prec S_2$ and $F_1 \prec F_2$. This means that whenever the construction process places an atom A in $\mathrm{neg}(T^k(I))(w)$, we must be sure that A will not be placed in $\mathrm{pos}(T^j(I))(w)$ for some j.

As our desired interpretation is $T^\omega(I_\perp)$, we are mainly interested in the ordinal powers of T so that we may construct $T^\omega(I_\perp)$, rather than interpretations and fixpoints per se. Before we define these ordinal powers, it is important to note that a consequence of the monotonicity of the operator given in [77] is that if $T^k(I_\perp), w \models G$, then $T^{k+1}(I_\perp), w \models G$. This may be seen by the fact that $I_\perp \sqsubseteq T(I_\perp)$, and so as T is monotonic, we have $T(I_\perp) \sqsubseteq T^2(I_\perp)$ and so on. This is perhaps a more important consequence of the monotonicity of T than the fact that a least fixed point exists. Whilst the fixpoint semantics does display a certain mathematical elegance, if the method used to construct the desired interpretation does not respect provability via $\models$, it is difficult to see what use it would be. The only thing we lose by using a direct construction rather than an operator is the ability to think of T as an operator on arbitrary interpretations, and so produce other constructions based on it. Traditionally the only other construction to gain much interest has been to produce the greatest fixpoint of T in order to deal with NAF. However, we have seen that the constructivist approach necessitates a more general notion of interpretation, which allows us to directly incorporate similar semantic properties to those of the greatest fixpoint of T. Thus we capture the

same descriptive power as the least fixpoint/greatest fixpoint approach, but in a more concrete way.

In some ways, the above definition of an interpretation is too general, in that it allows interpretations which do not correspond to our intuitive understanding of the program. For example, consider the program $\langle p(a), \langle \emptyset, \{p\}\rangle\rangle$, and the interpretation I for which $I(P) = \langle \emptyset, \{p(a)\}\rangle$ for all programs P. Clearly this interpretation is somehow "at odds" with the program, especially when it comes to comparisons with the operational notion of provability. Hence it seems more natural to construct the sequence of interpretations in which we are interested (i.e. $T^i(I_\perp)$) than to consider an operator on arbitrary interpretations.

It is possible to define a continuous operator which suits our purposes, but as the partial order involved does not form a complete lattice, the Knaster-Tarski fixpoint theorem used in [77] will not apply. In the next section we show how this operator may be defined, and discuss related issues. Below we present a more specific construction which builds the powers of T directly, and for which we may derive the desired results.

We define the powers of T in a slightly more intricate way than is strictly necessary, in order to facilitate some later proofs.

Definition 5.4.1 *Let I be an interpretation, and let $P = \langle D, N\rangle$ be a D_{HHF}-derivation state. We define the ordinal powers of T as follows:*

$$\begin{aligned}
\mathrm{pos}(Pr^0(I))(P) &= \{A \mid A \in (D)\} \\
\mathrm{neg}(Pr^0(I))(P) &= \{A \mid \mathrm{name}(A) \in \mathrm{den}(N) \textit{ and } \forall B \in (D) \textit{ and } \forall G \supset B \in (D),\ B \not\propto A\} \\
& T^0(I) = Pr^0(I)
\end{aligned}$$

$$\begin{aligned}
\mathrm{pos}(Pr^{k+1}(I))(P) &= \{A \mid \exists G \supset A \in (D) \textit{ such that } T^k(I), P \models^+ G\} \\
\mathrm{neg}(Pr^{k+1}(I))(P) &= \{A \mid \mathrm{name}(A) \in \mathrm{den}(N) \textit{ and } \forall B \in (D),\ B \not\propto A \textit{ and} \\
& \qquad \forall G \supset B \in (D) \textit{ such that } B \propto A, T^k(I), P \models^- G\} \\
& T^{k+1}(I) = Pr^{k+1}(I) \sqcup T^k(I)
\end{aligned}$$

$$T^\omega(I) = \bigsqcup_{i=1}^{\infty} T^i(I)$$

For an example of how this process works, consider the program below.

$$\begin{aligned} &\text{even}(0) \\ &\forall x\ \neg\text{even}(x) \supset \text{even}(s(x)) \end{aligned}$$

Let $P = \langle D, \langle \emptyset, \{\text{even}\}\rangle\rangle$ where D is the code in the even program above. Then we have

$$\begin{aligned} Pr^0(I_\perp)(P) &= \langle\{\text{even}(0)\}, \emptyset\rangle \\ Pr^1(I_\perp)(P) &= \langle\emptyset, \{\text{even}(s(0))\}\rangle \\ Pr^2(I_\perp)(P) &= \langle\{\text{even}(s^2(0))\}, \emptyset\rangle \\ Pr^3(I_\perp)(P) &= \langle\emptyset, \{\text{even}(s^3(0))\}\rangle \\ &\cdots \end{aligned}$$

and so

$$\begin{aligned} T^0(I_\perp)(P) &= \langle\{\text{even}(0)\}, \emptyset\rangle \\ T^1(I_\perp)(P) &= \langle\{\text{even}(0)\}, \{\text{even}(s(0))\}\rangle \\ T^2(I_\perp)(P) &= \langle\{\text{even})(0), \text{even}(s^2(0))\}, \{\text{even}(s(0))\}\rangle \\ T^3(I_\perp)(P) &= \langle\{\text{even})(0), \text{even}(s^2(0))\}, \{\text{even}(s(0)), \text{even}(s^3(0))\}\rangle \\ &\cdots \end{aligned}$$

Note that even(x) neither succeeds nor fails, as there are some instances of it which succeed and some which fail.

In this way we may think of the powers of T as using the program to define an increasing sequence of interpretations which is used to model the behaviour of the program. The final interpretation in this sequence (i.e. $T^\omega(I_\perp)$) can indeed be shown to capture this operational behaviour.

Below we show that the powers of T are interpretations.

Lemma 5.4.1 *Let I be an interpretation. Then for any $i \geq 0$, $Pr^i(I_\perp)$ and $T^i(I_\perp)$ are both interpretations.*

Proof: We proceed by induction on i. Let $P = \langle D, N \rangle$ be a D_{HHF-} derivation state.

In the base case, we have $Pr^0(I_\perp) = T^0(I_\perp)$, and so we need only show that $Pr^0(I_\perp)$ is an interpretation. Let $P = \langle D, N \rangle$ be a D_{HHF-} derivation state. It is clear that there can be no atom A such that $A \in (D)$ and $\forall B \in (D)$ $B \not\propto A$, and so $\text{pos}(Pr^0(I_\perp))(P) \cap \text{neg}(Pr^0(I_\perp))(P) = \emptyset$.

Now given $P_1 = \langle D_1, N_1 \rangle$ and $P_2 = \langle D_2, N_2 \rangle$ such that $P_1 \leq P_2$, we know that $(D_1) \subseteq (D_2)$, $\text{ass}(N_1) \subseteq \text{ass}(N_2)$, $\text{den}(N_1) \subseteq \text{den}(N_2)$, and for each $C \in D_2 \backslash D_1$, $\text{name}(\text{head}(C)) \in \text{ass}(N_1)$. As $(D_1) \subseteq (D_2)$, we have that $\text{pos}(Pr^0(I_\perp))(P_1) \subseteq \text{pos}(Pr^0(I_\perp))(P_2)$. Now if $A \in \text{neg}(Pr^0(I_\perp))(P_1)$, then $\text{name}(A) \in \text{den}(N_1)$ and $\forall B \in (D_1)$ and $\forall G \supset B \in (D_1)$ we have $B \not\propto A$. Now as for each $C \in D_2 \backslash D_1$ we have $\text{name}(\text{head}(C)) \notin \text{den}(N_1)$, it follows that $\forall B \in (D_2)$ and $\forall G \supset B \in (D_2)$ we have $B \not\propto A$, and so $A \in \text{neg}(Pr^0(I_\perp))(P_2)$.

Hence we assume that the lemma is true for all $0 \leq i \leq k$, so that $Pr^k(I_\perp)$ and $T^k(I_\perp)$ are interpretations. It will be sufficient to show that $Pr^{k+1}(I_\perp)$ is an interpretation, and that $Pr^{k+1}(I_\perp)$ and $T^k(I_\perp)$ are mutually consistent.

Let $P = \langle D, N \rangle$ be a D_{HHF-} derivation state. If $A \in \text{pos}(Pr^{k+1}(I_\perp))(P)$, then $\exists G \supset A \in (D)$ such that $T^k(I_\perp), P \models^+ G$, and so it is impossible that $\forall G \supset B \in (D)$ such that $B \propto A$, $T^k(I_\perp), P \models^- G$ by lemma 5.3.4, and so $\text{pos}(Pr^{k+1}(I_\perp))(P) \cap \text{neg}(Pr^{k+1}(I_\perp))(P) = \emptyset$.

Now as above, if P_1 and P_2 are two program such that $P_1 \leq P_2$, then $(D_1) \subseteq (D_2)$, and so $\text{pos}(Pr^{k+1}(I_\perp))(P_1) \subseteq \text{pos}(Pr^{k+1}(I_\perp))(P_2)$. Now if $A \in \text{neg}(Pr^{k+1}(I_\perp))(P_1)$, then $\text{name}(A) \in \text{den}(N_1)$ and $\forall B \in (D_1)$, $B \not\propto A$ and $\forall G \supset B \in (D_1)$ such that $B \propto A$ we have that $T^k(I_\perp), P_1 \models^- G$. As above, for each $C \in D_2 \backslash D_1$ we have $\text{name}(\text{head}(C)) \notin \text{den}(N_1)$, and so it follows that $\forall B \in (D_2)$ and $\forall G \supset B \in (D_2)$ such that $B \propto A$ we have $T^k(I_\perp), P_2 \models^- G$, and so $A \in \text{neg}(Pr^{k+1}(I_\perp))(P_2)$.

Hence $Pr^{k+1}(I_\perp)$ is an interpretation.

Let $P = \langle D, N \rangle$ be a D_{HHF-} derivation state. Now if $A \in \text{pos}(Pr^{k+1}(I_\perp))(P)$, then $\exists G \supset A \in (D)$ such that $T^k(I_\perp), P \models^+ G$. If $A \in \text{neg}(T^k(I_\perp))(P)$, then $A \in \text{neg}(Pr^i(I_\perp))(P)$ for some $0 \leq i \leq k$, and so $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ either $B \not\propto A$ or $B \propto A$ and $T^{i-1}(I_\perp), P \models^- G$, which by lemma 5.3.2 implies that $\forall G \supset B \in (D)$ either $B \not\propto A$ or $B \propto A$ and $T^k(I_\perp), P \models^- G$. Hence $\text{pos}(Pr^{k+1}(I_\perp))(P) \cap \text{pos}(T^k(I_\perp))(P) = \emptyset$.

Similarly, if $A \in \text{neg}(Pr^{k+1}(I_\perp))(P)$, then $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$, $T^k(I_\perp), P \models^- G$. If $A \in \text{pos}(T^k(I_\perp))(P)$, then $A \in \text{pos}(Pr^i(I_\perp))(P)$ for some $0 \leq i \leq k$, and so either $A \in (D)$ or $\exists G \supset A \in (D)$ such that $T^{i-1}(I_\perp), P \models^+ G$, and so by lemma 5.3.2 this implies that either $A \in (D)$ or $\exists G \supset A \in (D)$ such that $T^k(I_\perp), P \models^+ G$. Hence $\text{pos}(Pr^{k+1}(I_\perp))(P) \cap \text{pos}(T^k(I_\perp))(P) = \emptyset$.

Thus $Pr^{k+1}(I_\perp)$ and $T^k(I_\perp)$ are mutually consistent, and so by lemma 5.3.6, $T^{k+1}(I_\perp)$ is an interpretation.

□

Thus the construction gives us an increasing sequence of interpretations. This sequence respects $\models^+$ and $\models^-$ as shown below.

Proposition 5.4.2 *Let P be a D_{HHF-} derivation state and let G be a G_{HHF-} goal formula. Then*

1. $T^k(I_\perp), P \models^+ G \Rightarrow T^j(I_\perp), P \models^+ G$ *for any* $j \geq k$

2. $T^k(I_\perp), P \models^- G \Rightarrow T^j(I_\perp), P \models^- G$ *for any* $j \geq k$

Proof: $T^k(I_\perp)$ and $T^j(I_\perp)$ are interpretations by lemma 5.4.1, and as $T^k(I_\perp) \sqsubseteq T^j(I_\perp)$, 1 & 2 follow immediately by lemma 5.3.2. □

Thus our construction preserves important properties.

We now show the relationship between our construction and the relations $\vdash_s$ and $\vdash_f$. First we show that $\vdash_s$ and $\vdash_f$ are sound with respect to the Kripke-like model.

Proposition 5.4.3 *Let $\langle P, G\rangle$ be a D_{HHF-} derivation pair where $P = \langle D, N\rangle$. Then*

1. $P \vdash_s G \Rightarrow T^{\omega}(I_{\perp}), P \models^{+} G$
2. *If G is negatable, then* $P \vdash_f G \Rightarrow T^{\omega}(I_{\perp}), P \models^{-} G$

Note that the restriction in 2 is necessary. As $T^{\omega}(I_{\perp})$ is an interpretation, enlarging the program must preserve what is known to be true and what is known to be false. On the other hand, enlarging a program may mean that a goal which originally failed does not fail in the larger program.

Proof: We proceed by induction on the depth of the **O**-derivation of G. The base case occurs when G is an atom A and the sequent $P \longrightarrow^{+} A$ (resp. $P \longrightarrow^{-} A$) is initial.

1. As the sequent is initial, we have $A \in (D)$, and hence $A \in \text{pos}(T^{0}(I_{\perp}))(P)$, and so $T^{\omega}(I_{\perp}), P \models^{+} A$.
2. As the sequent is initial, we have $\forall B \in (D)$ and $\forall G \supset B \in (D)$ $B \not\propto A$ and as A is negatable, we have $\text{name}(A) \in \text{den}(N)$, and hence $A \in \text{neg}(T^{0}(I_{\perp}))(P)$, and so $T^{\omega}(I_{\perp}), P \models^{-} A$.

Hence we assume that the proposition is true for all **O**-derivations of no more than a given depth. There are seven cases:

A: 1. If the base case does not hold, we have that $\exists G \supset A \in (D)$ such that $P \vdash_s G$, and by the hypothesis this implies that $T^{\omega}(I_{\perp}), P \models^{+} G$. By lemma 5.3.9 we have that $T^{k}(I_{\perp}), P \models^{+} G$ for some k, and so $T^{k+1}(I_{\perp}), P \models^{+} A$, i.e. $T^{\omega}(I_{\perp}), P \models^{+} A$.

2. If the base case does not hold, we have that $\forall B \in (D)\ B \not\propto A$, and $\forall G \supset B \in (D)$ such that $B \propto A$, $P \vdash_f G$, and as G is negatable, by the hypothesis, $T^\omega(I_\perp), P \models^- G$. By lemma 5.3.9 we have that $T^k(I_\perp), P \models^- G$ for some k, and so $T^{k+1}(I_\perp), P \models^- A$, i.e. $T^\omega(I_\perp), P \models^- A$.

$\neg A$: 1. $P \vdash_s \neg A$ iff $P \vdash_f A$ and name$(A) \in$ den(N), and by the hypothesis, $T^\omega(I_\perp), P \models^- A$. By lemma 5.3.9, we have that $T^k(I_\perp), P \models^- A$ for some k, and so $T^k(I_\perp), P \models^+ \neg A$, i.e. $T^\omega(I_\perp), P \models^+ \neg A$.

2. $P \vdash_f \neg A$ iff $P \vdash_s A$, and by the hypothesis, $T^\omega(I_\perp), P \models^+ A$. By lemma 5.3.9, we have that $T^k(I_\perp), P \models^+ A$ for some k, and so $T^k(I_\perp), P \models^- \neg A$, i.e. $T^\omega(I_\perp), P \models^- \neg A$.

$G_1 \vee G_2$: 1. $P \vdash_s G_1 \vee G_2$ iff $P \vdash_s G_1$ or $P \vdash_s G_2$, and so by the hypothesis $T^\omega(I_\perp), P \models^+ G_1$ or $T^\omega(I_\perp), P \models^+ G_2$, i.e. $T^\omega(I_\perp), P \models^+ G_1 \vee G_2$.

2. $P \vdash_f G_1 \vee G_2$ iff $P \vdash_s G_1$ and $P \vdash_s G_2$, and so by the hypothesis $T^\omega(I_\perp), P \models^- G_1$ and $T^\omega(I_\perp), P \models^- G_2$, i.e. $T^\omega(I_\perp), P \models^- G_1 \vee G_2$.

$G_1 \wedge G_2$: 1. $P \vdash_s G_1 \wedge G_2$ iff $P \vdash_s G_1$ and $P \vdash_s G_2$, and so by the hypothesis $T^\omega(I_\perp), P \models^+ G_1$ and $T^\omega(I_\perp), P \models^+ G_2$, i.e. $T^\omega(I_\perp), P \models^+ G_1 \wedge G_2$.

2. $P \vdash_f G_1 \wedge G_2$ iff $P \vdash_s G_1$ or $P \vdash_s G_2$, and so by the hypothesis $T^\omega(I_\perp), P \models^- G_1$ or $T^\omega(I_\perp), P \models^- G_2$, i.e. $T^\omega(I_\perp), P \models^- G_1 \wedge G_2$.

$\exists x G$: 1. $P \vdash_s \exists x G$ iff $P \vdash_s G[t/x]$ for some $t \in \mathcal{U}$, and so by the hypothesis $T^\omega(I_\perp), P \models^+ G[t/x]$, i.e. $T^\omega(I_\perp), P \models^+ \exists x G$.

2. $P \vdash_f \exists x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_f G[t/x]$ for all $t \in R$, and so by the hypothesis $T^\omega(I_\perp), P \models^- G[t/x]\ \forall t \in R$, i.e. $T^\omega(I_\perp), P \models^- \exists x G$.

$\forall x G$: 1. $P \vdash_s \forall x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $P \vdash_s G[t/x]$ for all $t \in R$, and so by the hypothesis $T^\omega(I_\perp), P \models^+ G[t/x]\ \forall t \in R$, i.e. $T^\omega(I_\perp), P \models^+ \forall x G$.

2. $P \vdash_f \forall x G$ iff $P \vdash_f G[t/x]$ for some $t \in \mathcal{U}$, and so by the hypothesis $T^\omega(I_\perp), P \models^- G[t/x]$, i.e. $T^\omega(I_\perp), P \models^- \forall x G$.

$D' \supset G$: 1. $\langle D, N\rangle \vdash_s D' \supset G$ iff $\langle D \cup \{D'\}, N\rangle \vdash_s G$ and names(heads(D')) $\subseteq$ ass(N), and so by the hypothesis $T^\omega(I_\perp), \langle D \cup \{D'\}, N\rangle \models\!\!\!\models^+ G$, i.e. $T^\omega(I_\perp), \langle D, N\rangle \models\!\!\!\models^+ D' \supset G$.

2. $\langle D, N\rangle \vdash_f D' \supset G$ iff $\langle D \cup \{D'\}, N\rangle \vdash_f G$ and names(heads(D')) $\subseteq$ ass(N), and as $D' \supset G$ is negatable, G is negatable, and so by the hypothesis $T^\omega(I_\perp), \langle D \cup \{D'\}, N\rangle \models\!\!\!\models^- G$, i.e. $T^\omega(I_\perp), \langle D, N\rangle \models\!\!\!\models^- D' \supset G$.

□

Note that this result may be thought of as demonstrating the compactness of $\vdash_s$ and $\vdash_f$, in that if $P \vdash_s G$, then $T^\omega(I_\perp), P \models\!\!\!\models^+ G$, and by Lemma 5.3.9 $T^k(I_\omega), P \models\!\!\!\models^+ G$ for some k, and similarly for $\vdash_f$ when G is negatable.

Next we show that operational provability is complete with respect to the Kripke-like model.

Proposition 5.4.4 *Let $\langle P, G\rangle$ be a D_{HHF-} derivation pair where $P = \langle D, N\rangle$. Then*

1. $T^\omega(I_\perp), P \models\!\!\!\models^+ G \Rightarrow P \vdash_s G$
2. $T^\omega(I_\perp), P \models\!\!\!\models^- G \Rightarrow P \vdash_f G$

Proof: By lemma 5.3.9 we have that

$$T^\omega(I_\perp), P \models\!\!\!\models^+ G \Rightarrow \exists k \text{ such that } T^k(I_\perp), P \models\!\!\!\models^+ G$$
$$T^\omega(I_\perp), P \models\!\!\!\models^- G \Rightarrow \exists k \text{ such that } T^k(I_\perp), P \models\!\!\!\models^- G$$

In each case, let k be the smallest such number.

We proceed to show 1 & 2 simultaneously by formal induction on the ordinal measure $\omega.k + n$, where n is the number of connectives in G.

The base case occurs when $n = k = 0$. $T^0(I_\perp), P \models\!\!\!\models^+ A$ implies that $A \prec (D)$, i.e. $A \in (D)$, and hence $P \vdash_s A$. $T^0(I_\perp), P \models\!\!\!\models^- A$ implies that $\forall B \in (D)$ and $\forall G \supset B \in (D)$, $B \not\prec A$, and so $P \vdash_f A$.

Hence we assume that the proposition is true for all programs and goals for which $\omega.k + n$ does not exceed a certain value.

There are seven cases:

A: 1. $T^k(I_\perp), P \models^+ A$ implies that $\exists G \supset A \in (D)$ such that $T^{k-1}(I_\perp), P \models^+ G$, and by the hypothesis, $P \vdash_s G$, and so $P \vdash_s A$.

2. $T^k(I_\perp), P \models^- A$ implies that $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$, we have $T^{k-1}(I_\perp), P \models^- G$, and by the hypothesis, $P \vdash_f G$, and so $P \vdash_f A$.

$\neg A$: 1. $T^k(I_\perp), P \models^+ \neg A$ implies that $T^k(I_\perp), P \models^- A$, and so $\forall B \in (D)$ $B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$, $T^{k-1}(I_\perp), P \models^- G$ and name$(A) \in$ den(N). By the hypothesis this implies that $P \vdash_f A$ and name$(A) \in$ den(N), i.e. $P \vdash_s \neg A$.

2. $T^k(I_\perp), P \models^- \neg A$ implies that $T^k(I_\perp), P \models^+ A$, and so $\exists G \supset A \in (D)$ such that $T^{k-1}(I_\perp), P \models^+ G$. By the hypothesis this implies that $P \vdash_s A$, and so $P \vdash_f \neg A$.

$G_1 \vee G_2$: 1. $T^k(I_\perp), P \models^+ G_1 \vee G_2$ iff $T^k(I_\perp), P \models^+ G_1$ or $T^k(I_\perp), P \models^+ G_2$ and by the hypothesis this implies that $P \vdash_s G_1$ or $P \vdash_s G_2$, i.e. $P \vdash_s G_1 \vee G_2$.

2. $T^k(I_\perp), P \models^- G_1 \vee G_2$ iff $T^k(I_\perp), P \models^- G_1$ and $T^k(I_\perp), P \models^- G_2$ and by the hypothesis this implies that $P \vdash_f G_1$ and $P \vdash_f G_2$, i.e. $P \vdash_f G_1 \vee G_2$.

$G_1 \wedge G_2$: 1. $T^k(I_\perp), P \models^+ G_1 \wedge G_2$ iff $T^k(I_\perp), P \models^+ G_1$ and $T^k(I_\perp), P \models^+ G_2$ and by the hypothesis this implies that $P \vdash_s G_1$ and $P \vdash_s G_2$, i.e. $P \vdash_s G_1 \wedge G_2$.

2. $T^k(I_\perp), P \models^- G_1 \wedge G_2$ iff $T^k(I_\perp), P \models^- G_1$ or $T^k(I_\perp), P \models^- G_2$ and by the hypothesis this implies that $P \vdash_f G_1$ or $P \vdash_f G_2$, i.e. $P \vdash_f G_1 \wedge G_2$.

$\exists x G$: 1. $T^k(I_\perp), P \models^+ \exists x G$ iff $T^k(I_\perp), P \models^+ G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis this implies that $P \vdash_s G[t/x]$, i.e. $P \vdash_s \exists x G$.

2. $T^k(I_\perp), P \models^- \exists x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $T^k(I_\perp), P \models^- G[t/x]$ for all $t \in R$, and by the hypothesis this implies that $P \vdash_f G[t/x]\ \forall t \in R$, i.e. $P \vdash_f \exists x G$.

$\forall x G$: 1. $T^k(I_\perp), P \models^+ \forall x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $T^k(I_\perp), P \models^+ G[t/x]$ for all $t \in R$, and by the hypothesis this implies that $P \vdash_s G[t/x]\ \forall t \in R$, i.e. $P \vdash_s \forall x G$.

2. $T^k(I_\perp), P \models^- \forall x G$ iff $T^k(I_\perp), P \models^- G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis this implies that $P \vdash_f G[t/x]$, i.e. $P \vdash_f \forall x G$.

$D' \supset G$: 1. $T^k(I_\perp), \langle D, N\rangle \models^+ D' \supset G$ iff $T^k(I_\perp), \langle D \cup \{D'\}, N\rangle \models^+ G$ and $\langle D \cup \{D'\}, N\rangle \geq \langle D, N\rangle$, and by the hypothesis this implies that $\langle D \cup \{D'\}, N\rangle \vdash_s G$ and names(heads(D')) $\subseteq$ ass(N), i.e. $\langle D, N\rangle \vdash_s D' \supset G$.

2. $T^k(I_\perp), \langle D, N\rangle \models^- D' \supset G$ iff $T^k(I_\perp), \langle D \cup \{D'\}, N\rangle \models^- G$ and $\langle D \cup \{D'\}, N\rangle \geq \langle D, N\rangle$, and by the hypothesis this implies that $\langle D \cup \{D'\}, N\rangle \vdash_f G$ and names(heads(D')) $\subseteq$ ass(N), i.e. $\langle D, N\rangle \vdash_f D' \supset G$.

□

We now come to the main theorem.

Theorem 5.4.5 *Let $P = \langle D, N\rangle$ be a D_{HHF-} derivation state and let G be a G_{HHF-} goal formula. Then*

1. $P \vdash_s G \Leftrightarrow T^\omega(I_\perp), P \models^+ G$

2. *If G is negatable, then* $P \vdash_f G \Rightarrow T^\omega(I_\perp), P \models^- G$

3. $P \vdash_f G \Leftarrow T^\omega(I_\perp), P \models^- G$

Proof: Follows directly from propositions 5.4.3 and 5.4.4. □

5.5 Comparative Results

Although we have been concerned with the powers of T themselves, rather than interpretations in general and fixpoints, it may be instructive to investigate the precise nature of the relationship between our presentation and the traditional approach. Below we give a result which may be thought of as characterising the fixed point nature of $T^{\omega}(I_{\perp})$ without finding a particular operator of which this interpretation is a fixed point.

We also show how we may define a version of the T operator "in isolation", as is done in the traditional case, i.e., $T(I)$ is defined in terms of the interpretation I and that the T operator so defined is continuous. As mentioned above, we cannot apply the Knaster-Tarski fixed point theorem used in [77] to derive that $T^{\omega}(I_{\perp})$ is the least fixed point of T. However, the fact that $\sqsubseteq$ is a chain-complete partial order on interpretations together with the monotonicity of the operator means that the least fixpoint will indeed be $T^{\omega}(I_{\perp})$ [60,1]. It is also not hard to show directly that $T^{\omega}(I_{\perp})$ is the least fixpoint.

First we show the result that gives the implicit characterisation of $T^{\omega}(I)$ as a fixed point.

Proposition 5.5.1 *Let $P = \langle D, N\rangle$ be a D_{HHF-} derivation state, and let A be an atom. Then*

1. $T^{\omega}(I_{\perp}), P \models^{+} A \Leftrightarrow T^{0}(I_{\perp}), P \models^{+} A$ *or* $\exists G \supset A \in (D)$ *such that* $T^{\omega}(I_{\perp}), P \models^{+} G$
2. $T^{\omega}(I_{\perp}), P \models^{-} A \Leftrightarrow T^{0}(I_{\perp}), P \models^{-} A$ *or* $\forall B \in (D), B \not\propto A$ *and* $\forall G \supset B \in (D)$ *such that* $B \propto A$, $T^{\omega}(I_{\perp}), P \models^{-} G$, *and* $\mathrm{name}(A) \in \mathrm{den}(N)$

Proof:

1($\Rightarrow$): $T^{\omega}(I_{\perp}), P \models^{+} A$ iff for some i we have $T^{i}(I_{\perp}), P \models^{+} A$ by lemma 5.3.9, and so either $i = 0$, i.e. $T^{0}(I_{\perp}), P \models^{+} A$, or $i > 0$ and $\exists G \supset A \in (D)$ such that $T^{i-1}(I_{\perp}), P \models^{+} G$, which implies that $T^{\omega}(I_{\perp}), P \models^{+} G$.

($\Leftarrow$): $T^0(I_\perp), P \models^+ A \Rightarrow T^\omega(I_\perp), P \models^+ A$ by lemma 5.3.2. By lemma 5.3.9, if $T^\omega(I_\perp), P \models^+ G$ then $T^i(I_\perp), P \models^+ G$ for some i, and so $T^{i+1}(I_\perp), P \models^+ A$, and hence $T^\omega(I_\perp), P \models^+ A$ by lemma 5.3.2.

2($\Rightarrow$): $T^\omega(I_\perp), P \models^- A$ iff for some i we have $T^i(I_\perp), P \models^- A$ by lemma 5.3.9, and so either $i = 0$, i.e. $T^0(I_\perp), P \models^- A$, or $i > 0$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A, T^{i-1}(I_\perp), P \models^- G$ and name$(A) \in$ den(N), and so by lemma 5.3.2 $T^\omega(I_\perp), P \models^- G$.

($\Leftarrow$): $T^0(I_\perp), P \models^- A \Rightarrow 13\ T^\omega(I_\perp), P \models^- A$ by lemma 5.3.2. By lemma 5.3.9, if $T^\omega(I_\perp), P \models^- G$, then $T^i(I_\perp), P \models^- G$ for some i, and as $\forall B \in (D)\ B \not\propto A$, we have $T^{i+1}(I_\perp), P \models^- A$, and so by lemma 5.3.2 $T^\omega(I_\perp), P \models^- A$.

□

Thus we may think of $T^\omega(I_\perp)(P)$ as a fixed point of the definition of the powers of T, in that replacing $T^{k+1}(I_\perp)$ on the left by $T^\omega(I_\perp)$ and $T^k(I_\perp)$ on the right by $T^\omega(I_\perp)$ results in a valid equation.

Next, we show how to define our T operator "in isolation", i.e. define what T does to an arbitrary interpretation. This is not a definition in which we are particularly interested; it is included for purposes of comparison. The operator, which we will call S, is defined as follows.

Definition 5.5.1 *Let I be an interpretation, and let $w = \langle D, N\rangle$ be a D_{HHF}-derivation state. Then we define*

$$\text{pos}(S(I))(w) = \{A \mid A \in (D) \textit{ or } \exists G \supset A \in (D) \textit{ such that } I, w \models^+ G\}$$
$$\text{neg}(S(I))(w) = \{A \mid \text{name}(A) \in \text{den}(N) \textit{ and } \forall B \in (D)\ B \not\propto A \textit{ and } \forall G \supset B \in (D) \textit{ such that } B \propto A \textit{ we have } I, w \models^- G\ \}$$

$$S^\omega(I) = \bigsqcup_{i=1}^{\infty} S^i(I)$$

Note that in our earlier definition, we had $T^0(I_\perp), P \models^+ A$ iff $A \in (D)$. Here we have $S(I_\perp), P \models^+ A$ iff $A \in (D)$. No confusion should arise from this, as we are mainly concerned with $T^\omega(I_\perp)$ and $S^\omega(I_\perp)$.

Below we show that S is indeed a mapping from interpretations to interpretations.

Lemma 5.5.2 *Let I be an interpretation. Then $S(I)$ is an interpretation.*

Proof: Let P be an arbitrary D_{HHF-} derivation state.

$A \in \text{pos}(S(I))(P) \Rightarrow A \in (D)$ or $\exists G \supset A \in (D)$ such that $I, P \models^{+} G$.

$A \in \text{neg}(S(I))(P) \Rightarrow \forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$, $I, P \models^{-} G$.

Hence, it is clear that $\text{inst}(\text{pos}(S(I))(P)) \cap \text{inst}(\text{pos}(S(I))(P)) = \emptyset$.

Let $P_1 = \langle D_1, N_1 \rangle$ and $P_2 = \langle D_2, N_2 \rangle$ be D_{HHF-} programs. Let $P_1 \leq P_2$. If $A \in \text{pos}(S(I))(P_1)$, then either $A \in (D_1)$ and hence $A \in (D_2)$, or $\exists G \supset A \in (D_1)$ (and hence $G \supset A \in (D_2)$) such that $I, P_1 \models^{+} G$, and by lemma 5.3.1 we have $I, P_2 \models^{+} G$, i.e. $A \in \text{pos}(S(I))(P_2)$.

If $A \in \text{neg}(S(I))(P_1)$, then $\text{name}(A) \in \text{den}(N_1)$, $\forall B \in (D_1)$, $B \not\propto A$, and $\forall G \supset B \in (D_1)$ such that $B \propto A$, $I, P_1 \models^{-} G$, and by lemma 5.3.1, $I, P_2 \models^{-} G$. Now as $P_1 \leq P_2$, we have $\forall C \in (D_2) \backslash (D_1)$, $\text{name}(\text{head}(C)) \in \text{ass}(N_1)$ and as $\text{ass}(N_1) \subseteq \text{ass}(N_2)$, we have $\text{name}(A) \in \text{den}(N_2)$ and $\text{name}(\text{head}(C)) \notin \text{den}(N_1)$. Hence we have that $\forall B \in (D_2)$, $B \not\propto A$ and $\forall G \supset B \in (D_2)$ such that $B \propto A$, we have $I, P_2 \models^{-} G$, and so $A \in \text{neg}(S(I))(P_2)$.

Hence $S(I)$ is an interpretation.

□

Next we show that S is monotonic and continuous, as claimed above.

Lemma 5.5.3 *Let I_1 and I_2 be interpretations. Then*

$$I_1 \sqsubseteq I_2 \Rightarrow S(I_1) \sqsubseteq S(I_2)$$

Proof: Let P be a D_{HHF-} derivation state.

If $A \in \text{pos}(S(I_1))(P)$, then $A \in (D)$ or $\exists G \supset A \in (D)$ such that $I_1, P \models^+ G$, and by lemma 5.3.2 we have that $I_2, P \models^+ G$, and so $A \in \text{pos}(S(I_2))(P)$.

If $A \in \text{neg}(S(I_1))(P)$, then $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$, $I_1, P \models^- G$, and by lemma 5.3.2 we have that $I_2, P \models^- G$, and so $A \in \text{neg}(S(I_2))(P)$.

Hence $S(I_1) \sqsubseteq S(I_2)$.

□

Lemma 5.5.4 *Let $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$ be an increasing sequence of interpretations. Then*

$$\sqcup_{i=1}^{\infty} S(I_i) = S(\sqcup_{i=1}^{\infty} I_i)$$

Proof: $\sqsubseteq$: $I_j \sqsubseteq \sqcup_{i=1}^{\infty} I_i$, and so by lemma 5.5.3, we have that $S(I_j) \sqsubseteq S(\sqcup_{i=1}^{\infty} I_i)$ for any j, and hence $\sqcup_{i=1}^{\infty} S(I_i) \sqsubseteq S(\sqcup_{i=1}^{\infty} I_i)$.

$\sqsupseteq$: If $A \in \text{pos}(S(\sqcup_{i=1}^{\infty} I_i))(P)$, then either $A \in (D)$, in which case $A \in S(I_j)(P)$ for all $j \geq 0$, or $\exists G \supset A \in (D)$ such that $\sqcup_{i=1}^{\infty} I_i, P \models^+ G$, which by lemma 5.3.9 implies that $I_j, P \models^+ G$ for some $j \geq 0$, and so $A \in \text{pos}(S(I_j))(P)$. In either case we have $A \in \text{pos}(\sqcup_{i=1}^{\infty} S(I_i))(P)$.

If $A \in \text{neg}(S(\sqcup_{i=1}^{\infty} I_i))(P)$, then $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)$ $B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$, $\sqcup_{i=1}^{\infty} I_i, P \models^- G$, which by lemma 5.3.9 implies that $I_j, P \models^- G$ for some $j \geq 0$, and so $A \in \text{neg}(S(I_j))(P)$. Hence $A \in \text{neg}(\sqcup_{i=1}^{\infty} S(I_i))(P)$.

Hence, $S(\sqcup_{i=1}^{\infty} I_i) \sqsubseteq \sqcup_{i=1}^{\infty} S(I_i)$.

Hence $S(\bigsqcup_{i=1}^{\infty} I_i) = \bigsqcup_{i=1}^{\infty} S(I_i)$.

□

Next we show that S and T define the same sequence of interpretations.

Proposition 5.5.5 *Let the interpretations $S^i(I_\perp)$ and $T^j(I_\perp)$ be as defined above. Then for any $k \geq 0$*

$$T^k(I_\perp) = S^{k+1}(I_\perp)$$

Proof: We proceed by induction on k. Let $P = \langle D, N\rangle$ be an arbitrary D_{HHF}-derivation state.

For the base case, it is clear that $A \in \text{pos}(T^0(I_\perp))(P)$ iff $A \in (D)$ iff $A \in \text{pos}(S(I_\perp))(P)$. Similarly (as there is no goal G such that $I_\perp \models^- G$), we have that $A \in \text{neg}(T^0(I_\perp))(P)$ iff $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)\ B \not\propto A$ iff $A \in \text{neg}(S(I_\perp))(P)$.

Hence the induction hypothesis is that for all $0 \leq k \leq n$, $T^k(I_\perp) = S^{k+1}(I_\perp)$. Consider $T^{n+1}(I_\perp)$ and $S^{n+2}(I_\perp)$.

$A \in \text{pos}(T^{n+1}(I_\perp))(P)$ iff $A \in \text{pos}(T^n(I_\perp))(P)$ or $\exists G \supset A \in (D)$ such that $T^n(I_\perp), P \models^+ G$, and by the hypothesis this is equivalent to $A \in \text{pos}(S^{n+1}(I_\perp))(P)$ or $\exists G \supset A \in (D)$ such that $S^{n+1}(I_\perp), P \models^+ G$, and as $S^{n+1}(I_\perp) \sqsubseteq S^{n+2}(I_\perp)$, in either case we get that $A \in \text{pos}(S^{n+2}(I_\perp))(P)$.

$A \in \text{pos}(S^{n+2}(I_\perp))(P)$ implies that $A \in (D)$ or $\exists G \supset A \in (D)$ such that $S^{n+1}(I_\perp), P \models^+ G$. If $A \in (D)$, then $A \in \text{pos}(T^0(I_\perp))(P)$, and so by lemma 5.3.2, we have that $A \in \text{pos}(T^{n+1}(I_\perp))(P)$. Otherwise, by the hypothesis we have $\exists G \supset A \in (D)$ such that $T^n(I_\perp), P \models^+ G$, and so $A \in \text{pos}(T^{n+1}(I_\perp))(P)$.

$A \in \text{neg}(T^{n+1}(I_\perp))(P)$ iff $A \in \text{neg}(T^n(I_\perp))(P)$ or $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$ we have $T^n(I_\perp), P \models^- G$, which by the hypothesis is equivalent to $A \in \text{neg}(S^{n+1}(I_\perp))(P)$ or $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$ we have $S^{n+1}(I_\perp), P \models^- G$, and as $S^{n+1}(I_\perp) \sqsubseteq S^{n+2}(I_\perp)$, in either case we have $A \in \text{neg}(S^{n+2}(I_\perp))(P)$.

$A \in \text{neg}(S^{n+2}(I_\perp))(P)$ iff $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$ we have $S^{n+1}(I_\perp), P \models^- G$, and by the hypothesis, this implies that $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$ we have $T^n(I_\perp), P \models^- G$, which implies that $A \in \text{neg}(T^{n+1}(I_\perp))(P)$.

□

It follows directly from this lemma that the two interpretations $S^\omega(I_\perp)$ and $T^\omega(I_\perp)$ are the same.

Corollary 5.5.6 *Let $S^\omega(I)$ and $T^\omega(I)$ be defined as above. Then*

$$S^\omega(I_\perp) = T^\omega(I_\perp)$$

It is also clear that from proposition 5.5.5 and lemma 5.3.2 that $S^i(I_\perp), P \models^+ G \Rightarrow S^j(I_\perp), P \models^+ G$ for any $j \geq i$ and hence $S^i(I_\perp), P \models^+ G \Rightarrow S^\omega(I_\perp), P \models^+ G$. Similarly, $S^i(I_\perp), P \models^- G \Rightarrow S^j(I_\perp), P \models^- G$ for any $j \geq i$ and hence $S^i(I_\perp), P \models^- G \Rightarrow S^\omega(I_\perp), P \models^- G$.

As mentioned above, it follows from the monotonicity of S and the fact that $\sqsubseteq$ is a chain-complete partial order that the least fixed point of S is $S^\omega(I_\perp)$ [1, 60], which by the above corollary is the same as $T^\omega(I_\perp)$. However it is not hard to establish the same result directly, and as it is somewhat informative, we do so below.

Lemma 5.5.7 *Let S be defined as above, and so*

$$S^\omega(I_\perp) = \bigsqcup_{i=1}^{\infty} S^i(I_\perp) = S^1(I_\perp) \sqcup S^2(I_\perp) \sqcup S^3(I_\perp) \sqcup \ldots$$

Then $S^\omega(I_\perp)$ is a fixed point of S, i.e. $S(S^\omega(I_\perp)) = S^\omega(I_\perp)$.

Proof: Let $P = \langle D, N \rangle$ be any D_{HHF-} derivation state. By propositions 5.5.1 and 5.5.5, we know that $S^\omega(I_\perp), P \models^+ A \Leftrightarrow A \in (D)$ or $\exists G \supset A \in (D)$ such that $S^\omega(I_\perp), P \models^+ A$, which is equivalent to $S(S^\omega(I_\perp)) \models^+ A$.

By the same propositions we have $S^\omega(I_\perp), P \models^- A$ iff $\text{name}(A) \in \text{den}(N)$ and $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$, $S^\omega(I_\perp), P \models^- G$, which is equivalent to $S(S^\omega(I_\perp)), P \models^- A$.

Hence we have that $S^\omega(I_\perp) = S(S^\omega(I))$, i.e. that $S^\omega(I_\perp)$ is a fixed point of S.

□

Next we show that $S^\omega(I_\perp)$ is the least fixed point of S, so that if J is any fixed point of S, then $S^\omega(I_\perp) \sqsubseteq J$.

The key lemma needed to prove that $S^\omega(I_\perp) \sqsubseteq J$ is given below.

Lemma 5.5.8 *Let J be any fixed point of S. Then for any $i \geq 1$, $S^i(I_\perp) \sqsubseteq J$.*

Proof: Clearly $I_\perp \sqsubseteq J$, and as S is monotonic, we have $S(I_\perp) \sqsubseteq S(J)$, and so $S^i(I_\perp) \sqsubseteq S^i(J)$.

Now as J is a fixed point of S, we have that $J = S(J) = S^2(J) = \ldots = S^i(J)$, and so $S^i(I_\perp) \sqsubseteq J$. □

From this lemma we may easily derive the desired result, given below.

Theorem 5.5.9 *$S^\omega(I_\perp)$ is the least fixed point of S.*

Proof: We already have that $S^\omega(I_\perp)$ is a fixed point of S from lemma 5.5.7, so let J be any fixed point of S. By lemma 5.3.2 it will be sufficient to show that

1. $S^\omega(I_\perp), P \models^+ A \Rightarrow J, P \models^+ A$
2. $S^\omega(I_\perp), P \models^- A \Rightarrow J, P \models^- A$

Now by lemma 5.3.9 we have that $S^\omega(I_\perp), P \models^+ A$ implies that $S^k(I_\perp), P \models^+ A$ for some $k \geq 1$, and so as $S^k(I_\perp) \sqsubseteq J$ by lemma 5.5.8, by lemma 5.3.2 we have $J, P \models^+ A$.

Similarly, by lemma 5.3.9 we have that $S^\omega(I_\perp), P \models^- A$ implies that $S^k(I_\perp), P \models^- A$ for some $k \geq 1$, and so as $S^k(I_\perp) \sqsubseteq J$ by lemma 5.5.8, by lemma 5.3.2 we have $J, P \models^- A$.

Hence we have that

1. $S^\omega(I_\perp), P \models^+ A \Rightarrow J, P \models^+ A$
2. $S^\omega(I_\perp), P \models^- A \Rightarrow J, P \models^- A$

for any D_{HHF-} derivation state P, and so $S^{\omega}(I_{\perp}) \sqsubseteq J$, where J is any fixed point of S, and so $S^{\omega}(I_{\perp})$ is the least fixed point of S.

□

Thus even though our partial order does not form a lattice, our operator is continuous and $S^{\omega}(I_{\perp})$ is its least fixed point. Whilst we prefer to think of the powers of T without reference to arbitrary interpretations, this result may be seen as evidence that our approach does not stray too wildly from the traditional methods.

5.6 Problems with Inconsistency

The above results were obtained for the following class of programs:

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid \exists x G \mid \forall x G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G$$

A natural extension to this class of programs is the one given below.

$$L := A \mid \neg A$$
$$D := L \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset L$$
$$G := L \mid \exists x G \mid \forall x G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G$$

This class of programs is an obvious generalisation of the class given in [77] in that programs and goals are built up from literals rather than atoms, and so positive and negative information is treated symmetrically. This may thought of as allowing "negation in the head", in that clause heads are now literals rather than just atoms.

As discussed in chapter 2, an obvious problem with such an extension is the problem of inconsistency, and we cannot dodge this difficulty by restricting our attention to consistent programs because of the similarity between

$$\neg A \vdash_s A \supset G$$

and

$$\neg A \wedge A \vdash_s G$$

In this section we discuss the possibility of allowing programs to be inconsistent, and show briefly how our framework is not really suitable for this purpose.

As in [77] and chapters 2 and 3, we assume that there is a distinguished atom $\perp$ which stands for a contradiction and is computed in the same way as any other atom.

As mentioned above (and discussed in [77]), there is a choice to be made — are inconsistencies considered global, as in intuitionistic logic, so that if $P \vdash_s \perp$ then $P \vdash_s G$ for any goal G, or are they local, in that when $P \vdash_s A \wedge \neg A$, we wish to deal with the inconsistency rather than continue to make deductions from the program? Whichever approach is taken causes significant technical problems for our framework.

If we take the approach of intuitionistic logic, so that if $P \vdash_s \perp$ then $P \vdash_s G$ for any goal G, then we would wish this to be reflected in the model theory, i.e. that if $T^\omega(I_\perp), P \models^+ \perp$ then $T^\omega(I_\perp), P \models^+ G$ for any goal G. However, our definition of an interpretation makes this impossible, as for any interpretation I, we must have that $\text{inst}(\text{pos}(I)(P)) \cap \text{inst}(\text{neg}(I)(P)) = \emptyset$, and so we cannot have $I, P \models^+ A \wedge \neg A$, as there can be no atom A such that $A \prec \text{pos}(I)(P)$ and $A \prec \text{neg}(I)(P)$. So the first thing to do is to generalise the notion of an interpretation so that we may have *inconsistent* interpretations. This would be done by dropping the condition of internal consistency, so that the definition of an interpretation would become

An interpretation is any function $I : \mathcal{P} \to \mathcal{H}' \times \mathcal{H}'$ such that for all worlds w_1, w_2 for which $w_1 \leq w_2$ and $I(w_i) = \langle S_i, F_i \rangle$, $i = 1, 2$ then $S_1 \prec S_2$ and $F_1 \prec F_2$.

This more general notion of an interpretation will allow us to consider the desired interpretation above. In fact, we really only desire one extra interpretation:

this is the interpretation $I^{\top}$ where $I^{\top}(P) = \langle \mathcal{H}', \mathcal{H}' \rangle$ for any program P. This may be thought of as a dual to the null interpretation $I_{\perp}$ which is such that $I_{\perp}(P) = \langle \emptyset, \emptyset \rangle$, and so we think of this as "topping" the lattice of interpretations. In this way it is conceivable that we may model inconsistent programs by setting $T^{\omega}(I_{\perp})(P) = I^{\top}(P)$ whenever P is an inconsistent program.

One complication for an inconsistent program $\langle D, N \rangle$ is introduced by the necessity that if every goal is provable, then every goal of the form $D' \supset G'$ must be provable, so that

$$T^{\omega}(I_{\perp}), \langle D, N \rangle \models^{+} D' \supset G'$$

for all choices of D' and G'. In order to obey the rules of $\models^{+}$, we must have that $\langle D \cup \{D'\}, N \rangle \geq \langle D, N \rangle$ for *any* definite formula D', including clauses whose heads have names in common with $\mathrm{den}(N)$, i.e. clauses which extend the completely defined predicates of the program. Hence, for our relation between worlds to have any meaning,[1] either we must separate the inconsistent worlds from the consistent ones or we must have that $\mathrm{den}(N) = \emptyset$ and $\mathrm{ass}(N) = \emptyset$ for inconsistent programs. This per se may not seem a bad thing; after all no goal can fail when the program is inconsistent, and so the notion of completely defined predicates seems out of place. However, this may cause problems when a consistent program is inconsistently extended.

For example, let $P = \langle D, N \rangle$ be a consistent program with $\mathrm{den}(N) \neq \emptyset$, and A be an atom whose name does not occur anywhere in D and $\mathrm{name}(A) \in \mathrm{ass}(N)$. Now according to the accessibility relation between worlds given above, both $\langle D \cup \{A\}, N \rangle$ and $\langle D \cup \{\neg A\}, N \rangle$ are accessible from $\langle D, N \rangle$, as both are consistent with $\langle D, N \rangle$. However, the inconsistent program $\langle D \cup \{A\} \cup \{\neg A\}, \langle \emptyset, \emptyset \rangle \rangle$ is accessible from neither world. This may be seen to be reasonable, in that the choice made at the world $\langle D, N \rangle$ needs to distinguish between two mutually inconsistent alternatives, and so the inconsistent world should not be accessible. This

[1]Recall that this proposed extension must be conservative in that the previous results on consistent programs must still hold.

view then leads us to the conclusion that there is are really two lattices of worlds — one of consistent worlds and one of inconsistent worlds, and no inconsistent world is accessible from any consistent world, with the exception that the bottom world $\langle\emptyset, \langle\emptyset, \emptyset\rangle\rangle$ can access both consistent and inconsistent worlds. As there seems to be two distinct lattices, it seems more natural to separate the consistent programs from the inconsistent ones more formally, which is basically what was done above when we assumed that all programs were consistent.

The main objection to be made against this scheme is that it moves very much away from the close association between extensions of the program and accessibility between worlds. If we imagine the programming process as commencing at the empty world and progressing up the "cone" of worlds, it seems natural to consider an inconsistent extension to a program in the same context as a consistent extension. Naturally we will want to distinguish between the two, as we would normally think of an inconsistent extension as a dead end from which we need to backtrack so that a different choice can be made somewhere further down and the programming process resumed, hopefully terminating at a consistent program. In order for such a process to work, it is necessary that inconsistent programs be accessible from consistent ones, and so it is difficult to see how the framework given above can be made useful for such a scheme.

One possible modification that may be helpful is to take a minimal approach to inconsistency, rather than the full intuitionistic one. As discussed in [77], programming considerations suggest that the minimal approach is more appropriate, as it places the emphasis on detection of inconsistency, rather than trying to make sense of an inconsistent program. Also, as discussed in chapter 2, it is difficult to see how an inconsistency in the definition of the append predicate should force carcinogen(chocolate) $\wedge$ $\neg$carcinogen(chocolate) to be true. Again the notion of interpretation would need to be extended in the manner described above, but this time there will not just be one inconsistent interpretation of interest, but many. A related question is how to interpret $I, P \models^{+} A$ when $A \in \mathrm{pos}(I)(P) \cap \mathrm{neg}(I)(P)$. One answer may be to modify the rules for $I, P \models^{+} A$ and $I, P \models^{-} A$ as follows:

1. $I, P \models^{+} A$ iff $A \in \text{inst}(\text{pos}(I)(P)) \backslash \text{inst}(\text{neg}(I)(P))$

2. $I, P \models^{-} A$ iff $A \in \text{inst}(\text{pos}(I)(P)) \backslash \text{inst}(\text{neg}(I)(P))$

The other rules would be left the same, with the exception of the rule for goals of the form $D' \supset G'$, which would take the simpler form

$$I, \langle D, N \rangle \models^{+} D' \supset G' \text{ iff } I, \langle D \cup \{D'\}, N \rangle \models^{+} G'$$

This has the effect of pushing the consistency check inwards, in that $\models^{+}$ can only prove things from the consistent part of the program, and so whilst $\langle D \cup \{D'\}, N \rangle$ may be an inconsistent program, G' may still be provable from it in the above sense, in that the inconsistency may not "affect" G'. In this way the emphasis is on whether G' depends on the inconsistency (if any) or not, rather than on whether the extended program is consistent.

One obvious "safe" but rather uninteresting way to avoid such problems is to restrict the class of goals as follows:

$$G := A \mid \neg A \mid \exists x G \mid \forall x G \mid G_1 \vee G_2 \mid G_2 \wedge G_2$$

so that no implications can occur in goals. Less drastic restrictions are obviously desirable, but a full answer can only really be given by dealing with inconsistent programs in a manner similar to that of consistent ones, so that the model theory for both sorts of programs may be integrated.

Chapter 6

Semantic Properties of Hereditary Harrop Formulae

In this chapter we consider some semantic properties of first-order hereditary Harrop formulae. One important question is the precise strength of the class of formulae involved. In order to investigate this, we consider the redundant features of the language of [77], and show how they may be removed. This leads to a discussion of equivalence for this class of programs, and we show how intuitionistic logic is not quite strong enough for questions of equivalence, and we develop notions of equivalence for goals and programs based on a slightly stronger (but not classical) logic.

6.1 Structural Properties of Programs

In [63] it was shown how arbitrary first-order formulae may be converted into sets of D_{Horn-} and G_{Horn-} formulae, where equivalence is interpreted in classical logic. This means the Horn clauses may be thought of as a normal form for formulae of first-order classical logic, and so an interpreter for D_{Horn-} programs and G_{Horn-} goals may use this translation to interpret programs and goals of full first-order classical logic. This conversion relies on the fact that it is possible to define the connectives $\forall$, $\vee$ and $\supset$ in terms of $\exists$, $\wedge$ and $\neg$ in classical logic, and so once the latter trio have been implemented, as in G_{Horn-}, then the other three may be

implemented without extending the underlying programming engine. In this way first-order classical logic may be used for programming, rather than just D_{Horn-}.

This approach is not possible in intuitionistic logic, as the equivalences needed to prove the correctness of the interdefinability results are not intuitionistically valid. Given that intuitionistic logic seems to be a better context in which to interpret hereditary Harrop formulae, we may conclude that the approach using classical logic may be seen as more significant for automated theorem proving than for the semantics of logic programming and extensions to Horn clauses.

This, of course, does not mean that normal forms cannot exist for hereditary Harrop formula, when interpreted in intuitionistic logic. We may think of the translation of [63] as exploiting the redundancy of the connectives in classical logic. Here we examine what redundancies there may be in hereditary Harrop formula, in order to derive some kind of normal form.

An example of this redundancy may be given by the question of quantification in Horn clauses. We may consider all variables which appear in a Horn clause as being universally quantified at the front of the clause, e.g.

$$\forall x \forall y \; q(x,y) \supset p(x)$$

However, we may also consider variables which do not appear in the head of the clause as being existentially quantified at the front of the *body*, so that the above clause is equivalent to

$$\forall x \, (\exists y q(x,y)) \supset p(x)$$

Thus we may consider Horn clauses to be defined in the latter style (i.e. in which existential quantifiers are allowed in the bodies of clauses), implement the former language (i.e. in which existential quantifiers are not allowed in the bodies of clauses) and use the equivalence of the former and latter clauses to allow program of either definition to be used.

We can generalise this line of thought to a more general class of programs provided that we can perform a similar process of removing $\exists$ and $\vee$ from the

bodies of clauses. One step in this direction was given in [77], in which it was shown how disjunctions may be removed from D_{mod} programs. This result was established by showing that for any program which contains a disjunction, there is a program with the same operational behaviour, which, although usually much larger, contains no disjunctions. We may think of this as exploiting the structure of the program so that programs may be given a simpler definition, in that there is a smaller class of formulae with the same expressive power. We extend this result to the case when negations may be present, and also show how existential quantifiers may be similarly removed from programs. We may think of the transformed program as a normal form.

In this section, we will be considering D_{mod-} and G_{mod-} formulae, i.e. D formulae and G formulae of the form

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid \exists x G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G$$

Using D_{HHF-} and G_{HHF-} formulae present some technical difficulties which are discussed in section 7.2.

The process of removing disjunctions is inspired by the intuitionistic equivalence

$$(G_1 \vee G_2) \supset A \equiv_I (G_1 \supset A) \wedge (G_2 \supset A)$$

A similar process for removing existential quantifiers is specified by the corresponding equivalence

$$(\exists x G) \supset A \equiv_I \forall x (G \supset A)$$

where x does not appear in A.

The first of these two equivalences is proved in Appendix A, and the second is given in [51].

The formal definition of both of these processes are given below.

As in [77], dnf and dfnf denote respectively the *disjunctive normal form* of a goal, and the *disjunction-free normal form* of a program. The definitions in [77] are given in a slightly modified form here, so that we may consider dnf(G) as a formula, rather than a set. We introduce along similar lines the *existential normal form* of a goal and the *existential-free normal form* of a program, here denoted by enf and efnf respectively. We will assume that each existentially quantified variable is unique, so that a goal such as $(\exists x\, p(x)) \wedge (\exists x\, q(x))$ is written as $\exists x\, p(x) \wedge \exists y\, q(y)$.

Definition 6.1.1 *Let D be a D_{mod-} formula, G be a G_{mod-} goal formula, and $\tilde{x}$ be all the existentially quantified variables of G. Then we define* dfnf(D), efnf(D), dnf(G) *and* enf(G) *as follows:*

$$
\begin{aligned}
\mathrm{dnf}(G) &= \bigvee \mathrm{dnf}'(G) \\
\mathrm{dnf}'(A) &= \{A\} \\
\mathrm{dnf}'(\neg A) &= \{\neg A\} \\
\mathrm{dnf}'(G_1 \vee G_2) &= \mathrm{dnf}'(G_1) \cup \mathrm{dnf}'(G_2) \\
\mathrm{dnf}'(G_1 \wedge G_2) &= \{G' \wedge G'' \mid G' \in \mathrm{dnf}'(G_1), G'' \in \mathrm{dnf}'(G_2)\} \\
\mathrm{dnf}'(\exists x G) &= \{\exists x G' \mid G' \in \mathrm{dnf}'(G)\} \\
\mathrm{dnf}'(D \supset G) &= \{\mathrm{dfnf}(D) \supset G' \mid G' \in \mathrm{dnf}'(G)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{dfnf}(A) &= A \\
\mathrm{dfnf}(G \supset A) &= \bigwedge \{G' \supset A \mid G' \in \mathrm{dnf}'(G)\} \\
\mathrm{dfnf}(D_1 \wedge D_2) &= \mathrm{dfnf}(D_1) \wedge \mathrm{dfnf}(D_2) \\
\mathrm{dfnf}(\forall x D) &= \forall x\ \mathrm{dfnf}(D)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{enf}(G) &= \exists \tilde{x}\ \mathrm{enf}'(G) \\
\mathrm{enf}'(A) &= A \\
\mathrm{enf}'(\neg A) &= \neg A \\
\mathrm{enf}'(G_1 \vee G_2) &= \mathrm{enf}'(G_1) \vee \mathrm{enf}'(G_2) \\
\mathrm{enf}'(G_1 \wedge G_2) &= \mathrm{enf}'(G_1) \wedge \mathrm{enf}'(G_2) \\
\mathrm{enf}'(\exists x G) &= \mathrm{enf}'(G) \\
\mathrm{enf}'(D \supset G) &= \mathrm{efnf}(D) \supset \mathrm{enf}'(G)
\end{aligned}
$$

$$\begin{aligned}
\text{efnf}(A) &= A \\
\text{efnf}(G \supset A) &= \forall \tilde{x}\ (\text{enf}'(G) \supset A) \\
\text{efnf}(\forall x D) &= \forall x\ \text{efnf}(D) \\
\text{efnf}(D_1 \wedge D_2) &= \text{efnf}(D_1) \wedge \text{efnf}(D_2)
\end{aligned}$$

We define $\text{dfnf}(\{D_1, \ldots D_n\}) = \cup_{i=1}^{n} \text{dfnf}(D_i)$, *and* $\text{efnf}(\{D_1, \ldots D_n\}) = \cup_{i=1}^{n} \text{efnf}(D_i)$.

We may think of the process described by dnf(G) as pushing all disjunctions to the top level of the goal, so that dnf(G) is a disjunction of disjunction-free formulae. We may then apply the identity above to dnf(G) $\supset A$ to obtain the disjunction-free program. A similar remark applies to enf(G); all existential quantifiers are pushed to the top of the formula, and so we may apply the corresponding identity to enf(G) $\supset A$ to obtain a program free of existential quantifications. Thus the above equivalences imply that

$$\text{dfnf}(G \supset A) = \bigvee\{G' \supset A \mid G' \in \text{dnf}'(G)\} \equiv_I \text{dnf}(G) \supset A$$
$$\text{efnf}(G \supset A) = \forall \tilde{x}(\text{enf}'(G \supset A)) \equiv_I \text{enf}(G) \supset A$$

where $\tilde{x}$ are all the existentially quantified variables of G.

In this way we may specify the program equivalent to D which contains no occurrences of $\exists$ or $\vee$ as efnf(dfnf(D)). Note that there may be a slight syntactic difference between efnf(dfnf(D)) and dfnf(efnf(D)). For example, consider the program D below.

$$r \subset (\exists x\, p(x) \vee \exists y\, q(y))$$

It is easy to see that dfnf(D) and efnf(D) are just

dfnf(D)	efnf(D)
$\exists x\, p(x) \supset r \wedge \exists x y p(y) \supset r$	$\forall x \forall y\ (p(x) \vee p(y)) \supset r$

and so efnf(dfnf(D)) and dfnf(efnf(D)) are as follows:

efnf(dfnf(D))	dfnf(efnf(D))
$(\forall x\ p(x) \supset r) \wedge (\forall y\ q(y) \supset r)$	$\forall x \forall y\ (p(x) \supset r) \wedge (q(y) \supset r)$

Note also that the two processes only alter the bodies of clauses, possibly replacing a rule by several rules, and do nothing to facts. This is formally stated in the lemma below.

Lemma 6.1.1 *Let P be a D_{mod-} derivation state. Then*

1. $A \in (D) \Leftrightarrow A \in (\mathrm{dfnf}(D)) \Leftrightarrow A \in (\mathrm{efnf}(D))$
2. $G \supset A \in (D) \Leftrightarrow G_i \supset A \in (\mathrm{dfnf}(D))$ *for all* $i \Leftrightarrow G' \supset A \in (\mathrm{efnf}(D))$

   where $dnf(G) = G_1 \vee \ldots \vee G_n$ *and* $enf(G) = \exists \tilde{x} G'$

Proof: Obvious. □

It is easy to see from the definitions that

$$\mathrm{dnf}(G_1 \vee G_2) = \mathrm{dnf}(G_1) \vee \mathrm{dnf}(G_2)$$

and that

$$\mathrm{dnf}(G_1 \wedge G_2) = \bigvee\{G' \wedge G'' \mid G' \in \mathrm{dnf}'(G_1), G'' \in \mathrm{dnf}'(G_2)\}$$

Similarly it is clear that

$$\mathrm{enf}(D \supset \exists x G) = \exists x\ \mathrm{enf}(D \supset G)$$

This equivalences will be useful in the proofs below.

The formal results which establish the operational properties of D, dfnf(D) and efnf(D) are given below.

Proposition 6.1.2 *Let $\langle D, N\rangle$ be a D_{mod-} derivation state, and let G be a G_{mod-} goal formula. Then*

1. $\langle D, N\rangle \vdash_s G \Leftrightarrow \langle \mathrm{dfnf}(D), N\rangle \vdash_s G \Leftrightarrow \langle D, N\rangle \vdash_s \mathrm{dnf}(G)$

2. $\langle D, N\rangle \vdash_f G \Leftrightarrow \langle \mathrm{dfnf}(D), N\rangle \vdash_f G \Leftrightarrow \langle D, N\rangle \vdash_f \mathrm{dnf}(G)$

Proof: We proceed by induction on the depth of the **O**-derivation of G.

In the base case G is an atom A. It is then easy to see that both 1 and 2 follow, as $\mathrm{dnf}(A) = A$ and $A \in (D) \Leftrightarrow A \in \mathrm{dfnf}(D)$. A similar argument establishes 2 in this case.

Hence we assume that 1 and 2 are true for all programs and for all goals whose **O**-derivation is less than a given size. There are six cases:

A: 1. As $\mathrm{dnf}(A) = A$, it suffices to show that $\langle D, N\rangle \vdash_s A \Leftrightarrow \langle \mathrm{dfnf}(D), N\rangle \vdash_s A$. Now if the base case does not hold, then $\langle D, N\rangle \vdash_s A$ iff $\exists G \supset A \in (D)$ such that $\langle D, N\rangle \vdash_s G$ and by the hypothesis this is equivalent to $\langle \mathrm{dfnf}(D), N\rangle \vdash_s \mathrm{dnf}(G)$, i.e. $\langle \mathrm{dfnf}(D), N\rangle \vdash_s G_1 \vee \ldots \vee G_n$, which is just $\langle \mathrm{dfnf}(D), N\rangle \vdash_s G_i$ for some i. Now as $G \supset A \in (D)$ iff $G_i \supset A \in (\mathrm{dfnf}(D))$ for all $1 \leq i \leq n$ where $\mathrm{dnf}(G) = G_1 \vee \ldots \vee G_n$, this is equivalent to $\exists G' \supset A \in \mathrm{dfnf}(D)$ such that $\langle \mathrm{dfnf}(D), N\rangle \vdash_s G'$ which is equivalent to $\langle \mathrm{dfnf}(D), N\rangle \vdash_s A$.

2. As $\mathrm{dnf}(A) = A$, it suffices to show that $\langle D, N\rangle \vdash_f A \Leftrightarrow \langle \mathrm{dfnf}(D), N\rangle \vdash_f A$. Now if the base case does not hold, then $\langle D, N\rangle \vdash_f A$ iff $\forall B \in (D)\ B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$ we have $\langle D, N\rangle \vdash_f G$ and by the hypothesis this is equivalent to $\langle \mathrm{dfnf}(D), N\rangle \vdash_f \mathrm{dnf}(G)$, i.e. $\langle \mathrm{dfnf}(D), N\rangle \vdash_f G_1 \vee \ldots \vee G_n$, which is just $\langle \mathrm{dfnf}(D), N\rangle \vdash_f G_i$ for all i. Now as $G \supset A \in (D)$ iff $G_i \supset A \in (\mathrm{dfnf}(D))$ for all $1 \leq i \leq n$ where $\mathrm{dnf}(G) = G_1 \vee \ldots \vee G_n$, this is equivalent to $\forall B' \in (\mathrm{dfnf}(D))\ B' \not\propto A$ and $\forall G' \supset B' \in (\mathrm{dfnf}(D))$ such that $B' \propto A$ we have $\langle \mathrm{dfnf}(D), N\rangle \vdash_f G'$, which is equivalent to $\langle \mathrm{dfnf}(D), N\rangle \vdash_f A$.

$\neg A$: 1. As $\mathrm{dnf}(\neg A) = \neg A$, it suffices to show that $\langle D, N\rangle \vdash_s \neg A \Leftrightarrow \langle \mathrm{dnf}(D), N\rangle \vdash_s \neg A$. Now $\langle D, N\rangle \vdash_s \neg A$ iff $\mathrm{name}(A) \in \mathrm{den}(N)$ and $\langle D, N\rangle \vdash_f A$, and by the hypothesis this is equivalent to $\langle \mathrm{dnf}(D), N\rangle \vdash_f A$, which is just $\langle \mathrm{dnf}(D), N\rangle \vdash_s \neg A$.

2. As $\mathrm{dnf}(\neg A) = \neg A$, it suffices to show that $\langle D, N\rangle \vdash_f \neg A \Leftrightarrow \langle \mathrm{dnf}(D), N\rangle \vdash_f \neg A$. Now $\langle D, N\rangle \vdash_f \neg A$ iff $\langle D, N\rangle \vdash_s A$, and by the hypothesis this is equivalent to $\langle \mathrm{dnf}(D), N\rangle \vdash_s A$, which is just $\langle \mathrm{dnf}(D), N\rangle \vdash_f \neg A$.

$G_1 \vee G_2$: 1. $\langle D, N\rangle \vdash_s G_1 \vee G_2$ iff $\langle D, N\rangle \vdash_s G_1$ or $\langle D, N\rangle \vdash_s G_2$.
By the hypothesis this is just $\langle \mathrm{dnf}(D), N\rangle \vdash_s G_1$ or $\langle \mathrm{dnf}(D), N\rangle \vdash_s G_2$, which in turn is just $\langle \mathrm{dnf}(D), N\rangle \vdash_s G_1 \vee G_2$.
From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_s \mathrm{dnf}(G_1)$ or $\langle D, N\rangle \vdash_s \mathrm{dnf}(G_2)$ which is just $\langle D, N\rangle \vdash_s \mathrm{dnf}(G_1) \vee \mathrm{dnf}(G_2)$, i.e. $\langle D, N\rangle \vdash_s \mathrm{dnf}(G_1 \vee G_2)$.

2. $\langle D, N\rangle \vdash_f G_1 \vee G_2$ iff $\langle D, N\rangle \vdash_f G_1$ and $\langle D, N\rangle \vdash_f G_2$.
By the hypothesis this is just $\langle \mathrm{dnf}(D), N\rangle \vdash_f G_1$ and $\langle \mathrm{dnf}(D), N\rangle \vdash_f G_2$, which in turn is just $\langle \mathrm{dnf}(D), N\rangle \vdash_f G_1 \vee G_2$.
From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_f \mathrm{dnf}(G_1)$ and $\langle D, N\rangle \vdash_f \mathrm{dnf}(G_2)$ which is just $\langle D, N\rangle \vdash_f \mathrm{dnf}(G_1) \vee \mathrm{dnf}(G_2)$, i.e. $\langle D, N\rangle \vdash_f \mathrm{dnf}(G_1 \vee G_2)$.

$G_1 \wedge G_2$: 1. $\langle D, N\rangle \vdash_s G_1 \wedge G_2$ iff $\langle D, N\rangle \vdash_s G_1$ and $\langle D, N\rangle \vdash_s G_2$.
By the hypothesis this is just $\langle \mathrm{dnf}(D), N\rangle \vdash_s G_1$ and $\langle \mathrm{dnf}(D), N\rangle \vdash_s G_2$, which in turn is just $\langle \mathrm{dnf}(D), N\rangle \vdash_s G_1 \wedge G_2$.
From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_s \mathrm{dnf}(G_1)$ and $\langle D, N\rangle \vdash_s \mathrm{dnf}(G_2)$ which is just $\langle D, N\rangle \vdash_s \mathrm{dnf}(G_1) \wedge \mathrm{dnf}(G_2)$, i.e. $\langle D, N\rangle \vdash_s (\bigvee\{G' \mid G' \in \mathrm{dnf}'(G_1)\}) \wedge (\bigvee\{G'' \mid G'' \in \mathrm{dnf}'(G_2)\})$ which is equivalent to $\langle D, N\rangle \vdash_s \bigvee\{G' \wedge G'' \mid G' \in \mathrm{dnf}'(G_1), G'' \in \mathrm{dnf}'(G_2)\}$, which is just $\langle D, N\rangle \vdash_s \mathrm{dnf}(G_1 \wedge G_2)$.

2. $\langle D, N\rangle \vdash_f G_1 \wedge G_2$ iff $\langle D, N\rangle \nvdash_s G_1$ or $\langle D, N\rangle \vdash_f G_2$.

By the hypothesis, this is just $\langle \text{dfnf}(D), N\rangle \vdash_f G_1$ or $\langle \text{dfnf}(D), N\rangle \vdash_f G_2$, which in turn is just $\langle \text{dfnf}(D), N\rangle \vdash_f G_1 \wedge G_2$.

From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_f \text{dnf}(G_1)$ or $\langle D, N\rangle \vdash_f \text{dnf}(G_2)$ which is just $\langle D, N\rangle \vdash_f \text{dnf}(G_1) \wedge \text{dnf}(G_2)$, and as we saw above, this is just $\langle D, N\rangle \not\vdash_s \text{dnf}(G_1 \wedge G_2)$.

$\exists x G$: 1. $\langle D, N\rangle \vdash_s \exists x G$ iff $\langle D, N\rangle \vdash_s G[t/x]$ for some $t \in \mathcal{U}$.

By the hypothesis this is equivalent to $\langle \text{dfnf}(D), N\rangle \vdash_s G[t/x]$, which is just $\langle \text{dfnf}(D), N\rangle \vdash_s \exists x G$.

From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_s \text{dnf}(G[t/x])$, which is $\langle D, N\rangle \vdash_s \bigvee\{G' \mid G' \in \text{dnf}'(G[t/x])\}$ which in turn is equivalent to $\langle D, N\rangle \vdash_s G'[t/x]$ for some $G' \in \text{dnf}'(G)$, and this is clearly equivalent to $\langle D, N\rangle \vdash_s \exists x G'$, which in turn is just $\langle D, N\rangle \vdash_s \bigvee\{\exists x G' \mid G' \in \text{dnf}'(G)\}$, i.e. $\langle D, N\rangle \vdash_s \text{dnf}(\exists x G)$.

2. $\langle D, N\rangle \vdash_f \exists x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $\langle D, N\rangle \vdash_f G[t/x]$ for all $t \in R$.

By the hypothesis this is equivalent to $\langle \text{dfnf}(D), N\rangle \vdash_f G[t/x]$ for all $t \in R$, which is just $\langle \text{dfnf}(D), N\rangle \vdash_f \exists x G$.

From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_f \text{dnf}(G[t/x])$ for any $t \in R$, and by a similar argument to that above, this is equivalent to $\langle D, N\rangle \vdash_f \text{dnf}(\exists x G)$.

$D' \supset G$: 1. $\langle D, N\rangle \vdash_s D' \supset G$ iff $\text{names}(\text{heads}(D)) \subseteq \text{ass}(N)$ and $\langle D \cup \{D'\}, N\rangle \vdash_s G$.

By the hypothesis this is equivalent to $\text{names}(\text{heads}(D)) \subseteq \text{ass}(N)$ and $\langle \text{dfnf}(D \cup \{D'\}), N\rangle \vdash_s G$, which is just $\langle \text{dfnf}(D) \cup \{\text{dfnf}(D')\}, N\rangle \vdash_s G$, which by another application of the hypothesis is the same as $\langle \text{dfnf}(D) \cup \{D'\}, N\rangle \vdash_s G$, i.e. $\langle \text{dfnf}(D), N\rangle \vdash_s D' \supset G$.

From the hypothesis we may also deduce that the above is equivalent to $\text{names}(\text{heads}(D)) \subseteq \text{ass}(N)$ and $\langle D \cup \{D'\}, N\rangle \vdash_s \text{dnf}(G)$ which by another application of the hypothesis is equivalent to

$\langle D \cup \{\text{dfnf}(D')\}, N\rangle \vdash_s \text{dnf}(G)$, which is just $\langle D, N\rangle \vdash_s \text{dfnf}(D') \supset \text{dnf}(G)$.

This in turn is equivalent to $\langle D, N\rangle \vdash_s \text{dfnf}(D') \supset \bigvee\{G' \mid G' \in \text{dnf}'(G)\}$ which is just $\langle D, N\rangle \vdash_s \bigvee\{\text{dfnf}(D') \supset G' \mid G' \in \text{dnf}'(G)\}$, which is equivalent to $\langle D, N\rangle \vdash_s \text{dnf}(D' \supset G)$.

2. $\langle D, N\rangle \vdash_f D' \supset G$ iff $\text{names}(\text{heads}(D)) \subseteq \text{ass}(N)$ and $\langle D \cup \{D'\}, N\rangle \vdash_f G$.

   By the hypothesis this is equivalent to $\text{names}(\text{heads}(D)) \subseteq \text{ass}(N)$ and $\langle \text{dfnf}(D \cup \{D'\}), N\rangle \vdash_f G$, which is just $\langle \text{dfnf}(D) \cup \{\text{dfnf}(D')\}, N\rangle \vdash_f G$, which by another application of the hypothesis is the same as $\langle \text{dfnf}(D) \cup \{D'\}, N\rangle \vdash_f G$, i.e. $\langle \text{dfnf}(D), N\rangle \vdash_f D' \supset G$.

   From the hypothesis we may also deduce that the above is equivalent to $\text{names}(\text{heads}(D)) \subseteq \text{ass}(N)$ and $\langle D \cup \{D'\}, N\rangle \vdash_f \text{dnf}(G)$ which by another application of the hypothesis is equivalent to $\langle D \cup \{\text{dfnf}(D')\}, N\rangle \vdash_f \text{dnf}(G)$, which is just $\langle D, N\rangle \vdash_f \text{dfnf}(D') \supset \text{dnf}(G)$, and as above, this is equivalent to $\langle D, N\rangle \vdash_f \text{dnf}(D' \supset G)$.

□

An immediate consequence of this result is given in the corollary below.

Corollary 6.1.3 *Let $\langle P, G\rangle$ be a D_{mod-} derivation pair where $P = \langle D, N\rangle$. Then*

1. $\langle D, N\rangle \vdash_s G \Leftrightarrow \langle \text{dfnf}(D), N\rangle \vdash_s \text{dnf}(G)$

2. $\langle D, N\rangle \vdash_f G \Leftrightarrow \langle \text{dfnf}(D), N\rangle \vdash_f \text{dnf}(G)$

Proof: Follows immediately from proposition 6.1.2. □

This result, in the absence of negation, was shown in [77], but the result corresponding to proposition 6.1.2 was not.

Note that the stronger statement $D \equiv_I \text{dnf}(D)$ is not true. A counterexample, due to Dale Miller [75], is given by the program D below.

$$(r \supset (p \vee q)) \supset s$$

It is clear that dfnf(D) is

$$((r \supset p) \supset s) \wedge ((r \supset q) \supset s)$$

Whilst $D \vdash_I \text{dfnf}(D)$, the converse is not true. The reason is that the two goals $G_1 = r \supset (p \vee q)$ and $G_2 = (r \supset p) \vee (r \supset q)$ are not intuitionistically equivalent, as $G_1 \not\vdash_I G_2$. This will be discussed more fully in a later section.

We now turn to the corresponding result for existential quantifications.

Note that the statement corresponding directly to Proposition 6.1.2 does not hold. Essentially, this is due to the way that failure for existentially quantified goals is defined. For example, consider the program $P = \langle D, \langle \emptyset, \{p\} \rangle \rangle$ where D consists of the clauses below:

$$\begin{array}{r} p(a) \\ p(f(a)) \\ \forall x\, p(f(f(x))) \\ \exists x \neg p(x) \supset q \end{array}$$

It should be clear that $P \vdash_f q$, as $\neg p(a)$, $\neg p(f(a))$ and $\neg p(f(f(y)))$ all fail, and so $P \vdash_f \exists x \neg p(x)$. Consider now efnf($D$), given below.

$$\begin{array}{r} p(a) \\ p(f(a)) \\ \forall x\, p(f(f(x))) \\ \forall x\, (\neg p(x) \supset q) \end{array}$$

Here $\neg p(f(y)) \supset q \in \text{efnf}(D)$, and $\langle \text{efnf}(D), \{p\} \rangle \not\vdash_f \neg p(f(y))$, as $\langle \text{efnf}(D), \{p\} \rangle \not\vdash_s p(f(y))$. Hence, $\langle \text{efnf}(D), \{p\} \rangle \not\vdash_f q$.

This discrepancy is essentially due to the fact that (D) contains all instances of the clauses in D, and so contains all possible representations of those clauses,

whereas the failure of an existentially quantified goal requires only the failure of one representation. Hence, to show that q fails in the original program requires only that one representation fails, whereas to show that q fails in the transformed program requires that all representations fail.

However, it can be shown that $\langle D, N\rangle$ and $\langle \text{efnf}(D), N\rangle$ have the same operational behaviour provided that D does not contain any negations, which is done below.

Proposition 6.1.4 *Let $\langle P, G\rangle$ be a D_{mod} derivation pair where $P = \langle D, N\rangle$. Then*

1. $\langle D, N\rangle \vdash_s G \Leftrightarrow \langle \text{efnf}(D), N\rangle \vdash_s G \Leftrightarrow \langle D, N\rangle \vdash_s \text{enf}(G)$

2. $\langle D, N\rangle \vdash_f G \Leftrightarrow \langle \text{efnf}(D), N\rangle \vdash_f G \Leftrightarrow \langle D, N\rangle \vdash_f \text{enf}(G)$

Proof: As above, we proceed by induction on the depth of the **O**-derivation of G.

In the base case G is an atom A. It is then easy to see that both 1 and 2 follow, as $\text{enf}(A) = A$ and $A \in (D) \Leftrightarrow A \in \text{efnf}(D)$.

Hence we assume that 1 and 2 are true for all goals whose **O**-derivation is less than a given size. There are five cases:

A: 1. As $\text{enf}(A) = A$, it suffices to show that $\langle D, N\rangle \vdash_s A \Leftrightarrow \langle \text{efnf}(D), N\rangle \vdash_s A$. If the base case does not hold, then $\langle D, N\rangle \vdash_s A$ iff $\exists G \supset A \in (D)$ such that $\langle D, N\rangle \vdash_s G$ which by the hypothesis is equivalent to $\exists G \supset A \in (D)$ such that $\langle \text{efnf}(D), N\rangle \vdash_s G$. Now by the hypothesis, this is equivalent to $\langle \text{efnf}(D), N\rangle \vdash_s \text{enf}(G)$, i.e. $\langle \text{efnf}(D), N\rangle \vdash_s \exists \tilde{x} G'$, which is just $\langle \text{efnf}(D), N\rangle \vdash_s G''$ where G'' is some instance of G'. Now as $\exists G \supset A \in (D)$ iff $\exists G'' \supset A \in (\text{efnf}(D))$ where $\text{enf}(G) = \exists \tilde{x} G'$ and G'' is an instance of G', this is equivalent to $\exists G'' \supset A \in (\text{efnf}(D))$ such that $\langle \text{efnf}(D), N\rangle \vdash_s G''$, which is just $\langle \text{efnf}(D), N\rangle \vdash_s A$.

2. As $\text{enf}(A) = A$, it suffices to show that $\langle D, N\rangle \vdash_f A \Leftrightarrow \langle \text{efnf}(D), N\rangle \vdash_f A$. If the base case does not hold, then $\langle D, N\rangle \vdash_f A$ iff $\forall B \in (D)$

$B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$ we have $\langle D, N\rangle \vdash_f G$ which by the hypothesis is equivalent to $\forall B \in (D)$ $B \not\propto A$ and $\forall G \supset B \in (D)$ such that $B \propto A$ we have $\langle \text{efnf}(D), N\rangle \vdash_f G$, which by the hypothesis is equivalent to $\langle \text{efnf}(D), N\rangle \vdash_f \text{enf}(G)$, i.e. $\langle \text{efnf}(D), N\rangle \vdash_f \exists \tilde{x} G'$, which is just that there is a representative set of instances R of G' such that $\langle \text{efnf}(D), N\rangle \vdash_f G''$ for all $G'' \in R$. Note that without loss of generality we may assume that none of the variables of R occur free or bound in D or G. Hence by Proposition 2.3.4 this is equivalent to $\langle \text{efnf}(D), N\rangle \vdash_f G''$ for all instances G'' of G' such that no variables free in G'' occur bound in D or G'. Now as $\exists G \supset A \in (D)$ iff $\exists G'' \supset A \in (\text{efnf}(D))$ where $\text{enf}(G) = \exists \tilde{x} G'$ and G'' is an instance of G', this is equivalent to $\forall B \in (\text{efnf}(D))$ $B \not\propto A$ and $\forall G \supset B \in (\text{efnf}(D))$ such that $B \propto A$ we have $\langle \text{efnf}(D), N\rangle \vdash_f G$, which is equivalent to $\langle \text{efnf}(D), N\rangle \vdash_f A$.

$G_1 \vee G_2$: 1. $\langle D, N\rangle \vdash_s G_1 \vee G_2$ iff $\langle D, N\rangle \vdash_s G_1$ or $\langle D, N\rangle \vdash_s G_2$.
By the hypothesis this is just $\langle \text{efnf}(D), N\rangle \vdash_s G_1$ or $\langle \text{efnf}(D), N\rangle \vdash_s G_2$, which in turn is just $\langle \text{efnf}(D), N\rangle \vdash_s G_1 \vee G_2$.
From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_s \text{enf}(G_1)$ or $\langle D, N\rangle \vdash_s \text{enf}(G_2)$ which is just $\langle D, N\rangle \vdash_s \text{enf}(G_1) \vee \text{enf}(G_2)$, and this is the same as $\langle D, N\rangle \vdash_s \exists \tilde{x}\, \text{enf}'(G_1) \vee \exists \tilde{y}\, \text{enf}'(G_2)$. Now without loss of generality, we may assume that the variables in $\tilde{y}$ do not occur bound or free in G_1, and similarly for $\tilde{x}$ and G_2, and so this is equivalent to $\langle D, N\rangle \vdash_s \exists \tilde{x}\tilde{y}\, \text{enf}'(G_1 \vee G_2)$, which is just $\langle D, N\rangle \vdash_s \text{enf}(G_1 \vee G_2)$.

2. $\langle D, N\rangle \vdash_f G_1 \vee G_2$ iff $\langle D, N\rangle \vdash_f G_1$ and $\langle D, N\rangle \vdash_f G_2$.
By the hypothesis this is just $\langle \text{efnf}(D), N\rangle \vdash_f G_1$ and $\langle \text{efnf}(D), N\rangle \vdash_f G_2$, which in turn is just $\langle \text{efnf}(D), N\rangle \vdash_f G_1 \vee G_2$.
From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_f \text{enf}(G_1)$ and $\langle D, N\rangle \vdash_f \text{enf}(G_2)$ which is just

$\langle D, N\rangle \vdash_f \mathrm{enf}(G_1) \vee \mathrm{enf}(G_2)$, and as above, this is equivalent to $\langle D, N\rangle \vdash_f \mathrm{enf}(G_1 \vee G_2)$.

$G_1 \wedge G_2$: 1. $\langle D, N\rangle \vdash_s G_1 \wedge G_2$ iff $\langle D, N\rangle \vdash_s G_1$ and $\langle D, N\rangle \vdash_s G_2$.

By the hypothesis this is just $\langle \mathrm{efnf}(D), N\rangle \vdash_s G_1$ and $\langle \mathrm{efnf}(D), N\rangle \vdash_s G_2$, which in turn is just $\langle \mathrm{efnf}(D), N\rangle \vdash_s G_1 \wedge G_2$.

From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_s \mathrm{enf}(G_1)$ and $\langle D, N\rangle \vdash_s \mathrm{enf}(G_2)$ which is just $\langle D, N\rangle \vdash_s \mathrm{enf}(G_1) \wedge \mathrm{enf}(G_2)$, and this is the same as $\langle D, N\rangle \vdash_s \exists\tilde{x}\,\mathrm{enf}'(G_1) \wedge \exists\tilde{y}\,\mathrm{enf}'(G_2)$. Now without loss of generality we may assume that the variables in $\tilde{y}$ do not occur bound or free in G_1, and similarly for $\tilde{x}$ and G_2, and so this is equivalent to $\langle D, N\rangle \vdash_s \exists\tilde{x}\tilde{y}\,\mathrm{enf}'(G_1 \wedge G_2)$, i.e. $\langle D, N\rangle \vdash_s \mathrm{enf}(G_1 \wedge G_2)$.

2. $\langle D, N\rangle \vdash_f G_1 \wedge G_2$ iff $\langle D, N\rangle \vdash_f G_1$ or $\langle D, N\rangle \vdash_f G_2$.

By the hypothesis, this is just $\langle \mathrm{efnf}(D), N\rangle \vdash_f G_1$ or $\langle \mathrm{efnf}(D), N\rangle \vdash_f G_2$, which in turn is just $\langle \mathrm{efnf}(D), N\rangle \vdash_f G_1 \wedge G_2$.

From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_f \mathrm{enf}(G_1)$ or $\langle D, N\rangle \vdash_f \mathrm{enf}(G_2)$ which is just $\langle D, N\rangle \vdash_f \mathrm{enf}(G_1) \wedge \mathrm{enf}(G_2)$, and as we saw above, this is just $\langle D, N\rangle \vdash_f \mathrm{enf}(G_1 \wedge G_2)$.

$\exists x G$: 1. $\langle D, N\rangle \vdash_s \exists x G$ iff $\langle D, N\rangle \vdash_s G[t/x]$ for some $t \in \mathcal{U}$.

By the hypothesis this is equivalent to $\langle \mathrm{efnf}(D), N\rangle \vdash_s G[t/x]$, which is just $\langle \mathrm{efnf}(D), N\rangle \vdash_s \exists x G$.

From the hypothesis we may also deduce that the above is equivalent to $\langle D, N\rangle \vdash_s \mathrm{enf}(G[t/x])$, i.e. $\langle D, N\rangle \vdash_s \exists\tilde{x}\,\mathrm{enf}'(G[t/x])$, which is equivalent to $\langle D, N\rangle \vdash_s \exists\tilde{x}\exists x\,\mathrm{enf}'(G)$, which is just $\langle D, N\rangle \vdash_s \mathrm{enf}(G)$.

2. $\langle D, N\rangle \vdash_f \exists x G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $\langle D, N\rangle \vdash_f G[t/x]$ for all $t \in R$.

By the hypothesis this is equivalent to $\langle \mathrm{efnf}(D), N\rangle \vdash_f G[t/x]$ for all $t \in R$, which is just $\langle \mathrm{efnf}(D), N\rangle \vdash_f \exists x G$.

From the hypothesis we may also deduce that the above is equiva-

lent to $\exists R \in \mathcal{R}(\mathcal{U})$ such that $\langle D, N\rangle \vdash_f \text{enf}(G[t/x])$ for any $t \in \mathcal{R}$, and by a similar argument to that above, this is equivalent to $\langle D, N\rangle \vdash_f \text{enf}(\exists x G)$.

$D' \supset G$: 1. $\langle D, N\rangle \vdash_s D' \supset G$ iff names(heads(D)) $\subseteq$ ass(N) and $\langle D \cup \{D'\}, N\rangle \vdash_s G$.

By the hypothesis this is equivalent to names(heads(D)) $\subseteq$ ass(N) and $\langle \text{efnf}(D \cup \{D'\}), N\rangle \vdash_s G$, which is just $\langle \text{efnf}(D) \cup \{\text{efnf}(D')\}, N\rangle \vdash_s G$, which by another application of the hypothesis is equivalent to $\langle \text{efnf}(D) \cup \{D'\}, N\rangle \vdash_s G$, i.e. $\langle \text{efnf}(D), N\rangle \vdash_s D' \supset G$.

From the hypothesis we may also deduce that the above is equivalent to names(heads(D)) $\subseteq$ ass(N) and $\langle D \cup \{D'\}, N\rangle \vdash_s \text{enf}(G)$ which by another application of the hypothesis is equivalent to $\langle D \cup \{\text{efnf}(D')\}, N\rangle \vdash_s \text{enf}(G)$, which is just $\langle D, N\rangle \vdash_s \text{efnf}(D') \supset \text{enf}(G)$, i.e. $\langle D, N\rangle \vdash_s \text{efnf}(D') \supset \exists(\text{enf}'(G))$ which is equivalent to $\langle D, N\rangle \vdash_s \exists(\text{efnf}(D') \supset \text{enf}'(G))$ which is just $\langle D, N\rangle \vdash_s \text{enf}(D' \supset G)$.

2. $\langle D, N\rangle \vdash_f D' \supset G$ iff names(heads(D)) $\subseteq$ ass(N) and $\langle D \cup \{D'\}, N\rangle \vdash_f G$.

By the hypothesis this is equivalent to names(heads(D)) $\subseteq$ ass(N) and $\langle \text{efnf}(D \cup \{D'\}), N\rangle \vdash_f G$, which is just $\langle \text{efnf}(D) \cup \{\text{efnf}(D')\}, N\rangle \vdash_f G$, which by another application of the hypothesis is the same as $\langle \text{efnf}(D) \cup \{D'\}, N\rangle \vdash_f G$, i.e. $\langle \text{efnf}(D), N\rangle \vdash_f D' \supset G$.

From the hypothesis we may also deduce that the above is equivalent to names(heads(D)) $\cap N = \emptyset$ and $\langle D \cup \{D'\}, N\rangle \vdash_f \text{enf}(G)$ which by another application of the hypothesis is equivalent to $\langle D \cup \{\text{efnf}(D')\}, N\rangle \vdash_f \text{enf}(G)$, which is just $\langle D, N\rangle \vdash_f \text{efnf}(D') \supset \text{enf}(G)$, and as above this is equivalent to $\langle D, N\rangle \vdash_f \text{enf}(D' \supset G)$.

□

Note that this result depends upon Proposition 2.3.4, i.e. that all sets of representative instances of a goal fail if a single representative set of instances of it fails.

The lack of a corresponding result for success is what necessitates the restriction to programs which do not contain negations.

As in the previous case, the generalisation of the corresponding result in [77] now follows immediately.

Corollary 6.1.5 *Let $\langle P, G\rangle$ be a D_{mod} derivation pair where $P = \langle D, N\rangle$. Then*

1. $\langle D, N\rangle \vdash_s G \Leftrightarrow \langle \mathrm{efnf}(D), N\rangle \vdash_s \mathrm{enf}(G)$

2. $\langle D, N\rangle \vdash_f G \Leftrightarrow \langle \mathrm{efnf}(D), N\rangle \vdash_f \mathrm{enf}(G)$

Proof: Follows immediately from Proposition 6.1.4. □

As may be expected in the aftermath of corollary 6.1.3, the stronger statement $D \equiv_I \mathrm{efnf}(D)$ is not true. A counterexample is given by the program D below.

$$(p \supset \exists x q(x)) \supset r$$

It is clear that $\mathrm{efnf}(D)$ is

$$\forall x((p \supset q(x)) \supset r)$$

Whilst $D \vdash_I \mathrm{efnf}(D)$, the converse is not true. The reason is that the two goals $G_1 = p \supset \exists x\ q(x)$ and $G_2 = \exists x\ p \supset q(x)$ are not intuitionistically equivalent, as $G_1 \not\vdash_I G_2$. This will also be discussed more fully in a later section.

Note that a consequence of propositions 6.1.2 and 6.1.4 is that $P \vdash_s G$ iff $P \vdash_s \mathrm{dnf}(G)$ iff $P \vdash_s \mathrm{enf}(G)$ when P is a D_{mod} program and G is a G_{mod} goal.

It may be possible to strengthen Proposition 6.1.4 somewhat. For example, any derivation pair $\langle P, G\rangle$ for which $\mathrm{efnf}(P) = P$ and $\mathrm{enf}(G) = G$ will clearly have this property, whether they contain negations or not.

Clearly the definition of failure for existentially quantified goals affects whether Proposition 6.1.4 will hold for D_{mod-} programs or not. It seems clear that if $\exists x G$

were to fail precisely when $G[t/x]$ fails for all $t \in \mathcal{T}$, then Proposition 6.1.4 would indeed hold for D_{mod-} programs and G_{mod-} goals. Hence the failure of the efnf transformation to preserve operational equivalence under Negation as Failure is a consequence of the compactness properties of our definition of failure.

We now come to the result which shows the operational equivalence (for D_{mod} programs) of the original program and the program with the disjunctions and existential quantifications removed.

Theorem 6.1.6 *Let $\langle P, G\rangle$ be a D_{mod} derivation pair where $P = \langle D, N\rangle$. Then*

1. $\langle D, N\rangle \vdash_s G \Leftrightarrow \langle \text{efnf}(\text{dfnf}(D)), N\rangle \vdash_s G$

2. $\langle D, N\rangle \vdash_f G \Leftrightarrow \langle \text{efnf}(\text{dfnf}(D)), N\rangle \vdash_f G$

Proof: Follows immediately from propositions 6.1.2 and 6.1.4.

□

Thus the theorem above not only ensures that for any D_{mod} program there exists another program free of disjunctions and existential quantifications which is equivalent to the original, but also shows how such a program may be derived from the original one.

We may interpret the above result as showing that the class of programs in which universal quantifiers cannot occur positively in goals is of no greater expressive power than the core programs. This may be seen by the fact that any program in this class which contains neither $\exists$ nor $\vee$ is a core program, and the above result shows that for any program containing $\exists$ or $\vee$ there is an equivalent program with no such occurrences. In fact this class of programs is strictly less expressive than the core programs, as a core program may contain a positive occurrence of a universal quantifier in a goal, i.e. in the body of a clause.

6.2 Normal Forms and Representation

In the light of the above results, we may think of D_{object} programs as a normal form for D_{mod} programs. In other words, for any D_{mod} program D, there is an operationally equivalent D_{object} program D', where D' satisfies

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid Q \supset A$$
$$Q := A \mid Q_1 \wedge Q_2 \mid D \supset Q$$

Note that the Q formulae are only used to define D'; the queries remain the same as before, i.e. G_{mod} formulae, which may be given as

$$G := A \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid \exists x G \mid D \supset G$$

where D is a D_{object} formula.

Computation takes place in the same way as before, in that we search for a proof of a goal G from a program $P = \langle D, N \rangle$, and so we may think of the above processes as statically converting the program into a more specific form.

As noted above, these computations take the form of searching for uniform proofs of goal formulae, but it is interesting to note that a successful search for a uniform proof of a goal G may be thought of as converting G into a more definite statement. Naively, we may think of this as converting G into a D formula. For example, it is well-known that the usual unification methods used to implement the search for a proof of a goal of the form $\exists x G$ will return an answer substitution θ which not only provides a witness, i.e. a term $t \in \mathcal{U}$ such that $G[t/x]$ succeeds, but where θ is such that *any* instance of $G\theta$ succeeds, so that we may in fact conclude $\forall(G\theta)$. Thus the search process "converts" the $\exists$ quantifier into a number (possibly zero) of $\forall$ quantifiers. In this way we may view the result of a successful computation not as "yes with the answer substitution θ", but as "$\forall(G\theta)$", in the sense that $\forall(G\theta) \vdash_I \exists x G$.[1]

[1]We assume that $\mathcal{U}$ is not empty, and so this will always be true.

It is easy to see that a similar property holds for disjunction, in that a uniform proof of $G_1 \vee G_2$ will prove one of the disjuncts, and so we may consider the proof search process as determining which of the disjuncts is true, and so we again get that $G_i \vdash_I G_1 \vee G_2$, where i is either 1 or 2. Now if we carry out this conversion procedure for each positive occurrence of $\exists$ and $\vee$ in the goal, then it is easy to see that the resulting formula looks very much like a definite formula.

For example, consider the program $P = \langle p(a) \wedge q(b), \emptyset \rangle$. It is easy to see that the goal $G = \exists x\, p(x) \vee \exists y\, q(y)$ succeeds, i.e. $P \vdash_s G$, as both $p(a)$ and $q(b)$ succeed. Now we may think of either of these as an "answer form" of the original goal, in that not only does the computation process supply witnesses for the existential quantifications, but also determines which of the disjuncts holds. In this way we may think of $p(a)$ and $q(b)$ as *realizers* of G in the sense of Kleene [51], in that either supplies sufficient information "missing" from the statement that $P \vdash_s G$ so that intuitionistic truth is established. We may think of these realizers as the real objects of computation, in that the search process may be considered to operate by establishing that $p(a)$ succeeds, and from that deducing that the success of $p(a)$ implies the success of $\exists x\, p(x) \vee \exists y\, q(y)$. A similar remark applies to $q(b)$. In this way the computation process finds a goal G' which contains no existential quantifiers or disjunctions such that $P \vdash_s G'$ and $G' \vdash_I G$. The analogy should not be pushed too far, but it seems intriguing that the answer forms may be thought of as supplying the information needed to establish the truth of the goal, and in a computational way.

As noted above, this formula G' contains no disjunctions nor existential quantifiers, but it may contain universal quantifiers. For example, consider the program $P = \langle D, \langle \emptyset, \emptyset \rangle \rangle$ where D is $\forall x\ p(f(x))$. The goal $\exists y\ p(y)$ succeeds with the answer substitution being $y \leftarrow f(x)$, and so the answer form of the goal is $\forall x\ p(f(x))$. A more specific definition of the possible answer forms for a goal may be given by another of the results above, namely that the success of G is equivalent to the success of $\mathrm{enf}(\mathrm{dnf}(G))$, which we may write as

$$\exists(Q_1 \vee \ldots \vee Q_n)$$

where each of the Q_i are goals which contain no disjunctions nor existential quantifications, and so are just Q formulae (defined above). Thus any answer form for G must be just $\forall(Q_i\theta)$ for some $1 \leq i \leq n$ and some substitution θ, and so we may consider a successful search for a proof of G from P as finding an i and a θ such that

$$P \vdash_s \forall(Q_i\theta) \text{ and } \forall(Q_i\theta) \vdash_I G$$ [2]

In this way an answer form plays a similar role to a cut formula (in the proof-theoretic sense), in that in order to establish that $P \vdash_s G$, we find a formula $\forall(Q_i\theta)$ such that $P \vdash_s \forall(Q_i\theta)$ and $\forall(Q_i\theta) \vdash_I G$, but in our case we know that the second statement is trivially established, whereas the first may take considerable effort to derive. Thus this is really a special case of the cut rule, in that G defines a finite number of possible choices for Q_i, and so we do not need to search for an arbitrary formula, and that one of the two sub-derivations will require no effort to establish. Hence we may view this form of computation as a restricted form of the cut rule.

In order to view an answer form $\forall(Q_i\theta)$ as a definite clause, the main difficulty is to accommodate the different classes of formulae which may appear on either side of the $\supset$ connective. For example, $D \supset Q$ is a Q formula, and hence a possible answer form, but it is not necessarily a D formula. However, it is clear that it is not far away from one in some sense, and so it should not be too difficult to convert an answer form into an equivalent D formula. This process is defined formally below.

There is one other potential difficulty, and that is the possible occurrence of a negation in an answer form, which precludes it from being a D formula. We can circumvent this problem by allowing negated atoms as definite formula under the restriction that negation is still only applied to completely defined predicates and that if $\neg A$ is a clause in a program P, then $P \vdash_f A$. This process will be discussed

---

[2]This is a slight abuse of notation, as we do not allow universal quantifiers in goals. This matter will be dealt with shortly.

more fully in a later section, but for now we note that we are mainly interested in the structure of the answer forms, and that we may think of an answer form as being stored in a cache somewhere, and hence is separate from the program itself. In this way we can allow a limited form of negation in the known consequences of the program without allowing this form of negation to appear in the original program.

The process of re-writing an answer form to make it a definite formula is given below. For convenience we use the extended form of $D_{object-}$ formulae, in which the conclusion of an implication in a program may be an arbitrary D formula rather than just an atom. As remarked previously, this does not increase the power of the language.

Definition 6.2.1 *Let D be an extended $D_{object-}$ program and a Q an extended $G_{object-}$ goal, that is, D and Q belong to the class of formulae defined below.*

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid Q \supset D$$
$$Q := A \mid \neg A \mid Q_1 \wedge Q_2 \mid D \supset Q$$

We define defp(D) *and* defq(Q) *as follows:*

$$\begin{aligned}
&\mathrm{defp}(A) &&= A\\
&\mathrm{defp}(\forall x D) &&= \forall x\ \mathrm{defp}(D)\\
&\mathrm{defp}(D_1 \wedge D_2) &&= \mathrm{defp}(D_1) \wedge \mathrm{defp}(D_2)\\
&\mathrm{defp}(Q \supset \forall x D') &&= \forall x\ \mathrm{defp}(Q \supset D')\\
&\mathrm{defp}(Q \supset (D_1 \wedge D_2)) &&= \mathrm{defp}(Q \supset D_1) \wedge \mathrm{defp}(Q \supset D_2)\\
&\mathrm{defp}(Q \supset (Q' \supset D')) &&= \mathrm{defp}((Q \wedge Q') \supset D')\\
&\mathrm{defp}(Q \supset A) &&= \mathrm{defq}(Q) \supset A
\end{aligned}$$

$$\begin{aligned}
&\mathrm{defq}(A) &&= A\\
&\mathrm{defq}(\neg A) &&= \neg A\\
&\mathrm{defq}(Q_1 \wedge Q_2) &&= \mathrm{defq}(Q_1) \wedge \mathrm{defq}(Q_2)\\
&\mathrm{defq}(D \supset (Q_1 \wedge Q_2)) &&= \mathrm{defq}(D \supset Q_1) \wedge \mathrm{defq}(D \supset Q_2)
\end{aligned}$$

$$\begin{aligned}
\text{defq}(D \supset (D' \supset Q')) &= \text{defq}((D \wedge D') \supset Q') \\
\text{defq}(D \supset A) &= \text{defp}(D) \supset A \\
\text{defq}(D \supset \neg A) &= \text{defp}(D) \supset \neg A
\end{aligned}$$

Note that we assume that all bound variables are unique and distinct from all free variables, so that we do not need to worry about variable capture when moving quantifiers around.

Proposition 6.2.1 *Let D be an extended $D_{object-}$ program, and let Q be an extended $G_{object-}$ goal. Then*

$$D \equiv_I \text{defp}(D)$$
$$Q \equiv_I \text{defq}(Q)$$

Proof: We proceed by simultaneous induction on the structure of D and Q. The base case occurs when D is an atom and when Q is a literal. It is obvious that in this case $D \equiv_I \text{defp}(D)$ and $Q \equiv_I \text{defq}(Q)$.

Thus the hypothesis is that for all D and Q formulae of no more than a given size, $D \equiv_I \text{defp}(D)$ and $Q \equiv_I \text{defq}(Q)$. There are several cases to examine.

The cases for D:

$\forall x D$: By the hypothesis, $D \equiv_I \text{defp}(D)$, and so $\forall x D \equiv_I \forall x\ \text{defp}(D)$, which is just $\text{defp}(\forall x D)$.

$D_1 \wedge D_2$: By the hypothesis, $D_1 \wedge D_2 \equiv_I \text{defp}(D_1) \wedge \text{defp}(D_2)$, which is just $\text{defp}(D_1 \wedge D_2)$.

$Q' \supset D'$: There are four sub-cases here. We proceed via an inductive argument on the size of D'. The base case occurs when D' is just an atom A. By the hypothesis $Q' \equiv_I \text{defq}(Q')$, and so $Q' \supset A \equiv_I \text{defq}(Q') \supset A$, which is just $\text{defp}(Q' \supset A)$.

Hence we will assume, in addition to the main hypothesis, that for all Q formulae with less connectives than $Q' \supset D'$ and for all D formula of no more than a given size that $Q \supset D \equiv_I \text{defp}(Q \supset D)$.

The three inductive subcases are given below

$Q \supset \forall x D$: It is clear that $Q \supset \forall x D \equiv_I \forall x\, Q \supset D$ as x is not free in Q, and by the hypothesis, this is equivalent to $\forall x$ defp$(Q \supset D)$, which is just defp$(Q \supset \forall x D)$.

$Q \supset (D_1 \wedge D_2)$: $Q \supset (D_1 \wedge D_2)$ is clearly equivalent to $(Q \supset D_1) \wedge (Q \supset D_2)$, and by the hypothesis this is equivalent to defp$(Q \supset D_1) \wedge$ defp$(Q \supset D_2)$, which is just defp$(Q \supset (D_1 \wedge D_2))$.

$Q \supset (Q'' \supset D)$: $Q \supset (Q'' \supset D)$ is equivalent to $(Q \wedge Q'') \supset D$, and as $Q \wedge Q''$ clearly has less connectives than $Q \supset (Q'' \supset D)$, by the hypothesis this is equivalent to defp$((Q \wedge Q'') \supset D)$, which is just defp$(Q \supset (Q'' \supset D))$.

Thus we establish the result for any formula of the form $Q \supset D$.

The cases for Q:

$Q_1 \wedge Q_2$: By the hypothesis, $Q_1 \wedge Q_2 \equiv_I$ defq$(Q_1) \wedge$ defq(Q_2), which is just defq$(Q_1 \wedge Q_2)$.

$D' \supset Q'$: There are four cases here, and as above, we use an inductive argument to establish the overall case. The base case occurs when Q' is a literal L, and so by the hypothesis, $D' \supset L \equiv_I$ defp$(D') \supset L$, which is just defq$(D' \supset L)$.

Hence we will assume, in addition to the main hypothesis, that for all D formulae with less connectives than $D' \supset Q'$ and for all extended $G_{object-}$ formulae of no more than a given size that $D \supset Q \equiv_I$ defq$(D \supset Q)$.

There are two sub-cases:

$D \supset (Q_1 \wedge Q_2)$: It is clear that $D \supset (Q_1 \wedge Q_2)$ is equivalent to $(D \supset Q_1) \wedge (D \supset Q_2)$, and by the hypothesis this is equivalent to defq$(D \supset Q_1) \wedge$defq$(D \supset Q_2)$, which is just defq$(D \supset Q_1 \wedge Q_2)$.

$D \supset (D'' \supset Q)$: $D \supset (D'' \supset Q)$ is clearly equivalent to $(D \wedge D'') \supset Q$, and as $D \wedge D''$ must have less connectives than $D \supset (D'' \supset Q)$, by the hypothesis we have that this is equivalent to defq$((D \wedge D'') \supset Q)$, which is just defq$(D \supset (D'' \supset Q))$.

Thus we establish the result for any formula of the form $D \supset Q$.

□

It is clear from the above proof that for any program in the larger class there is an equivalent program in the smaller class, i.e. the class defined by using $Q \supset A$ in place of $Q \supset D$ in the above definition.

For the smaller class of programs, consider defq($D \supset Q$). The only changes that the process can make are made by the defq process rather than the defp one, as there can be no sub-formulae of D of the form $Q \supset D'$ unless D' is just an atom. Hence, defq($D \supset Q$) will be just $\bigwedge_i\{D_i \supset L_i\}$ for some object level programs D_i and literals L_i. The important observation is that each D_i must have less connectives than $D \supset Q$, although it may contain more than D. This is due to the fact that any D_i of greater size than D must have "gained" the relevant connectives from Q, and so cannot exceed this number.

Recall that D_{meta} and G_{meta} formulae may be defined as follows:

$$L := A \mid \neg A$$
$$D := L \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset L$$
$$G := L \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid \forall x G \mid D \supset G$$

Recall also that an M_{meta} formula is any formula satisfying

$$L := A \mid \neg A$$
$$M := L \mid \forall x M \mid M_1 \wedge M_2 \mid M \supset L$$

so that any M_{meta} formula is both a D_{meta} formula and a G_{meta} formula.

Proposition 6.2.2 *Let D be an $D_{object-}$ program, Q be a $G_{object-}$ goal. Then*

1. defp(D) *is a Q_{meta} formula.*

2. defq(G) *is a D_{meta} formula.*

Proof: We will show that defp(D) and defq(Q) are both M_{meta} formulae.

We proceed by simultaneous induction on the structure of D and Q. The base case occurs when D is an atom and when Q is a literal. It is obvious that in this case that both defp(D) and defq(Q) are M_{meta} formulae.

Thus we assume that for all D and Q formulae of no more than a given size, defp(D) is an M_{meta} formula and defq(Q) is an M_{meta} formula. We then proceed by cases as follows.

1. There are three cases for D:

$\forall x D'$: defp($\forall x D'$) = $\forall x$ defp(D'), and by the hypothesis, defp(D') is some core formula M, and so defp($\forall x D'$) is just $\forall x M$, and it is clear that $\forall x M$ is a core formula.

$D_1 \wedge D_2$: defp($D_1 \wedge D_2$) = defp(D_1) $\wedge$ defp(D_2), and by the hypothesis, defp(D_1) and defp(D_2) are core formulae M_1 and M_2, and so defp($D_1 \wedge D_2$) is $M_1 \wedge M_2$, which is obviously a core formula.

$Q \supset A$: defp($Q \supset A$) = (defq(Q)) $\supset A$, and by the hypothesis defq(Q) is a core formula M, and so defq($Q \supset A$) is $M \supset A$, which is clearly a core formula.

2. There are two cases for Q:

$Q_1 \wedge Q_2$: defq($Q_1 \wedge Q_2$) = defq(Q_1)$\wedge$defq(Q_2), and by the hypothesis defq(Q_1) and defq(Q_2) are core formulae M_1 and M_2, and so defq($Q_1 \wedge Q_2$) = $M_1 \wedge M_2$, which is transparently a core formula.

$D' \supset Q'$: There are four cases here, and as above, we use an inductive argument to establish the overall case. The base case occurs when Q' is a literal L, and so defq($D \supset L$) = defp(D) $\supset L$, and by the hypothesis defp(D) is a core formula M, and so defq($D \supset L$) = $M \supset L$, which is clearly a core formula.

Hence we will assume, in addition to the main hypothesis, that for all D formulae with less connectives than $D' \supset Q'$ and for all Q formula of no more than a given size that defq($D \supset Q$) is a core formula.

There are two remaining cases:

$D \supset (Q_1 \wedge Q_2)$: $\text{defq}(D \supset (Q_1 \wedge Q_2)) = \text{defq}(D \supset Q_1) \wedge \text{defq}(D \supset Q_2)$, and by the hypothesis this is just $M_1 \wedge M_2$ where M_1 and M_2 are core formulae, and obviously so is $M_1 \wedge M_2$.

$D \supset (D'' \supset Q)$: $\text{defq}(D \supset (D'' \supset Q)) = \text{defq}((D \wedge D'') \supset Q)$ and by the hypothesis this is a core formula M.

Thus we establish the result for any formula of the form $D \supset Q$.

□

This result shows that we may consider computation as converting G_{mod-} formulae into definite clauses of a certain form, which may then be stored in addition to the program, in that a G_{mod-} goal G may be re-written as $\exists(Q_1 \vee \ldots \vee Q_n)$ where each Q_i is a $G_{object-}$ formula, and if the goal succeeds, we get an answer form $\forall(Q_i\theta)$ for some $1 \leq i \leq n$ and so $\forall(\text{defq}(Q_i)\theta) = \forall(\text{defq}(Q_i\theta))$ is a D_{meta} formula.

It is interesting to note that Horn clauses have a similar property. Consider the class of $D_{Horn\vee}$ programs and $G_{Horn\vee}$ goals, i.e. D and G formulae satisfying

$$D := A \mid D_1 \wedge D_2 \mid \forall x D \mid G \supset A$$
$$G := A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G$$

As noted above, an equivalent class of programs is given the programs for which D and G formulae are given by

$$D := A \mid D_1 \wedge D_2 \mid \forall x D \mid G \supset A$$
$$G := A \mid G_1 \wedge G_2$$

Now in this case, for any $G_{Horn\vee}$ goal G, $\text{enf}(\text{dnf}(G))$ may be written as $\exists(G_1 \vee \ldots \vee G_n)$, and so any answer form is of the form $\forall(G_i\theta)$, where the G_i are the G formulae above. As G_i can only be a conjunction of atoms, it is obvious that $\forall(G_i\theta)$ is a D_{Horn} formula.

In this way we may see the above results as restoring properties lost when generalising from Horn clauses to the class of programs defined above. Not surprisingly, this required a more sophisticated approach than for Horn clauses, but this more complex approach still retains the essential features of the original class of programs in a natural way.

6.3 Minimal Conditions for Operational Equivalence

For the rest of this chapter, we omit consideration of negation for the sake of simplicity. There is no obvious problem in incorporating negation into what follows, but the required detail would obscure the salient points of the discussion.

Note that $\vdash_s$ and $\vdash_o$ coincide on D_{mod} programs and G_{mod} goals when there are no completely defined predicates.

One way to interpret the results of section 6.1 is that they define two program transformations which preserve the operational behaviour of programs, in that the original program and the transformed program are indistinguishable if the only available method of inspection is the evaluation of goals. If two programs produce the same results, i.e. the same goals succeed and fail, then we say that the two programs are *operationally equivalent.* An interesting question which arises is the question of "minimal testing" for observational equivalence. More formally, the question is what is the smallest class of goals for which operational equivalence needs to be established in order to show operational equivalence for all goals. It is easy to see that this smallest class must properly include all atomic goals. The reason that atomic goals alone are not sufficient is demonstrated by the following example.

p	p
$r \supset q$	$q \supset r$

It is clear that for both programs p succeeds and all other atoms fail, but that the goal $q \supset r$ succeeds from one program but not the other.

An "upper bound" for this minimal class is given in the lemma below.

Lemma 6.3.1 *Let Q formulae be defined by*

$$Q := A \mid D \supset Q$$

where D is any D_{HHF} formula.

Let D_1 and D_2 be sets of D_{HHF} definite formulae, and let G range over G_{HHF} goal formulae. Then

$$(\forall Q\ D_1 \vdash_s Q \Rightarrow D_2 \vdash_s Q) \Rightarrow (\forall G\ D_1 \vdash_s G \Rightarrow D_2 \vdash_s G)$$

Note that the Q formulae defined above are not a subclass of G_{object} formulae, as the implications in the Q formulae may be D_{HHF} formulae, rather than D_{object} formulae.

Proof: Assume that $\forall Q\ D_1 \vdash_s Q \Rightarrow D_2 \vdash_s Q$.

First note that from the assumption we may derive that

$$\forall Q\ \forall D\ D_1 \cup \{D\} \vdash_s Q \Rightarrow D_2 \cup \{D\} \vdash_s Q$$

as for any Q formula and for any D_{HHF} program D, $D \supset Q$ is a Q formula.

We will show that $\forall G\ \forall D\ D_1 \cup \{D\} \vdash_s G \Rightarrow D_2 \cup \{D\} \vdash_s G$. It is clear that this will establish the result.[3]

Assume that $D_1 \cup \{D\} \vdash_s G$. We proceed by induction on the structure of G.

---

[3]We allow the possibility that D may be the empty program.

The base case occurs when G is an atom A, and it is clear from the initial assumption that $D_2 \vdash_s A$.

Hence the induction hypothesis is that the lemma is true for all programs D and for all goals of less than a given size.

There are five cases:

$G_1 \vee G_2$: $D_1 \cup \{D\} \vdash_s G_1 \vee G_2$ iff $D_1 \cup \{D\} \vdash_s G_1$ or $D_1 \cup \{D\} \vdash_s G_2$, and by the hypothesis this implies $D_2 \cup \{D\} \vdash_s G_1$ or $D_2 \cup \{D\} \vdash_s G_2$, i.e. $D_2 \cup \{D\} \vdash_s G_1 \vee G_2$.

$G_1 \wedge G_2$: $D_1 \cup \{D\} \vdash_s G_1 \wedge G_2$ iff $D_1 \cup \{D\} \vdash_s G_1$ and $D_1 \cup \{D\} \vdash_s G_2$, and by the hypothesis this implies $D_2 \cup \{D\} \vdash_s G_1$ and $D_2 \cup \{D\} \vdash_s G_2$, i.e. $D_2 \cup \{D\} \vdash_s G_1 \wedge G_2$.

$\exists x G$: $D_1 \cup \{D\} \vdash_s \exists x\ G$ iff $D_1 \cup \{D\} \vdash_s G[t/x]$ for some $t \in \mathcal{U}$, and by the hypothesis this implies $D_2 \cup \{D\} \vdash_s G[t/x]$ for some $t \in \mathcal{U}$, i.e. $D_2 \cup \{D\} \vdash_s \exists x\ G$.

$\forall x G$: $D_1 \cup \{D\} \vdash_s \forall x\ G$ iff $\exists R \in \mathcal{R}(\mathcal{U})$ such that $D_1 \cup \{D\} \vdash_s G[t/x]$ for all $t \in R$, and by the hypothesis this implies $D_2 \cup \{D\} \vdash_s G[t/x]$ for all $t \in R$, i.e. $D_2 \cup \{D\} \vdash_s \forall x\ G$.

$D' \supset G'$: $D_1 \cup \{D\} \vdash_s D' \supset G'$ iff $D_1 \cup \{D\} \cup \{D'\} \vdash_s G'$, and by the hypothesis this implies $D_2 \cup \{D\} \cup \{D'\} \vdash_s G'$, i.e. $D_2 \cup \{D\} \vdash_s D' \supset G'$.

□

We may think of the above result as establishing that operational equivalence for atoms and implications is sufficient to establish operational equivalence for all G_{HHF} goals. We shall see in section 6.4 how this result is useful in another context. A stronger result may be possible, in that operational equivalence for a smaller class of goals than those in the above lemma may imply operational equivalence for G_{HHF} goals. Whilst such a result seems to be true, it will take more work to establish, and, as discussed in 6.4, the above is usually adequate.

Alternatively, the above result may be interpreted as stating that if we can establish that all *extensions* of two programs enable the same atoms to be derived, then we have established operational equivalence. Thus if we cannot distinguish between two programs by arbitrary mutual extensions of them, even when the only goals which may be asked are atomic, then the two programs are operationally equivalent. There may be weaker conditions under which operational equivalence holds, but it would seem that any weakening of this condition would come from restricting the extensions that may be made to the programs.

6.4 Notions of Equivalence

The above deliberations involve detailed consideration of the precise relationship between the relations $\vdash_s$ and $\vdash_I$. Whilst we know that $D \vdash_s G$ iff $D \vdash_I G$ for D_{mod} and G_{mod} formulae, we may use $\vdash_I$ between programs, i.e. we may ask whether $D_1 \vdash_I D_2$ or not, but we cannot use $\vdash_s$ in the same way unless D_2 is a G formula. Thus the greater restriction placed upon the relationship $\vdash_s$ may have some subtle affects, and so it seems worthwhile to investigate the relationship between the two more closely.

One way to explore this relationship is to consider different ways in which consequence relations give rise to equivalence relations between programs. Maher [67] has shown that there are several meaningful conceptions of equivalence between logic programs; here we consider two such notions. One is the operational equivalence referred to above, so that two programs D_1 and D_2 are operationally equivalent if for every goal G, $D_1 \vdash_s G \Leftrightarrow D_2 \vdash_s G$. This may be thought of as treating the programs as two black boxes, so that the only way that we may distinguish between them is by how they react to queries from the outside world. We will denote the operational equivalence of two programs D_1 and D_2 as $D_1 \equiv_o D_2$. Another obvious notion of equivalence is given by the consequence relation $\vdash_I$, so that two programs D_1 and D_2 are considered equivalent if $D_1 \equiv_I D_2$. We may think of this as logical equivalence.

A natural question which then arises is whether these two notions of equivalence coincide, i.e. whether two programs D_1 and D_2 are operationally equivalent iff D_1 and D_2 are logically equivalent. It is easy to see that logical equivalence implies operational equivalence, as if $D_1 \vdash_I D_2$ and $D_2 \vdash_s G$, then as $D_2 \vdash_s G$ implies that $D_2 \vdash_I G$, we have that $D_1 \vdash_I G$, i.e. $D_1 \vdash_s G$. The corresponding case for $D_2 \vdash_I D_1$ is similar. Hence, the question is whether operational equivalence implies logical equivalence.

It is known that for full first-order hereditary Harrop formulae under the "new constant" interpretation of the universal quantifier, the notions of equivalence do not coincide. A counterexample, due to Frank Pfenning, is given by the two programs D_1 and D_2 given below.

D_1	D_2
$\forall x\,(p(x) \vee q(x)) \supset r$	$(\forall x\; p(x) \vee \forall x\; q(x)) \supset r$

Let $G_1 = \forall x\,(p(x) \vee q(x))$, $G_2 = \forall x\, p(x) \vee \forall x\, q(x)$. It is clear that $G_2 \vdash_I G_1$, and so $G_1 \supset r \vdash_I G_2 \supset r$, i.e. $D_1 \vdash_I D_2$. However, $G_1 \not\vdash_I G_2$, and so $D_2 \not\vdash_I D_1$. All that remains is to show that $\forall G\; D_1 \vdash_s G \Rightarrow D_2 \vdash_s G$ (the converse is obvious from the fact that $D_1 \vdash_I D_2$). From the operational interpretation of $\forall$ as a new constant, it is clear that we may equivalently re-write D_1 and D_2 as D_1' and D_2' as follows:

D_1'	D_2'
$(p(c) \vee q(c)) \supset r$	$(p(c_1) \vee q(c_2)) \supset r$

It should now be clear that any goal provable from D_1' is provable from D_2' under the operational rules given. This is because the goal $\forall x\,(p(x) \vee q(x))$ can only succeed when either $\forall x\, p(x)$ succeeds or when $\forall x\, q(x)$ succeeds, due to the fact that in the goal $p(c) \vee q(c)$ the new constant c is treated in the same way as an "old" constant, rather than as a meta-variable. Hence there can be no interaction between the two disjuncts, which is necessary to prove $\forall x\, p(x) \vee q(x)$ in the case when neither $\forall x\, p(x)$ nor $\forall x\, q(x)$ succeed.

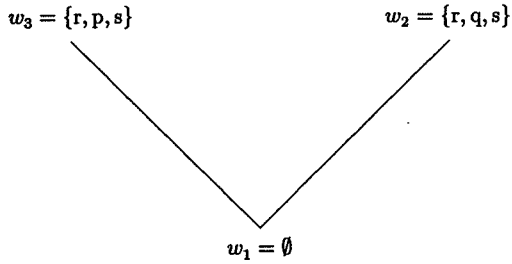

Figure 6–1: Kripke model in which D_4 is true but D_3 is not

This still leaves the question open for the class of programs in which universal quantifiers are not allowed in goals, as well as for full first-order hereditary Harrop formulae under the extensional interpretation of the universal quantifier. However, operational equivalence does not imply logical equivalence for either class of programs, as is shown by the following counterexample (mentioned earlier in another context):

D_3	D_4
$(r \supset (p \vee q)) \supset s$	$((r \supset p) \supset s) \wedge ((r \supset q) \supset s)$

Whilst $D_3 \equiv_o D_4$ due to the fact that $D_4 = \mathrm{dfnf}(D_3)$, D_3 and D_4 are not logically equivalent. A Kripke model in which D_4 is true but D_3 is not is given in Figure 6–1. The reason that D_3 is not true is that $w_1 \models r \supset (p \vee q)$, and so for D_3 to be true in w_1, we require that s be true in w_1, which it is not, and so D_3 is not true in w_1. On the other hand, we have that $w_1 \not\models r \supset p$ and $w_1 \not\models r \supset q$, and so for D_4 to be true in w_1, we do not require that s be true in w_1, and in fact D_4 is trivially true in w_1, as $w_1 \not\models r$. Hence D_4 is true for every world in the model, whereas D_3 is not true in world w_1.

Thus operational equivalence is not strong enough to establish (intuitionistic) logical equivalence. One way to interpret this result is that the natural choice of

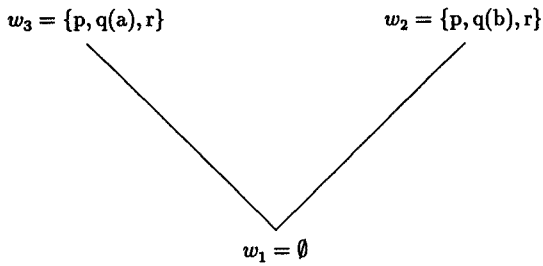

Figure 6–2: Kripke model in which D_6 is true but D_5 is not

logic for programming in this context needs to have a stronger consequence relation than $\vdash_I$, so that operationally equivalent programs are logically equivalent. Thus some intermediate logic is required, as $D_3 \equiv_C D_4$, and we saw earlier that $\vdash_C$ was too strong to be suitable in this context.

Not surprisingly, there is a property dual to that above which involves existential quantification, in that the programs D_5 and D_6 below are operationally equivalent but not logically equivalent.

D_5	D_6
$(p \supset \exists x\ q(x)) \supset r$	$\forall x\ ((p \supset q(x)) \supset r)$

Note that $D_6 = \text{efnf}(D_5)$, which establishes the operational equivalence, but whilst $D_5 \vdash_I D_6$, the converse is not true. A Kripke model in which D_6 is true but D_5 is not is given in Figure 6–2. As in the previous case, it is easy to see that both formulae are true in worlds w_2 and w_3, but whilst D_6 is true at w_1, (as $w_1 \not\models \exists x\ p \supset q(x)$ and so the condition that if $\exists x p \supset q(x)$ is true then r is true is satisfied), D_5 is not true at w_1, as $w_1 \models p \supset \exists x\ q(x)$ but $w_1 \not\models r$.

Thus any intermediate logic, say I', appropriate for this application will need to satisfy the following equivalences:

$$D \supset (G_1 \vee G_2) \equiv_{I'} (D \supset G_1) \vee (D \supset G_2)$$
$$D \supset \exists x\, G \equiv_{I'} \exists x\, D \supset G$$

where x is not free in D. Note that the $\Leftarrow$ direction of both equivalences hold in intuitionistic logic. These equivalences are natural ones to choose given that we expect that goals which are operationally equivalent are logically equivalent. We will see in section 6.5 that this is indeed the case.

The $\Rightarrow$ directions of these rules are known as the *Independence of Premise* axioms [109], and although there are some results regarding the addition of these and similar axioms to Heyting arithmetic [109], not much seems to be known about the logic obtained by adding these rules to intuitionistic logic.

We may think of such a logic as a logic of "present choice", in that the choice of witness for the existentially quantified variable cannot be postponed; if we can ever choose such a witness, then we can do so immediately, without investigating future worlds. Similar remarks apply to the disjunctive case, in that if we can ever choose between the two alternatives we can do so immediately, without waiting to see what will happen in the future. This immediacy is reflected in the computation process by the fact that in order to establish the truth of r from either D_5 or D_6 we do so by trying to prove the body of the clause, rather than considering possible future choices. In this way the knowledge which we may use to make the relevant choice is already present in the program, and so it is merely a matter of seeing if such a choice can be made. If it cannot, then we have no other way to proceed. Hence, the logic of present choice will reflect the fact that we can only use information encoded in the program.

We consider the precise nature of the desired logic in section 6.5; for now, we look at the properties of such a logic.

The feature which causes the failure of operational equivalence to imply equivalence in intuitionistic logic is the mutual interaction of $\supset$ with $\exists$ or $\vee$. Not surprisingly, when one or the other of these is absent from the body of clauses in programs, then we do have that operational equivalence implies equivalence in intuitionistic logic. It is easy to see this when D_1 is a core formula as if

$$\forall G\ D_1 \vdash_s G \Rightarrow D_2 \vdash_s G$$

then as D_1 is a G formula, it immediately follows that $D_2 \vdash_s D_1$, and so $D_2 \vdash_I D_1$.

Similarly, it is not hard to show that if there are no implications in the body of a clause in D_1, then

$$D_1 \equiv_I \mathrm{dfnf}(D_1) \equiv_I \mathrm{efnf}(D_1)$$

and this gives $D_1 \vdash_I \mathrm{defp}(\mathrm{dfnf}(\mathrm{efnf}(D_1)))$ which may be similarly used to derive that $D_2 \vdash_I D_1$.

Hence, intuitionistic logic performs precisely as computational intuition would suggest when there is no "mixing" of $\supset$ and either $\exists$ or $\vee$, but otherwise some stronger logic, whose nature seems difficult to elucidate, is needed.

Naturally we expect a similar result to hold in the presence of such mixing for an intermediate logic I' in which the equivalences mentioned above hold. We may think of this requirement on I' as demanding that all connectives be treated similarly, since the corresponding equivalences for the other connectives, given below, are intuitionistically valid.

$$D \supset (G_1 \wedge G_2) \equiv_I (D \supset G_1) \wedge (D \supset G_2)$$
$$D \supset (D' \supset G) \equiv_I (D \wedge D') \supset G$$
$$D \supset \forall x\ G \equiv_I \forall x\ (D \supset G)$$

where x is not free in D.

In any such logic I', it may be shown that $D \equiv_{I'} \mathrm{dfnf}(D)$ and $D \equiv_{I'} \mathrm{efnf}(D)$. This is done below.

Proposition 6.4.1 *Let D be a D_{mod} definite formula and G be a G_{mod} goal formula. Then*

$$D \equiv_{I'} \mathrm{dfnf}(D)$$
$$G \equiv_{I'} \mathrm{dnf}(G)$$

Proof: We proceed by mutual induction on the structure of D and G. When D is an atom, $D = \mathrm{dfnf}(D)$, and when G is an atom, $G = \mathrm{dnf}(G)$, and so the proposition is obviously true for the base case.

Hence the hypothesis is that for all D and G formulae with less than a given number of connectives, the proposition holds.

There are three cases for D:

$\forall x D'$: By the hypothesis $D' \equiv_I \mathrm{dfnf}(D')$, and so $\forall x D'$ is equivalent to $\forall x\ \mathrm{dfnf}(D')$, which is just $\mathrm{dfnf}(\forall x D')$.

$D_1 \wedge D_2$: By the hypothesis each D_i is equivalent to $\mathrm{dfnf}(D_i)$, and so $D_1 \wedge D_2$ is equivalent to $\mathrm{dfnf}(D_1) \wedge \mathrm{dfnf}(D_2)$, which is just $\mathrm{dfnf}(D_1 \wedge D_2)$.

$G \supset A$: By the hypothesis $G \supset A$ is equivalent to $\mathrm{dnf}(G) \supset A$, which in turn is just $(\bigvee\{G' \mid G' \in \mathrm{dnf}'(G)\}) \supset A$. This is equivalent to $\bigwedge\{G' \supset A \mid G' \in \mathrm{dnf}'(G)\}$, i.e. $\mathrm{dfnf}(G \supset A)$.

There are four cases for G:

$G_1 \vee G_2$: By the hypothesis each G_i is equivalent to $\mathrm{dnf}(G_i)$, and so $G_1 \vee G_2$ is equivalent to $\mathrm{dnf}(G_1) \vee \mathrm{dnf}(G_2)$ which is just $\mathrm{dnf}(G_1 \vee G_2)$.

$G_1 \wedge G_2$: By the hypothesis each G_i is equivalent to $\mathrm{dnf}(G_i)$, and so $G_1 \wedge G_2$ is equivalent to $\mathrm{dnf}(G_1) \wedge \mathrm{dnf}(G_2)$, which is just $(\bigvee\{G' \mid G' \in \mathrm{dnf}'(G_1)\}) \wedge (\bigvee\{G' \mid G' \in \mathrm{dnf}'(G_2)\})$. This is equivalent to $\bigvee\{G'_1 \wedge G'_2 \mid G'_1 \in \mathrm{dnf}'(G_1), G'_2 \in \mathrm{dnf}'(G_2)\}$, which in turn in just $\mathrm{dnf}(G_1 \wedge G_2)$.

$\exists x G$: By the hypothesis G is equivalent to $\mathrm{dnf}(G)$, and so $\exists x G$ is equivalent to $\exists x \mathrm{dnf}(G)$, which is just $\exists x \bigvee\{G' \mid G' \in \mathrm{dnf}'(G)\}$ which in turn is equivalent to $\bigvee\{\exists x G' \mid G' \in \mathrm{dnf}'(G)\}$, i.e. $\mathrm{dnf}(\exists x G)$.

$D' \supset G'$: By the hypothesis $D' \supset G'$ is equivalent to $\mathrm{dfnf}(D') \supset \mathrm{dnf}(G')$, which is just $\mathrm{dfnf}(D') \supset \bigvee\{G'' \mid G'' \in \mathrm{dnf}'(G')\}$. Due to the equivalence in I' of $D \supset (G_1 \vee G_2)$ and $(D \supset G_1) \vee (D \supset G_2)$, this in turn is equivalent to $\bigvee\{\mathrm{dfnf}(D') \supset G'' \mid G'' \in \mathrm{dnf}'(G')\}$, i.e. $\mathrm{dnf}(D' \supset G')$.

□

The corresponding result for the existential case is given below.

Proposition 6.4.2 *Let $\langle D, N\rangle$ be a D_{mod} definite formula, and let G be a G_{mod} goal formula. Then*

$$D \equiv_{I'} \text{efnf}(D)$$
$$G \equiv_{I'} \text{enf}(G)$$

Proof: We proceed by mutual induction on the structure of D and G. When D is an atom, $D = \text{efnf}(D)$, and when G is a literal, $G = \text{enf}(G)$, and so the proposition is obviously true for the base case.

Hence the hypothesis is that for all D and G formulae with less than a given number of connectives, the proposition holds.

There are three cases for D:

$\forall x D'$: By the hypothesis $D' \equiv_I \text{efnf}(D')$, and so $\forall x D'$ is equivalent to $\forall x\ \text{efnf}(D')$, which is just $\text{efnf}(\forall x D')$.

$D_1 \wedge D_2$: By the hypothesis $D_1 \wedge D_2$ is equivalent to $\text{efnf}(D_1) \wedge \text{efnf}(D_2)$, which is just $\text{efnf}(D_1 \wedge D_2)$.

$G \supset A$: By the hypothesis $G \supset A$ is equivalent to $\text{enf}(G) \supset A$, which in turn is just $(\exists \tilde{x}\ (\text{enf}'(G))) \supset A$. This is equivalent to $\forall \tilde{x}\ \text{enf}'(G) \supset A$, i.e. $\text{efnf}(G \supset A)$.

There are four cases for G:

$G_1 \vee G_2$: By the hypothesis $G_1 \vee G_2$ is equivalent to $\text{enf}(G_1) \vee \text{enf}(G_2)$, which is just $(\exists \tilde{x}\ \text{enf}'(G_1)) \vee (\exists \tilde{y}\ \text{enf}'(G_2))$, which in turn is equivalent to $\exists \tilde{x}\tilde{y}\ (\text{enf}'(G_1) \vee \text{enf}'(G_2))$, i.e. $\text{enf}(G_1 \vee G_2)$.

$G_1 \wedge G_2$: By the hypothesis $G_1 \wedge G_2$ is equivalent to $\text{enf}(G_1) \wedge \text{enf}(G_2)$ which is just $(\exists \tilde{x}\ \text{enf}'(G_1)) \wedge (\exists \tilde{y}\ \text{enf}'(G_2))$, which is in turn equivalent to $\exists \tilde{x}\tilde{y}\ (\text{enf}'(G_1) \wedge \text{enf}(G_2))$, i.e. $\text{enf}(G_1 \wedge G_2)$.

$\exists x G$: By the hypothesis G is equivalent to $\text{enf}(G)$, which is just $\exists \tilde{y}\ \text{enf}'(G)$, and so $\exists x G$ is equivalent to $\exists x\ \exists \tilde{y}\ \text{enf}'(G)$, i.e. $\text{enf}(\exists x G)$.

$D' \supset G'$: By the hypothesis $D' \supset G'$ is equivalent to $\text{efnf}(D') \supset \text{enf}(G')$, which is just $\text{efnf}(D') \supset \exists \tilde{x}\ \text{enf}'(G')$. Due to the equivalence in I' of $D \supset \exists x G$ and $\exists x\, D \supset G$, this in turn is equivalent to $\exists \tilde{x}\ \text{efnf}(D') \supset \text{enf}'(G')$, i.e. $\text{enf}(D' \supset G')$.

□

We will see later how these two results may be seen as stronger versions of propositions 6.1.2 and 6.1.4 for a particular choice of I'.

It is clear from proposition 6.2.1 that $D' \equiv_{I'} \text{defp}(D')$ when D' contains no existential quantifiers or disjunctions. Thus from propositions 6.4.1, 6.4.2 and 6.2.1 we have that

$$D \equiv_{I'} \text{defp}(\text{efnf}(\text{dfnf}(D)))$$

We will abbreviate the latter formula by $\text{core}(D)$. From proposition 6.2.2 we have that $\text{core}(D)$ may be considered as a G formula if we allow universal quantifications in goals. This gives us a method for establishing that operational equivalence implies logical equivalence. A precise statement and proof is given below.

Theorem 6.4.3 *Let D_1 and D_2 be D_{mod} programs, and let G range over G_{mod} goals.*

Then

$$(\forall G\ D_1 \vdash_s G \Leftrightarrow D_2 \vdash_s G) \Rightarrow D_2 \equiv_{I'} D_1$$

Proof: We will show that if $\forall G\ D_1 \vdash_s G \Rightarrow D_2 \vdash_s G$, then $D_2 \vdash_{I'} D_1$. It is clear that this will establish the result.

Assume that $\forall G\ D_1 \vdash_s G \Rightarrow D_2 \vdash_s G$. From lemma 6.3.1, we know that this implies that $\forall G'\ D_1 \vdash_s G' \Rightarrow D_2 \vdash_s G'$, where G' ranges over G_{HHF} goals.

As mentioned above, we know from propositions 6.4.1, 6.4.2 and 6.2.1 that $D \equiv_{I'} \text{core}(D)$, and from proposition 6.2.2 that $\text{core}(D)$ is a G_{HHF} formula, as it contains no negations. Thus we have that $D_1 \vdash_s \text{core}(D_1)$, and so $D_2 \vdash_s \text{core}(D_1)$, which gives us that $D_2 \vdash_{I'} \text{core}(D_1)$, i.e. $D_2 \vdash_{I'} D_1$.

□

Thus for any such logic I', operational equivalence implies logical equivalence. As the two desired equivalences are true classically, this property also holds for classical logic.

An interesting interpretation of this result it is that for any such logic I', if two programs D_1 and D_2 differ, i.e. $D_1 \not\equiv_{I'} D_2$, then there is a goal which distinguishes the two. More precisely, from the contrapositive of theorem 6.4.3 we have that if $D_2 \not\vdash_{I'} D_1$, then there is a goal G such that $D_1 \vdash_s G$ but it is not the case that $D_2 \vdash_s G$.

6.5 Choice of Intermediate Logic

Whilst theorem 6.4.3 holds when I' is interpreted as classical logic, it is clear from earlier discussions that this is not an ideal choice, and so the desired logic I' lies strictly between intuitionistic logic and classical logic.

One intermediate logic of interest in this context is the extension of intuitionistic propositional calculus (IPC) called *lc* by Gabbay [36]. This may be characterised by adding to IPC the rule (in a Hilbert-style proof system)

$$(p \supset (q \vee r)) \supset ((p \supset q) \vee (p \supset r))$$

However, *lc* appears to be too strong, as it loses the disjunctive property, i.e. it is not true that if $A \vee B$ is provable then either A is provable or B is provable. Also, some of the alternative characterisations of *lc* do not fit in well with computational intuition. For example, another way to characterise this logic is to add the following rule to IPC:

$$(p \supset q) \vee (q \supset p)$$

This does not seem to be a rule which can be justified in terms of computation, especially due to the restrictions which first-order hereditary Harrop formulae place on implications. In addition, the models are characterised as all finite partially ordered sets for which $\forall xy\ (x \leq y \vee y \leq x)$, and hence are linear in that any given worlds w_1 and w_2 must be related by either $w_1 \leq w_2$ or $w_2 \leq w_1$. This is in direct contradiction with the Kripke-like model theory given in [77], in which it is certainly not true that all worlds are comparable. Thus it seems that the best way to proceed is to use the Kripke-like model theory as the guiding intuition for the particular choice of logic I'. This is particularly relevant since we are not interested in all first-order formulae, but only in hereditary Harrop formulae, and so the fact that all models of *lc* are linear only shows that it is not of great interest, rather than indicating problems.

The Kripke-like model clearly relies heavily on the fact that the formulae involved are hereditary Harrop formulae and not arbitrary first-order formulae. An interesting observation is that programs have similar properties to prime theories (also known as saturated theories) [21,107] which are used in the proof of the completeness of intuitionistic provability with respect to Kripke models. This suggests that hereditary Harrop formulae are of interest as the worlds of some special kind of Kripke model. Further work in this area would presumably indicate more precisely the relationship between hereditary Harrop formulae and full first-order logic, particularly in relation to the Independence of Premise rules.

As mentioned in the previous chapter, another intermediate logic of interest is the logic of constant domains. As the domain is constant for all worlds in the Kripke-like model, it is clear that any results which we derive for the Kripke-like model will hold for the logic of constant domains.

One way to analyse the nature of I' is to explore the relation $\vdash_s \subseteq \mathcal{D} \times \mathcal{G}$ in terms of a relation on $\mathcal{D} \times \mathcal{D}$ and a relation on $\mathcal{G} \times \mathcal{G}$. The main obstacle to viewing $\vdash_s$ directly as a consequence relation in the traditional sense is that such relations are usually defined on $\mathcal{F} \times \mathcal{F}$ where $\mathcal{F}$ is the set of all well-formed

(first-order) formulae. Hence, as is done in [77], we may view $\vdash_s$ as the restriction of $\vdash_I$ to $\mathcal{D} \times \mathcal{G}$, as it is known that $D \vdash_o G$ iff $D \vdash_I G$ [77] , and as $\vdash_o$ and $\vdash_s$ coincide on $D_{mod} \times G_{mod}$ when there are no completely defined predicates, this implies $D \vdash_s G$ iff $D \vdash_I G$. However, we cannot use $\vdash_s$ directly for questions of equivalence between programs or between goals, and so for such equivalences we need to use $\vdash_{I'}$.

An important property of the Kripke-like model is that whilst worlds are D formulae, the formulae which are provable from a given world are G formulae. Nevertheless, there is still an underlying notion of truth for D formulae in a given world. As discussed in the previous chapter, we may think of this second notion of truth along the lines of "D is assumed in world w", rather than "G is provable in world w". In this way there is already an implicit relation on $\mathcal{W} \times \mathcal{D}$. Also in the same chapter we saw how we may find an interpretation $T^{\omega}(I_{\perp})$ such that $T^{\omega}(I_{\perp}), P \Vdash G$ iff $P \vdash_s G$. We may think of this as a semantic characterisation of the relation $\vdash_s$. In order to think of this in terms of a relation on $\mathcal{D} \times \mathcal{G}$, we will write $T^{\omega}(I_{\perp}), P \Vdash G$ as just $P \Vdash G$. [4] In this way we fix the choice of interpretation I in the triple of I,P and G, as we do not wish to consider arbitrary interpretations, but to investigate the nature of $\vdash_s$. This makes it possible to interpret the Kripke-like model directly as a Kripke model, as we can now consider which worlds prove which goals, and we may abbreviate $\forall w\ w \Vdash G$ as $\Vdash G$ as is usually done. We then have that $\Vdash D \supset G$ iff $\forall w\ w \Vdash D \supset G$ iff $\forall w\ w \cup \{D\} \Vdash G$. Now from the access relation between worlds we know that this is just $\forall w \geq D\ w \Vdash G$, as $w \cup \{D\} \geq D$ for all worlds w. This in turn is just $D \Vdash G$, which is not surprising given that we consider $\Vdash$ as a semantic characterisation of $\vdash_s$. However, in a Kripke model one expects $w \Vdash D \supset G$ to be equivalent to $\forall w' \geq w\ w' \models D \Rightarrow w' \models G$ for some relation $\models$. As mentioned above, in our case we need to consider two different consequence relations, and so

[4]Strictly we should write $\Vdash_{T^{\omega}(I_{\perp})}$, to denote our choice of interpretation, but it will always be clear from the context what is meant.

we interpret this as $\forall w' \geq w\ w' \models_D D \Rightarrow w' \models\!\!\!\models G$, for some appropriate definition of $\models_D$. An obvious choice is given below.

Definition 6.5.1 *Let D_1 and D_2 be D_{mod} programs. Then*

$$D_1 \models_D D_2 \text{ iff } T^\omega(I_\perp)(D_2) \sqsubseteq T^\omega(I_\perp)(D_1)$$

The results below show that $\models_D$ behaves in the expected manner.

Lemma 6.5.1 *Let D_1 and D' be D_{mod} programs and let G be a G_{mod} goal.*

$$\text{If } D_1 \models\!\!\!\models G \text{ and } D_2 \models_D D_1 \text{ then } D_2 \models\!\!\!\models G$$

Proof: From $D_2 \models_D D_1$ we have that $T^\omega(I_\perp)(D_1) \sqsubseteq T^\omega(I_\perp)(D_2)$, and so by lemma 5.3.2 $T^\omega(I_\perp), D_1 \models\!\!\!\models G \Rightarrow T^\omega(I_\perp), D_2 \models\!\!\!\models G$. Hence, as $D_1 \models\!\!\!\models G$, we have that $D_2 \models\!\!\!\models G$.

□

In this way $\models_D$ respects operational equivalence, as a consequence of the above result is that if two programs D_1 and D_2 are such that $D_1 \equiv_D D_2$, then the programs are operationally equivalent. Another interesting property of $\models_D$ is given in the lemma below.

Lemma 6.5.2 *Let D be a D_{mod} program and G be a G_{mod} goal. Then*

$$(\forall w\ w \geq D \Rightarrow w \models\!\!\!\models G) \Leftrightarrow (\forall w\ w \models_D D \Rightarrow w \models\!\!\!\models G)$$

Proof:

($\Leftarrow$): $w \geq D$ implies that $\{D\} \subseteq w$ and so $T^\omega(I_\perp)(D) \sqsubseteq T^\omega(I_\perp)(w)$, i.e. $w \models_D D$. Hence $w \models_D D \Rightarrow w \models\!\!\!\models G$ implies that $w \geq D \Rightarrow w \models\!\!\!\models G$.

($\Rightarrow$): $\forall w\ w \geq D \Rightarrow w \models\!\!\!\models G$ implies that $D \models\!\!\!\models G$, and so if $w \models_D D$, then by lemma 6.5.1 we have that $w \models\!\!\!\models G$, and so we have $\forall w\ w \models_D D \Rightarrow w \models\!\!\!\models G$.

□

Having defined a model-theoretic notion of consequence for programs, we next look for a proof-theoretic consequence relation which matches this one, so that we expect $D_1 \vdash_{I'} D_2$ iff $D_1 \models_D D_2$. As mentioned above, we want $\vdash_{I'}$ to include all intuitionistic consequences (i.e. $\vdash_I \subseteq \vdash_{I'}$), but also for the Independence of Premise rules to hold, i.e.:

$$D \supset \exists x G \vdash_{I'} \exists x D \supset G$$
$$D \supset (G_1 \vee G_2) \vdash_{I'} (D \supset G_1) \vee (D \supset G_2)$$

Note that the converses of both these rules hold in intuitionistic logic.

The simplest way to define $\vdash_{I'}$ is to add the two rules above to the deduction rules of first-order intuitionistic logic. A sequent-style proof system for I' is given below. This is the standard sequent system for intuitionistic logic augmented by the two desired rules.

$$\frac{B, C, \Gamma \longrightarrow F}{B \wedge C, \Gamma \longrightarrow F}\ \wedge\text{-L} \qquad \frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge\text{-R}$$

$$\frac{B, \Gamma \longrightarrow F \quad C, \Gamma \longrightarrow F}{B \vee C, \Gamma \longrightarrow F}\ \vee\text{-L} \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow B \vee C}\ \frac{\Gamma \longrightarrow C}{\Gamma \longrightarrow B \vee C}\ \vee\text{-R}$$

$$\frac{\Gamma \longrightarrow B \quad C, \Gamma \longrightarrow F}{B \supset C, \Gamma \longrightarrow F}\ \supset\text{-L} \qquad \frac{B, \Gamma \longrightarrow C}{\Gamma \longrightarrow B \supset C}\ \supset\text{-R}$$

$$\frac{\Gamma, B[t/x] \longrightarrow F}{\Gamma, \forall x B \longrightarrow F}\ \forall\text{-L} \qquad \frac{\Gamma \longrightarrow B[y/x]}{\Gamma \longrightarrow \forall x B}\ \forall\text{-R}$$

$$\frac{\Gamma, B[y/x] \longrightarrow F}{\Gamma, \exists x B \longrightarrow F}\ \exists\text{-L} \qquad \frac{\Gamma \longrightarrow B[t/x]}{\Gamma \longrightarrow \exists x B}\ \exists\text{-R}$$

$$\frac{\Gamma \longrightarrow D \supset \exists x G}{\Gamma \longrightarrow \exists x D \supset G}\ \exists\text{-}\supset \qquad \frac{\Gamma \longrightarrow D \supset (G_1 \vee G_2)}{\Gamma \longrightarrow (D \supset G_1) \vee (D \supset G_2)}\ \vee\text{-}\supset$$

$$\frac{\Gamma \longrightarrow \bot}{\Gamma \longrightarrow B}\ \bot\text{-R}$$

The rules $\forall$-R and $\exists$-L have the side condition that y is not free in Γ, B or F, and the rule $\exists$-$\supset$ has the side condition that x is not free in D.

We also include the standard structural rules of interchange, contraction and thinning. As is done in [77], we may omit the interchange and contraction rules if we allow the antecedents of sequents to be sets.[5] Thinning may be introduced by a technique of antecedent thinning described in [77], but it is not vital to the discussion here.

An *initial sequent* is a sequent $\Gamma \longrightarrow F$ where F is either an atomic formula or $\perp$ and $F \in \Gamma$. A *proof* for the sequent $\Gamma \longrightarrow F$ is a finite tree, constructed using the above rules, whose root is $\Gamma \longrightarrow F$ and whose leaves are initial sequents.

We refer to proofs in the above system as $\mathbf{I}'$-proofs.

Note that the two extra rules are only applicable when the consequents are a particular kind of G formulae, but that the extra rules ensure that there are programs D_1 and D_2 for which $D_1 \not\vdash_I D_2$ but $D_1 \vdash_{I'} D_2$. For example, $p \supset \exists x\, q(x) \vdash_{I'} \exists x\, p \supset q(x)$, and so $(\exists x\, p \supset q(x)) \supset r \vdash_{I'} (p \supset \exists x\, q(x)) \supset r$.

The relation $\vdash_{I'}$ then gives us both the desired relation on $\mathcal{D} \times \mathcal{D}$ and on $\mathcal{G} \times \mathcal{G}$. One important feature of the above proof rules is that the restriction of $\vdash_{I'}$ to $\mathcal{D} \times \mathcal{G}$ is precisely $\vdash_s$. In this way we have not altered the way that goals are derived from programs, but strengthened the provability relation between programs and between goals. A formal proof of this property is given below.

The following lemmas are analogous to lemmas 9 and 10 in [77], and are extended to include $\mathbf{I}'$-proofs.

---

[5]The succedent can only be a singleton set, and so we write succedents as single formulae.

Lemma 6.5.3 *Let Γ be a set of D_{mod} and G_{mod} formulae. Then there is no $\mathbf{I}'$-proof of $\Gamma \longrightarrow \bot$.*

Proof: Suppose there is such a proof Ξ of $\Gamma \longrightarrow \bot$ for some Γ.

We proceed by induction on the height of Ξ. Clearly there is no initial sequent for which $\bot \in \Gamma$.

Hence the hypothesis is that for all sets Γ of D and G formulae, there is no $\mathbf{I}'$-proof of $\Gamma \longrightarrow \bot$ of a given height or less.

Consider the possibilities for the final rule in Ξ. None of $\forall$-R, $\wedge$-R, $\vee$-R, $\supset$-R, $\exists$-R, $\exists$-$\supset$ and $\vee$-$\supset$ are applicable, as the consequent in the resulting sequent must be just $\bot$.

This leaves the following cases:

$\wedge$-L: For this rule to be applicable, the previous sequent must be $B, C, \Gamma \longrightarrow \bot$, and as $B \wedge C$ is a D or G formula iff B and C are both D formulae or both G formulae, this contradicts the induction hypothesis.

$\vee$-L: For this rule to be applicable, the previous sequent must be $B, \Gamma \longrightarrow \bot$, and as $B \vee C$ is a D or a G formula iff B and C are G formulae, this contradicts the induction hypothesis.

$\forall$-L: For this rule to be applicable, the previous sequent must be $\Gamma, B[t/x] \longrightarrow \bot$, and as $\forall x B$ is a D or G formula iff $B[t/x]$ is a D formula, this contradicts the induction hypothesis.

$\exists$-L: For this rule to be applicable, the previous sequent must be $\Gamma, B[y/x] \longrightarrow \bot$, and as $\exists x B$ is a D or G formula iff $B[y/x]$ is a G formula, this contradicts the induction hypothesis.

$\supset$-L: For this rule to be applicable, the previous two sequents must be $\Gamma \longrightarrow B$ and $C, \Gamma \longrightarrow \bot$. Now as $B \supset C$ must be a D or a G formula, there are two cases.

In the first case, B must be a goal G and C must be an atom A, making the two prior sequents $\Gamma \longrightarrow G$ and $A, \Gamma \longrightarrow \bot$ respectively, which contradicts the induction hypothesis.

In the second case, B must be a program D and C must be a goal G, making the two prior sequents $\Gamma \longrightarrow D$ and $G, \Gamma \longrightarrow \bot$ respectively, which contradicts the induction hypothesis.

$\bot$-R: For this rule to be applicable, the previous sequent to $\Gamma \longrightarrow \bot$ must be just $\Gamma \longrightarrow \bot$, which contradicts the induction hypothesis.

Thus there can be no proof of $\Gamma \longrightarrow \bot$ when Γ is a set of D and G formulae.

□

As we often wish to restrict our attention to $\mathcal{D} \times \mathcal{G}$, the following notion will be useful.

Definition 6.5.2 *A sequent $\Gamma \longrightarrow F$ is an* **O***-sequent iff Γ is a set of D_{mod} formulae and F is a G_{mod} formula.*

The following lemma is analogous to lemma 9 in [77].

Lemma 6.5.4 *Let Γ be a set of D_{mod} formulae, G be a G_{mod} goal and Ξ be an $\mathbf{I}'$-proof of $\Gamma \longrightarrow G$. Then Ξ contains no instances of the rules $\vee$-L, $\exists$-L or $\forall$-R, and all sequents which appear in Ξ are* **O***-sequents.*

Proof: We proceed by induction on the height of Ξ. It is clear that if the root sequent is initial, then the antecedent must be a set of D formulae and the consequent just an atom A, which is clearly an **O**-sequent, and so the property holds when Ξ has height 1.

So the inductive hypothesis is that the property holds for all $\mathbf{I}'$-proofs whose height is less than a given value. Consider the last rule used in the proof Ξ of the sequent $\Gamma \longrightarrow G$.

It is clear that the final rule used in Ξ cannot be $\vee$-L, $\exists$-L or $\forall$-R as the sequent resulting from such a rule will not be an **O**-sequent. By lemma 6.5.3, we need not consider the $\bot$-R rule either.

It only remains to show that applying the remaining rules to **O**-sequents results in **O**-sequents, as then we know that the last rule used in Ξ is applied to an **O**-sequent, to which we may apply the hypothesis.

The cases are:

$\wedge$-L: For this rule to be applicable, we must have that $B \wedge C$ is a D_{mod} formula and that F is a G_{mod} formula, and so B and C must both be D_{mod} formulae, and so the previous sequent is an **O**-sequent.

$\wedge$-R: For this rule to be applicable, we must have that $B \wedge C$ is a G_{mod} formula, and so B and C must both be G_{mod} formulae, and so the previous sequents are **O**-sequents.

$\vee$-R: For this rule to be applicable, we must have that $B \vee C$ is a G_{mod} formula, and so B and C must both be G_{mod} formulae, and so the previous sequents are **O**-sequents.

$\supset$-L: For this rule to be applicable, we must have that B is a G_{mod} formula, C is an atom and F is a G_{mod} formula, and so the previous sequents are **O**-sequents.

$\supset$-R: For this rule to be applicable, we must have that $B \supset C$ is a G_{mod} formula, and so B is a D_{mod} formula and C is a G_{mod} formula, and so the previous sequent is an **O**-sequent.

$\forall$-L: For this rule to be applicable, we must have that $\forall x B$ is a D_{mod} formula, and so $B[t/x]$ is a D_{mod} formula, and so the previous sequent is an **O**-sequent.

$\exists$-R: For this rule to be applicable, we must have that $\exists x B$ is a G_{mod} formula, and so $B[x/t]$ is a G_{mod} formula, and so the previous sequent is an **O**-sequent.

$\exists$-$\supset$: Clearly $\exists x D \supset G$ is a G_{mod} formula iff $D \supset \exists x G$ is a G_{mod} formula, and so the previous sequent is an **O**-sequent.

$\vee$-$\supset$: Clearly $(D \supset G_1) \vee (D \supset G_2)$ is a G_{mod} formula iff $D \supset (G_1 \vee G_2)$ is a G_{mod} formula, and so the previous sequent is an **O**-sequent.

□

An easy extension to this result is given below.

Definition 6.5.3 *A sequent $\Gamma \longrightarrow F$ is an $\mathbf{I}'$-sequent iff every element of Γ is either a D_{mod} or G_{mod} formula, and F is either a D_{mod} formula or a G_{mod} formula.*

Note that Γ may contain both D_{mod} and G_{mod} formulae, so that

$$\{\forall x p(x), \exists x q(x)\} \longrightarrow p(a) \wedge p(b)$$

is an $\mathbf{I}'$-sequent.

Lemma 6.5.5 *Let $\Gamma \longrightarrow F$ be an $\mathbf{I}'$-sequent and let Ξ be a proof of $\Gamma \longrightarrow F$. Then every sequent which appears in Ξ is an $\mathbf{I}'$-sequent.*

Proof: We proceed by induction on the height of Ξ. It is clear that the base case holds, as then $\Gamma \longrightarrow F$ must be an initial sequent. Hence the property holds when Ξ has height 1.

So the inductive hypothesis is that the property holds for all $\mathbf{I}'$-proofs whose height is less than a given value. Consider the proof Ξ of the sequent $\Gamma \longrightarrow F$.

Consider the final rule used in Ξ.

The are a number of cases:

$\wedge$-L: For this rule to be applicable, the previous sequent must be $B, C, \Gamma \longrightarrow F$, and as $B \wedge C$ is a D or G formula iff B and C are both D formulae or both G formulae, this is clearly an $\mathbf{I}'$-sequent.

$\wedge$-R: For this rule to be applicable, the previous sequents must be $\Gamma \longrightarrow B$ and $\Gamma \longrightarrow C$, and as $B \wedge C$ is a D or G formula iff B and C are both D formulae or both G formulae, these are both clearly $\mathbf{I}'$-sequents.

$\vee$-L: For this rule to be applicable, the previous sequents must be $B, \Gamma \longrightarrow F$ and $C, \Gamma \longrightarrow F$, and as $B \vee C$ is a D or G formula iff B and C are both G formulae, these are both clearly $\mathbf{I}'$-sequents.

$\vee$-R: For this rule to be applicable, the previous sequent must be either $\Gamma \longrightarrow B$ or $\Gamma \longrightarrow C$, and as $B \vee C$ is a D or G formula iff B and C are both G formulae, in either case it is clearly an $\mathbf{I}'$-sequent.

$\supset$-L: For this rule to be applicable, the previous sequents must be $\Gamma \longrightarrow B$ and $C, \Gamma \longrightarrow F$. now as $B \supset C$ must be a D or a G formula, there are two cases.

In the first case, B must be a G formula, and C must be an atom, and as F must be either a D or a G formula, clearly both prior sequents are $\mathbf{I}'$-sequents.

In the second case, B must be a D formula, and C must be a G formula, and as F must be either a D or a G formula, clearly both prior sequents are $\mathbf{I}'$-sequents.

$\supset$-R: For this rule to be applicable, the previous sequent must be $B, \Gamma \longrightarrow C$. As $B \supset C$ must be a D or G formula, there are two cases.

In the first case, B must be a G formula, and C must be an atom, and so clearly the previous sequent is an $\mathbf{I}'$-sequent.

In the second case, B must be a D formula, and C must be a G formula, and so clearly the previous sequent is an $\mathbf{I}'$-sequent.

$\forall$-L: For this rule to be applicable, the previous sequent must be $\Gamma, B[t/x] \longrightarrow F$, and as $\forall x B$ is a D or G formula iff B is a D formula, the previous sequent is clearly an $\mathbf{I}'$-sequent.

$\forall$-R: For this rule to be applicable, the previous sequent must be $\Gamma, \longrightarrow B[y/x]$, and as $\forall x B$ is a D or G formula iff B is a D formula, the previous sequent is clearly an $\mathbf{I}'$-sequent.

$\exists$-L: For this rule to be applicable, the previous sequent must be $\Gamma, B[y/x] \longrightarrow F$, and as $\exists x B$ is a D or G formula iff B is a G formula, the previous sequent is clearly an $\mathbf{I}'$-sequent.

$\exists$-R: For this rule to be applicable, the previous sequent must be $\Gamma \longrightarrow B[y/x]$, and as $\exists x B$ is a D or G formula iff B is a G formula, the previous sequent is clearly an $\mathbf{I}'$-sequent.

$\exists$-$\supset$: For this rule to be applicable, the previous sequent must be $\Gamma \longrightarrow D \supset \exists G$, which is clearly an $\mathbf{I}'$-sequent.

$\vee$-$\supset$: For this rule to be applicable, the previous sequent must be $\Gamma \longrightarrow D \supset (G_1 \vee G_2)$, which is clearly an $\mathbf{I}'$-sequent.

$\perp -R$: By lemma 6.5.3, this rule cannot occur in Ξ.

□

As noted above, the Kripke-like model incorporates features of the logic of constant domains, and so we desire I' to be at least as powerful as this logic. The standard proof-theoretic way to enhance intuitionistic logic in this way is to add the following rule (in a Hilbert-type proof system):

$$\forall x\, (\phi \vee \psi(x)) \supset (\phi \vee \forall x\, \psi(x))$$

We may incorporate this rule into the sequent system for I' by adding the following inference rule:

$$\frac{\Gamma \longrightarrow \forall x\, (\phi \vee \psi(x))}{\Gamma \longrightarrow \phi \vee \forall x\, \psi(x)}\ \text{CD}$$

We refer to the extended proof system as I'_{CD}.

However, if we restrict our attention to D and G formulae, then this rule can never be used, as neither of the above two consequents are D formulae, and as we do not allow universal quantification in goals, they are not G formulae either. Hence, this rule will not affect the provability relations between D formulae or between G formulae. We give a formal proof of this result below.

Proposition 6.5.6 *Let $\Gamma \longrightarrow F$ be an $\mathbf{I}'$-sequent. Then any $\mathbf{I}'_{CD}$-proof Ξ of $\Gamma \longrightarrow F$ is an $\mathbf{I}'$-proof.*

Proof: We need only show that the CD rule is never used in Ξ.

We proceed by induction on the height of Ξ. It is clear that the base case holds, as the formula $\forall x\, (\phi \vee \psi(x))$ is not an atom.

Hence we the induction hypothesis is that all I'_{CD} proofs of $\Gamma \longrightarrow D$ or $\Gamma \longrightarrow G$ of no more than a given size contain no occurrences of the CD rule.

Consider the final rule used in Ξ. As $\phi \vee \forall x\, \psi(x)$ is neither a D formula nor a G formula, the final rule cannot be the CD rule. Hence by lemma 6.5.5 all the previous sequents must be $\mathbf{I}'$-sequents, and so by the hypothesis, the CD rule is not used in the sub-proofs of any of the prior sequents.

Hence by induction we get that the CD rule does not occur anywhere in Ξ, and so Ξ is an $\mathbf{I}'$-proof.

□

Next we show that when we restrict $\vdash_{I'}$ to $\mathcal{D} \times \mathcal{G}$, we get precisely $\vdash_s$, i.e. that our extension to intuitionistic logic does not affect the provability relation between programs and goals.

Proposition 6.5.7 *Let D be a D_{mod} program and G be a G_{mod} goal. Then*

$$D \vdash_{I'} G \Leftrightarrow D \vdash_s G$$

Proof: As $D \vdash_s G \Leftrightarrow D \vdash_I G$ and $\vdash_I \subseteq \vdash_{I'}$, we only need to show that $D \vdash_{I'} G \Rightarrow D \vdash_I G$. Let Ξ be an I'-proof of $D \longrightarrow G$. By lemma 6.5.4, all antecedents in Ξ are sets of D formulae, and so the only uses of the rules $\exists$-$\supset$ and $\vee$-$\supset$ are of the form

$$\frac{\Gamma \longrightarrow D' \supset \exists x\, G'}{\Gamma \longrightarrow \exists x\, D' \supset G'} \qquad \frac{\Gamma \longrightarrow D' \supset (G_1 \vee G_2)}{\Gamma \longrightarrow (D' \supset G_1) \vee (D' \supset G_2)}$$

where Γ is a set of D formulae. It is clear that we may view Γ as either a set of D formulae or as a conjunction of D formulae.

Now

$$\begin{aligned}\Gamma \vdash_I D' \supset \exists x\, G' &\Leftrightarrow \Gamma \vdash_s D' \supset \exists x\, G' \\ &\Leftrightarrow \Gamma, D' \vdash_s \exists x\, G' \\ &\Leftrightarrow \Gamma, D' \vdash_s G'[t/x] \text{ for some term } t \in \mathcal{U}\end{aligned}$$

$$\Leftrightarrow \Gamma, D'[t/x] \vdash_s G'[t/x] \text{ as } x \text{ is not free in } D'$$
$$\Leftrightarrow \Gamma \vdash_s D'[t/x] \supset G'[t/x]$$
$$\Leftrightarrow \Gamma \vdash_s \exists x\, D' \supset G'$$
$$\Leftrightarrow \Gamma \vdash_I \exists x\, D' \supset G'$$

Similarly

$$\begin{aligned}\Gamma \vdash_I D' \supset (G_1 \vee G_2) &\Leftrightarrow \Gamma \vdash_s (D' \supset (G_1 \vee G_2)\\ &\Leftrightarrow \Gamma, D' \vdash_s G_1 \vee G_2\\ &\Leftrightarrow \Gamma, D' \vdash_s G_1 \text{ or } \Gamma, D' \vdash_s G_2\\ &\Leftrightarrow \Gamma, D' \supset G_1 \text{ or } \Gamma \vdash_s D' \supset G_2\\ &\Leftrightarrow \Gamma \vdash_s (D' \supset G_1) \vee (D' \supset G_2)\\ &\Leftrightarrow \Gamma \vdash_I (D' \supset G_1) \vee (D' \supset G_2)\end{aligned}$$

Hence the forms of the two rules $\exists$-$\supset$ and $\vee$-$\supset$ are derived rules of intuitionistic logic on the fragment $\mathcal{D} \times \mathcal{G}$, and so any $\mathbf{I}'$-proof of $D \longrightarrow G$ is an $\mathbf{I}$-proof of $D \longrightarrow G$, i.e. $D \vdash_{I'} G \Rightarrow D \vdash_I G$.

□

In this way I' proves the desired equivalences between programs and between goals, but does not affect computations, i.e. the consequence relation between programs and goals. This property is also reflected in the following two corollaries of the above proposition.

Corollary 6.5.8 *Let D_1 and D_2 be D_{mod} programs and let G be a G_{mod} goal.*

$$\textit{If } D_1 \vdash_s G \textit{ and } D_2 \vdash_{I'} D_1 \textit{ then } D_2 \vdash_s G$$

Corollary 6.5.9 *Let D be a D_{mod} program and let G_1 and G_2 be G_{mod} goals.*

$$\textit{If } D \vdash_s G_1 \textit{ and } G_1 \vdash_{I'} G_2 \textit{ then } D \vdash_s G_2$$

Thus the same goals may be derived from programs which are provably equivalent in I', and goals which are provably equivalent in I' behave identically.

We are now in a position to prove the equivalence of the two relations on D formulae.

Theorem 6.5.10 *Let D_1 and D_2 be D_{mod} programs. Then*

$$D_1 \vdash_{I'} D_2 \Leftrightarrow D_1 \models_D D_2$$

Proof:

($\Rightarrow$): $D_1 \vdash_{I'} D_2$ implies that

$$\forall G\ D_2 \vdash_{I'} G \Rightarrow D_1 \vdash_{I'} G$$

and by proposition 6.5.7 this is equivalent to

$$\forall G\ D_2 \vdash_s G \Rightarrow D_1 \vdash_s G$$

which in turn is equivalent by theorem 5.4.5 to

$$\forall G\ T^\omega(I_\perp), D_2 \models G \Rightarrow T^\omega(I_\perp), D_1 \models G$$

Now in particular, we have that

$$T^\omega(I_\perp), D_2 \models A \Rightarrow T^\omega(I_\perp), D_1 \models A$$

for any atom A, and so

$$A \in \mathrm{pos}(T^\omega)(I_\perp)(D_2) \Rightarrow A \in \mathrm{pos}(T^\omega)(I_\perp)(D_1)$$

i.e.

$$D_1 \models_D D_2$$

($\Leftarrow$): $D_1 \models_D D_2$ implies that $T^\omega(I_\perp)(D_2) \sqsubseteq T^\omega(I_\perp)(D_1)$, and so by lemma 5.3.2 we have

$$\forall G\ T^\omega(I_\perp), D_2 \models G \Rightarrow T^\omega(I_\perp), D_1 \models G$$

which is equivalent by theorem 5.4.5 to

$$\forall G\ D_2 \vdash_s G \Rightarrow D_1 \vdash_s G$$

and by theorem 6.4.3 this implies that $D_1 \vdash_{I'} D_2$.

□

The corresponding result for G_{mod} formulae is more problematic. One way to derive a model-theoretic notion of equivalence between goals is to consider two goals to be equivalent if there is no program on which they behave differently. This is a natural dual to theorem 6.4.3, as we may interpret that result as stating that if there is no goal which behaves differently for two given programs, then the programs are equivalent. For these reasons the following definition seems a natural way to define $\models_G$.

Definition 6.5.4 *Let G_1 and G_2 be G_{mod} goals. Then*

$$G_1 \models_G G_2 \text{ iff } \forall w \; w \not\models G_1 \Rightarrow w \not\models G_2$$

It is easy to show an analogous lemma to lemma 6.5.1. This is done below.

Lemma 6.5.11 *Let G_1 and G_2 be G_{mod} goals and let D be a D_{mod} program. Then*

$$\text{If } D \not\models G_1 \text{ and } G_1 \models_G G_2 \text{ then } D \not\models G_2$$

Proof: $G_1 \models_G G_2$ implies $D \not\models G_1 \Rightarrow D \not\models G_2$, and so clearly the result holds.

□

We also expect the analogous result to theorem 6.4.3 to hold. A formal statement is given below.

Conjecture 6.5.12 *Let G_1 and G_2 be G_{mod} goals. Then*

$$(\forall D \; D \vdash_s G_1 \Rightarrow D \vdash_s G_2) \Rightarrow G_1 \vdash_{I'} G_2$$

The problem in proving this result is deriving the analogous result to lemma 6.3.1; in particular we require that the operational equivalence of G_1 and G_2 over all D_{mod} programs is sufficient to establish the operational equivalence of G_1 and G_2 over all D_{HHF} programs. The difficulty is that in the corresponding case for the operational equivalence of programs, there was a simple way to decompose goals into simpler goals, and so there was a straightforward induction argument. However programs are not so easily decomposed, and so it seems unlikely that an inductive argument will work. A model-theoretic argument seems to be the most promising, but has not borne fruit yet.

If we had such a result, then we may expect a proof of the above conjecture to run as follows: from $\forall D\ D \vdash_s G_1 \Rightarrow D \vdash_s G_2$, by the assumed result and proposition 6.2.1 we may conclude that $\mathrm{defp}(\forall(Q_i\theta)) \vdash_s G_2$, where $\forall(Q_i\theta)$ is an answer form of G_1. From this, proposition 6.5.7 and another application of proposition 6.2.1, we get that $\forall(Q_i\theta) \vdash_{I'} G_2$, i.e. that any answer form of G_1 is also an answer form of G_2, and as this must hold for *any* answer form of G_1, we conclude that $G_1 \vdash_{I'} G_2$.

A further conjecture is that the two notions of consequence for G formulae coincide. One direction may be easily shown, and the other follows from the above conjecture.

Proposition 6.5.13 *Let G_1 and G_2 be G_{mod} goals.*

$$\textit{If } G_1 \vdash_{I'} G_2 \textit{ then } G_1 \models_G G_2$$

Proof: If $D \vdash_{I'} G_1$ then by proposition 6.5.7, $D \vdash_s G_1$, and so by corollary 6.5.9, if $D \vdash_s G_1$ and $G_1 \vdash_{I'} G_2$ then $D \vdash_s G_2$. By theorem 5.4.5 this in turn is just $D \models\!\!\!\models G_1$ and $G_1 \vdash_{I'} G_2$ implies $D \models\!\!\!\models G_2$, and so $G_1 \vdash_{I'} G_2$ implies that $G_1 \models_G G_2$.

□

The conjecture below is the converse to proposition 6.5.13, and follows immediately from the conjecture above.

Conjecture 6.5.14 *Let G_1 and G_2 be G_{mod} goals.*

$$\textit{If } G_1 \models_G G_2 \textit{ then } G_1 \vdash_{I'} G_2$$

6.6 Discussion

We have seen that the natural logic in which to express the equivalence of D_{mod} programs and G_{mod} goals is slightly stronger than intuitionistic logic, but still less than classical logic, and is at least as strong as the logic of constant domains. However the properties of interest are affected by the fact that we are interested in only a fragment of first-order logic – namely the D_{mod} and G_{mod} formulae, although it is desirable to extend this fragment to include D_{HHF} and G_{HHF} formulae. Looking at a restricted class of formulae means that the possible worlds need not necessarily have a linear relation between them, as is suggested by *lc*. The Kripke-like model seems to be the best way to investigate I', as it is relatively well understood. One problem with the proof theory outlined above is that it is obviously hacked to derive the required results, rather than inherently natural. It is clearly desirable to find a more informative proof system.

An interesting result reported in [34] says that for formulae which do not contain $\exists$ or $\vee$, the logic of constant domains is equivalent to intuitionistic logic, and so the extensional interpretation of $\forall$ is then a natural way to implement universal quantification. This also suggests that our choice of I' is a natural one.

One way to think of the difference between I' and intuitionistic logic is to think of I' as preserving the constructive spirit of intuitionistic logic but using a slightly stronger proof system in order to allow a more uniform method of finding proofs, and so being more amenable to implementation. The main problem with intuitionistic logic for first-order hereditary Harrop formula is the treatment of the connectives on either side of $\supset$, as specified by the five equivalences below.

$$D \supset (\exists x G) \equiv_{I'} \exists x (D \supset G)$$

$$D \supset (\forall x G) \equiv_{I'} \forall x (D \supset G)$$
$$D \supset (G_1 \vee G_2) \equiv_{I'} (D \supset G_1) \vee (D \supset G_2)$$
$$D \supset (G_1 \wedge G_2) \equiv_{I'} (D \supset G_1) \wedge (D \supset G_2)$$
$$D \supset (D' \supset G) \equiv_{I'} (D \wedge D') \supset G$$

As the first and third equivalences are not intuitionistically valid, this uniformity is not maintained in intuitionistic logic. We may think of this uniform property as reflecting the underlying structure of the formulae involved. A uniform proof proceeds by decomposing the structure of the goal, and when the simplest structural component, i.e. an atom, is reached, the usual unification and backtracking methods may be used to generate another goal, which is then structurally decomposed, and so on. Clearly this process relies on the fact that atomic goals may be easily handled, and hence uses a particularly restricted form of consequent (i.e. the right hand side of $\supset$) in programs. In this way it is the structural properties of the formulae used as programs that leads to a proof system which is sufficiently straightforward that it may be interpreted directly as computation.

As the result of section 6.1 suggest, the class of formulae used may have the same expressive power as a larger class of formulae, and so may be used as an implementation vehicle for the larger class of formulae. As mentioned earlier, we may allow $G \supset D$ as a definite formula, as we know that we can re-write such a formula as a conjunction of definite formulae in which the conclusions of all implications are just atoms. In this way the smaller class of formulae represents a compromise between expressive power and feasibility, in that we can express the same information in a more restrictive language, which is all that is necessary to be implemented. This principle of maximality of information seems important when discussing the semantic nature of I'. From programming principles, it seems intuitively clear that it is best if the ratio of information to structure is as large as possible. This not only leads to greater feasibility, but also presumably to greater clarity.

This principle of maximality of information may also be used to explain the choice of formulae. Due to the fact that we may define D formulae in either of the following ways,

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset D$$

if we were to allow $\exists x D$ and $D_1 \vee D_2$ as definite formulae, then all formulae of the positive fragment of first-order logic may be used as programs. However, neither of the formulae $\exists x D$ and $D_1 \vee D_2$ may be considered to convey a maximal amount of information, as if $\exists x D$ is true, there must be an instance t of x such that $D[t/x]$ is true, and so if the programmer knew such a t, more information would be conveyed by the program $D[t/x]$ than by the program $\exists x D$. Similarly, in order that $D_1 \vee D_2$ be true, it must be the case that either D_1 or D_2 is true, and so either of the latter two programs will convey more information than the former program. In this way hereditary Harrop formulae force the programmer to give the maximal amount of information.

From the point of view of possible worlds, we may think of this property as minimising the necessity to examine future worlds in order to prove something about the current world. This is clear from the definitions of truth under $\not\models$ for $D \supset G$, $\neg A$ and $\forall x G$ when compared to their counterparts in a Kripke model. The only time it is necessary to move to a world above the current one is in order to prove an implication, and the only world that is considered is the extension to the current world given by assuming the antecedent of the implication. In this way our philosophical thrust is somewhat different from that of intuitionistic logic. The Kripke model semantics is usually motivated by the idea of an ideal mathematician who increasingly builds up knowledge and who has unlimited resources. We may think of this ideal mathematician as answering the question "What is true?", and hence searching for truth in a timeless fashion. If a given statement involves examining several possible futures, then he, not unlike the approach of a brute force chess program, explores them all and comes to a conclusion.

This is not really in keeping with the notion of programming, which is more concerned with finding out what is known *now*, rather than what is currently known and what will be discovered in the future. In contrast with the global approach of the ideal mathematician, who starts from no assumptions and aims

to explore all possible worlds eventually, we think of exploring the possible worlds from within, in that we envisage our explorer setting out from a given spot in the partial order and seeing what may be deduced from there. Hence our exploration is a part of the overall exploration, but is local rather than global in nature.

The difference in perspective seems to be reflected in the difference between intuitionistic logic and the intermediate logic I'. In the former case, we are concerned with a particular logical system, and wish to explore a variety of possibilities. In the latter case, we are concerned with a particular class of formulae, and we are thus able to come to stronger conclusions than is possible in intuitionistic logic. We do not wish to consider all possible models, but generally only the models of a particular program. In this way we are generally more interested in consequence in I' than theoremhood in I', and as such we are able to use a more interesting model theory than is given by related intermediate logics such as *lc*. Overall, the fact that we have a narrower motivation than that of intuitionistic logic means that we can derive appropriately stronger results.

Chapter 7

Meta-programming Features of Hereditary Harrop Formulae

In this chapter we use some results from the previous chapter to allow memoisation to take place for a large class of programs. Due to the kind of formulae needed to memoise goals, this leads to a natural separation into an object level and a meta-level. This separation may be used for several common programming tasks, and seems to be a natural way to explore memoisation properties. We also discuss the possibilities for extending memoisation to full first-order hereditary Harrop formulae, which would allow a greater degree of flexibility at the object level. Another issue is the use of implication in the bodies of clauses as a meta-programming device. Whilst there are many meta-programming tasks which require the use of programs as objects, and are hence necessarily higher-order, there are some which may be performed in a first-order framework. We give several examples of how this may done.

7.1 Memoisation Properties

In chapter 5 we gave a Kripke-like model for D_{HHF-} programs, that is

$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$
$$G := A \mid \neg A \mid \exists x G \mid \forall x G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G$$

As has been mentioned above, the Kripke-like model naturally incorporates a notion of program development in the form of the reachability relation between worlds. We have seen how the reachability relation given above may be thought of as requiring that all extensions of the current program be consistent. We may use this notion to study the programming process, as we may consider the programmer to begin at the bottom world and work his way up to the final program via the access relation. In order to carry out such a process, the programmer needs some specification of the desired program's behaviour, which we may think of as some particular interpreted world, i.e. $\langle S, F\rangle$ where S is the set of atoms which the programmer desires to succeed, and F is the set of atoms which the programmer desires to fail. We can then think of the development/debugging process as finding a program P such that $T^{\omega}(I_{\perp})(P) = \langle S, F\rangle$. Naturally one way to specify $\langle S, F\rangle$ is via an interpretation I and a program P such that $I(P) = \langle S, F\rangle$. In this way our framework may be an interesting one in which to study formal program development.

At the conclusion of the development of the program, so that we now have $T^{\omega}(I_{\perp})(P)$ coinciding with the program's specification in the above sense, we may consider that the programmer passes the program to the machine so that deductions may be made from the knowledge encoded in the program. These deductions take the form of searching for uniform proofs of goal formulae, but it is interesting to note that a successful search for a uniform proof of a goal G may be thought of as converting G into a more definite statement.

As discussed in chapter 6, we may think of this process as searching for information to complete our knowledge of the truth of the goal G. We saw that if $\langle D, N\rangle \vdash_s G$, then in many cases the extra information that has been computed enables us to express the success form of G as a definite formula D'. [1] We may then record this information as a larger program $\langle D \cup \{D'\}, N\rangle$ which has the same consequences as $\langle D, N\rangle$. This larger program may lead to shorter proofs than the

[1]The exceptions are the cases where the failure of Proposition 6.1.4 in the presence of negation is relevant.

smaller one, as we may use previous computations in subsequent proofs, rather than recompute known results. Thus the larger program has the same meaning, but may be more efficient, as we can record the results of previous computations, rather than throwing them away.

This technique is known as *memoisation* [72], and is known to be useful for avoiding redundant computation, as it allows known results to be stored for later use, thus reducing the amount of work that must be done. In our case, this consists of storing consequences of the program, so that the proof search process need not start from scratch each time that a new goal is presented. This is akin to the way that a mathematical theory is built up; one starts from the axioms of a given theory, and derives some basic results. Subsequent proofs then refer to this list of results, rather than resort to the axioms each time, and in so doing often re-discovering known proofs. In this way the reasoning process produces an increasing set of consequences of the axioms, and later proofs refer to this set of consequences, rather than the original axioms alone. Some aspects of memoisation for D_{mod} programs were discussed in [77], where the emphasis was on the memoisation of atomic goals.

We saw in the previous chapter how we may think of the computation process as removing disjunctions from goals and replacing existential quantifiers with zero or more universal quantifiers, which is very similar to the processes described in section 6.1 which exploit the structure of programs to show that disjunctions and existential quantifiers are not necessary. Thus the memoisation process seems to be a natural extension of this process of "tightening up" the program.

However there are some technical difficulties with the answer form of the goal. As shown above, this is of the form $\forall(G_i\theta)$ where G_i is a $G_{object-}$ formula [2], i.e. G_i is of the form

[2]Of course, if the goal contains a negation, it may not have such an answer form, due to the failure of Proposition 6.1.4 for such goals. However, for the purposes of this discussion we will assume that such an answer form exists.

$$G := A \mid \neg A \mid G_1 \wedge G_2 \mid D \supset G$$
$$D := A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A$$

It is clear that $\forall(G_i\theta)$ is very close to a $D_{object-}$ formula. That is, by proposition 6.2.2 we have that defq$(G_i\theta)$ is a D_{meta} formula and so we may memoise $\forall(G_i\theta)$ by adding $\forall(\text{defq}(G_i\theta))$ to the program if this is a $D_{object-}$ formula, and by proposition 6.2.1 this is equivalent to the original formula.

There are two possibilities which may ensure that $\forall\text{defq}(G_i\theta)$ is not such a formula: the possible occurrence in G_i of a subformula $D' \supset G'$, and the possible occurrence of a negation in G_i. The first possibility will require that universal quantifiers be allowed in goals, as whilst $D \supset A$ is a $G_{object-}$ formula, in order to view it as a D formula, we need to allow the possibility that D contains universal quantifiers. The second possibility will require that we treat negated literals in a symmetric way to positive literals (i.e. atoms).

For example, if we find that the goal $(\forall x p(x)) \supset \neg q$ succeeds, then we cannot store this formula as a clause due to the fact that the conclusion is not an atom, and the premise is not a G_{mod-} formula.

The first possibility destroys the property of the proof search process that disjunctions are not necessary in any answer form of a goal, as in a goal such as $\forall x$ even$(x) \vee$ odd(x), the disjunction cannot be removed as it was previously. As discussed in section 2.4.3, the second possibility involves non-trivial questions of consistency. These considerations would tend to imply that there is no appropriate memoisation property for D_{mod-} formulae.

Fortunately, this does not seem to be the case. We imagine that the programmer desires the greatest amount of flexibility possible when writing programs, and so whilst any program he or she writes which contains existential quantifiers or disjunctions may be re-written as an operationally equivalent program which contains neither connective, the programmer would presumably wish to use the more general form for writing programs and the more restrictive one for proving properties of programs and so forth, which may be easier in the more restrictive case. In this way there are two ways to view the program: the human view, in which we

desire as great a degree of generality and flexibility as possible, and the machine view, in which we desire as great a degree of specific information as possible. The precise level of generality desired in each view is not the issue here; what is important is that the programmer and the machine tend to have two different attitudes towards the same program.

These two different views may be characterised as an object level and a meta-level. The programmer is generally concerned with the data which the program will manipulate, and so thinks of the program as a way to capture and describe the properties of the real objects of his or her concern. On the other hand, the machine sees the same program as an object which will require certain resources, so that the program itself, rather than the objects it manipulates, is the prime concern.

As we envisage memoisation being performed by the system and not by the programmer, a natural way to resolve the above difficulties is to define object level programs as above, but to allow a more general class of programs as meta-level programs, i.e. D_{meta} formula, which may be defined as follows:

$$D := A \mid \neg A \mid \forall x D \mid D_1 \wedge D_2 \mid G \supset A \mid G \supset \neg A$$
$$G := A \mid \neg A \mid \exists x G \mid \forall x G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G$$

Whilst we allow some potentially more troublesome programs at the meta-level, memoising the object level programs will not "use" all the possibilities. One example of this is given by theorem 6.1.6, which states that existential quantification and disjunction are redundant constructs in D_{object} programs. Also, the fact that we allow negations in the heads of clauses in meta-level programs may lead to inconsistent programs, but as we shall see, memoising object level programs can never lead to an inconsistency. Thus we wish any program which satisfies the appropriate definition to be an object level program, but we do not wish for a corresponding property for meta-level programs. This is due to the fact that meta-level programs are constructed in a known way from object level programs. Hence, any meta-level program will satisfy the above definition, but we do not necessarily wish to consider anything which satisfies the definition as a meta-level program,

as we will only consider meta-level programs which arise from the memoisation of object level programs. So in this case the only genuine difference between the two classes of programs will be that in meta-level programs, universal quantifiers may appear in the body of clauses and negations may occur in the heads of clauses. The combination of these two properties will then allow us to consider the answer form $\forall(G_i\theta)$ of a goal G as a program clause, and so we may consider that the program at the meta-level is now $\langle D \cup \{\forall\text{defq}((G_i\theta))\}, N\rangle$.

This seems a natural way in which to view memoisation, in that the program has not changed, and so the programmer's view of it should be the same, whereas the machine's view has been extended, in that some of the consequences of the program have been calculated and then reflected back into the program. Hence the internal knowledge contained in the program remains the same; external knowledge about the program has been increased.

For example, consider the program

$$p(f(a)) \supset q$$
$$r(f(a))$$

where r is completely defined. The goals $(\forall x\, p(x)) \supset q$ and $\neg r(a)$ both succeed, and so the memoised meta-level program is

$$p(f(a)) \supset q$$
$$r(f(a))$$
$$(\forall x\, p(x)) \supset q$$
$$\neg r(a)$$

It is obvious that $\forall(A)$ and $\forall(\neg A)$ are both meta-level programs. Note also that for $\forall(\neg A)$ to be an answer form, we must have that every instance of $\neg A$ succeeds, i.e. that every instance of A fails, and so there can be no inconsistency arising from memoising negated formulae in this way. Thus we do not allow any new form of negation at the meta-level; all we are concerned with here is memoising the object level program, and so whilst $\neg A$ may appear in a meta-level program,

we still require that negation only be applied to completely defined predicates. It is this restriction that allows the negations in the meta-level programs to be consistent, as then we may apply the NAF rule, and it is obvious that if A fails it is consistent to add $\neg A$ to the program.

In this way we do not expect the programmer to use negated atoms as definite clauses, but that the memoisation procedure may, and so whilst we may not extend completely defined predicates *positively*, we may extend such predicates *negatively*, provided that the extension is consistent with the program. For example, if $P = \langle D, N\rangle$ is the standard append program (so that append $\in$ den(N)), then we wish to be able to memoise $\forall x\ \neg\text{append}([],[],[x])$, and so we desire that the program $P' = \langle D \cup \{\forall x\ \neg\text{append}([],[],[x])\}, N\rangle$ be accessible from P. Note that the goal $\text{append}([],[],[1,2]) \supset G$ will cause no problems, because the program $\langle D \cup \{\forall x \neg\text{append}([],[],[x])\} \cup \{\text{append}([],[],[1,2])\}, N\rangle$ is not accessible from P', due to the fact that the larger program extends the positive definition of a completely defined predicate of P', and so $\text{append}([],[],[1,2]) \supset G$ will fail. Thus the access relation between worlds needs only to look at the *positive* extensions to a completely defined predicate, as all negative extensions which are added in the safe way described above are consequences of the program.

Now that we have a precise description of how memoisation may be performed in this context, we may consider the computation process as a natural continuation of the programming process, which started from the empty program and proceeded to larger and larger worlds as the process went on. After this phase is concluded, we may consider the process of moving upwards in the cone of worlds to continue, as we may memoise each successful goal, and so move to successively larger worlds which are accessible from (and hence consistent with) the original program. Thus computation in this setting is envisaged in precisely the same way as the ideal mathematician of the intuitionistic school; a process of ongoing acquisition and assimilation of knowledge.

7.2 Memoisation for Larger Classes of Programs

The results of sections 6.1 and 6.2 depend crucially on the fact that we are using G_{mod} formulae and not G_{HHF} formulae as goals. We may interpret these results as indicating that the above class of programs and goals is in some sense maximal, as any extension to the class of goals (other than extending the negative fragment) will lose the memoisation property. We cannot extend the above approach in the obvious way to the case when universal quantification is allowed in goals. The problem is that a universal quantifier may "block" the process of pushing the disjunctions outward. For example, consider the program below.

$$(\forall x\, p(x) \vee q(x)) \supset r$$

There is no way to move the disjunction outside the universal quantification, as there was previously.

An alternative conclusion is that the above class of programs needs to be extended so that goals such as

$$\forall x\ \mathrm{even}(x) \vee \mathrm{odd}(x)$$

may be memoised. This requires a significant extension to the class of programs defined above.

Such an extension seems undesirable, as an essential property of uniform proofs is that they depend only on the structure of the goal, and so a large goal may be systematically broken down into smaller goals (possibly increasing the program along the way), resulting eventually in an atomic goal, which may then be matched against the head of a clause in the program, and a new goal generated. To allow the above goal as a clause would mean that this systematic method of searching for a proof of a goal would break down, in that the search for a proof of the goal even(0) $\vee$ odd(0) may need to do more than search either for a proof of even(0) or for a proof of odd(0). It is not clear that such a search procedure still retains

the flavour of logic programming due to the fact that uniform proofs will not be complete for these programs. Whilst the approach of disjunctive databases has been along these lines [83,98], these are designed to deal with uncertain information rather than as a programming language per se. In this way we need to keep our information certain in order to maintain the property that uniform proofs are complete.

One important observation about clauses is that they represent information in a maximal way. For example, given the program

$$\begin{aligned} &\text{even}(0) \\ \forall x\ &\text{even}(x) \supset \text{even}(s^2(x)) \\ &\text{odd}(s(0)) \\ \forall x\ &\text{odd}(x) \supset \text{odd}(s^2(x)) \end{aligned}$$

it is clear that $\text{even}(s^n(0)) \vee \text{odd}(s^n(0))$ succeeds for all $n \geq 0$, but it is not possible to derive the above program from the statement $\forall x\ \text{even}(x) \vee \text{odd}(x)$. In this way the fact that the heads of clauses are just atoms allows us to build up to formulae of arbitrary depth, and so we can use the program to analyse goals of an arbitrary size. If however the heads of the clauses were of some larger complexity, e.g. containing at least three logical connectives, then it is not clear how we could ascertain the truth or otherwise of an atomic goal. In this way clauses allow us, indeed *require* us, to express information in a manner which allows the greatest number of conclusions to be reached, and it is this maximality property which allows us to use uniform proofs, rather than arbitrary proofs.

Thus in order to allow D_{HHF} programs to have the memoisation property, we need to find some way of expressing goals such as $\forall x\ \text{even}(x) \vee \text{odd}(x)$ in a clausal form.

One method of deriving results similar to those of section 6.1 in the presence of universal quantifiers in goals is to interpret universal and existential quantifiers as shorthands for an infinite conjunction or disjunction respectively. In this way the versions of the program which do not contain disjunctions or conjunctions need

not be finite. The infinitary version of the program is not particularly interesting in its own right, as we cannot hope to write it down explicitly, but it may serve as a useful way to manipulate the program. Hence, our interest in these constructs is restricted to those infinitary programs which are generated from finite first-order hereditary Harrop formula programs in a certain way, just our interest in meta-level programs was restricted to the meta-level programs which were generated from memoisation of object level programs.

Under this scheme a program such as

$$(\forall x\ p(x)) \supset q$$

may be considered equivalent to

$$(\bigwedge_{t \in \mathcal{T}} p(t)) \supset q$$

and similarly a program

$$\exists x\ p(x) \supset q$$

may be considered equivalent to

$$(\bigvee_{t \in \mathcal{T}} p(t)) \supset q$$

which in turn is just

$$\bigwedge_{t \in \mathcal{T}} p(t) \supset q$$

Once we have replaced the quantifiers which occur negatively in the program (i.e. those which occur as part of a goal) by such infinitary constructs, we then need to remove the disjunctions from the program. This is a little more tricky than in the previous case, as we may need to use an infinite number of infinitary conjunctions or disjunctions. For example, consider the goal

$$\forall x\, \exists y\ p(x, y)$$

We replace the universal quantifier by an infinite conjunction to get

$$\bigwedge_{t \in \mathcal{T}} \exists y\, p(t, y))$$

so that we have

$$\bigwedge_{t \in \mathcal{T}} (\bigvee_{t' \in \mathcal{T}} p(t, t')))$$

Now in order to push the infinitary disjunction outside the conjunction, we require an infinite number of infinitary disjunctions, as we need to represent all mappings of $\mathcal{T}$ to itself, i.e. $p(t, f(t))$ for any such map f. This we need to allow not only the infinitary constructs, but to allow them to be used infinitely often. Hence, the size of the "formulae" involved is considerable, and so this approach, whilst retaining some attractive features, does not appear to be the best solution to the problem.

An alternative solution is to use Skolem functions to push existential quantifications outwards, and so the problem with the interaction between universal and existential quantifiers may be resolved by using quantification over functions as well as variables. This allows us to use Skolem constants in the standard way, so that the goal

$$\forall x\, \exists y\, p(x, y)$$

may be re-written as

$$\exists f\, \forall x\, p(x, f(x))$$

This function f is considered semantic rather than syntactic, in the sense that f is not necessarily one of the functions named in the signature, but represents a functional relationship between terms in $\mathcal{T}$, and so dealing with such a function will involve considering all mappings of $\mathcal{T}$ to itself.

This requires an extension to the usual unification procedure, but such an extension should not incorporate any unusually difficult problems, as it is "essentially" first-order [74].

This approach will not help us memoise goals such as

$$\forall x \ \mathrm{even}(x) \vee \mathrm{odd}(x)$$

and so one way to tackle the problem may be to combine the two approaches, so that the Skolem functions are used to replace existential quantifications within universal quantifications, and infinite disjunctions are used to replace disjunctions within universal quantifications.

Note that the first approach may be considered as an explicit form of Skolemisation, in that rather than giving a name to the semantic function mapping $\mathcal{T}$ to itself, we consider all possibilities for the value of such a function. This is really a different side of the same coin: in the implicit case, the work is done in the extension to the unification process; in the explicit case, the work is done in manipulating the program generated. Either way we need to consider the possible mappings of $\mathcal{T}$ to itself.

7.3 Programming at the Meta-Level

There has been much attention given recently to meta-programming issues in logic programming [2]. An obvious setting for meta-level logic programming is higher-order logic. This has the natural advantage that all computation, at the object or meta-level, may be thought of as logic programming, rather than being logic programming at the object level, and some other programming paradigm at the meta-level, and possibly in a significantly different meta-language. Given that the meta-language itself may be interpreted as some logic, then we may claim that any features explained by the meta-language but not by the object level have been given a logical interpretation, thus giving a more appropriate semantics to a logic programming language. Better still, if the meta-language is a logic programming language, then we may easily program at the meta-level, with a natural and well-understood relation to the object level.

A computational form of a higher-order logic has been given in [79], and is based on a subset of higher-order intuitionistic logic known as higher-order hereditary

Harrop formulae. Some other approaches in this area may be found in [91,47,97, 92,102]. There are some features of the meta-theory of Horn clauses which require the use of programs as first-class objects, and thus are necessarily higher-order. One such feature is that of program transformation, and has been studied in [44]. However, there are aspects which may be studied in a first-order setting, i.e. by using first-order hereditary Harrop formulae as a meta-language.

An example of such an aspect was given by Gabbay et al. In [37] it is shown how some procedural control rules for Horn clauses may be given a logical explanation in N-Prolog, an extension of Prolog, which is similar to first-order hereditary Harrop formulae. We show how the same explanation will hold for first-order hereditary Harrop formulae. One example of such a control rule is the restriction that a goal may only use certain clauses in the program. As the meta level represents a subset of first-order intuitionistic logic, seemingly non-logical or procedural operations at the object level, such as adding a clause to a program or specifying that only certain clauses are to be used in the computation of a particular goal, may be given a logical interpretation at the meta level.

Meta-theoretic aspects of logic programming have been considered by Bowen and Kowalski [10], Bowen and Weinberg [11], and Gallaire and Lasserre [40]. The approach by Bowen and Kowalski, extended by Bowen and Weinberg, is centred around a Demo predicate, which takes a program as its first argument and a goal as its second, and $\text{Demo}(P, G)$ is true if the derivation of the goal G from the program P is successful. This approach is strictly higher-order, as it involves the use of programs as objects. However, we can achieve something equivalent in hereditary Harrop formulae by the use of implication. [3]

An interesting consequence of the results of chapter 6 is that we may use the techniques described above to achieve a similar function to that of the Demo

---

[3]Note that for Horn clauses (i.e. D_{Horn} formulae), classical logic and intuitionistic logic coincide, so that the fact that we use intuitionistic logic and Bowen and Kowalski use classical logic makes no difference to this discussion.

predicate for a wider class of programs and goals than Horn clauses – namely D'_{mod} programs and G_{mod} goals. Now as we know for such programs that $D \vdash_s G$ iff there is an answer form $\forall(G_i\theta)$ such that efnf(dfnf(D)) $\vdash_s \forall(G_i\theta)$ and $\forall(G_i\theta) \vdash_I G$ (when considered as meta-level programs), if we can memoise the formula efnf(dfnf(D)) $\supset \forall(G_i\theta)$, then this is effectively representing the Demo predicate in our language. We know that we can indeed memoise this formula, as we know that

$$\text{efnf}(\text{dfnf}(D)) \supset \forall(G_i\theta)$$

is intuitionistically equivalent to

$$\text{defp}(\text{efnf}(\text{dfnf}(D))) \supset \forall(\text{defq}(G_i\theta)))$$

and we may clearly write this formula as D_{meta} clause, as indicated by proposition 6.2.2.

Thus whilst our conception of object and meta level does not amalgamate the two, as is done in [10], it can achieve the same end.

A further observation that may be made in the light of proposition 6.2.2, as D_{mod} programs may be interpreted as meta-level goals and G_{mod} answer forms may be interpreted as meta-level programs, is that we may represent the equivalence of object level programs as meta-level programs. For example, if $\langle D_1, N\rangle$ and $\langle D_2, N\rangle$ are D_{mod} programs, then the formula $D_1 \supset D_2$ may be re-written as a G_{meta} formula provided that D_2 may be re-written as a G_{meta} formula, and by proposition 6.2.2 we know that this is the case. Similarly, the same formula may be considered as a D_{meta} formula provided that we may re-write D_1 as a G_{meta} formula, which we know is possible, and we may then use the techniques suggested by the definition of defp to produce the desired D_{meta} formula. This means that we may ask $D_1 \supset D_2$ as a goal, and if it succeeds, we may state it as a program. In this way we have not only a method for determining the equivalence of object level programs, but also a way to record the equivalences so derived. Thus we may use our conception of object level and meta-level programs as a calculus for the determining the equivalence of object level programs.

For example, let D_1 be $\exists x p(x) \supset q$ and D_2 be $p(a) \supset q$. Then $D_1 \supset D_2$ is $(\exists x p(x) \supset q) \supset (p(a) \supset q)$, and it should be clear that this is a G_{meta} formula as D_2 is a G_{meta} formula. Now D_1 may be re-written as $\forall x(p(x) \supset q)$, which is a G_{meta} formula, and hence $D_1 \supset D_2$ is equivalent to a D_{meta} formula.

Another possibility is to represent the information that two D_{mod} programs prove the same goal. We may rewrite the formula $(D_1 \supset G) \supset (D_2 \supset G)$ as a G_{meta} formula by first re-writing it as $((D_1 \supset G) \wedge D_2) \supset G$, and then re-writing the first occurrence of G as $\exists(G_1 \vee \ldots \vee G_n)$, so that we get $(\exists(D_1 \supset (G_1 \vee \ldots \vee G_n)) \wedge D_2) \supset G$, which is operationally equivalent to $\forall(D_1 \supset (G_1 \vee \ldots G_n)) \wedge D_2 \supset G$. This is turn is operationally equivalent to $\forall((\bigvee_i(D_1 \supset G_i)) \wedge D_2) \supset G$, which is clearly equivalent to $\forall \bigcup_i((D_1 \supset G_i) \supset G)$, and finally we get the G_{meta} formula

$$\forall(\bigwedge_i((\mathrm{defq}(D_1 \supset G_i) \wedge D_2) \supset G))$$

It is difficult to see how the original formula can be re-written as a D_{meta} formula due to the possibility that there may be existential quantifiers or disjunctions occurring positively in G. However, it is not hard to see that if $\forall(G_i\theta)$ is an answer form of G, then we may re-write the following formula as a D_{meta} formula

$$(D_1 \supset \forall(G_i\theta)) \supset (D_2 \supset \forall(G_i\theta))$$

This is done by moving D_2 to the left of the $\supset$ as is done above, and then using the defp technique to derive a D_{meta} formula.

This is perhaps not surprising, given that we know that the existential quantifier and disjunction are redundant constructs in D_{mod} programs, and so D_{mod} programs may be considered equivalent to D_{object} programs. These in turn are a sub-class of the meta-level core, i.e. those formulae which are both meta-level programs and meta-level goals, which may be given as

$$M := A \mid \neg A \mid \forall x M \mid M_1 \wedge M_2 \mid M_1 \supset M_2$$

Thus we are effectively dealing with programs which are within the core, and hence there is great potential for reflecting information back into the program.

Gallaire and Lasserre's approach, like that of Bowen and Kowalski, was also strictly higher-order, and used a system with a similar structure to Horn clauses to specify metarules. We shall see how we can imitate some similar features in hereditary Harrop formulae, especially clause ordering.

The work of Gabbay et al. had a slightly different motivation, which was to extend the formulae available to the programmer, in a manner similar to the extension provided by hereditary Harrop formulae. It was shown in [37] that this extension allowed for the description of control information, and so the language was used to express some of its own meta-theory. In our approach, all such control information would be encoded at the meta level, thus allowing the clear separation of the (object level) program and the control information, and yet retaining a natural connection between the object and meta-level.

Gabbay and Reyle [37] have shown how the meta-language property of first-order hereditary Harrop formulae extended to include negated atoms may be put to good effect. This revolves around the trick of "naming" clauses, so that we write

$$(D \supset \text{name}) \supset \text{name}$$

rather than just D, and similarly we write

$$(G \supset \text{name}) \supset \text{name}$$

in place of G, where name does not appear anywhere in either D or G. We may use this device to control which clauses are used in the computation and which are not, so that we may specify that the computation for the goal G may only use the clause D_1 and not the clause D_2 by renaming both clauses as

$$(D_1 \supset \text{name}_1) \supset \text{name}_1$$
$$(D_2 \supset \text{name}_2) \supset \text{name}_2$$

and the goal G as $(G \supset \text{name}_1) \supset \text{name}_1$, where name_1 and name_2 do not appear anywhere in D_1, D_2 or G. The idea is that due to the fact that the names are

new, the computation of G can only gain access to the clause D_1 and not to D_2. The first step is to add $G \supset \text{name}_1$ to the program, so that we get

$$G \supset \text{name}_1$$
$$(D_1 \supset \text{name}_1) \supset \text{name}_1$$
$$(D_2 \supset \text{name}_2) \supset \text{name}_2$$

and the goal is now name_1. As neither name_1 nor name_2 appear in G, the second clause cannot lead to a success, and so the only way for name_1 to succeed is to use the first clause above, so that the goal is now

$$D_1 \supset \text{name}_1$$

Now the program becomes

$$D_1$$
$$G \supset \text{name}_1$$
$$(D_1 \supset \text{name}_1) \supset \text{name}_1$$
$$(D_2 \supset \text{name}_2) \supset \text{name}_2$$

and the goal is name_1. Now provided that we can prevent name_1 from matching against the third clause above and thus looping, the only possible match is the second clause, so that the goal becomes G. As neither name_1 nor name_2 appear in D_1, D_2 or G, only D_1 can be used in the computation of G, and not D_2.

It is then shown in [37] how a simple loop detection mechanism may be provided, so that the above process may work. However, it is possible to provide the same facility in such a way that the loop detection is not required. If we name the two clauses in the following manner

$$(D_1 \supset \text{fire}) \supset \text{load}_1$$
$$(D_2 \supset \text{fire}) \supset \text{load}_2$$

and the goal as $(G \supset \text{fire}) \supset \text{load}_1$, where fire, load_1 and load_2 do not appear anywhere in D_1, D_2 or G, then we may achieve the same effect. The computation may be performed as follows: first we add $G \supset \text{fire}$ to the program, giving us

$$G \supset \text{fire}$$
$$(D_1 \supset \text{fire}) \supset \text{load}_1$$
$$(D_2 \supset \text{fire}) \supset \text{load}_2$$

and the goal load_1. The only clause that this can match is the second one, and so the new program is

$$D_1$$
$$G \supset \text{fire}$$
$$(D_1 \supset \text{fire}) \supset \text{load}_1$$
$$(D_2 \supset \text{fire}) \supset \text{load}_2$$

and the new goal is fire. This can only match the second clause, and so the next goal is G, thus allowing G only to use D_1 and not D_2, as none of fire, load_1 and load_2 can match anything in G.

In this way we can get a more straightforward implementation of deterministic exclusion, i.e. using one given clause but not another during computation. A refinement of the above idea can give non-deterministic exclusion, where it does not matter which clause is used, provided that once some clause is chosen, the other clause cannot be used. All we need to do is to use the same name for both clauses, so that D_1 and D_2 become

$$(D_1 \supset \text{fire}) \supset \text{load}$$
$$(D_2 \supset \text{fire}) \supset \text{load}$$

and G becomes $(G \supset \text{fire}) \supset \text{load}$. The computation proceeds by adding the clause $G \supset \text{fire}$ to the program and the goal is then load. Now there are two clauses it

may match, and so the next goal is $D_i \supset \text{fire}$, where $i = 1$ or 2, which involves adding D_i to the program, and the next goal is G. Hence, which clause is added may be decided arbitrarily, but once it is chosen, only that clause and not the other may be used in the computation of G.

If we consider the order of the clauses in the program to be significant (as happens in Prolog systems), then, as shown in [37], we may use this device to arbitrarily re-order clauses in the program. This is done by encoding the desired order of the clauses in a goal so that the clauses are added in the desired order before the goal is asked. For example, if there are three clauses

$$(D_1 \supset \text{fire}_1) \supset \text{load}_1$$
$$(D_2 \supset \text{fire}_2) \supset \text{load}_2$$
$$(D_3 \supset \text{fire}_3) \supset \text{load}_3$$

and the desired order is D_2 followed by D_3 followed by D_1, then this order may be achieved if we use the goal

$$(((((G \supset \text{fire}_2) \supset \text{load}_2) \supset \text{fire}_3) \supset \text{load}_3) \supset \text{fire}_1) \supset \text{load}_1$$

and we assume that clauses are added to the beginning of the program. Thus we first load D_1, then D_3, and finally D_2, thus giving the required order, before the computation of G. Thus we represent the six different object level programs (as the order of the clauses is significant here) by the same meta-level program, but with six different goals, one corresponding to each arrangement of the clauses. This is intuitively attractive, as the meta-level program is the same in each case, but the way it is used in computation varies, which seems to capture our intuitive notion of the difference between the six object level programs, i.e. that they all represent the same declarative information, but vary on the operational details.

Another possible application of this separation into an object level program and a meta-level program is to deductive databases. A deductive database often contains a set of conditions which must be satisfied at all times in order to guarantee that the information represented in the database is consistent and does not

get garbled. Such conditions are known as integrity constraints, and are usually checked after each database update to ensure that the changes will not render the database useless or redundant. Hence, the constraints are not themselves part of the database, but conceptually separate, and we consider a change to the object level program, i.e. a database update, to be conceptually very different from a change to the integrity constraints. By considering the integrity constraints as a meta-level program in the above sense, we get a natural representation of a database and its associated information.

The integrity constraints may be represented by G_{meta} formulae, and the database by two $D_{object-}$ formulae D_1 and D_2, where D_1 is the database before the change and D_2 is the database after the change. Checking the integrity constraints then reduces to checking whether $D_2 \vdash_I G$, which specifies a meta level computation. If this succeeds, then the changes are allowed and the database is now D_2. If this fails, then the changes are not allowed and the database remains D_1.

For example, consider a database containing information on student enrolments. The institution has a rule that no student may be enrolled in both the Science faculty and the Arts faculty. This rule may be expressed by the fact that the goal

$$\exists x\, \text{enrolled}(x, \text{science}) \wedge \text{enrolled}(x, \text{arts})$$

must fail, and so if D is the (object level) database, we may think of the meta-level view of the program as

$$D \wedge (\exists x\, \text{enrolled}(x, \text{science}) \wedge \text{enrolled}(x, \text{arts})) \supset \text{ic\_violation}$$

After making a change to D, we then ask the goal ic_violation of the meta-level program, and if it succeeds, then the change is disallowed. If it fails, then the change is allowed. This computation is done at the meta-level, as the integrity constraints are not able to be changed by the user, and so should not be visible at the object level. Hence the "internal", or machine view of the database is the

meta-level one, whilst the "external", or user's view of the database is the object level one.

The point of these examples is to show that the separation of computation into an object level program and a meta-level program seems to be a natural one, and this difference in level may be naturally studied in the context of hereditary Harrop formulae. This approach may be used to show how some typical programming techniques, such as memoisation and clause re-ordering, may be given an explanation in terms of logic, rather than purely operational notions. One may think of the completion of a program, as defined in chapter 3, as a meta-level program corresponding to an object level program, and hence interpreting Negation as Failure in a logical context.

Clearly this approach cannot be a full answer to the problems raised, such as the fact that $\wedge$ is not necessarily commutative, as the order in which the conjuncts are selected may affect termination properties. However, it does show that such properties are not as "extra-logical" as may be initially thought. Fuller and better answers are probable using a higher-order approach, but it is interesting to note that "meta" and "higher-order" need not be synonymous. Rather than merely let the meta-theory of logic programming be expressed in the (usually) informal language used by computer scientists to communicate with one another, we interpolate a third level between the object level and this informal meta-level. As formulae of this new level may be given a computational interpretation, we may think of this as an executable meta-language, in which programming and to some extent meta-programming may both be accommodated.

Chapter 8

Conclusion and Further Work

We have seen how first-order hereditary Harrop formulae may be used as basis for logic programming via the paradigm of uniform proofs. This notion of proof requires that we use intuitionistic logic as the semantic basis for the language, and allows us to expand the class of formulae which may be used as programs and goals whilst retaining features which are natural to a logic programming system.

We have shown how negation may be incorporated into this system, and have shown how the usual techniques may be interpreted in a constructive framework. We also saw how to give a extensional interpretation of universally quantified goals.

The model theory most appropriate in this context seems to be Kripke or Kripke-like models, and so we have explored model-theoretic issues in regard to negation and universal quantification. A closer examination of the model theory has shown how we may exploit the structure of the formulae to explore issues of equivalence, and this in turn led us to intermediate logics. Finally, similar considerations of structure allowed us to explore some issues of memoisation and meta-programming.

More work remains to be done, and below we briefly summarise the various possibilities.

8.1 Completions and Inconsistencies

As mentioned in chapter 3, it is important to restrict the class of programs so that the completion of every program is consistent. The completion is an explicit form of dealing with NAF, in that an explicit formula P^c is given for which $P^c \vdash_I A \supset \bot$ iff $P \vdash_s \neg A$. As NAF is inherently implicit, in that the programmer only writes down the definition of success rather than the definition of success and failure, this approach requires some restriction to be placed on the class of programs. There are less restrictive assumptions than that of local stratification, which will presumably lead to similar result to those reported here; there have been some efforts in this direction given by Przymusinska and Przymusinski [93], as well as Sato [103] and Cavedon [14].

It seems difficult to define precisely which class of programs leads to consistent completions, other than that it must exclude programs such as $\neg p \supset p$. Whilst the precise definition of this exact class may be possible, an easier approach may be to use a weaker logic, in which inconsistencies are less problematic, such as minimal logic. Inconsistency is more of a problem for the choice of logic than from programming considerations, and so this latter approach has some appeal.

One such method of dealing with inconsistent completions was given by Gabbay [36], in which, in essence, the failure of an atom A is perceived as the modal statement $\neg\Box A$, i.e. that it is not necessary that A be true. This view of NAF as a modal operator means that the completion of the program $\neg A \supset A$ becomes $A \leftrightarrow \neg\Box A$, and so whilst being locally inconsistent (i.e. nothing sensible can be said about A), the completion of $P \cup \{\neg A \supset A\}$ is not empty. Indeed, it has the nice property that $\text{comp}(P_1 \wedge P_2) = \text{comp}(P_1) \cup \text{comp}(P_2)$. In this way the completion may be inspected piece by piece and packed together, and so may be a useful way of dealing with the completion of non-locally stratified programs.

8.2 The Role of Induction

As mentioned in chapter 2, an inductive method of establishing the truth of $\forall x\, p(x)$ from the program

$$p(a)$$
$$\forall x\, p(x) \supset p(f(x))$$

where the signature of p is $\{a/0, f/1\}$ will be useful in order to implement a stronger form of extensional universal quantification in goals. This may be more useful at the meta-level than the object level, due to the nice memoisation properties of G_{mod} formulae. One obvious area of further work is to define the inductive strategy in an operationally feasible way. The methods of [12,13] may be useful in this regard.

Another application of such an induction strategy is to the problem of finding answer substitutions for existentially quantified goals. The methods of chapter 4 had some difficulties with programs such as the one above. However, the methods used to find a proof of $\forall x\, p(x)$ and of $\exists x \neg p(x)$ are closely related, in that we may consider both as looking for instances of $p(x)$ which either succeed or fail. Hence, if the search for a proof of $p(c)$, where c is a meta-variable (i.e. an arbitrary term) succeeds, then $\forall x\, p(x)$ succeeds and $\exists x\, \neg p(x)$ fails, whereas if $p(c)$ fails due to the fact that $p(t)$ fails, then a correct answer substitution for $\exists x \neg p(x)$ is $x \leftarrow t$. For example, given the program

$$p(a)$$
$$\forall x\, p(x) \supset p(f(x))$$

and the signature $\{a/0, b/0, f/1\}$, then the goal $\forall x\, p(x)$ fails, as $p(b)$ fails, and so $\exists x \neg p(x)$ succeeds.

Clearly such an inductive method will make use of implications in goals, as often we would wish to add $p(c)$ to the program (where c is a new constant), and

see if $p(f(c))$ succeeds from this larger program. This seems to give an elegant connection to lemma 6.3.1, in that one may consider the inductive method as showing that the two programs

$$\begin{array}{ll} p(a) & \forall x\, p(x) \\ \forall x\, p(x) \supset p(f(x)) & \end{array}$$

are operationally equivalent for the signature $\{a/0, f/1\}$, and in order so to do it is usually necessary to look at certain extensions of the longer program. In this way, if we find that a few choice mutual extensions of the programs prove the same select goals, then we may conclude that the two programs are operationally equivalent for a certain class of goals. Hence the inductive method gives us a way of checking whether two programs are equivalent for a given goal, rather than for all goals, which may be thought of as a localisation, or top-down implementation, of the ideas expressed in lemma 6.3.1.

Our approach to equivalence may be strengthened by using canonical programs [50,8], so that there is a stronger notion of equivalence between programs. It may be interesting to combine the notion of a canonical program with that of a normal form, which may lead to an interesting notion of equivalence.

Another possibility is to explore the relationship between negation and the efnf transformation more closely, so that a stronger result than Proposition 6.1.4 may be shown.

8.3 Model Theory

The relation between the Kripke-like model of chapter 5 and standard Kripke models may be worthy of deeper investigation. It has recently been shown how the model theory of [77] may be extended to hereditary Harrop formulae under the "new constant" interpretation of the universal quantifier [76], and that this is also an S4 model. An investigation along these lines would make it easier to

identify the role of negation in the intermediate logic I', as well as leading to a better understanding of I' itself. This will presumably help in settling conjecture 6.5.12 about the equivalence of $\vdash_I$ and $\models_G$ on G_{mod} formulae. The notion of increasing knowledge which is inherent in Kripke models may be useful for formal development of logic programs, and so this approach to model theory may have some interesting connections to program specification.

It is well-known that intuitionistic logic may be interpreted in a topological space [64,21], and so another natural direction to take is to give a topological interpretation of first-order hereditary Harrop formulae. Just as classical propositional calculus is sound and complete with respect to interpretations over Boolean algebras, one may define a Heyting algebra over a topological space for which intuitionistic propositional calculus is sound and complete. The same line of thought may be extended to intuitionistic predicate calculus, and so it may be interesting to explore the difference between the interpretations of full first-order logic and hereditary Harrop formulae in this context.

Kripke models may be thought of as a special case, in that the topologies involved are restricted to be partially ordered sets. Hence, a less restrictive topology may allow us to use a more semantically meaningful structure than that given by Herbrand interpretations.

This may also be useful in order to characterise I' more naturally.

Clearly there is also the problem of constructing the T^{ω} interpretation when universal quantifiers are allowed in goals (under the inductive interpretation). This is not as straightforward as it may appear, as it is not clear how to build the interpretation within ω steps. Naturally it may be possible to do so by using larger ordinals.

Another possibility is that a more detailed analysis of the operational method of proving $\forall x\, p(x)$ may lead to a more subtle construction. In the above example, it is clear that by using induction it is possible to derive the truth of $\forall x\, p(x)$ in only a finite number of steps, and so the construction process may imitate such a derivation.

As mentioned earlier, intuitionistic logic may not be the most natural way to deal with inconsistencies. An alternative approach may be to use *relevant* logic [3,22,99], in which inconsistencies remain localised. A step in this direction has already been taken by Fitting [31], although the programs involved did not contain any inconsistencies. Another possibility is that such a logic may be able to incorporate the features of the Kripke-like model, so that there is no change for consistent worlds, but still deal with inconsistencies in a desirable way.

A related issue is the side condition on implicative goals, i.e. that a goal $D \supset G$ only succeeds or fails when the addition of D to the program only extends the definition of predicates which cannot be completely defined. As we saw in chapter 5, this restriction is made principally for model-theoretic reasons. Whilst this ensures that the Kripke-like model is significantly simplified and this approach does concur with programming intuitions, it seems desirable to remove this side condition. Not only would this be a conservative extension of the work of Miller [77], unlike the version above, it would also simplify the notion of an O-derivation. Whilst the notion of separating predicates into completely defined and otherwise may correspond to programming intuition, there do not seem to be any strong proof-theoretic arguments against allowing arbitrary extensions of the program to be used; if a more sophisticated model theory can be found to deal more cogently with the problem of extending completely defined predicates, then there would be no point in keeping the side condition. Hence, if we can find a natural way to deal with inconsistencies in the manner discussed above, we will presumably be able to lift this restriction.

8.4 Meta- and Higher-Order Programming

It was seen in chapter 7 how certain meta-programming tasks may be handled in a first-order setting. However, there is clearly a limit to what may be done along these lines compared to a higher-order approach. We have seen how attempting to get a memoisation property for first-order hereditary Harrop formulae leads

naturally into higher-order issues. Higher-order logic programming is an expanding area, and there are several papers in the literature which deal with various issues, and hereditary Harrop formulae have been shown to be particularly amenable to higher-order extensions [27,44,45,79,87,80]. It would be interesting to see how the results reported here in the first-order case may be generalised to the higher-order case. One may view the thrust of this thesis as exploring to some extent the limits of what may be achieved in a first-order framework; this will presumably lead to a clearer idea of the precise advantages and pitfalls of a higher-order approach to logic programming.

Bibliography

[1] S. Abian and A.B. Brown, A Theorem on Partially Ordered Sets with Applications to Fixed Point Theorems, *Canadian Journal of Mathematics*:13:78-82, 1961.

[2] H. Abramson and M.H. Rogers (eds.), *Meta-Programming in Logic Programming*, MIT Press, 1989.

[3] A. Anderson and N. Belnap, *Entailment: the Logic and Relevance and Necessity*, volume 1, Princeton University Press, 1975.

[4] K.R. Apt, H. Blair, and A. Walker, Towards a Theory of Declarative Knowledge, in *Foundations of Deductive Databases and Logic Programming* 89-144, J. Minker (ed.), Morgan Kaufmann, 1988.

[5] K.R. Apt and M.H. van Emden, Contributions to the Theory of Logic Programming, *Journal of the Association for Computing Machinery* 29:841-862, July, 1982.

[6] C. Aquilano, R. Barbuti, P. Bocchetti and M. Martelli, Negation as Failure. Completeness of the Query Evaluation Process for Horn Clause Programs with Recursive Definitions, *Journal of Automated Reasoning* 2:155-170, 1986.

[7] R. Barbuti, P. Mancarella, D. Pedreschi and F. Turini, Intensional Negation of Logic Programs, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, Pisa, Italy, 1987.

[8] H. Blair, Canonical Conservative Extensions of Logic Program Completions, *Proceedings of the Symposium on Logic Programming* 154-161, San Francisco, August, 1987.

[9] G.S. Boolos and R.C. Jeffrey, *Computability and Logic*, Cambridge University Press, 1980.

[10] K.A. Bowen and R.A. Kowalski, Amalgamating Language and Metalanguage in Logic Programming, in *Logic Programming* 153-172, K.L. Clark and S.-A. Tärnlund (eds.), Academic Press, 1982.

[11] K.A. Bowen and T. Weinberg, *A Meta-Level Extension of Prolog*, Technical Report CIS-85-1, Syracuse University, 1985.

[12] A. Bundy, F. van Harmelen, J. Hesketh and A. Smaill, Experiments with Proof Plans for Induction, *Journal of Automated Reasoning*, 1989.

[13] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill and A. Stevens, A Rational Reconstruction and Extension of Recursion Analysis, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* 359-365, 1989.

[14] L. Cavedon, Continuity, Consistency and Completeness Properties for Logic Programs, *Proceedings of the International Conference on Logic Programming* 571-584, Lisbon, June, 1989.

[15] D. Chan, Constructive Negation Based on the Completed Database, *Proceedings of the International Conference and Symposium on Logic Programming* 111-125, Seattle, August, 1988.

[16] D. Chan, An Extension of Constructive Negation and its Application in Coroutining, Proceedings of the North American Conference on Logic Programming 477-493, Cleveland, October, 1989.

[17] K. Clark, Negation as Failure, in *Logic and Databases*, H. Gallaire and J. Minker (eds.), Plenum Press, 1978.

[18] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1984.

[19] A. Colmerauer, H. Kanoui, R. Pasero and P. Roussel, *Un Systeme de Communication Homme-Machine en Francais*, Technical Report, Groupe de Intelligence Artificielle, Universitae de Aix-Marseille II, Marseille, 1973.

[20] R.L. Constable and S.F. Smith, Computational Foundations of Basic Recursive Function Theory, *Proceedings of the Symposium on Logic in Computer Science* 360-371, Edinburgh, July, 1988.

[21] D. van Dalen, Intuitionistic Logic, in *Handbook of Philosophical Logic Volume III: Alteratives to Classical Logic* 225-339, D.M. Gabbay and F. Guenthner (eds.), Reidel, 1986.

[22] J.M. Dunn, Relevant Logic and Entailment, in Handbook of Philosophical Logic Volume III: Alternatives to Classical Logic 117-224, D.M. Gabbay and F. Guenther (eds.), Reidel, 1986.

[23] E.W. Elcock, *Absys: The First Logic Programming Language – a Retrospective and Commentary*, DAI Research Report 26, Department of Computer Science, University of Western Ontario, July, 1988. To appear in the *Journal of Logic Programming.*

[24] E.W. Elcock, Absys: The Historical Inevitability of Logic Programming, *Proceedings of the North American Conference on Logic Programming* 1201-1214, Cleveland, October, 1989.

[25] M.H. van Emden and R.A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *Journal of the Association for Computing Machinery* 23:4:733-742, October, 1976.

[26] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi, A New Declarative Semantics for Logic Languages, *Proceedings of the International Conference and Symposium on Logic Programming* 993-1005, Seattle, August, 1988.

[27] A. Felty and D. Miller, Specifying Theorem Provers in a Higher-Order Logic Programming Language, *Proceedings of the International Conference on Automated Deduction* 61-80, Argonne, May, 1988.

[28] M. Fitting, *Intuitionistic Logic, Model Theory and Forcing*, North-Holland, 1969.

[29] M. Fitting, A Kripke-Kleene Semantics for Logic Programming, *Journal of Logic Programming* 2:295-312, 1985.

[30] M. Fitting, Partial Models and Logic Programming, *Theoretical Computer Science* 48:229-255, 1986.

[31] M. Fitting, Negation as Refutation, *Proceedings of the Symposium on Logic in Computer Science* 63-70, Paolo Alto, June, 1989.

[32] M. Fitting and M. Ben-Jacob, Stratified and Three-valued Logic Programming Semantics, *Proceedings of the International Conference and Symposium on Logic Programming* 1054-1069, Seattle, August, 1988.

[33] T. Flannagan, The Consistency of Negation as Failure, *Journal of Logic Programming* 3:93-114, 1986.

[34] D. Gabbay, *Semantical Investigations in Heyting's Intuitionistic Logic*, Reidel, 1981.

[35] D. Gabbay, N-Prolog: An Extension of Prolog with Hypothetical Implication.II. Logical Foundations and Negation as Failure, *Journal of Logic Programming* 2:251-283, 1985.

[36] D. Gabbay, *Modal Provability Foundations for Negation by Failure 1*, Technical Report 86/4, Imperial College, London, 1987.

[37] D. Gabbay and U. Reyle, N-Prolog: An Extension of Prolog with Hypothetical Implications. I., *Journal of Logic Programming* 1:319-355, 1984.

[38] D. Gabbay and U. Reyle, *Computation with Run Time Skolemisation (N-Prolog Part 3)*, Technical Report, Imperial College, London, 1987.

[39] D. Gabbay and M.J. Sergot, Negation as Inconsistency.I, *Journal of Logic Programming* 3:1-35, 1986.

[40] H. Gallaire and C. Lasserre, Metalevel Control for Logic Programs, in *Logic Programming* 173-185, K.L. Clark and S.-A. Tärnlund (eds.), Academic Press, 1982.

[41] M. Gelfond and V. Lifschitz, The Stable Model Semantics for Logic Programming, *Proceedings of the International Conference and Symposium on Logic Programming*, Seattle, August, 1988.

[42] J-Y. Girard, Y. LaFont and P. Taylor, *Proofs and Types*, Cambridge University Press, 1989.

[43] A. Grumbach, Learning with Prolog: some needed features, *Logic Programming Newsletter*, May, 1987.

[44] J. Hannan and D.A. Miller, Uses of Higher-Order Unification for Implementing Program Transformers, *Proceedings of the International Conference and Symposium on Logic Programming*, Seattle, August, 1988.

[45] J. Hannan and D.A. Miller, Deriving Mixed Evaluation From Standard Evaluation for a Simple Functional Language, *Proceedings of the International Conference on the Mathematics of Program Construction*, The Netherlands, June, 1989.

[46] J. Harland, A Kripke-like Model for Negation as Failure, *Proceedings of the North American Conference on Logic Programming* 626-642, Cleveland, October, 1989.

[47] R. Harper, F. Honsell and G. Plotkin, A Framework for Defining Logics, *Proceedings of the Symposium on Logic in Computer Science*, Ithaca, June, 1987.

[48] J. Jaffar and J-L. Lassez, Constraint Logic Programming, *Proceedings of the Conference on Principles of Programming Languages*, Munich, January, 1987.

[49] J. Jaffar, J-L. Lassez and J. Lloyd, Completeness of the Negation as Failure Rule, *Proceedings of the International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, 1983.

[50] J. Jaffar and P.J. Stuckey, Canonical Logic Programs, *Journal of Logic Programming* 3:2:143-155, 1986.

[51] S.C. Kleene, *Introduction to Metamathematics*, North-Holland, 1952.

[52] R.A. Kowalski, Predicate Logic as a Programming Language, *Information Processing 74*, North-Holland, Amsterdam, 1974.

[53] K. Kunen, Answer Sets and Negation as Failure, *Proceedings of the International Conference on Logic Programming* 219-228, Melbourne, May, 1987.

[54] K. Kunen, Negation in Logic Programming, *Journal of Logic Programming* 4:289-308, 1987.

[55] K. Kunen, Signed Data Dependencies in Logic Programs, Computer Sciences Technical Report 719, University of Wisconsin, October, 1987. To appear in the *Journal of Logic Programming.*

[56] K. Kunen, Some Remarks on the Completed Database, *Proceedings of the International Conference and Symposium on Logic Programming* 978-992, Seattle, August, 1988.

[57] G.M. Kuper, K.W. McAloon, K.V. Palem and K.J. Perry, Efficient Parallel Algorithms for Anti-Unification and Relative Complement, *Proceedings of the Symposium on Logic in Computer Science* 112-120, Edinburgh, July, 1988.

[58] J-L. Lassez and M.J. Maher, Optimal Fixedpoints of Logic Programs, *Theoretical Computer Science* 39:15-25, 1985.

[59] J-L. Lassez and K. Marriott, *Explicit Representation of Terms Defined by Counter Examples*, Journal of Automated Reasoning 3:301-317, 1987.

[60] J-L. Lassez, V.L. Nyugen and E.A. Sonenberg, Fixed Point Theorems and Semantics: A Folk Tale, *Information Processing Letters*:14:112-116.

[61] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984.

[62] J.W. Lloyd, E.A. Sonenberg and R.W. Topor, Integrity Constraint Checking in Stratified Databases, *Journal of Logic Programming* 4:331-344, 1987.

[63] J.W. Lloyd and R.W. Topor, Making Prolog More Expressive, *Journal of Logic Programming* 1:225-240, 1984.

[64] D.C. McCarty, Intuitionism: An Introduction to a Seminar, *Journal of Philosophical Logic* 12:105-149, 1983.

[65] L.T. McCarty, Clausal Intuitionistic Logic I. Fixed Point Semantics, *Journal of Logic Programming* 5:1:1-32, 1988.

[66] L.T. McCarty, Clausal Intuitionistic Logic II. Tableau Proof Procedures, *Journal of Logic Programming* 5:2:93-132, 1988.

[67] M.J. Maher, Equivalences of Logic Programs, in *Foundations of Deductive Databases and Logic Programming* 627-658, J. Minker (ed.), Morgan Kaufmann, 1988.

[68] J. Maluszyński and T. Näslund, Fail Substitutions for Negation as Failure, *Proceedings of the North American Conference on Logic Programming* 461-476, Cleveland, October, 1989.

[69] P. Mancarella, S. Martini and D. Pedreschi, Complete Logic Programs with Domain-Closure Axiom, *Journal of Logic Programming* 5:263-276, 1988.

[70] P. Martin-Löf, Constructive Mathematics and Computer Programming, in *Logic, Methodology and Philosophy of Science VI*, L.J. Cohen, J. Los, H. Pfeiffer and K.D. Podewski (eds.), North-Holland, 1982.

[71] E. Mendelson, *An Introduction to Mathematical Logic*, van Nostrand, 1964.

[72] D. Michie, Memo functions and machine learning, *Nature* 218:19-22, April, 1968.

[73] D.A. Miller, private communication, January, 1988.

[74] D.A. Miller, private communication, August, 1988.

[75] D.A. Miller, private communication, June, 1989.

[76] D.A. Miller, private communication, October, 1989.

[77] D.A. Miller, A Logical Analysis of Modules in Logic Programming, *Journal of Logic Programming* 6:79-108, 1989.

[78] D.A. Miller, Lexical Scoping as Universal Quantification, *Proceedings of the International Conference on Logic Programming* 268-283, Lisbon, June, 1989.

[79] D.A. Miller and G. Nadathur, Higher-Order Logic Programming, *Proceedings of the International Conference on Logic Programming* 448-462, London, July, 1986.

[80] D.A. Miller and G. Nadathur, Some Uses of Higher-Order Logic in Computational Linguistics, *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics* 247-255, 1986.

[81] D.A. Miller, G. Nadathur, F. Pfenning and A. Scedrov, Uniform Proofs as a Foundation for Logic Programming, Technical Report MS-CIS-89-36, Department of Computer and Information Science, University of Pennsylvania, June, 1989. To appear in the *Annals of Pure and Applied Logic.*

[82] D.A. Miller, G. Nadathur and A. Scedrov, Hereditary Harrop Formulas and Uniform Proof Systems, *Proceedings of the Symposium on Logic in Computer Science* 98-105, Ithaca, June, 1987.

[83] J. Minker, On Indefinite Databases and the Closed World Assumption, *Proceedings of the Conference on Automated Deduction*, 292-308, Lecture Notes in Computer Science 138, Springer-Verlag, 1982.

[84] P. Mishra, Towards a Theory of Types in Prolog, *Proceedings of the Symposium on Logic Programming* 289-298, Atlantic City, February, 1984.

[85] A. Mycroft and R. O'Keefe, A Polymorphic Type System for Prolog, *Artificial Intelligence* 23:295-307, 1984.

[86] G. Nadathur, *A Higher-Order Logic as the Basis for Logic Programming*, Ph.D. Thesis, University of Pennsylvania, May, 1987.

[87] G. Nadathur and D.A. Miller, An Overview of λProlog, *Proceedings of the International Conference and Symposium on Logic Programming*, Seattle, August, 1988.

[88] L. Naish, *An Introduction to Mu-Prolog*, Technical Report 82/2, Department of Computer Science, University of Melbourne, 1982.

[89] L. Naish, *Negation and Control in Prolog*, Lecture Notes in Computer Science 238, Springer-Verlag, 1986.

[90] L. Naish, Specification = Program + Types, *Proceedings of the Conference on the Foundations of Software Technology and Theoretical Computer Science* 326-339, Pune, December, 1987. Also published as Lecture Notes in Computer Science 287, Springer-Verlag.

[91] M.A. Nait Abdallah, Procedures in Horn-clause Programming, *Proceedings of the International Conference on Logic Programming* 433-447, London, July, 1986.

[92] L. Paulson, Natural Deduction as Higher-Order Resolution, *Journal of Logic Programming* 3:237-258, 1986.

[93] H. Przymusinska and T. Przymusinski, Weakly Perfect Semantics for Logic Programs, *Proceedings of the International Conference and Symposium on Logic Programming* 1106-1121, Seattle, August, 1988.

[94] T. Przymusinski, On the Declarative Semantics of Deductive Databases and Logic Programs, *Foundations of Deductive Databases and Logic Programming* 193-216, J. Minker, (ed.), Morgan Kaufmann, 1988.

[95] T. Przymusinski, Perfect Model Semantics, *Proceedings of the International Conference and Symposium on Logic Programming* 1081-1096, Seattle, August, 1988.

[96] T. Przymusinski, On Constructive Negation in Logic Programming, *Proceedings of the North American Conference on Logic Programming*, Cleveland, October, 1989.

[97] D. Pym, *Proofs, Search and Computation in General Logic*, Ph.D. Thesis, University of Edinburgh, 1990.

[98] A. Rajasekar and J. Minker, A Stratification Semantics for General Disjunctive Programs, *Proceedings of the North American Conference on Logic Programming* 573-586, Cleveland, October, 1989.

[99] S. Read, *Relevant Logic*, Blackwells, 1988.

[100] R. Reiter, On Closed World Databases, in *Logic and Databases*, H. Gallair and J. Minker (eds.), Plenum Press, 1978.

[101] J.A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the Association for Computing Machinery* 12:1:23-41, 1965.

[102] J.A. Robinson, The Future of Logic Programming, *Information Processing 86* 219-224, North-Holland, 1986.

[103] T. Sato, Completed Logic Programs and their Consistency, manuscript, Electrotechnical Laboratory, Ibaraki, 1988. To appear in the *Journal of Logic Programming.*

[104] M.J. Sergot, F. Sadri, R.A. Kowalski, F. Kriwaczek, P. Hammond and H.T. Cory, The British Nationality Act as a Logic Program, *Communications of the Association for Computing Machinery* 29:5:370-386, May, 1986.

[105] J. Sheperdson, Negation in Logic Programming, in *Foundations of Deductive Databases and Logic Programming* 19-88, J. Minker (ed.), Morgan Kaufmann, 1988.

[106] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, 1986.

[107] C. Smorynski, Applications of Kripke Models, Chapter V in [109].

[108] S.-A. Tärnlund, Horn Clause Computability, *BIT* 17:215-226, 1977.

[109] A.S. Troelstra, *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis*, Lecture Notes in Mathematics 344, Springer-Verlag, Berlin, 1973.

[110] M. Wallace, Negation by Constraints, *Proceedings of the Symposium on Logic Programming* 253-263, San Francisco, August, 1987.

[111] J. Zobel, Derivation of Polymorphic Types for Prolog Programs, *Proceedings of the International Conference on Logic Programming* 817-838, Melbourne, May, 1987.

Appendix A

Proof of the Disjunctive Identity

Lemma A.0.1 $(G_1 \vee G_2) \supset A \equiv_I (G_1 \supset A) \wedge (G_2 \supset A)$

Proof:

($\Leftarrow$): From figure A–1 it is clear that

$$G_i, (G_i \supset A), (G_j \supset A) \vdash_I A$$

when either $i = 1, j = 2$ or $i = 2, j = 1$. Thus we may derive the uppermost sequents in figure A–2, which in turn clearly shows the result.

($\Rightarrow$): It is clear that the uppermost sequents of figure A–3 may be derived, as it is obvious that

$$G_i \vdash_I G_1 \vee G_2$$
$$A, G_i \vdash_I A$$

where $i = 1, 2$. Hence, the result follows.

□

$$\frac{G_i, G_j \supset A \longrightarrow G_i \qquad A, G_i, G_j \supset A \longrightarrow A}{G_i, G_i \supset A, G_j \supset A \longrightarrow A} \supset\text{-L}$$

Figure A–1: Useful subproof

$$
\begin{array}{ll}
\dfrac{G_1, G_1 \supset A, G_2 \supset A \longrightarrow A \qquad G_2, G_1 \supset A, G_2 \supset A \longrightarrow A}{G_1 \vee G_2, G_1 \supset A, G_2 \supset A \longrightarrow A} & \vee\text{-L} \\
\dfrac{G_1 \vee G_2, G_1 \supset A, G_2 \supset A \longrightarrow A}{G_1 \supset A, G_2 \supset A \longrightarrow (G_1 \vee G_2) \supset A} & \supset\text{-R} \\
\dfrac{G_1 \supset A, G_2 \supset A \longrightarrow (G_1 \vee G_2) \supset A}{(G_1 \supset A) \wedge (G_2 \supset A) \longrightarrow (G_1 \vee G_2) \supset A} & \wedge\text{-L}
\end{array}
$$

Figure A–2: Proof that $(G_1 \supset A) \wedge (G_2 \supset A) \vdash_I (G_1 \vee G_2) \supset A$

$$
\begin{array}{ll}
\dfrac{\dfrac{\dfrac{G_1 \longrightarrow G_1 \vee G_2 \qquad A, G_1 \longrightarrow A}{G_1, (G_1 \vee G_2) \supset A \longrightarrow A}\ \supset\text{-L}}{(G_1 \vee G_2) \supset A \longrightarrow G_1 \supset A}\ \supset\text{-R} \qquad \dfrac{\dfrac{G_2 \longrightarrow G_1 \vee G_2 \qquad A, G_2 \longrightarrow A}{G_2, (G_1 \vee G_2) \supset A \longrightarrow A}\ \supset\text{-L}}{(G_1 \vee G_2) \supset A \longrightarrow G_2 \supset A}\ \supset\text{-R}}{(G_1 \vee G_2) \supset A \longrightarrow (G_1 \supset A) \wedge (G_2 \supset A)} & \wedge\text{-R}
\end{array}
$$

Figure A–3: Proof that $(G_1 \vee G_2) \supset A \vdash_I (G_1 \supset A) \wedge (G_2 \supset A)$