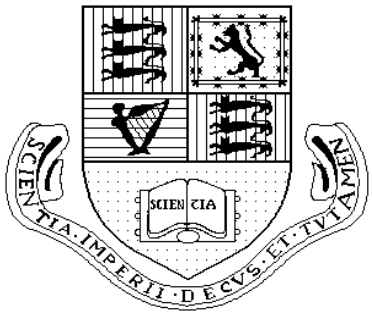
1

Imperial College of Science, Technology & Medicine

(University of London)

Department of Computing



Extending Logic Programming

by

Ethan Richard Collopy

Submitted in partial fulfilment of the requirements for the MSc degree in Engineering of the University of London and for the Diploma of Imperial College of Science, Technology & Medicine

September 1994

Acknowledgements

I would particularly like to thank Krysia Broda for her help and advice throughout the duration of the project.

Abstract

Negation cannot effortlessly be implemented in logic programming. The negation as failure rule has become the accepted model. This project compares and contrasts various possible implementations of negation theories, presents an implementation of a transformer of programs to their finite failure counterparts together with techniques (a constraints meta-interpreter, extending SLDNF) for dealing with universal conditions that arise from transformation. Induction is also considered as an option for the last technique. The report concludes that negation as failure will remain the standard until efficiency is greatly improved in other methods.

Contents

List of Figures

Introduction

Logic Programming was created under the auspices of efficient computation from deduction. As David Warren said, "Programming should be an intellectually rewarding activity. Prolog helps to make it so."

In the most general terms, this project aims to examine and extend the ways that Prolog handles negation. The meta-logical manner that nearly all existing systems employ is not particularly graceful.

Preliminary Definitions

Quantifiers ∀ - for all, the universal quantifier
∃ - there exists, the existential quantifier

Literals Atom A is a positive literal whilst ~A, the negation of A, is a negative literal.

Clausal Form A clausal form sentence is the normal form of a first order logic sentence where all of its conditions are conjoined together and all of its conclusions disjoined. Variables are implicitly universally quantified. In clausal form $B_1,\ldots,B_n \leftarrow A_1,\ldots,A_v$ represents 'For All variables X1,...,Xn, B_1 *or* K *or* B_n if A_1 *and* ... *and* A_n.

Horn Clause A Horn clause is a definite program clause of the form $A \leftarrow B_1,\mathrm{K}\,,B_v$ or a definite clause goal of the form $\leftarrow B_1,\mathrm{K}\,,B_n$

Negation As Failure Negation as Failure uses the **not** relation and **not G** fails if **G** fails to establish itself as a consequence of the program under evaluation.

First Order Logical Expression A first order logical expression is a well formed formulae taking one of the following forms, where A and B are well formed formulae - A, ~A, A ∧ B, A ∨ B, A if B and A iff B.

Non-monotonic Logic Non-monotonic Logic is where the rules of inference may be extended to reason with incomplete information. In non-monotonic logic the discovery/augmentation of additional information may invalidate previous conclusions in contrast with any monotonic logic.

Induction Mathematical induction is the basis of inductive proofs of program properties. A property is proven if it can be shown inductively for all values which possess the property.

Logic Programming is distinguished from first order logic by its treatment of negation and non-explicit **quantification**. Negation is the reason that Logic Programming has been analysed as operating **non-monotonically**. This is due to the incorporation of the **negation as failure** rule yet it can be put to good use in many A.I. and expert system applications. Logic Programming consists of a series of rules defining relations between objects in the form of horn clauses. Any **first order logical expression** can be converted to **clausal form. Horn clauses** are used primarily because they are explicitly quantifier free and this allows simple assignment of a procedural interpretation as well as being easily represented computationally. Their logic is *decidable* unlike full first order logic in the sense of only having one conclusion literal. It limits expressiveness whilst simultaneously enhancing efficiency. In the context of this report Prolog can be understood to be definitive of logic programming. In Prolog all quantification is implicit thus $p(X) :- \theta(\Xi, \Psi)$ is logically equivalent to $\forall X[\exists Y\, q(X,Y) \rightarrow p(X)]$.

Negation cannot be represented explicitly in Horn clause logic on which Prolog is based. This is the underlying motivation for this project; examining theories for negation in logic programming and analysis of the major themes with regard to implementation and efficiency. Negation theories in logic programming require extensions in one form of another to the standard clausal form. As we will see, these may involve the evaluation of universal and existential properties. Computational ambiguities arise in conflict with the implicit quantification of Prolog. Many of the methods exhibited incorporate explicit quantification.

The formal interpretation of quantification in logic is a vast area. Frege discovered the use of multiple quantifiers in first-order logic by examining predicates with singular terms and substituting generality signs. Under finite domains the universal quantifier $\forall$ is equivalent to conjunction and the existential quantifier $\exists$ to disjunction. Quantifiers may comprise of distinctive classes of variables where, for example $\forall X{:}\, List$, specifies X is of type *List.* Two main interpretations exist for quantifiers; objectual and substituitional. The objectual interpretation is defined as:

$\forall$X F(X) is read as ' For all objects X, in domain D, F(X)'

or substitutionally as,

$\forall$X F(X) is read as ' All substitution instances of ' F' are true'

Both have their uses. Substitutional Interpretation is closer to what we deal with in logic programming and so this is the reading we shall adopt for the purposes of this study.

There are a number of reasons as to why there is not a clear implementation of first order logic. Efficiency and decidability are important factors; how can we implement a theory without any procedural interpretation? When it does become necessary to evaluate quantified sentences there exist a number of techniques. Inductive methods are a possibility for verification of program properties when standard resolution techniques fail or loop infinitely.[11] The justification of induction for proving properties is beyond the scope of this project if if it is at all necessary. Properties are attempted to be proved inductively using techniques such as reduction, substitution and simplification thus transforming the goal to prove into a trivial formulae set. An induction schema is used to direct the attempted proof for possible cases. Computational induction implements a form of this, detailed in §7.

Prerequisites

Prerequisites are a knowledge of logic programming theory and practice. Deductive database theory and a background of computational induction will also be helpful. Suitable logic background is given in both [Llo87] and [Hog90].

Structure of Report

Part one features a coverage of the possible implementations of negation in logic programming concentrating on the viability of each approach. Each section commences with definitions of relevance to the section and concludes with a note on the usefulness, efficiency and robustness of the particular method. The preliminary definitions are based on those presented in [Hog90, Llo87, Ser90, Ster86, Bre91].

Part Two details the implementation of the transformation approach, extensions provided for this method with negative clauses in the candidate for transformation and the separate handling of datalogic and defining (constructor) clauses. It then proceeds to focus on the implementation of a negation - by - constraints meta- interpreter that can potentially be extended to deal with certain predicates in an inductive fashion. Inductive solution techniques are introduced.

The report concludes with a brief discussion on the further work on negation in logic programming as well as extensions to the techniques studied.

1[1] The possibility of infinite recursion analysis is another problem. It may be solved by a combination of dependency (of predicates) and variable/term analysis.

Part One §1 Introduction

Preliminary Definitions

Classical Negation Here ~P is defined as P implying all propositions. ~P is true iff P is false.

Closed World Assumption For a ground atom A, ~A is a consequence of program P if A cannot be proved from P.

Clark Completion Given a set of predicates of the form $p(t_1,\mathrm{K},t_n) \leftarrow A_1,\mathrm{K},A_k$ the completed definition is $\forall X_1,\mathrm{K},\forall X_n\ p(X_1,\mathrm{K},X_n) \leftrightarrow E_1 \vee \mathrm{K} \vee E_n$ where each E_i corresponding to a predicate definition for p is of the form $\exists Y_1,\mathrm{K},\exists Y_w((X_1 = t_1) \wedge \mathrm{K} \wedge (X_n = t_n)) \wedge L_1 \wedge \mathrm{K} \wedge L_m$. Axioms for defining the '=' predicate used in the above are included to define the action of unification. The Clark Completion together with the equality theory will be denoted by $\mathrm{Comp}_{\mathrm{EQ}}$.

Herbrand Interpretation A Herbrand Interpretation is a free assignment of truth values to all of the atom in the Herbrand Base, the set of all ground atoms from a given language.

Minimal Herbrand Model The minimal Herbrand model of program P is the least number of atoms which are true so that P is satisfied.

Safe Computation Rule Given a set of normal goals the computation rule ensures that the selected literal for evaluation is a positive literal or a ground negative literal in the goal.

Unsafe Computation Rule A computation rule that does not possess **safeness**, where only ground negative literals can be selected. Prolog has an unsafe computation rule.

Floundering A goal G and program P flounders in the computation of P U {G} reaches a goal containing non-ground negative literals.

Allowed Clause $\mathrm{A} \leftarrow \mathrm{B}_1,\mathrm{K},\mathrm{B}_v$ is allowed if all variables in A occur positively in $\mathrm{B}_1,\mathrm{K},\mathrm{B}_n$.

Admissible Clause $\mathrm{A} \leftarrow \mathrm{B}_1,\mathrm{K},\mathrm{B}_v$ is admissible if all variables occur in A or in positive positive literals in $\mathrm{B}_1,\mathrm{K},\mathrm{B}_n$. Admissibility is more general than Allowedness.

Stratified Program A program is stratified if if has a strata such that all clauses $p(t_1,\mathrm{K},t_n) \leftarrow \Lambda_1,\mathrm{K},\Lambda_\mu$ occur such that if L_i is positive in a strata less or equal to p or if L_i is negative in a strata less than the clause. Each separate strata is mapped to an integer.

Declarative Semantics These are the model theoretics from first order logic where goals are answered by evaluating there truth values. Informally it is the interpretation of meaning based on logic inference.

Procedural Semantics This is the proof procedure of the logic programming system consisting of SLD-resolution and a control strategy.

Occur Check Checks when unifying a variable with a term that the variable itself is not embedded in the term

In logic programming success and failure are not handled uniformly. All the successful proofs of a query are presented but a proof of failure just returns one answer to signify that evaluation of all branches of the sub-tree have resulted in failure. Negation in the realms of logic programming is vastly different from that of classical negation: *the negation of a formula is true on some elements if the formula is not true on these elements*. Problems with negation stem from it being impossible to give a complete implementation of negation. This is due to

a) universality of the Horn clause language. All computable predicates can be represented by a collection of Horn clauses. Therefore the negation of these computable predicates is not computable which is the motivation behind negation as failure. Negation as failure is incomplete as answers depend on the selection rule used in the query evaluation procedure.

b) semantic restrictions including the restrictions of programs to admissible and stratified programs to obtain a least model. The definitions of these are in [Llo87], [Hog90] and [Ser90].

[Llo87] states that it is difficult to implement anything more than negation as failure, especially the **Closed World Assumption** that is a syntactic reasoning process. The completion as presented by Clark attempts to capture the idea that information not explicitly stated by the program is false. When a program is **stratified** its completion will have a **minimal normal Herbrand model**. Thus the results of queries dpend not only upon the logical consequences but also on the way that they are ordered within the program or goal. Safeness is another cautionary technique for **negation as failure** flounder prevention. It allows only the selection of ground negative queries in

program resolution. **Prolog** does not work in this manner. Under negation as failure, the failure of negative subgoals prove a universal rather than an existential property.

Negation in Prolog is constrained by only allowing ground negative literals to return an affirmation answer. This answer may frequently not be the desired one. Given goal **<-~r(X,Y),q(Y),p(X)** and program

r(a,b) **q(c)**
p(c) **p(e)**

the answer will return no when it would succeed for {Y/c,X/c} and {Y/c,X/e} with the safe computation rule (in this case right-to-left would be suitable). A **safe computation rule** [Llo87, Hog90] is designed to prevent floundering under SLDNF. Prolog systems at present do not use this but variants MU-Prolog and NU-Prolog allow complicated selection rules that are capable of minimising floundering. Neither do the majority of systems possess an **occur check** [Llo87, Hog90] which disallows self-recursive bindings that create infinite terms. Thus Prolog is not sound for SLD- resolution, a central foundation of logic programming. Ideally, we desire the following equivalence for any logic programming language,
declarative semantics ≡ προχεδυραλ σεμαν
and this is definitely not true of Prolog.

Unsafe computation rules and the absence of the occur check together with non-logical predicates (var, assert ...) cause many Prolog programs to have minimal declarative semantics and somewhat complex procedural semantics. Many such programs might just as well be written in C.

For Negation as Failure (NAF, in the sequel) to have any declarative meaning logic programs have to be completed by the addition of an 'only-if' converse and a constraining equality theory. Unsafe negation detracts from the procedural semantics. It can be prevented by formulation of pre-processing rules, admission of a certain class of programs or rearrangements of goals etc. The inadequacies of NAF are described in §2.

This has led to a number of other theories being researched including negation by constraints, constructive negation and program transformation. None of these inherently possess the efficiency of NAF at runtime yet they may possess a superior declarative semantics. The primary reference theory of NAF is the Clark Completion. It was proved by Clark that inferences under NAF are equivalent to inferences from the completion of a program which essentially replaces each 'if' by 'if and only-if'. [Shep84] proved that NAF deductions are equivalent to Reiter's Closed World Assumption (CWA, in the sequel). He also stated that the results can differ in specific circumstances, namely where the program has infinite failure branches.

Other semantics are also relevant such as iterated fix points and those based on the non-monotonicity of NAF. Semantics for constructive negation and negation by constraints rely on the Clark Completion only. The transformational approach is also carried out with respect to the Clark Completion. These are introduced throughout Part one as the different methods are studied.

§2
Negation As Failure

This section covers the implementation and theory of negation as failure, the primary negation system in logic programming at present.

Preliminary Definitions

Call-Consistency A program is call consistent iff its dependency graph contains no odd number of negatives in any cycles so that no relation wil depend oddly on itself. In the dependency graph for any clause A:-...,B_i,... there is a postive edge in the graph from B_i to A if B_i is a positive literal and a negative edge if B_i is a negative literal. Call-Consistency is due to Sato.

Independence of the Computation Rule All distinct answers will be returned regardless of any computation rule.

Covering ¥ is a covering of term t iff for each ground Herbrand instance g* of t there exists t" in ¥ such that g is a ground instance of t". {0,s(X)} and {X} are coverings of the language L = {0,s(X)}. ¥ is an **exact covering** of t iff it is a covering and for all terms t" in ¥, t" is a ground instance of t. {s(0),s(s(X))} is an exact covering of {s(X)}.

Abduction The most basic definition of abduction, due to Poole, is: *from B and B:- A we can derive A*. A is said to be the hypothesis that explains observation B.

Modal Negation as Failure Provability under negation as failure is treated as a modal operator and the semantics are formed accordingly.

Ground Instantiation The ground instantiation of a program P is the set of all ground instances for all of its constituent clauses, denoted by **G(P)**.

2.1 The Clark Completion

For a program P and a goal A, under the NAF rule we infer ~A if A is not a logical consequence of the Clark Completion of the program, $Comp_{EQ}(P)$.

~A succeeds whenever A finitely fails (on all branches)

~A fails whenever A finitely succeeds for some substitution answer ß.

The Completion rewrites clauses replacing 'if' by 'iff', generalising headings as much as possible and adding axioms constraining equality. These axioms include:-

$\forall(t[X] \neq X)$, *for each term* $t[X]$ *containing* X, *different from* X

which defines the occur check not present in most Prolog systems for efficiency reasons. When negative consequences are derivable from the completion it is natural to extend these programs to include negative assumptions. However, as [Apt94] notes, once a program includes negative assumptions it is liable to become self-referential and paradoxical. The completion is liable to be inconsistent if the program is not call-consistent; that is there exists dependency on a clause which occurs in a negative instance. Call-consistency is demonstrated by positive and negative dependency analysis.

The clause *not* is implemented in prolog as:

not(X):- X,!,fail.

not(X).

If the clause is used with a non-ground goal then it will flounder. Given the following program clause set:

p(a) **p(b)** **q(c)**

and the query **<- not p(X), q(X)**, for a safe computation rule and under the assumption of independence of the computation rule, this should succeed for {X/c}. In Prolog this will flounder as the appearance of **p** in the **not** query is non-ground.

2.2 Reiter's Closed World Assumption

The semantics of NAF may also be interpreted under the CWA. The CWA depends on the logical content of the program which is ideally how we would desire NAF to operate. [Apt94] defines the program conclusions under the CWA as a *static non-monotonicity* where all of the positive and negative information can be realised at any time. This is in contrast with what he refers to as a *dynamic non-monotonicity* where we are forced to draw conclusions from incomplete information, such as belief revision. The CWA takes a program P1 and then uses this to derive all negative consequences. This can formally be defined as:

P1 is a set of formulae, $\leftarrow\pi\ \iota\sigma\ \delta\varepsilon\rho\iota\varpi\alpha\beta\lambda\varepsilon\ \iota\phi\phi$

$\Pi\!\!1\ \cup\ A\Sigma\Sigma(\Pi\!\!1)\ \alpha\ \ \leftarrow\pi\ \omega\eta\varepsilon\rho\varepsilon$

$A\Sigma\Sigma(\Pi\!\!1)\ =\ \{\leftarrow\theta \mid \theta\ \iota\sigma\ \alpha\tau o\mu\iota\chi\in\ \Gamma(\Pi\!\!1)\ \wedge\ \nu o\tau\ \Pi\!\!1\ \alpha\ \ \theta\}$

It captures the notion that any ground clause not implied is false. For the program

P1 = { visits(dan, X)
visits(Y, new_zealand) }

the CWA adds the clause {~visits(new_zealand, dan)} which makes sense. Much like the completion, the CWA may be inconsistent, generally where there are clause heads in the program that are *not implied* by the operation of the program which results in 'indefinite knowledge.' For the program P where

P = {q(b):- not q(c)} we form

ASS(P) = {~q(b),~q(c)}

Here ASS(P), the assumptions under the CWA, are inconsistent with the original program P.

Occasionally one may be consistent when the other is not. [Shep84] presents:-

p(a):- not p(a).

Here the CWA is consistent. The completion gives us:-

$Comp_{EQ}(P)\ \equiv\ \forall\xi\ (\pi(\xi)\ \leftrightarrow\ (\xi\ =\ \alpha\ \wedge\ \leftarrow\pi(\xi))$ which is clearly inconsistent. Using classical logic we arrive at the conclusion **p(a)** which is another alternative for NAF. All of this background is presented in more detail in the following:- [Llo87], [Hog90], [Bre91], [Ser90]. [Shep85] questioned the logical status of NAF given the applicability of either the Clark Completion or the CWA.

We now present some mathematical notation to present a class of program where the set of possible inferences under NAF is equivalent to the CWA. Given a program P and a Herbrand Interpretation P, the immediate consequence mapping T_P is:

$T_P(I)\ =\ \{A \mid (A\ \iota\phi\ A_1,K\ ,A_\nu)\in\Gamma(\Pi\!\!1)\ \wedge\ \{A_1,K\ ,A_\nu\}\ \iota\sigma\ \tau\rho\upsilon\varepsilon\ \iota\nu\}]$

$T_P \downarrow \omega$ is the limit of the chain defined by iteratively applying T_P to the Herbrand Base B_P and $T_P \uparrow \omega$ is the limit of the chain defined by iteratively applying T_P to the empty set. When $T_P \downarrow \omega = T_P \uparrow \omega$ for definite program p then NAF is equivalent to the CWA as the infinite failure set of the program is empty. Hierarchical programs do satisfy

this constraint. The iterative $T_P \uparrow \omega$ was shown by Kowalski and van Emden to be equivalent to the minimal Herbrand Model for definite programs. Given

p(c) q(c) r(X):-q(X),p(X)

$T_P \uparrow \omega$ = {p(c), q(c), r(c)}.

Once negative information can be inferred of a program it is acceptable for negative subgoals to be included in programs and then T_P adopts undesired behaviour due to loss of monotonicity and continuity. Fixpoint semantics attempts to create restrictions for a stronger theory.

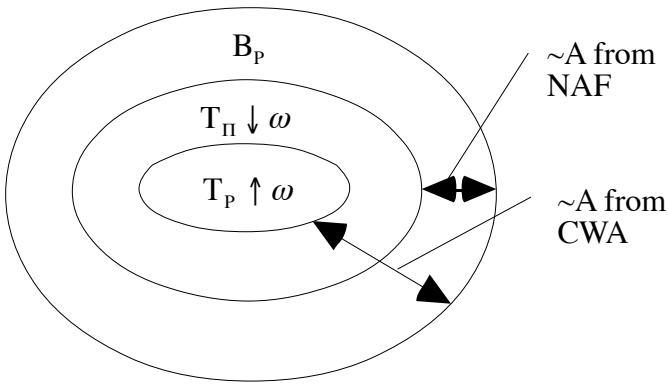


Figure 1.1 Onion depiction of theoretical relations

A limitation of the CWA is that it only assumes a program is complete with respect to the properties it describes. It will not infer any negative consequences for relations that do not have at least one instance in the original program as it is a syntactic reasoning process. It also falters when all of the information is not present in the program.

[Shep85] states that a query with a finite tree property, that is all branches end in failure or success, is **answer complete**. A query is said to be **answer complete** iff:

Q succeeds iff $Comp_{EQ}$ a $(\exists)\Theta$

Θ φαιλσ ιφφ$Xo\mu\pi_{E\Theta}$ α $\leftarrow\Theta$

Θ συχχεεδσ ωιτη ανσωερ ινχλυδινφφ$Xo\mu\pi_{E\Theta}$ α $\Theta\theta$

In the first and third case success may be deemed dependent on a selection rule whilst failure is independent of a selection rule. A hierarchical database, where the body of every non-ground database statement is defined on a *lower strata* that satisfies the allowability condition (all bodies of clauses are allowed) has the property defined above of being **answer complete**. Therefore, any program that adheres to these relations will never flounder for an allowed goal. The class of these programs is very small and that is why other methods have to be considered. It should not really be the case that answers returned from a program depend on its ordering. Ideally we desire a negation rule that does not need such restrictions

for both the database and goals; the other negation rules are possibly a step in the right direction.

2.3 The Domain Closure Axiom

The completion augmented with the Domain Closure Axiom (DCA, in the sequel) is another contender for NAF reference theory and is introduced thoroughly in [Man88]. The DCA, in addition to the equality theory, forces any model to have an interpretation domain where each object is constructed using interpretations of the given constants and functions. The Domain Closure Axiom is:-

$$\forall x \left(\bigvee_{f \text{ is a function}} \exists y_1,\ldots,y_n (x = f(y_1,\ldots,y_n)) \vee \bigvee_{c \text{ is a constant}} x = c \right)$$

Maher stated, "EQ_{DCA} is a complete theory," where EQ is the completion equality axioms that themselves characterise the unification process. Therefore the EQ_{DCA} interpretations are much closer to Herbrand interpretations. The domain closure axiom is innate within the CWA. It cannot be expressed finitely if functions are in the language but there is no need for this as it is not required to compute logical consequence.

The major advantage is that $Comp_{EQ+DCA}$ has universally quantified positive consequences given the theory of *term covering*. Essentially it ensures any possible ground term instance can be accounted for in the term covering. For instance if we have:

p(0) **p(s(X))**

The logical consequences of this are p(0) and $\forall X\, p(s(X))$ so if we have a covering of the variable Y as {0,s(Y)} then $\forall Y\, p(Y)$ is a logical consequence which is not true without the DCA as $\forall X\, p(s(X))$ is not a logical consequence under $Comp_{EQ}$ alone

This is a **proof by case analysis** technique and is not a logical consequence without the domain closure axiom. Proof by case analysis is weaker than induction and similar to implementations of universal quantification where it may be explicitly stated. With proof by case analysis and covering finite failure over a set of subcases does guarantee finite failure in the general case. It should be noted that an infinite number of coverings are not allowed, for if they were then the general property would never be reached.

[Man88] prove that equality theory interpretations are similar to herbrand interpretations when extended by the DCA. The DCA proves a stronger form of completeness than $Comp_{EQ}$ alone and this is that finite failure for each element of a covering guarantees finite failure for the term that is being covered. Formally,

$Comp_{EQ+DCA}$ a $\forall \leftarrow \pi(\underline{\tau})\ \iota\phi\phi\ \pi(\underline{\tau}) \in \Phi\Phi_{\Sigma\Lambda\Delta}(\Pi)$

[Man88] sums it up well, 'The proof by case analysis inference rule, induced by DCA, gives us an opportunity to reach a deeper insight into the operational properties of SLD resolution.' This is true in the sense of being able to draw generalised conclusions from specific operation. What we desire in an implementation is to be able to use the DCA (inherently) and work in the reverse direction in an attempt to prove a universal property.

2.4 The Strict Completion

[Dra91] introduce a slight variation of the Clark Completion which is shown to always be consistent. The strict completion is developed in such a way that examines how the undesirable aspects of the completion arise and alters the program so that they are removed. The original program is transformed which possesses the same answer and finite failure sets yet forms a consistent completion.

The strict completion has the following definition,

$comp_{STRICT}(P) = comp(split(P))$

Informally split(P) is a function which exchanges predicate symbols that occur in negated literals for exact renamed copies so that an entire copy of the original program is formed. The defintion of split is given in [Dra91]. The equivalence of $P \cup \{\leftarrow \Gamma\}$ and split(P) $\cup$ {split(G)} is proven by Drabent. If we have the following program,

p:- q
p:- ~q
q:- q

the action of the split function yields,

p:-~q'	q:- q
p':- q'	p':- ~q
q':- q'	

Call-consistency analysis is now performed for both programs.

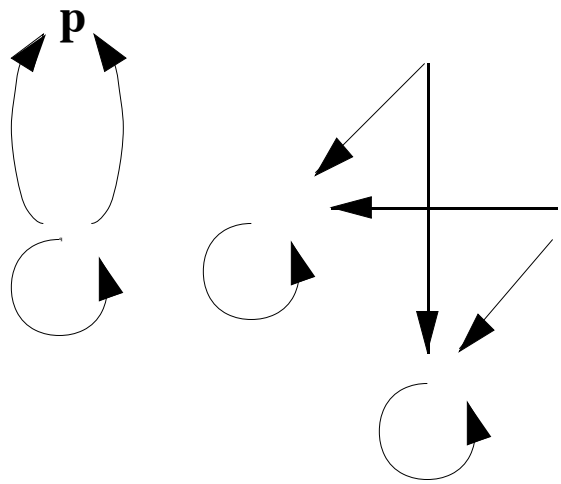


Figure 1.2 Call-consistency analysis

Essentially, its completions always satisfy the call-consistency of [Hog90]. Drabent justifies the strict completion as being the actual meaning of a program, preferable to the Clark completion by referring to negation as failure as a meta - rule that has to deal with a copy of the predicate anyway. The split feature may be applied to a program which is going to be transformed by creation (implicitly) of its completion so we know that it is consistent for logical consequences.

2.5 Modal Negation as failure

NAF can also be understood with a modal reference theory. Under a modal examination, ~A can be seen as ~*L*A. Here the 'not' of logic programming as an operator that tests the provability of the clause in its scope. It is an attempt to provide a complete semantics for NAF. SLDNF provability is defined as a modal operator such that under a modal system X (this could be S4, S5, KT4 ...) we have:

$P \; a_{SLDNF} \; l$ iff $P^{*} \; a_{X} a_{SLDNF} \; l$ where P* is the modal completion. Non-provability with respect to SLDNF is defined as:

$P \leftarrow_{\Sigma\Lambda\Delta N\Phi} \lambda \; \iota\phi\phi \; \Pi\alpha_{\Xi} \leftarrow_{\Sigma\Lambda\Delta N\Phi} \lambda$

[Bal93] states, "If one wants to conclude ~A from a logic program, it is not sufficient to say it is impossible to conclude A in that program; one has to build the impossible proof." One advantage of a modal interpretation is that no longer is A succeeds or A fails a tautology - it becomes **L**A or **L**~A which is not a tautology. The modal completion of program P is presented as a conjunction of positive and negative refutations of an atom if there is a literal such that it has a refutation in P. If we have the program:

$\sim B :- A \quad B$ then the modal completion is:
$(\mathbf{L}\leftarrow B \rightarrow \Lambda A) \wedge (\Lambda B \rightarrow \Lambda\leftarrow A) \wedge \Lambda B$ so $\mathbf{L}\sim A$ and $\mathbf{L}B$ are theorems if the modal system is K, considered standard. The Clark Completion has ~A and B as logical consequences.

2.6 Non-monotonic Techniques
Perfect/ Stable models

Other semantic theories for NAF are now presented briefly. Perfect models [Ser90] give special attention to circumscribed predicates in models where their extensions are as small as possible. Informally, perfect model creation drops as many of the circumscribed predicates as possible so that we still have a model. Priorities for circumscription may be asserted to create an ordering for the minimisation of predicates. The highest degree of circumscription is achieved via the prioritisation of specific atoms. Priorities may capture different readings of the predicates from non-Horn clauses. For example, the clause $p \rightarrow \theta \vee \rho$ can have the different readings of $q \leftarrow p,\sim r$ or $r \leftarrow p,\sim q$ depending on prioritisation.

Perfect models then use these prioritised predicates to build the canonical model repeatedly applying the prioritied predicates. Priorities are found from strata of the stratified program. A perfect model is one which has the most atoms of a low priority and is minimal. Perfect model semantics can also be applied to constructive negation (§3).

Stable models do not require stratification, they use the notion of a belief modal operator; ~q expresses the fact that q is not believed. A set of beliefs is stable if it represents beliefs of a 'rational' agent where the program is a premise. A stable set of beliefs is equivalent to logical consequences of its beliefs. If this is the case then it has been proven [Ser90] that we have a minimal model.

Negation as failure can be read abductively based on the work of Eshgi and Kowalski in 1989. The semantics of any program P is a set Δ of variable free negative literals whose union satisfies the following integrity constrants:
$\leftarrow [q(\bar{t}) \wedge not\, q(\bar{t})]$ and
$q(\bar{t}) \vee \nu o \tau \theta(\bar{\tau})$
The contents of the set Δ are used to decide the truth (or falsity) of all ground clauses. The first constraint decides consistency and the second totality. Problems on deciding whether the integrity constraints are too stringent or weak have still to be resolved as we see for $p \leftarrow \nu o \tau \theta$ which has two sets $\Delta 1 = \{\sim q\}$ and $\Delta 2 = \{\sim p\}$.

The presented theories give us a wide ranging choice as to the declarative reading of negation as failure. Certain theories have advantages to different applications yet these incongruities arise from negation as failure being a procedural concept. NAF theory can be brought further into question as we have shown that it can be equally well explained by perfect and stable models, modal logic [Bal93] as well as methods not covered like iterated fixpoint semantics.

The remaining chapters in part one examine different negation implementations. Their semantics are generally based on the completion. Indeed, it may be said that these theories do not require the theoretical spaciousness of negation as failure. The widespread acceptance of negation as failure is summarised well in [Cha88] where he notes the compatibility of NAF and Prolog where subrefutation meshes with Prolog implementations with great efficiency.

§3
Negation By Constraints

Negation By Constraints (NBC, in the sequel) is introduced as a solution to the problem of floundering in Prolog.

Preliminary Definitions

Negation By Constraints NBC is an extension to NAF that allows any universal and existentially quantified formulae and inequalites to be in the set of goals and subgoals. In this mode answers may be returned from non-ground negative literals. Negation By Constraints is only called into operation when NAF would flounder.

Deductive Databases A deductive database, according to John Lloyd, is a finite collection of statements of the form $A \leftarrow \Omega$ where W is any typed first order formula. A typed formula implies that all variables and terms belong to a finite set of elements called types.

Constraint Sets A constraint set is a list of constraints, one for each variable in the goal. Ordering of variables in a constraint set is dictated by their locality.

Constraint Dropping A constraint is dropped after successful evaluation when it is not required for the final answer. If the goal is ?r(X) for the programr(X):- q(X,Y) q(X,Y):- Y ≠ 2 then {X/a} is an answer and the constraint Y≠2 is dropped from the constraint set.

3.1 Introduction

[Wal87] only presents Negation By Constraints for the evaluation of languages without function symbols. The most prominent of these are deductive databases. Constraints are not called into operation when presented with a negative ground literal; here NAF suffices. NBC is only called upon when presented with a non-ground negative goal or subgoal in a program clause.

A universal evaluation algorithm is presented by Wallace that is solved by the use of constraints. This is of particular value when related to the negation of existential variables that occurs when a program has its negative converse calculated by transformation (cf.§5). In

this respect the standard range of queries that can be presented to an NBC Prolog extends what is allowable under standard Prolog.

The problem of floundering which NBC resolves is shown clearly for an applied example [Wal87].

h(C,V) denotes, "conference C was held at V"
lp(C) denotes, "C was logic programming"

If the query is "Which venue held all logic programming conferences?" then this is in the predicate calculus:-

$\forall Conf(lp(Conf) \rightarrow h(Conf, Venue))$

Given the transformations of [LLo87] which are proved to be equivalent to the above by examining logical consequences of the completion, this maps into:-

query(V):- not test(V)
test(V):- lp(C), not h(C,V).

The query **?query(V)** will flounder in Prolog but will succeed under negation by constraints. Under NBC quantifiers may be stated explicitly and all answers are substitutions or inequalities. [Wal87] notes that all systems with explicit negation require a check for inconsistency of the whole program. Such is grossly inefficient.

Given,

p(1) p(2)

The query **?p(X)** succeeds with {X/1} and {X/2} as expected. The query **?not p(3)** succeeds under NAF and the query **?not p(X)** which would flounder under NAF returns the inequality { X≠1, X ≠ 2 }. If there were a finite set of constants the complement of this might be expressed explicitly. Variable domains [Van89] are given in many constraint problems and the solutions require much less backtracking as unsatisfiable values are removed from the domains in program operation.

All constraint sets at any stage are satisfiable; this is central to answering a given goal. Following [Van89] these constraints may be used in the evaluation process in a forward checking technique that will prune the search space effectively. Constraint sets have to be negatable throughout their use. Naïve evaluation of constraints where the constraint set is simply the negation of the all-solution set is not used in Wallace's implementation. The aim of this is to reduce redundancy as much as possible and a straightforward negation and use of De Morgan's laws results in many conjunctions that are entirely redundant. An example of

this is, for program P =

q(a,b) q(c,d)

which has, for goal ?q(X,Y) the answer set A = ((X=a,Y=b) ∨ (X = c,Y=d)). Now, negation and application of De Morgan's laws provides us with the negated answer set ~A = ((X ≠ a, X≠c) ∨ (X ≠ a, Y ≠ d) ∨

(Y ≠ b, X ≠ c) ∨ (Y ≠ c, Y ≠ d)).

Now the goal ?~q(b,Y) will succeed four times, redundancy which we do not desire.

It is also the theoretical base for constructive negation but does not create as much redundancy as the operation technique is not governed by a constraint set. It can be shown that a negative goal with m arguments and n solutions of the unnegated goal will create m^n negated conjunctions. Implementationally, this will lead to the same answer substition being returned many times over as the example shows.

Wallace's implementation intentionally incorporates techniques to reduce the problem of redundancy. Reconciliation, developed by Khabaza, is fundamentally intersecting constraint sets when there are multiple literals in a goal or subgoal.

3.2 Enhanced Constraint Methods

Focussing of the search is introduced. Its two main applications are:-

a) to examine clause heads before processing bodies so that constants are introduced at the earliest possible evaluation stage. This reduces the processing of goals that result eventual inconsistency.

b) to direct the search towards a solution with minimal backtracking or the need for all solution evaluation to precede negation.

As stated in the introduction, explicitly quantifed goals can be given to the system and they are then evaluated by negation by constraints. Universal quantification rears its head in the negation of subgoals that include local variables. Existential quantification operates under NBC by simply *dropping* the constraint(s) on the quantified variable(s) and returning substitutions/constraints on the free variable(s); automatically indicating existence of the quantified variable. In the dropping of constraints some unifying processing may be neccessary,

?∃Y p(Y, X,W) and the program clause p(X, V, X):- X ≠ 1, V ≠ X

gives the constraints Y ≠ 1, X ≠ Y, W = Y and dropping Y results in { X ≠ W,W ≠ 1 } as a solution after some processing.

3.3 Universal Goal Evaluation

Informally, universal quantification under NBC takes a quantified goal with respect to a program and attempts to solve the goal for all possible values of the universally quantified variable. Wallace presents an algorithm for achieving this for a goal with two arguments. Presented here is a variant of the algorithm generalised to handle any length of arguments within a goal. Firstly, the algorithm operation is described to make its behaviour perspicuous.

If the constraints processor is presented with a goal $\forall Y_i\ q(Y_1,\mathrm{K}\ ,Y_m)$ a solution is attempted for $q(Y_1,\mathrm{K}\ ,Y_m)$ (**step 2**). If one is found the constraints for the variables are instantiated and the goal is retried for the same constraints except that the constraint(s) on the universally quantified variables are negated. The constraint set is reconciled with input and output constraints before retrying the goal. Reconciliation ensures the set is satisfiable otherwise evaluation fails (**step 4**). If the goal succeeds for this constraint set again the universal quantified variable constraint set that is returned is negated (**step 5**). This procedure continues until the input constraint matches the output constraint for the universally quantified variable (**step 3**). Intuitively, this signifies success for all values as it has succeeded for a value initially and also the negation of this value which covers the whole domain.

1. Set Variable Constraints in Goal $\mathbf{Cons_i, \cdots, Cons_n}$ to **null**
2. Solve **Goal && Cons$_i$,.⋯ Cons$_n$**, returning the new constraints set **NCons$_i$,.⋯ NCons$_n$**.
3. If universally quantified goal (UQ) returns with $\mathbf{Cons_{UQ} == NCons_{UQ}}$ then return constraints $\mathbf{Cons_i,\cdots,Cons_{UQ-1}, Cons_{UQ+1},\cdots,Cons_n}$ otherwise proceed to step 4.
4. Negate $\mathrm{Cons_{UQ}}$ to give $\mathrm{NCons_{UQ}}$ and call **reconcile**($\mathrm{Cons_{UQ}}$,$\mathrm{NCons_{UQ}}$,$\mathrm{RCons_{UQ}}$) where it has the mode **reconcile**(? input,?input,^output).
5. For i = 1,...,n, $\mathbf{Cons_i = NCons_i}$
   For UQ, $\mathbf{Cons_{UQ} = RCons_{UQ}}$
   Goto step 2.

The details of the reconcile procedure are presented in Part two Sample evaluations of this are now presented. The use of this algorithm occurs when we are given a negated goal that itself may be defined in terms of a negated clause containing local variables. The algorithm terminates upon finding a returned constraints set equivalent to the

previous input constraints for the universally quantified variable. This signifies success.

Presented are three examples which detail the operation of the algorithm and display its flaws. The first example displays the simple operation of a universal goal. Suppose we have the following clause set:

p(1,1) p(3,1)
p(2,1) p(3,Z):- ineq(Z,1).

Given the goal $\leftarrow \forall Y\ p(X,Y)$ the algorithm creates null constraints for the two goal arguments. The first branch returns [eq(X,1), eq(Y,1)]. The universal constraint is negated and the goal is re-evaluated under the constraint [eq(X,1), ineq(Y,1)]. This fails. The next branch returns [eq(X,2),eq(Y,1)] and fails for the re-evaluated constraint [eq(X,2),ineq(Y,1)]. The last branch has a re-evaluated goal of [eq(X,3),ineq(Y,1)] which succeeds and the output constraint [eq(X,3)] is returned. This is a rather simple example without any nested constraints or initial input constraints.

If the goal is $\leftarrow \forall Y\ friend(X,Y)$ and is run against the database
friend(rupert, jackie).
friend(daniel, diane).
M M
friend(charlie, jane).
where we assume that all clauses and atoms are distinct, the goal will fail but only after evaluating every single friend clause in the database. This is a drawback of the algorithm in that it can be very inefficient when run against certain databases.

Our third example is the following goal ?p(X) against the following program

p(X):-~q(X,Y)
q(X,Y):- ~r(X),m(Y)
r(X):- n(X,Y)
n(a,b) n(e,f)
m(1) m(2)

and we can see that it requires NBC as the first subgoal would flounder for a safe computation rule. The program processes the subgoals until evaluated against a ground clause. Thus the first answer is (X = a,Y = b) returned from the subquery n(X,Y). For ~r(X), {Y = b} is dropped from the constraint set as it is local and therefore existentially quantified so is removed once it is satisfied. Then {X = a} is negated to return {X ≠ a} and the evaluation of m(_Y1) returns {_Y1 = 1}. This

clause set is conjoined together for the answers from the separate subgoals and negated. This gives {X = a or _Y1 ≠ 1}. The constraint on the existentially quantified _Y1 is dropped and {X = a] is returned as an answer. Upon backtracking {X = e} is given as an answer because {X ≠ a} will be an input constraint. This example displays how constraint dropping works in practice.

When an **unconstrained** variable appears in a clause head and it is a current universally quantified variable under evaluation then its input and output constraints will be the same and so success will occur, an obligatory action. This correlates with behaviour of the universal quantifier as it will succeed for all possible values. It is a feature that is inherent in NBC operation.

NBC operation forces backtracking to give subsequent solutions to a goal using the backtracking mechanism of prolog by giving the last solution set appended to the input constraints list in a negated constraint for the new goal.

The constraint processor makes the following two assumptions:-

a) a unique name assumption ensures that every different variable refers to a different constraint.

b) the infinite constant pool ensure that no two constants denote the same object.

3.4 Conclusion

NBC has a number of operational drawbacks. Firstly, it can only deal with pure prolog, that which has no control features. This means it is either of little use in procedural programs or requires embedding in a processing shell outside of which the control mechanisms reside. This may be a tedious but, as often, necessary overhead. Secondly, NBC can not deal with functions, only datalogic. Without functions the constraints processor only has to cope with constants of the program and those presented in goals in operation and does not have to express infinite inequalities which is impossible given the handling mechanism of NBC as it is presented. If functions were allowed, the algorithm for universally quantified goals would generally lead to infinite evaluation loop due to the presence of local variables. Functions, or what we may term defining clauses may be allowed, if we ensure that no variables are local or introduce other proof methods such as induction. This is examined in §9.

NBC is a solution for floundering yet it is not particularly flexible even in deductive database applications for which it was created. Implemented 'on top' of NAF and with the ability to handle quantifiers (again restricted) it is of most use in basic deductive databases and simple program clause sets. The constraints approach is examined in Part two with a view to extending the number of cases which the universal quantifier algorithm can handle.

§4
Constructive Negation

[Cha88] attempts to provide the most minimal extension to Prolog that solves the inadequacies of NAF. As with NBC, constructive negation (CNF, in the sequel) does solve the problem of floundering.

Preliminary Definitions

Constructive Negation This explicitly negates the disjunction given by the completion to a goal. Formally it is:

Given goal G has answers $A_1,\ldots,A_n$ the completion provides

$G \equiv A_1 \vee \ldots \vee A_\nu$

ΧΝΦ σιμπλψ νεγατεσ τηε Ρ(Ηενχε δισφυνχτιωε ανσωερ)τοεμπτψ

$\leftarrow \Gamma \equiv \leftarrow(A_1 \vee \ldots \vee A_\nu)$

4.1 Introduction

It is very closely linked to the naïve handling of constraints set in NBC. The constraint set specifies the substitution of variables so the result of a query by either method is equivalent. CNF makes more use of Prolog's backtracking than NBC so that negative literals in the program clauses are handled as efficiently as possible, particularly when given a conjunctive goal. CNF has the advantage of allowing functions.

Subgoals in CNF can be equality, inequality, universal inequality or literals. SLD-CNF refutations terminate in either the empty clause or a conjunction of primitive inequalities.

Negative subgoals are handled as follows:-

Goal $\leftarrow \Lambda_1,\ldots,\Lambda_\mu$

Σελεχτεδ λιτεραλ $\iota\omega = \Lambda \leftarrow \Pi$

There are two possibilities:

i) no answers $\Rightarrow$ Goal becomes $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$

ii) answers $A_1, \ldots, A_m \Rightarrow$ Goal becomes $L_1, \ldots, L_{i-1}, B_{j1}, \ldots, B_{jk}, L_{i+1}, \ldots, L_n$

where $B_{j1} \ldots B_{jk}$ is NA_j such that

$NA_1 \vee \ldots \vee NA_m \equiv \leftarrow (A_1 \vee \ldots \vee A_m)$

Again explicitly quantified negative subgoals are allowed such as $\forall \overline{X} \neg Q$ and $\neg \exists \overline{X}\, Q$. In these cases the quantified variable tuple is treated as a non goal variable so that the problem is solved for the free variables only. This simulates enforcing that the quantified variables hold for all cases which is the objective of universal quantification.

4.2 Constructive Negation Examples

The following examples show CNF evaluation for the given program P1:

```
p(U,f(U)):- U ≠ 1, q(U).
q(X):-X ≠ 3, r(X).
r(X):- ¬p1(X).
p1(X):- X ≠ 2.
p1(3).
p2(4).
```

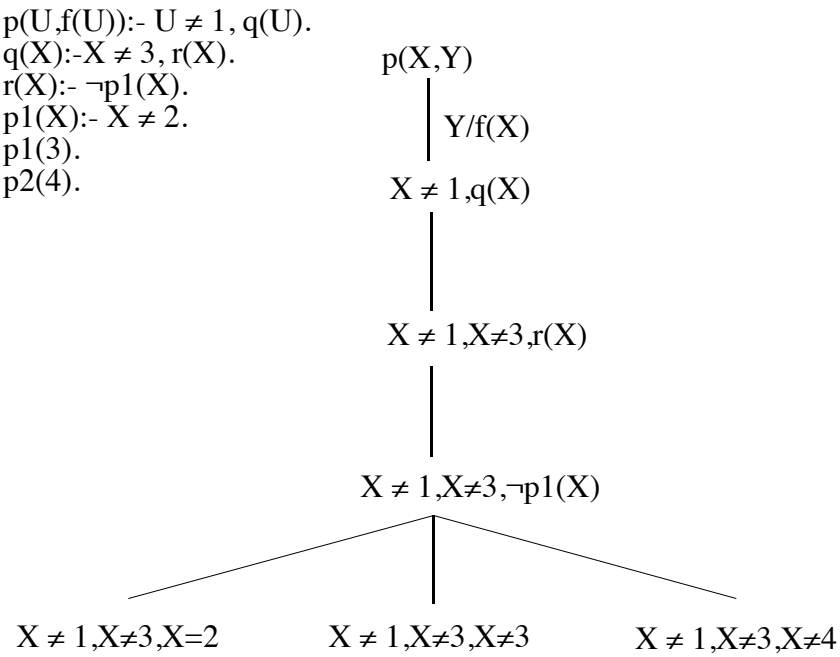


Figure 4.1 SLD-CNF evaluation for program P1

The above computation exits with the following answer set Ω.

Ω = [[X ≠1,X≠3,X=2],
[X≠1,X≠3],
[X≠1,X≠3,X≠4]].

We now present the same query in a negative form using the answer set Ω which will be negated by CNF processing.

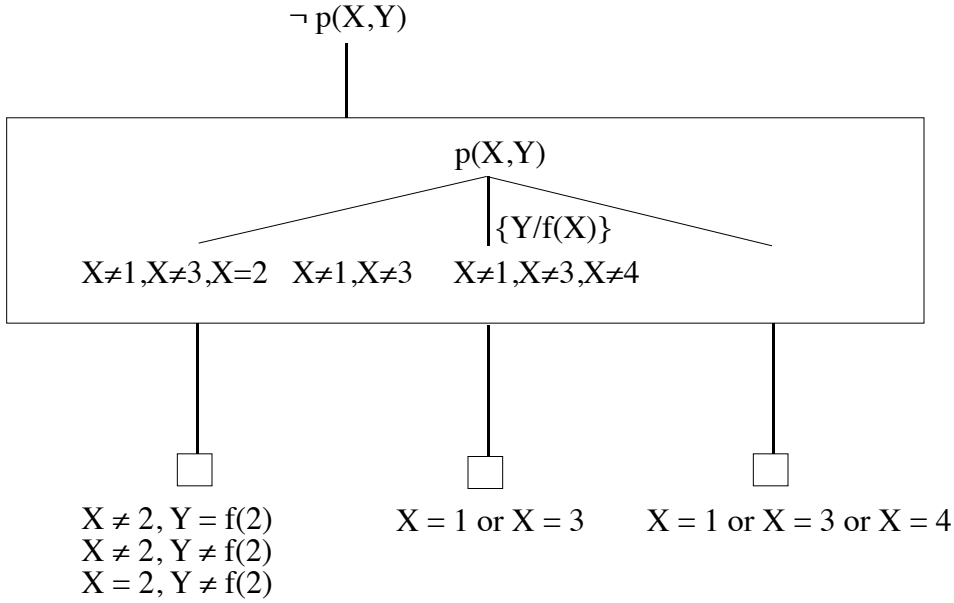


Figure 4.2 SLD-CNF Negative Query evaluation for program P1

With CNF, no longer does the order of goals remain a prime factor if there are non-ground negative literals. Indeed, the inequalities returned as the solutions of non-ground negative goals may sufficiently constrict the search space for the evaluation of remaining literals and in doing so improve efficiency. Redundancy is reduced more efficiently than in NBC. It is a major consideration for future logic systems that negative clauses are implemented as effectively as their definite counterparts and constructive negation goes some way towards this.

[Cha89] provides instructive examples showing where CNF may enhance program efficiency by using negative subgoals to generate inequalities (or equalities) to constrain the subsequent search tree whereas NAF would require ground negative goals at all times.

The treatment of universal variables under constructive negation where they are treated as 'non-goal' variables is not reported in any implementational depth. However, queries that are universally quantified may return answers that contain universally quantified variables due to their status of being 'non-goals'. Given the program,

bridge(1,1)
bridge(3,1).
bridge(3,Z):- Z ≠ 1

and the query $?\forall Y \neg \text{bridge}(X,Y)$ then the variable Y is normalised during CNF resolution and X is the only goal variable and $\{X \neq 1, X \neq 3\}$ is an answer. Unfortunately [Cha89] does not cover the implementation of 'non-goal' variables.

In his '89 paper on constructive negation, Chan states that the intensional negation technique is a form of constructive negation but one that cannot handle negative subgoals. This is due to the possible presence of local variables within these clauses and the inability of standard clauses to deal with universal clauses without SLDN resolution. There are methods or restrictions for allowance of negative subgoals and these are now detailed in the next section.

§5
The Transformational Approach

Preliminary Definitions

Intensional Negation This is a transformation technique that given a set of predicates p synthesises the definition $\overline{p}$ whose success set is equivalent to p's finite failure set.

Herbrand Generator A predicate for generating all Herbrand terms from the given language. For every constructor in the language the generator requires a recursive instance for all terms that will behave fairly to all terms if possible.

Tamaki-Sato Transforms These refer in the most basic context to unfolding, where the body of a clause replaces its head in the body of another and folding where a conjunct of literals in a clause body may be replaced by a corresponding clause head in the program.

Left-Linear Program No clause head contains more than one occurrenceof the same variable. Explicit unification must occur in the body for variables to be unified in the head. Non-left linear programs will lose the unification in clause heads during set-theoretict complement calculation.

SLDN Resolution As SLD resolution but incorporates a special procedure to deal with universally quantified variables that occur during transformation.

5.1 Transformation Introduction

Introduced in the papers [Bar90] and [Bar87] initially under the title of Intensional negation, these papers refer to a transformation approach for definite clauses that is an extension of the Tamaki-Sato transforms. [Ng90] expanded this approach to include negative literals for partially evaluated transformations. What is presented here is a complete transformation procedure that includes negative literals. Negative subgoalsl are acceptable providing SLDN is not the resolution method, the reasons for this are given later. Transforming a program allows non-ground querying on negative literals as all clauses are treated positively. All transformations are conducted with respect to

the completed database and the coinciding equality theory which for the purposes of calculating the complement of a term includes the domain closure axiom. It is the completion that is used to introduce the negative half of a program. The negative half is explicitly created using the language of the original program and in this manner positive and negative information are handled symmetrically. SLD-resolution does not suffice for the transforms as they generally do not correspond to horn clauses. SLDN-resolution is presented in [Bar90] and is required to deal with local variables that arise in the negation process, possessing the following property:-

$$\mathbf{SS_{SLDN}(P) = FF_{SLD}(P)}.$$

5.2 Set Theoretic Complement

A major part in the transformation process is the calculation of the set theoretic complement of a term. This is vital to the creation of the converse of the program as all terms are a representation of a set of ground instances with respect to the interpretation domain that is characterised by the DCA. Therefore all terms in the language different from the original may be calculated. An important property of this process is its use of the DCA as covered in §2. If the term did contain multiple occurrences of the same variable the ability to finitely represent the complement would be lost as an infinite amount of different terms within functions would require calculation.

Given an unrestricted term t, *that which has no multiple occurrences of the same variable*, and a first order language then t', the set theoretic complement of t, can be calculated with the following algorithm where X is a variable and $\overline{X}, \overline{Y}$ are variable tuples:-

$$Not_L(X) = \{\ \}$$

$$Not_\Lambda(\chi_\iota) = \{\chi_\varphi \mid \chi_\varphi \text{ is a constant and } \chi_\varphi \neq \chi_\iota\} \cup \{\chi_\varphi(\overline{\Xi}) \mid \chi_\varphi \text{ is a function symbol and } \chi_\varphi \neq \chi_\iota\}$$

$$\text{if } \chi_\iota \text{ is a constant}$$

$$Not_\Lambda(\chi_\iota(\tau_1, \ldots, \tau_\mu)) = \{\chi_\varphi \mid \chi_\varphi \text{ is a constant and } \chi_\varphi \neq \chi_\iota\}$$

$$\cup \{\chi_\varphi(\overline{\Psi}) \mid \chi_\varphi \text{ is a function symbol and } \chi_\varphi \neq \chi_\iota\}$$

$$\cup \{\chi_\iota(\xi_1, \ldots, \xi_{\kappa-1}, \theta, \xi_{\kappa+1}, \ldots, \xi_\mu) \mid \kappa = 1, \ldots, \mu, \theta \in Not_\Lambda(\tau_\kappa)\}$$

Given the following term cons(X,cons(Y,cons(Z,nil))) and the signature L = [nil,cons(_,_)] the set theoretic complement is:-

{ nil
cons(V, nil),
cons(V1, cons(V2, nil)),
cons(V3, cons(V4, cons(V5, cons(V6, V7)))) }

Note here the representation for lists which is used for our purposes in all list transforming programs as the complement in conventional notation is {[], [V], [V1,V2], [V1,V2,V3|V4]}. The bracket representation could be used if the complement function were extended to accept such input.

The complement of a tuple is embedded in the above recursive complement algorithm. Its formal definition is:-

$$Not_L(t_1,\ldots,t_m) = \{(\xi_1,\ldots,\xi_{\kappa-1},\tau,\xi_{\kappa+1},\ldots,\xi_\mu) \mid \kappa = 1,\ldots,\mu, \tau \in Not_\Lambda(\tau_\kappa)\}$$

Without the DCA, if an object is not one of the constants of the language we have no implication that it is one of the language constructors. The DCA ensures that fresh variable tuples are constrained to be a member of the complement set of a term if they are not equivalent to the original tuple (cf. Theorem 3.1, [Bar90]).

[Bar90] develops the transformation theory extensively using factorised logic programs (FLPs, in the sequel). The paper notes that the more concise negated logic programs are harder to be tackled formally. Obviously, all FLPs preserve the semantics of their original, unfactorised counterparts. FLPs have separate predicate calls for all disjunctive clause heads, unification and recursive body calls. The implementation created for the purpose of this report did not use FLPs and so the formal presentation of the transformation procedure is developed with reference to the intensional negation technique which requires more steps when transforming the completed definition. The theoretical transformation can be covered in seven steps. These are:

1. Rewrite in left-linear form, if necessary
2. Complete the program
3. Apply logical negation
4. Apply equivalence lemma, if necessary
5. Replace negative inequalities by computation of set theoretic complements
6. Find the most general unifiers of conjunctions of the same variables
7. Extract the negative program from the completed definition

The naïve reverse program transfromation is presented step-by-step with reference to $Comp_{EQ+DCA}$ in Appendix 4. This is now carried out for a program that tests if a numeral is odd,

odd(s(0)).
odd(s(s(X))):- odd(X).

The completion of this program is:-

$\forall X\ odd(X) \Leftrightarrow (X = s(0)) \vee \exists Y\ (X = s(s(Y)) \wedge odd(Y))$ together with the equality theory augmented with the DCA. We now apply logical negation to obtain:-

$\forall X\ \neg odd(X) \Leftrightarrow \neg(X = s(0)) \wedge \neg\exists Y\ (X = s(s(Y)) \wedge odd(Y))$

All occurrences of the negative odd clause are now replaced by a positive $\overline{odd}$ and applying the negations procures:-

$\forall X\ \overline{odd}(X) \Leftrightarrow (X \neq s(0)) \wedge \forall Y\ (X \neq s(s(Y)) \vee \overline{odd}(Y))$

An equivalence, the proof of which is in [Cha88], is now applied:-

$\forall X\ \overline{odd}(X) \Leftrightarrow (X \neq s(0)) \wedge (\forall Y\ (X \neq s(s(Y))) \vee \exists Y\ (X = s(s(Y)) \wedge \overline{odd}(Y)))_{No}$

w the intensional complements of terms in negative guards on the RHS are computed (using Not_L). This gives us:-

$$\begin{aligned}\forall X\ \overline{odd}(X) \Leftrightarrow\ & (X = 0, X = 0) \vee \\ & (X = 0, X = s(0)) \vee \\ & (\exists Y\ X = s(s(Y)), X = 0) \vee \\ & (\exists Y\ X = s(s(Y)), X = s(0)) \vee \\ & X = 0, (\exists Y\ X = s(s(Y)) \wedge \overline{odd}(Y)) \vee \\ & (\exists Y\ X = s(s(Y)), \exists V\ (X = s(s(V)) \wedge \overline{odd}(V))\end{aligned}$$

Conjunctions of the positive guards are now unified so that the difference in the heads of clause can become syntactically equivalent. We have the following most general unifications $\{ <0/0>, <Y/V>\}$ for the above to give the final completed definition.

$\forall X\ \overline{odd}(X) \Leftrightarrow X = 0 \vee \exists Y(X = s(s(Y)) \wedge \overline{odd}(Y))$

The predicate for the negated clause can now be presented;

$\overline{odd}$ (0).
$\overline{odd}$ (s(s(Y))):- $\overline{odd}$(Y).

It is obvious how advantageous it is in many situations to be able to query with non-ground instances of the negative part of a program. Generally, it is a suitably

effortless task to extract the predicate definitions from the completion. The only difficulty resides in the presence of universally quantified local variables that do not implicitly correspond to any form of horn clause.

A simple example describes an interesting property. If we have the following database clauses for the language L = [chris,colin]:

happy(chris)
happy(colin)

then the transformed completion for this after logical negation and renaming of negative clauses gives us:

$\forall Y\ \overline{happy}(Y) \leftrightarrow (Y \neq chris, Y \neq colin)$

and not_L is now applied to the negative guards to give

$\forall Y\ \overline{happy}(Y) \leftrightarrow (Y = colin, Y = chris)$

Unification now fails and so the clause happy has no complement under the restriction that the language does not vary from its contents under which the transformed clause was created. Ideally the transformation approach would be combined with a constraints evaluator to allow flexibility in the language as the complement clause would be:

$\overline{happy}(Y)$:- $Y \neq \chi o \lambda \iota \gamma \Psi \neq \chi \eta \rho \upsilon$

If the language is now modified the transformation process does not have to be repeated. Obviously any changes to the happy clauses require repetition of the transformation process if the converse $\overline{happy}$ clause set is still desired.

Successful Inclusion of a Negative Subgoal

To make things easier we use the above predicate for the language L2 = [chris, colin] and the new clause **dead(X):- not happy(X)**. The completion of this provides:

$\forall X\ (\overline{dead}(X) \leftrightarrow happy(X))$ and this will succeed for both {X/chris} and {X/colin}. In this instance the negative subgoal caused no problems. If the subgoal had included local variables in a negative instance problems occur with the interpretation of the existential quantifiers and therefore their transformation. It is much safer to stick with negative subgoals that do not possess local variables.

5.3 Evaluating Universally Quantified Subgoals.

As one solution to the problem of universally quantified subgoals, [Bar90] present an extension of SLD resolution, namely SLDN resolution. This solves a $\forall \overline{Y}\, q''(\overline{X},\overline{Y})$ predicate with following conjunction:-

$\langle q''(\overline{X},\overline{Y}), naf_{\overline{Z}}\, q(\overline{X},\overline{Z})\rangle$

This evaluates q" obtaining an instance of the variables that are not universally quantified and ensures they succeed for all values of the quantified variables by calling the original clause with the universally variable uninstantiated. As NAF only succeeds for ~q if it fails on all braches, the universal property is therefore proved in this generate and test fashion. NAF in the context of SLDN does not flounder as its use is only for universal variable checking. It would fail however under a safe computation rule that selects positive literals or ground negative literals.

SLDN Resolution

There are a number of deficiencies in SLDN resolution caused by local variables. An example in [Bar90] displays how local variables have different bindings and so the NAF part of SLDN may fail for certain bindings when the original would succeed due to the incompatibility between a conjunction of goals. This is lost after the transformation procedure. Suppose we have thefollowing program:

s(X):- r(X,Y)
r(X,Y):- q(X,Y),p(X,Y)
q(X,a)
p(X,b) which has the transformed converse:

$\bar{s}(X)$:- $\forall\Psi\, \bar{\rho}(\Xi,\Psi)$
$\bar{\rho}(\Xi,\Psi)$:–$\bar{\theta}(\Xi,\Psi)$
$\bar{\rho}(\Xi,\Psi)$:–$\bar{\pi}(\Xi,\Psi)$
$\bar{\theta}(\Xi,\beta)$
$\bar{\pi}(\Xi,\alpha)$

The universal quantifier is evaluated under SLDN as
$\bar{s}(X)$:- $\bar{r}(X,Y)$, not r(X,W) where the transform is used to obtain a candidate solution that is tested under negation as failure. The problem here is that an original query **?s(a)** which would fail due to incompatibility of the **p** and **q** clauses is not generated by the transformation.

A solution to this is proposed. Namely, instantiate the non-quantified variables in the NAF conjunct of the SLDN clauses using the language provided. A procedure to achieve this is referred to as a *Herbrand generator*. Doing this is theoretically sound as the non-local variable may be uninstantiated and can only be, for the purposes of this technique, a member of the set of Herbrand terms. Computation rules that delay selection of negative goals will also specialise the query further and will provide more

successful transformation results. As an aside, in the transformation method implemented for this report, if the **p** and **q** clauses were transformed as datalogic and the other clauses as for defining clauses we would have the following definitions:

$\bar{s}(X)$:- $\forall\Psi\, \bar{\rho}(\Xi,\Psi)$

$\bar{\rho}(\Xi,\Psi)$:–$\bar{\theta}(\Xi,\Psi)$

$\bar{\rho}(\Xi,\Psi)$:–$\bar{\pi}(\Xi,\Psi)$

$\bar{\theta}(\alpha,\beta)$ $\bar{\theta}(\beta,\beta)$

$\bar{\pi}(\alpha,\alpha)$ $\bar{\pi}(\beta,\alpha)$

and a Herbrand generator is not required as the finite failure set contains $\bar{s}(b)$ and $\bar{s}(a)$. This is detailed in §9.

Universal subgoals require the evaluation of the transformed clause to provide a possible answer substitution and it is obvious that the answer substitution will satisfy the Herbrand generator. Situations where this is not the case are presented further on. Herbrand generators prevent the allowance of NAF in programs for transformation and this is a key issue when other techniques for proving universal goals are examined that allow NAF. The implementation of Herbrand generators require a significant extension when used in practice to that shown in [Bar90] due in the main to the infinite retrieval of terms and thus the success of a goal infinitely many times.

[Bar90] prove the soundness and completeness of SLDN-resolution with respect to the completion extended with the DCA. Completeness ensures answer substitutions to negative theorems are provided by $\text{Comp}_{\text{EQ+DCA}}$.

$\text{Comp}_{\text{EQ}+\Delta\text{XA}}(\Pi)\ \alpha\ \ \forall \leftarrow \pi(\bar{\tau}\ominus)$ ιμπλιεσ $\text{Χομπ}_{\text{ΕΘ}+\Delta\text{XA}}(\Pi)\ \alpha\ \leftarrow \pi(\bar{\gamma}_{\iota})$

φορ αλλ γρουνδ ινστα$\bar{\gamma}\chi\theta\phi\ \bar{\tau}\ominus$

An SLDN-refutation supplies a covering of a universally quantified negative theorem.
The answer substitution ß given by SLDN are a covering for the negatively proven tuple when ß is applied to a ground instance of the tuple. The proof is detailed in Theorem 6.3 in [Bar90].

Because of the requirement that all clauses are unrestricted, a necessity for the set theoretic complements to be fully calculated under the DCA, the eq clause that unifies variables cannot be transformed by the given technique and therefore has to be created manually. It is defined as:-

$\overline{eq}(c_i(\overline{X}),\chi_\varphi(\overline{\Psi}))$. φορ αλλ χονστρυχτορσ

$\overline{\varepsilon\theta}(\chi_\kappa(\xi_1,\ldots,\xi_{\kappa\iota}),\chi_\kappa(\psi_1,\ldots,\psi_{\kappa\iota})):-\overline{\varepsilon\theta}(\xi_\varphi,\psi_\varphi)$. φορ αλλ αργυμε

A significant implementational disadvantage of the $\overline{eq}$ predicate is that for it to be implemented wholly it requires an inbuilt occur check to be included in the predicate definition. The eq predicate may be asserted for a language as an addition to the transformation process. For example, if we have the language L = [nil,cons(_,_)], $\overline{eq}$ is asserted to be:-

$\overline{eq}$ (nil,cons(X,Y)).
$\overline{eq}$ (cons(X,Y),nil).
$\overline{eq}$ (cons(X,Y),cons(V,W)):- $\overline{eq}$(X,V).
$\overline{eq}$ (cons(X,Y),cons(V,W)):- $\overline{eq}$(Y,W).

Another use of the $\overline{eq}$ predicate is to replace the need to calculate the set theoretic complement. It may improve transformation efficiency at the cost of more run-time calls. This method requires the use of 'Skolem' constants. Given, $\forall Y\ X \neq s(s(Y))$ the transformation creates $\overline{eq}$ (X,s(s(strange))) where strange is a Skolem constant preventing infinite evaluation. The standard theoretic complement would create X = 0 or X = s(0) which is the same answer set that $\overline{eq}$ will return for the language L = [0,s(_)].

5.4 Other Techniques for Universal Evaluation

Many of the techniques for the evaluation of universally quantified (sub-)goals rely on analysis of the goal in question. If it known that the clause corresponds to a database of ground clauses then Wallace's constraint evaluation algorithm suffices.

$\forall \overline{Y}\ \overline{p}(\overline{X},\overline{Y})$ may be the result of transforms from a number of different clause formulations. If the original clause had a conjunction of literals in the body the transform corresponds to a disjunction and so values may be instantiated from an original clause and then tested in a transformed clause. An example of this is:

p(X,Y):- q(X,Y),q2(X,Y) which results in the transform

$\overline{p}(X,Y):-\overline{\theta}(\Xi,\Psi)$

$\overline{\pi}(\Xi,\Psi):-\overline{\theta 2}(\Xi,\Psi)$ ανδ σο τηε υνιωερσαλ θυαντιφιερ ισ

$\forall\Psi(\overline{\theta}(\Xi,\Psi)\vee\overline{\theta 2}(\Xi,\Psi))$ ανδ τηισ χαν βε ιντερπρετεδ ασ

$\forall\Psi(\theta(\Xi,\Psi)\rightarrow\overline{\theta 2}(\Xi,\Psi))$

Now both the original and its transform may be used together to make the task of universal evaluation specialised to the term values. This relies on the disjunctive theory that if $\overline{q}(X,Y)$ (= ~q(X,Y)) fails then $\overline{q2}(X,Y)$ must succeed for the universal goal to be satisfied. This process is now shown for the following program:

q(X):- p(X,Y)
p(X,Y):- q(X,Y),q2(X,Y)
q(a,b)
q(a,a)
q2(b,a)

The transformation of this gives us:

$\overline{q}(X) :- \forall \Psi\, \overline{\pi}(\Xi,\Psi)$

$\overline{\pi}(\Xi,\Psi) :- \overline{\theta}(\Xi,\Psi) \vee \overline{\theta 2}(\Xi,\Psi)$

$\overline{\theta}(\beta,\alpha) \quad \overline{\theta}(\beta,\beta)$

$\overline{\theta 2}(\alpha,\alpha) \quad \overline{\theta 2}(\beta,\beta) \quad \overline{\theta 2}(\alpha,\beta)$

The subgoal $\forall Y\ \overline{p}(X,Y)$ now reduces to $\forall Y\,(q(X,Y) \rightarrow \overline{q2}(X,Y))$ and this is satisfied for the answer substituion to $\overline{q}$ of {X/a}. Such a transformation is useful when the original clause evaluation has a finite solution set and does not flounder (in this case referring to p(X,Y) only).

Implementation of this may be achieved with a solve_all predicate detailed in §9 that is a variant of find_all for specified variables that returns all solutions to unbound variables and then a restriction test on the values themselves so that the universal property is or is not satisfied.

Evaluating Classes of solutions

A subgoal for universal satisfaction may not admit a finite set of solutions but we may know from the domain closure axiom that it does admit a finite class of solutions. Assessment of a solve_all for this will lead to infinite evaluation, obviously impossible to detect computationally. $\forall Y\ \overline{p}(X,Y)$ becomes $\forall Y$ (herbrand(Y) $\rightarrow$ $\overline{p}(X,Y)$). The predicate 'herbrand' actually defines the DCA. As we are assuming an infinite solution set this is imminently likely to be recursive. [Ng90] introduces this method and shows how it might be solved with Natural Deduction mechanism where it is attempted for all disjuncts of the DCA using Skolem constants where functions are concerned. A simpler situation is when the classes correspond to an equality and its converse inequality over the Herbrand Universe. The intuition of this is equivalent to that of Wallace's universal evaluation algorithm.

5.5 Conclusion

The advantage of specifying sections of clauses from a program to be either defining clauses or ground database clauses is of primary value when required to evaluate a universal variable that would require a Herbrand instantiation with [Bar90]'s mechanism. Negative programs allow the user to gain answers about which properties do not hold. This is of important value within the realms of expert systems and intensional negation could prove to be of worth in planning and machine learning. In the most basic planning situation we could ask of a blocks world which blocks were not on top of block A to see if certain pre-conditions are satisfied from an initial stage and from the result generate a plan efficiently. Such an application requires development. [Bar88] concluded that one of the most important issues is optimisation techniques for the transformer and this still applies when the transformer is given a large input program.

§6
Qualified Answers

Qualfied Answers are examined not with regard to partial evaluation, their more general use, but for obtaining transformed goals so that a query will return more answers than negation as failure.

Preliminary Definitions

Qualified Answer Given a program P, goal G and an answer substitution ß such that G·ß follows from the program. A Qualified Answer imposes a hypothesis H such that the theorem [G·ß if H·ß] follows from the program. Answers are qualified by H·ß and so are not the more standard answers substutions.

6.1 Introduction

The main use of qualified answers is that of the compile time uses of program transformation and macro expansions. These avenues and not traversed for the purposes of this analysis. [Vas86] presents an abstract interpreter that generates qualified answers using the additional arguments (to those of program and query in Kowalski's interpreter) of an explicit substitution parameter and an argument for representing the qualifications. The abstract interpreter will reduce a query to a collection of subgoals corresponding to the tips of branches on its search tree.

Additional answers may be returned when the right hand side of the original query is resolved with its constituent parts and, possibly, folded with other clause instances. Using a meta-interpreter to achieve this allows for potentially many more results than negation as failure which prevents solutions from returning values.[22]

---

[22] In the more simpler cases these would be negations of variable bindings which occurred during the resolution of the goal in question.

6.2 Qualified Answer Motivation

Qualified answers show the solution space much more precisely than usual goal evaluation where not just successful bindings are returned but all of the unbound clauses which lead to the generation of these bindings where there is the possibility of multiple answers. In the case of negative clauses (those prefixed by 'not') standard evaluation would not return any successful substitutions from these clauses whereas qualified answers would still return the unbound clauses. This is why their use may return more results than the use of negation as failure by providing more expansive solutions.

A qualified answer interpreter may generate results and assert these results for the prolog interpreter and then ground queries (for results with negative clauses) may be given to the interpreter to test their validity. This process may be of use in a number of situations. If qualified answers are applied to the transformation process (§5) then no additional solutions will be given but additional clause instances that provide the same answer substitutions may be provided by the process. Such can be applied to many meta-interpreting tools, particularly for debugging purposes. The standard application for qualified answers, not described here, is that of a specification tool.

6.3 Illustrative Example

The following example shows how qualified answers may return additional answers to negation as failure. The following program is used for this example. It is rather contrived:

```
number(0)
number(s(X)):-number(X).
number(s(X)):- not letter(X).
equiv(0,0)
equiv(1,s(0)) etc.
same_numerals(X,Y):-number(X),equiv(X,Y)
```

This rather contrived example shows that in addition to the standard answers substitutions the qualified answer **same_numerals(X,Y) if not letter(X),equiv(X,Y)**

is returned that would not generate any result with negation as failure.

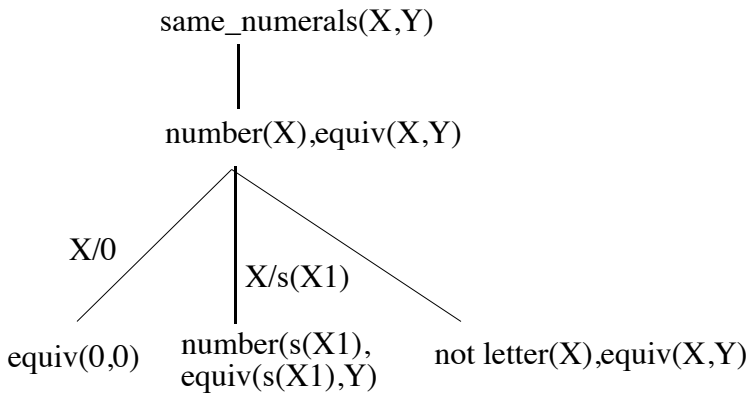


Figure 6.1 Qualified Answer Evaluation

If, as suggested, these generated answer substutions are asserted they can be tested with ground queries.[33] This example is of little practical value but it does convey the worth of the technique. Qualified answers may have a place in constraints processing and this method is certainly more efficient than the transformation process when it comes to retrieving answers that would not be accessible with NAF.

[3]3 Ground because of the inclusion of a negative subgoal.

§7
Inductive Techniques

7.1 Introduction

Induction has been implemented in a number of systems using Prolog [Kan84, Kan87] and these systems refer to the inductive proofs of program properties. The relevance of such proofs for this approach is to prove universal conditions that occur as a result of transformation of a program p to its finite failure counterpart $\overline{p}$.

[Ng90] suggested making use of the natural well-founded ordering of the Herbrand Universe and implementing induction as follows. If the universal condition is $\overline{p}(X)$ and X is universally quantified and the language is L = [0,s(_)] then to show $\forall X\, \overline{p}(X)$ show $\overline{p}(0)$ as the base case (of which there will frequently be more than one) and then assume $\overline{p}(X)$ to be true for an arbitrary depth n and then show $\overline{p}(X2)$ for depth n+1 which will reduce to $\overline{p}(X)$ where the hypothesis can be used. It may be probable that an implementation of this can be formed from the constraints approach. This would be achieved by incorporation of an inductive universal algorithm as a variant of the universal algorithm which Wallace presents. It would use the form of the clauses in the program to generate an induction schema and then prove all of the base cases as for a standard constraints evaluation and assert the hypothesis as a possible (satisfied) constraint and prove the inductive case using this assertion.

The detailing of a sample induction schema is now provided and then details of how this is used. Unfortunately concentration of other aspects of negation implementations meant that no inductive techniques were implemented. Hopefully this may provide an outline for future projects. [Kan84] present a formulation of computational induction that can easily by applied to programs given the least fixpoint semantics of Prolog that allows first - order formulas. A type concept is introduced so that clauses which define particular data structures are separated from the clauses defining procedures.

According to the model-theoretic semantics a (possibly) quantified formula is a logical consequence of the least Herbrand model. Under proof theoretic semantics a formula is proved from the completion by computational induction.

7.2 Induction Scheme Generation

This induction is applied by:

a) select the key atom

b) find conditions required for soundness

c) simplify induction schemes

Terminology for induction according to [Kan87]. The notion of positive and negative subformula of a formula are introduced, defined as follows:-

a) F is a +ve subformula of F

b) ¬ G is +ve (-ve) then G is a -ve (+ve) subformula of F

c) $G \wedge \Xi$ or $\Gamma \vee \Xi$ is a +ve (-ve) subformula then G and X are +ve (-ve) subformulas of F

d) $G \supset \Xi$ is a +ve (-ve) subformula then G is a -ve (+ve) subformula of F and X is a +ve (-ve) subformula of F

e) $\forall XG, \exists XG$ are +ve (-ve) subformulas then Gx(t) is a +ve (-ve) subformula of F.

Free variables are universally quantified and undecided variables are existentially quantified. Specification formulas (S-formulas) are

a) no free variable in X is quantified in scope on an undecided variable.

b) no undecided variable appears in the negative atoms of S

Manipulation of goal formulas through substitutions are also introduced. We detail generation of an induction schema using the following program taken from [Ster86]:

```
sumlist([],0).
sumlist([I|Is],Sum):-
        sumlist(Is,Partsum),
        add(I,Partsum,Sum).
```

This relation includes the smallest set of pairs of terms that includes a pair ([],0) and that for any term I, includes ([I|Is],S) whenever it includes (Is,Ps) and (I,Ps,S) is in the add relation. For any Herbrand Interpretation we let Q(A,B) denote a binary relation over terms. Hence the following computational induction scheme is arrived at:-

$$\frac{Q([],0) \qquad \forall A,B,X,Z(\Theta(A,B) \wedge \alpha\delta\delta(Z,B,X) \supset \Theta([Z|A],X)}{\forall A,B(\sigma\upsilon\mu\lambda\iota\omega(A,B) \supset \Theta(A,B))}$$

The application of this induction scheme is now achieved by resorting to inference rules such as definite clause inference, and, or and implication deletion, negation as
failure inference and simplification where a goal is assumed to be true or false. It is highly probable that these rules could be combined with a constraints interpreter so that all general

universal conditions can be solved within Prolog.

Part Two §8 Introduction

Now that the theoretical backgrounds of the different methods of treating negation besides NAF have been covered, the design and program overview of the transformation program and a constraints meta-interpreter are provided.

The implementation details of the transformation procedure include designs of the set theoretic complement, linerisation procedures, negation for a set of clauses and then for a whole program and enumerations of dealing separately with ground and defining clauses. Execution examples are provided as well as aspects of the data structures required for successful program operation.

The constraints sections concentrate on:

- a naive constraints meta-interpreter that evaluates universal conditions not by the actions of Wallace's Universal Evaluation algorithm but by examination of results achieved via a 'solve_all' variant of the findall predicate in most prolog systems which returns all solutions for the desired variables and is able to perform logical negation on the returned answer set if necessary. It is naive as its solutions are not dictated by the constraint values at run-time.

- a constraints meta-interpreter that incorporates the algorithms of Wallace into its evaluation and rejects/accepts solutions as the the current statue of variables in the constraint structures.

Inductive implementation methods and other further work is covered in §10.

§9
The Transformation Implementation

The transformation technique was implemented in Edinburgh syntax Prolog with the LPA MacProlog compiler. Any features that were used within the program specific to LPA MacProlog are noted so that the program may easily be adapted to other Prolog systems with minimal modification. Transformation implementation code is given in Appendix 5 and the transformation system user guide is presented in Appendix 1.

Preliminary Definitions

Database clause	Ground Clauses of the form **parent(deborah, malcolm).** It is referred to, for implementation purposes, as 'Datalogic'.
Defining clause	A clause, possibly non-ground and/or recursive that defines a function in the program

9.1 Transformation Requirements

The implementation of the procedure for transforming a program from its original form to its finite failure (FF, in the sequel) counterpart is now presented with specification of all of the individual program requirements.

Program Requirements

Linearisation

Programs which are non-left linear (those that contain restricted arguments having multiple occurrences of the same variable in clause heads), lose this unification upon calling the complement operation which returns an empty set when given a variable as input thus losing the necessary unification performed in the head.

Set Theoretic Complement

The complement of a term in a clause head has to be calculated when a program's negative counterpart is being created. This is done using the Not_L algorithm of §5. If we refer to the transformation of naïve reversal, presented in appendix 4, with

reference to the completion it is obvious that complement calculation becomes necessary when a variable is constrained to be unequal to a specific term implying that it is one of the complements of this term in the language. For reverse, when we arrive at X ≠ [V|Vs] X must be an empty list according to the provided language.

It was intentionally left to the user to input the language of constants and constructor symbols with respect to which the set theoretic complement is calculated. The main reason for this is the allowance of constants or functions which may not be gleamed from parsing the input program to form a set of constants and functions. It allows for the language to be altered easily without inserting, removing or changing predicate definitions already present in the program.

Clause Negation

For each clause C of the form $p(\bar{t}) :- p_1(\bar{t_1}),\ldots,p_k(\bar{t_k})$ we create NegC(C), its negative counterpart from the following rules of [Bar90];

$\bar{p}(\bar{s}) \leftarrow$ φορ εαχη $\bar{\varphi} \in No\bar{\tau}_i(\bar{\tau})$

$\bar{\pi}(\bar{\tau}) \leftarrow \forall \bar{\zeta}_i\ \bar{\pi}_i(\bar{\tau}_i)$ φορ εαχη $i = 1,\ldots,\kappa,$ ωιτη $\bar{\zeta}_i = \varpi\alpha\rho(\bar{\sigma}_i) \cdot \varpi\alpha\rho(\bar{\sigma})$

and if the second formula has no local variables ($\bar{\bar{z}}_i = \{\}$) then the clause is to be read as $\bar{p}(\bar{t}) \leftarrow \bar{\pi}_i(\bar{\tau}_i)$,

so it is much simpler when local variables are absent as the transformation results in standard Horn clauses.

The Unification Procedure

Given the two clauses R1 and R2 which can be read as

$A_1 \leftarrow B_1$ ανδ $A_2 \leftarrow B_2$ respectively, [Bar90] introduce the commutative, associative and idempotent '@' operator defined as

$R_1 \ @\ R_2 \equiv A_1\theta \leftarrow (B_1, B_2)\theta$ ιφφ $\exists\ \theta = \mu\gamma\upsilon\,(A_1, A_2)$

Idempotency of the operator guarantees that for a clause $A\theta = (A\theta)\theta$ and that the substitution is legal and that all variables which are unified with are distinct (termed the functionality principle) so that we do no obtain, for example, {V/[],V/[X|Xs]} as a substitution instance. Associativity provides a final result if unification succeeds irrelevant of the ordering. Given the two clauses,

p(f(X)):- q(X)

p(f(X1)):- v(X1) unifies with **mgu(X,X1)** to give

p(f(X)):- q(X),v(X)

Note how the body is also unified and the ordering of body clauses is not taken into account, a factor that may prove deciding with an unsafe computation rule.

In forming the set of clauses that define a predicate the set of successful unifications from all possible clause combinations that are obtained from the negation of a clause procedure form the definition of that predicate. Formally Neg(P) is

$$\mathrm{Neg}(p) = \{\overline{\pi}(\overline{\Xi}) \leftarrow\} \quad \iota\phi\, X\Lambda(\pi) = \varnothing$$

$$N\varepsilon\gamma(\pi) = \{P \mid P = P_1 \cong P_2 \cong \ldots \cong P_\nu,\ P_\iota \in N\varepsilon\gamma X(X_\iota)\}$$

$$\iota\phi\, X\Lambda(\pi) = \{X_1,\ldots,X_\nu\}$$

for each distinct predicate p in P. The operation of the 'set' uniqueness property of the second case is questioned later as the standard set rule leading to distinct elements is not robust enough for database querying. An example of this is shown later in the section to display the implementation process.

Ground Clause Construction for Database Clauses.

As the negation of a predicate rule stands, under a language L = [a,b,c] and a binary predicate p we are likely to obtain clauses such as $\overline{p}(c,X)$ as a result of an original program p =

p(a,b) p(b,c).

The problem of this result is apparent for non-ground querying such as $?\,\overline{p}(c,X)$. Non-ground querying is one of the primary motivations of the transformation process yet it runs aground for database clauses when there are constants in the language that are not present in the program. That is why it is a key issue to handle the transformation of data and defining clauses separately.

9.2 Set Theoretic Complement Design

The complement procedure follows directly from the Not_L algorithm shown in §5.

Variable Handling

As in all Prolog meta-programs which deal with clause arguments of clauses themselves which are likely to be variables special attention is required. Firstly, they cannot be treated as terms because the Prolog system recognises variables for binding and in doing so renames them to local variables denoting memory positions on the stack. If a variable instance is bound to a non-variable value all instances of that particular variable will be bound throughout the program which, when the variable itself requires comparison, lead to failure. Careful use of testing and of the non-logical *copy_term*/2 (where the output is a copy of the input with variables renamed) is necessary.

The standard solution is to use a toground/3 predicate which replaces variables with terms by assigning single quotes (variable X becomes atom 'X'). When programs are manipulated in such a way problems of dealing with unwanted variable bindings during program operation are removed.

Grounding of variables was not selected for the following reasons:-

- In calculating the complement of a term every argument would have to be matched against the given signature whilst, if maintaining variables, a simple call to *var* performs the task.
- In the operation of the Not_L algorithm variables return an empty set; again, making the explicit use of variables means the empty markers are not required.
- A potentially infinite number of distinct fresh variables may have to be introduced when replacing arguments in the complement operation. These are intended to map to any element in the language but generating terms would lead to further comparison checks in subsequent executions of the algorithm increasing computation time.

The algorithm for generating the complement of a term is:-

$Not_L(X)$ =

X is a variable - return the empty list

X is a constant - return all other language members

X is a function with k arguments -

calculate complement of all function arguments by forming a structure of arguments in the form of t/N referring to term t being argument in position N. For each t/N apply complement algorithm and assign /N to all $\bar{t}$ terms returned. Successively create terms from the complement structure list by systematically placing $\bar{t}$/N in Nth argument position of function and replacing arguments 1 to N-1 and N+1 to k with **fresh variables**.

This complement procedure is relatively efficient in that the list structures map directly to the generated sets of the Not_L algorithm. As a practical note it was decided to represent lists in the language as the constructors nil and cons(_,_) representing [] and [_|_] respectively. A side effect of this is if the user is not solely transforming defining clauses and presents the program with a language consisting of constructors and constants for both database and defining clauses. An example of the potential problem is L = [a,b,nil,cons(_,_)] the complement will return a and b as complements to either nil and cons(_,_) when it will usually be the case that these constants will only be required to appear inside the functions. The answer, however, is theoretically

correct but operationally clouded. Given **nil** we may only require **cons(_,_)** as the complement. Solutions to this ambiguity are

- to separate the language into database and defining constants. Once the defining complements have been formed the database iterms may be inserted if necessary.
- to remove non-defining constants from the language to obtain only the base instances so that other terms can be inserted independently to the user's requirements once transformation has been achieved.

9.3 Linearisation

This is a simple procedure. Wherever the head of a clause contains more than one occurrence of a variable all equivalent variables are renamed distinctly and the unifying clause eq(X,Y) is appended to the body of the program for each required unification. The ground definition of the unifying clause is eq(X,X). Being non-left linear itself any transformation for it must be constructed by a different means that the transformation. Given

all_the_same(X,X,X) as input we return

all_the_same(X,Y,Z):- eq(X,Y),eq(Y,Z).

Implementation of this procedure requires special consideration as the data structure consists of a list of variables and recursive application is necessary for functions. The non-unifying ==/2 has to be used to test if two variables are the same without unifying them if one or both are variables.

9.4 Unrestricted Clause Negation

Ground Clause Negation

Each argument of the ground clause has its set theoretic complement calculated according to the following design based on the implementation design of [Bar90]

1. Obtain clause from input list
2. Access clause arguments 1,...,K where clause has K arguments.
3. For each argument i call the set complement function and assign /i to each individual complement in the resulting list.
4. For each complement Cdash of original term C in position /j , Cdash is inserted into position /j and all other arguments of the clause are replaced by fresh and distinct variables.[44]

[44] All constant permutations may be inserted into the variable position for a complement clause so that non-ground querying is productive. This is discussed later in the section.

Defining Clause Negation

Defining clauses are negated by applying negC to their heads as for ground clause. They are also negated for the literals in their body from which disjunctive solutions are obtained. It is in blindly forming the disjunctive clauses from an original conjunction that failure due to an incompatible conjunction is lost and hence the need for SLDN resolution.

Without Local Variables

The conjunction of literals in the body of a clause are simply replaced by a disjunction of clauses corresponding to each literal in the body. From

q(X,Y):- p(X),r(X,Y) we attain

$\overline{q}(X,Y) :- \overline{p}(X)$

$\overline{q}(X,Y) :- \overline{r}(X,Y)$

With local variables

If we refer to the completed transformation, negation of existential variables, those that appear only in clause bodies, create universally quantified variables. Therefore from the clause

q(X):- p(X,Y),r(Y) we attain

$\overline{q}(X) :- \forall \Psi \overline{\pi}(\Xi, \Psi)$

$\overline{\theta}(\Xi) :- \forall \Psi \overline{\rho}(\Psi)$

as the transformed answer. Any variables not in the clause head will be universally quantified. Such mechanisms may not always be conjuncted together in the unification process and therefore the universal quantifier is not explicitly true across a disjunction. This is essentially too weak as solutions failing from incompatibility of the disjunction may be lost. The answer is to form a predicate p2(X,Y) defined as

p2(X,Y):- p(X,Y),r(Y) and re-create the program as

q(X):- p2(X,Y) to obtain from transformation

$\overline{q}(X) :- \forall \Psi \overline{\pi 2}(\Xi, \Psi)$.

9.5 Negative Program Generation

A whole set of program clauses is obtained by performing the clause negation function on each clause in the program. Clauses that form the set defining predicate **p** are grouped into structures so that each structure contains the negated predicates obtained from a specific clause.

The unification algorithm is then applied in the following way. Each predicate in a structure is attempted to be unified with one predicate from all of the other structures that define one predicate **p**. Every separate structure correlates to the complements reached by applying negC to one clause. The result of this successful unification then has the same procedure applied to it. This is repeated for all clauses in the program.

Unification Algorithm Implementation

The unification algorithm is an extension of a simple unification procedure in [Ster86]. This includes the **occur check** preventing the unification of variables with terms containing instance(s) of the variable(s) achieved using the meta-logical '=='/2 and '\=='/2.

X == Y succeeds iff X and Y are identical constants, variables or functors (to whose arguments it is applied recursively. The predicate **not_occurs_in/2** succeeds if the occur check succeeds for its two arguments.

Unification of the system is equivalent to [Bar90]'s '@' operator. The first step is to gather the input program and separate each clause set which defines a predicate into its own structure.[55]

Every transformed predicate set CL(p) = $\{C_1,...,C_n\}$ which defines a clause transformation add to the output list every successful unification $R = R_j$ @ ... @ R_k where R_i is selected from CL(p). The input is processed so that this unification is attempted for all members C_i in CL(p) irrelevant of ordering due to the associativity of '@'.

An example is **CL(member) = {C1,C2}** where

C1 = [mem*(X,nil),mem*(X,[V|Vs]):- eq*(V,Vs)] and

C2 = [mem*(X,nil),mem*(X,[V|Vs]):- mem*(V,Vs)].

The successful unifications are

mem*(X,nil) @ mem*(X,nil) = mem*(X,nil) and

mem(X,[V|Vs]):- eq*(V,Vs) @ mem*(X,[V|Vs]):- mem*(V,Vs)

= mem*(X,[V|Vs]):- eq*(V,Vs),mem*(V,Vs).

5[5] It was not assumed that the user will automatically group all predicate definitions sequentially in the program. Therefore a parser for the whole input was implemented.

The unification predicate **eq** cannot be transformed using the method described (cf §5). This is due to its restricted nature which cannot be altered as it is a solution to the requirement of left-linearity. It must be created in an *ad-hoc* fashion. It is a simple procedure to wholly create an **eq*** predicate from the language provided by the user. Simply create an **eq*** clause for each different element given in the language. This must include for every constructor a disjunctive recursive call for all of its arguments. It is disjunctive as **eq*** is satisfied if just one argument pair is different. Suppose we have L = [a,f(_,_,_)] then CL(eq*) is

```
eq*(a,f(_,_,_)).
eq*(f(_,_,_),a).
eq*(f(X,Y,Z),f(X1,Y1,Z1)):-eq*(X,X1).
eq*(f(X,Y,Z),f(X1,Y1,Z1)):-eq*(Y,Y1).
eq*(f(X,Y,Z),f(X1,Y1,Z1)):-eq*(Z,Z1).
```

9.6 Ground Clause Unification Problems

The transformation approach's motivation is to allow non-ground querying of negative clauses. Unfortunately this behaviour may not occur when the language present the unification process with constants that do not occur in the original program. The 'set' operation with regard to uniqueness is indeterminate.

If L = [a,b,c] and program P, not defined as datalogic, is

```
r(b,b)          r(b,a)
```

the following negative program P* is returned using unification

```
r*(X,c)r*(a,a) r*(c,a) r*(c,Y)r*(c,b)
r*(c,c) r*(a,V)r*(a,b) r*(a,c)
```

r*(b,c) is absent as unification only creates r*(X,c) which 'covers' it according to Barbuti et al.'s algorithm. Clearly the set property of uniqueness has not been imposed on the output. This is because we could desire either

- r*(X,c) which succeeds for any query <-r*(Y,C) but returns no answer.[66] This style of success is similar to that of negation as failure so no real benefits are returned.
- to delete all non-ground clauses. In this case we only have r*(a,c) and r*(c,c) and so r*(b,c) would incorrectly fail.

6[6] This assumes that the clauses r*(a,c) and r*(c,c) have been filtered as they are 'covered' by r*(X,c). Unification does not create r*(b,c) as the transformation is based on the completion, those clauses that are in the original program.

- install a generation procedure for the non-ground clauses. Again there are two main possible solutions to this, which are:
    i. Place all the constants into the variables in the clauses and remove duplicates. This method is not the most efficient, particularly for a large database but it does ensure a ground database that will return answers for, what were originally, non-ground negative clauses. When there is more than one variable in the head all possible constant permutations must be inserted into the head. For the above program this is

        r*(b,c) r*(a,a) r*(c,a) r*(c,b) r*(a,b) r*(c,c) r*(a,c)

    ii. Remove clauses which are covered by non-ground cases. Turn each non-ground clause into a predicate which will instantiate the non-ground predicates in the head via Herbrand generation procedures. Values do not then have to be generated until the user presents the program with a relevant query. For the above, this would correspond to

        r*(X,c):- herbrand(X)
        r*(c,X):- herbrand(X)
        r*(a,V):- herbrand(X)

The latter is preferable if many answers to non-ground queries will be needed. For the purposes of forming the transformation every time the language is modified the Herbrand generator has to be modified as well. Under this circumstance, the former was implemented as it inherently possesses more flexibility as it is a one-step, albeit rather large, process. The result of transformation is post-processed after the unification stage. All non-ground clauses have all constant permutations inserted in to them and a filter mechanism removes duplicates from the database.

These procedures are useful within the realms of deductive databases, (cf. §3) particularly for integrity constraint checking where the negative part of the program could easily be called in a conjunctive manner. Suppose we had the integrity constraint for an employment database that every job has a wage. In first order logic this is,

$\forall V\,[\,job(V) \rightarrow \exists Vs\; wage(V,Vs)]$

and then the goal $\leftarrow job(V), \overline{wage}(V,Vs)$ could be run and success would correspond to failure of the integrity constraint.

§10
The Constraints Meta-Interpreter

10.1 The Naïve Constraints Processor

This naïve process was implemented at an early stage. It is intended to solve universal goals for database clauses which it does heavy-handedly achieve. Informally, the idea is that whenever a negative goal or subgoal is attempted a *solve_all/3* is called which finds all solutions to the goals for the variables and performs a logical negation on the result, theoretically similar to the theory of constructive negation only substantially less flexible. If the goal is explicitly universally quantified the set of solutions may be processed in a similar methods to Wallace's algorithm except that the search is performed initially, hence the naïveté. The **solve_all/3** predicate is presented for its effectiveness:

```
solve_all(Goal,GoalStruct,Xs):-
        asserta(instance(mark)),
        list_call(Goal),
        varcheck(GoalStruct,Sols),
        asserta(instance(Sols)),
        fail.

solve_all(Goal,GoalStruct,Xs):-
        varcheck(GoalStruct,Sols),
         retract(instance(Sols)),
        reap(Sols,Xs,[]),!.
```

The GoalStruct contains an instance of the desired solution structure. *reap/3* retracts the input argument returning an instantiated structure of solutions. *varcheck/2* places the variables from the input into a solution structure and *list_call/1* successively calls a list of subgoals so that the predicate can accept conjunction of clauses in a query. *solve_all/3* is a simple variant of *find_all/3* but it may be of use for meta-logical processing.

10.2 The Constraints Meta-Interpreter

A meta-interpreter for constraint processing was implemented upon the basis of Wallace's negation by constraints theories [Wal87]. The motivations for this were to provide another mechanism for solving the universally quantified predicate calculus formulae that often arise when transforming a logic program P containing local variables to its finite failure counterpart $\overline{P}$. The extension of constraints to handle universally quantified predicate calculus formulae is to deal with local variables in clauses that are to be negated, the same reasoning as for the transformation approach of [Bar90]. Universal evaluation works for database clauses under the simple intuition that a universal variable is satisfied if it holds for both a value and the negation of this value which encapsulates the Herbrand Universe in question.

The design of a constraints meta-interpreter relies on a recreation of Prolog's computational process. It extends Prolog by emulating Wallace's universal constraints algorithm and solves standard goals using constraint evaluation to direct the search in preference to the inbuilt backtracking of Prolog. Negation by Constraints procedures are only required for the evaluation of non-ground negative subgoals where Prolog would normally flounder. This dictates that the efficiency of the Prolog system is not penalised for the addition of constraints.

A Basic Meta-Interpreter

The simplest meta-interpreter that can be written in Prolog is the following:-

```
solve(true)
solve((A,B)):- solve(A),solve(B)
solve(A):- clause(A,B),solve(B)
```

It is the simplest meta-interpreter, sometimes referred to as the "Vanilla" meta-interpreter, and examined in greater depth in [Ster86]. The declarative interpretation reads: true is true, (A,B) is true if both A and B are true and A is true when A if B and B are true. A procedural interpretation of the clauses clearly displays how simulation of standard Prolog goal evaluation is achieved via this meta-interpreter. Note the import of **clause(A,B)** which unifies a clause head A with a corresponding body B from the program.

10.3 Standard Constraint Goal Evaluation

The constraints meta-interpreter requires extended clauses comparable to the above so that the constraints, in effect, direct the search. The general form for solving a goal

with constraints is an extension of the vanilla meta-interpreter. The basic clauses for standard goal evaluation are:

```
solve(true, _, true).

solve(Goal, Input_constraints1, Result_constraints):-
        copy_term(Goal, Goal_Clone),
        clause(Goal_Clone, true),
        obtain_constraints(Goal, Goal_Clone, Cons),
        reconcile(Input_constraints1, Cons, Result_constraints).

solve(Goal, Input_constraints1, Result_constraints):-
        clause(Goal, GoalBody),
        GoalBody \= true,
        separate(GoalBody, Body_Clauses, Body_Cons),
        reconcile(Input_constraints1, Body_Constraints, Partial_res_constraints),
        solve(Body_Cons, Partial_res_constraints, Result_constraints).

solve( (Goal1, Goal2), Input_constraints1, Result_constraints):-
        solve(Goal1, Input_constraints1, Partial_res_constraints),
        solve(Goal2, Partial_res_constraints, Result_constraints).
```

The primary divergence from the vanilla meta-interpreter is that two clauses are required; one for where the clause correlates to a ground database clause and the latter for where the clause is a non-ground predicate relation. The reason for two separate cases is to ease computation of both ground and non-ground relations. For the former a simple call to **clause(Goal, Body)** would instantiate all occurrences corresponding to $\mathbf{X \neq 1}$ and for the latter the body of the goal requires separation between further constraints in the body and calls to other literals.

obtain_constraints/3 takes the uninstantiated goal, the goal_clone and creates the required constraints set that corresponds to the variables in the goal. Even with a ground clause, success of this procedure is not guaranteed as the resulting constraints must satisfy the reconciliation procedure.

The **reconciliation/3** predicate fails if the two constraint sets are not satisfiable together. Its arguments consist of the goal's input constraints and the resulting constraints set of the goal once evaluated. Reconciliation essentially intersects the sets whilst ensuring satisfiability.

The second clause is called for non-ground relations where analysis of the goal body is required as it may not only contain literals but further constraints on the variables in question. All constraints in the body are extracted and an attempt at reconciliation is made with the initial input constraints. For example,

p(X,Y):- X ≠ 1, q(X,Y)

has **X ≠ 1** reconciled with input constraints to give a constraints set **R1** and **q(X,Y)** is then called under the initial constraints set of **R1**.

It can clearly be seen that standard goal evaluation under constraints is an extension of the most basic meta-interpreter and the procedural reading of both interpreters is very similar.

10.4 Constraint Specification

All constraints are either equalities and inequalities and Wallace in [Wal87] as with most other theoretical constraint papers uses X ≠ Y to depict inequality and X = Y to depict equality. Implementation difficulties emanate from this as nearly all Prolog systems treat = as unification and ≠ (written \=) as non-unification. Most Prolog systems additionally include '==' and '\==' for a representation of equality and inequality but they were not used for this particular implementation as it is safer to use new syntax for the meta-interpreter under situations where constraint clauses may receive a ground level reading. To exemplify, **X == 1** as a constraint would fail for an equality test that was never desired anyhow.

For simplicity, the following forms were used for implementation:

X = 1 is represented by eq(X,1) and

X ≠ 1 is represented by ineq(X,1).

As we can see from the operation of the meta-interpreter for standard goals, it is vital to ensure that variables given in the original goal do not become bound to any values as standard backtracking is not relied upon to do the 'dirty work' of unbinding variables. This is performed throughout operation of the meta-interpreter using the following techniques:

- copy_term/2, which recreates any input with new variables, is called before instantiating predicates with their ground values so that we do not have to process constraint sets of the form [eq(3,1)] unless receiving partial ground input which is dealt with inside reconciliation.
- avoidance of the possibly instantiating **member** test which cannot be used for variable evaluation. A simple variant which does not lead to any unwanted bindings was created.

Inclusion of Predicate Calculus Formulae

Arbitrary predicate calculus formulae have to be evaluated as negation by constraints is attempting to retrieve answers from negated literals. These literals may contain local variables and therefore their negations are no longer in clausal form. Any mechanism attempting to solve negated literals requires extensions to accept and process universally quantified goals. With regard to the meta-interpreter created, it was implemented to solve the universal subgoals that occur during the transformation process. For the program

p(X):- ~q(X,Y)

the query ?p(V) reduces to $\leftarrow \exists Y\, q(X,Y)$ which is equivalent to $\forall Y \neg q(X,Y)$ and such a formula requires special attention.

Example Solutions

The program clause set is:

p(a,b) **p(a,c)**

and for a goal **?p(a,Y),ineq(Y,c)** where the initial constraint set is not null the only solution is **{eq(Y,b)}** as the initial constraint prevents the second solution of **{eq(Y,c)}**.

Answers to negative goal solutions inherently rely on the focussing method introduced in §3. Suppose we have query **?~p(X,Y)** for the following program:

p(X,b):- q(X)
q(1)
q(2)

then {ineq(Y,b)} is an immediate solution due to the focussing mechanism that does not need all possible solutions to be evaluated before returning an answer set. A naïve constraints processor would require the evaluation of all q/1 ground clauses. In database terms, such focussing could save a great deal of redundancy incurred by straightforward negation. Further solutions to the above query are {ineq(Y,1), ineq(Y,2)} to which De Morgan's Laws are applied over a disjunction with the first solution.

Often there will be a situation where partial non-ground negative literals need to be solved. The constraints meta-interpreter is called into operation for this. As an aside, many negative goals would not necessarily flounder with Prolog's unsafe computation rule and occasionally the desired affirmation answer will be returned. This behaviour cannot be relied upon and it is safer to use the constraints meta-interpreter. If the query ?~bridge(a,Y) is posed to the database:

bridge(b,c)

bridge(d,e)

then success is returned with no constraints on variable Y. The meta-interpreter deals with such queries by referring to a value check that examines the constraints themselves and reconcilation with each database clause will fail for **a and b** and then **a and d** so the goal succeeds without constraints on Y. These partial non-ground clauses are not explicitly mentioned by Wallace in [Wal87] but his constraints interpreter will deal with them succinctly as it incorporates directly in its resolution mechanism both equality (=) and inequality (≠).

Existential Goal Evaluation

Existential goals are solved simply by solving as a standard goal and dropping the constraint on the existential variable. The intuition behind this is clear; an existential goal is satisfied if at least one instance of it is satisfied regardless of its actual value.

10.5 Universal Goal Evaluation

Universal Goal Evaluation is a straightforward implementation of Wallace's algorithm that is presented in §3. The meta-interpreter code is presented:

```
solve(X forall Goal, Input_constraints, Result_cons):-
        solve(Goal, Init_cons, Res_cons1),
        obtain_Q_cons(X, Res_cons1, Quant_cons,Rc2),
        obtain_Q_cons(X, Input_constraints, Quant_cons2, Icons),
        not equiv(Quant_cons, Quant_cons2),
        negate(Quant_cons, Neg_Quant_cons),
        append(Rc2, Neg_Quant_cons, Conslist),
        solve(X forall Goal, Conslist, Result_cons).

solve(X forall Goal, Init_cons, Result_constraints):-
        solve(Goal, Init_cons, Res_cons1),
        obtain_Q_cons(X, Res_cons1, XCons, Rc2),
        obtain_Q_cons(X, Init_cons, XC2, Icons),
        equiv(XCons, XC2).
```

obtain_Q_cons/4 takes the quantified variable and the given constraints set and separates the constraints from the universally quantified variable constraints. **negate/2** clearly negates a quantified constraint set that is to be provided as input to the system so that the whole value domain will be covered if successful for these negated constraints.

equiv/2 is given the constraints for the universally quantified variable and tests to see if they are equivalent. This corresponds to success in the algorithm where the input and output constraints for the universally quantified variable are the same for the input constraints are last goals with the universal variable constraint negated.

It is clear that the constraints solution technique is highly useful. The meta-interpreter shown here is neither efficient nor particularly robust. Full integration with a Prolog system is the next step together with extension to process a wider class of program.

§11
Further Work

This section covers extensions for the techniques studied here. Some are basic extensions whilst others will not come to fruition until theoretical issues at the heart of logic programming are tackled, namely the divide between logic programming theory and an average application implemented in Prolog. Other languages, such as Gödel, are being constructed with much sounder declarative semantics and a completely fresh start may be the only way to improve on the current implementations of logic languages.

11.1 Negation By Constraints

The motivation for implementing constraints is to remove the floundering problem that occurs when a non-ground negative goal is processed. As a side-effect of the negation of subgoals containing local variables the handling of arbitrary predicate calculus formulae is needed which includes universally quantified variables. The intuition behind the processing of unviersal quantification works for a potentially infinite domain. The next step for this is, as Wallace notes, full integration with a Prolog interpreter at the unification stagae. This step would improve efficiency as well as making the constraint processing innate at the variable binding stage rather than a check after it has occurred.

The major drawback of the NBC approach is that it only works for database program sets i.e. any program without functions. The following clause,

p(0)
p(f(X)):- p(X)

cannot be solved under NBC as the goal ? $\forall X\ p(X)$ has an infinite branch in the search tree that will lead the constraints processor into an infinite loop that cannot be expressed by constraints. Another feature of the constraints system is that they can only involve the finite set of constants present in the program or database implying that no non-termination can occur through constraint processing. This dictates, as the above program displays, only the program structures in goal evaluation can lead to

infinite evaluation. For real applications outside of databases the constraints evaluator must be expanded to deal with a number of program properties.

Inductive solutions may be applicable for evaluating universally quantified goals as the well founded nature of the Herbrand Universe can be exploited and the base cases solved almost as for any goal in the constraints mechanism and inductive cases would be proved by a previous assertion of the induction hypothesis (depending on the goal's nature) as constraints which are resorted to when the recursive call is being processed. A combination of such an approach with constraints might be very profitable yet it requires more work at present.

The handling of arbitrary predicate calculus formulae needs expanding for successful management of compound expressions so that a larger class of goals or subgoals may be directly evaluated.

11.2 The Transformational Approach

The main requirements for further work on the transformational approach are:-

- an extension to successfully cope with negative subgoals which are present in many programs and
- a provision of disparate options to evaluate universal subgoals that occur during transformation of subgoals with local variables including some or all of NBC, CNF, Induction, Natural Deduction and the process of Barbuti et. al., namely SLDN-resolution.

SLDN-resolution as introduced in [Bar90] for universal subgoal evaluation has a number of deficiencies. Unless the universally quantified subgoal which is transformed into a generate and test form is **allowed** then a Herbrand generator is required.

Herbrand generators need further evolution than what [Bar90] present. For an infinite Herbrand Universe the generator is itself infinite and so the user must be careful not to use standard NAF on a transformed program $\overline{p}$ where the original program p contains local variable within subgoals. If the universal subgoal generates an incorrect candidate solution a standard computation rule would backtrack to the Herbrand generator that has an infinite branch with an infinite number of finite sub-branches. Therefore a 'cut' feature is required solely for the Herbrand generator. The basic method given by Barbuti is a relatively naïve procedure and needs amplification.

Transformation is only successful for clauses which fail finitely in the original program. The problem of infinite failure is not infringed by the transformation process. Ideally programs should be created to satisfy the hierarchical constraint so that the infinite failure set of a program is null. Theoretically the transformed program will then have an answer set which can be understood under the Closed World Assumption to which it is equivalent in this situation.

As this report has displayed when a language is specified (a requirement in the implemented transformation process) to contain additional constants and/or function symbols than what is present in the program, the transformation implementation may not present all finite failure clauses as programs are transformed according to their completions that relate to clauses instances of the program. This vague area needs to be formalised in the specification provided by the user - Should the language for transformation be restricted to constants and functions appearing solely in the program? The datalogic extension detailed in §9 and implemented here creates all clauses based on the LANGUAGE given. This allows answer substitutions to be returned from non-ground queries using language items which may not be in the original program though the user has specified them as members of the language.

[Ng90] introduces a variation of the transformation theory which transforms programs for partial evaluation by specialising a transformation to a particular query. The next step is to form an implementation of this. Practical purposes of this *partial transformation* are various optimisation techniques and also to aid the generation of "stopping" conditions used in standard partial evaluation. Partial evaluation techniques themselves can benefit from examination of a program transformed in this regard.

Obviously, one of the more definite extensions of transformation is the inclusion of 'impure' Prolog programs which cause significant complications. These features such as the cut(!) and other meta-logical control primitives reside in the chasm outside of the theory of logic programming. Once these 'impurities' are ironed out or strengthened with a theoretical basis then transformation may be applicable to a far greater class of program.

Efficiency of transformation is a key factor. Optimisation of the transformation theory is required and as such has not been researched. For the ultimate aims of Barbuti et al., presented in [Bar90] are

- the definition of a functional meta-language for logic programming
- equipping functional languages with a logic data type

and the recognition of these desires depends to a large extent on efficiency enhancements. A possible objective is the integration of program compilation with transformation.

11.3 Induction

As described in §7, inductive methods are a viable mode of proving program properties. §7 describes the formal application of induction to prove program properties and this area alone may form many future projects.

In creating $\overline{p}$, the negation of program p, the instances of universally quantified subgoals tend to be far simpler than those properties required to prove in [Kan87]. Such was not pursued in this project due to focussing on other areas and time restrictions.

11.4 Constructive Negation

Constructive negation is relevant for all classes of program unlike NBC's limitation to function free programs. Chan in [Cha89] notes that the constructive negation described makes the assumption that all inequalities in answers are ground and this frequently this will not be the case. The CNF method is a useful tool but, like the other mechanisms, needs integration with a Prolog system. As section 3 pointed out, the procedure for solving universally quantified goals or subgoals requires refinement and clearer documentation.

From the issues presented in this section there are more than a few avenues open to future research. Most of those detailed will, if implemented, be of great use to the standard logic programming environment. It seems that the reasons for all of these required extensions are inabilities existent within Prolog implementations due to the divide between logic programming theory and practice. It is probable that Prolog will be superseded by a new logic programming model which will either not demand or inherently include such extensions.

§12 Conclusion

Preliminary Definitions

SLD	Defined as Linear resolution with selection function for Definite programs, it is the refutation procedure used by Prolog for its resolution inference rule.
Parlog	A parallel logic programming language that has the capacity for AND and OR parallelism based on committed choice non-determism.

The relationship between the transformation of a logic program and its evaluation under negation as failure can be stated as:

If $\overline{p}(t)$ succeeds for SLDN-resolution with answer substitution θ then $\leftarrow p(t\theta)$ can be inferred under NAF.

This relationship displays the primary advantage of creating the negative program $\overline{p}$ from the original program p - answer substitutions may be returned from what we deem to be the *negative* program. Negation as Failure is unable to achieve this as its success for query ?~A relies on the inability to logically infer A from the program p. [Shep85] notes the compatibility of NAF with a number of reference theories and notes, '...this raises the question of exactly what its logical status is.' Theoretically NAF may not rest on the strongest of structures yet there can be no question as to its emminent suitability for using the Prolog resolution mechanism, **SLD**, to great effect and its ease of implementation within a computational system. There can be no other contender for an implementation of negation with regard to computational efficiency.

The Transformational approach implemented here is of use in a number of logic programming fields. Barbuti, Mancarella, Pedreschi and Turini refer to it in [Bar87] as, ' a notable extension to the expressive power of logic programming.' There is no doubt it could prove to be very useful in such areas as knowledge engineering, knowledge acquisition and deductive databases. Applications in parallel logic languages such as **Parlog** are as yet unnoticed but an environment where variables can be returned from negative clauses does increase the language's flexibility. An example of this is the forming of communication channels where there can now be

negative clauses containing shared variables knowing that answers will not suspend if they are unbound and occur negatively.

As discussed in section 11, efficiency issues need much work before any transformation methods become more widely used. Examination of the procedures required in the implemented transformation system indicate how inefficient the present mechanism is for a reasonable program or database. The creation of negative programs also requires extensions to handle negative subgoals and, possibly, meta-logical primitives.

Negation By Constraints and Constructive Negation methods were examined for their ability to handle universally quantified subgoals which arise in transformation of negative literals with local variables. The most direct solution to this problem is to avoid using local variables. Constructive Negation is exemplified by its simple theoretical negation of the completion and a full-integration with a Prolog system would be advantageous for many implementations.

A constraints meta-interpreter was implemented to deal with such universal goals that may occur from transformation. It solves universal goals for datalogic following the algorithm presented by Wallace in [Wal87] yet is rather inefficient in its search control methods. Possible improvements are examined in section 11 of which the most useful would be an extension to include processing the whole class of Prolog programs. This may require inductive techniques or Natural Deduction and the meshing with constraints could ease any implementational problems.

Given systems which contain the extensions outlined in this report, once sufficiently optimised, then John Lloyd's statement, 'Logical Inference is about to become the fundamental use of computation,' may not seem so very extravagant.

Appendices

A1

Transformation User Interface

The User Interface has the following front-end.

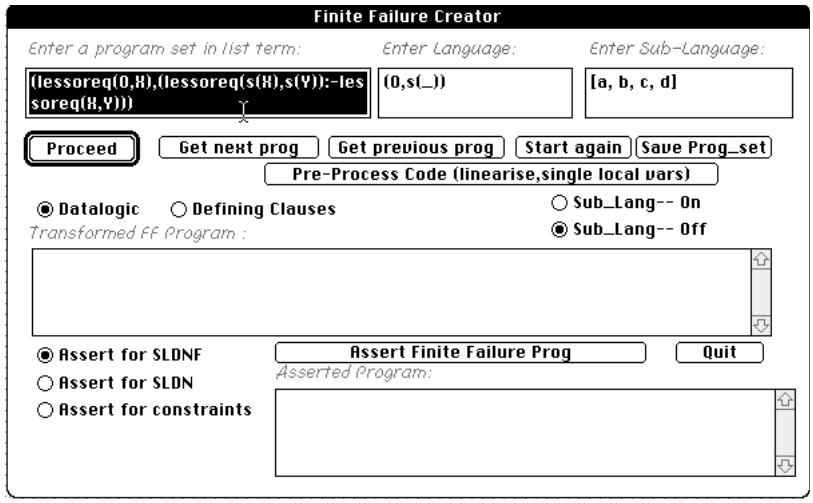


Figure A-1. The Transformation Screen Display

The window is now shown as it appears during operation. The program clause sets are dispayed in the appropriate windows.

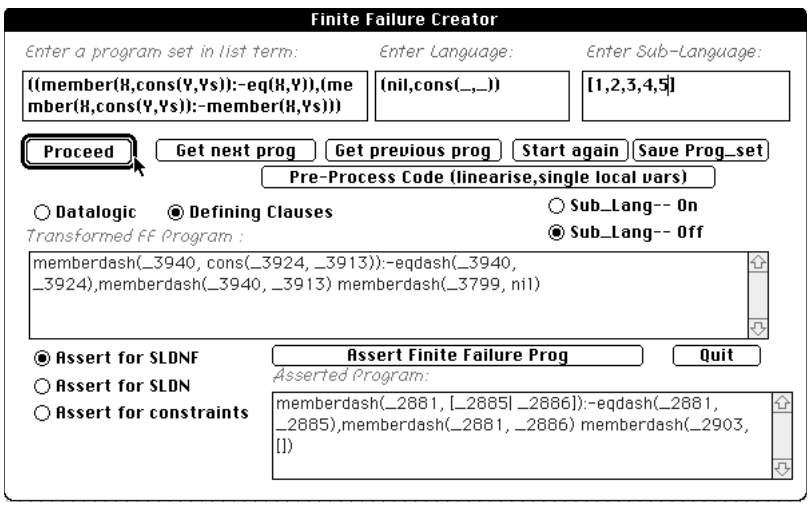


Figure A-2. The Transformation Display in operation

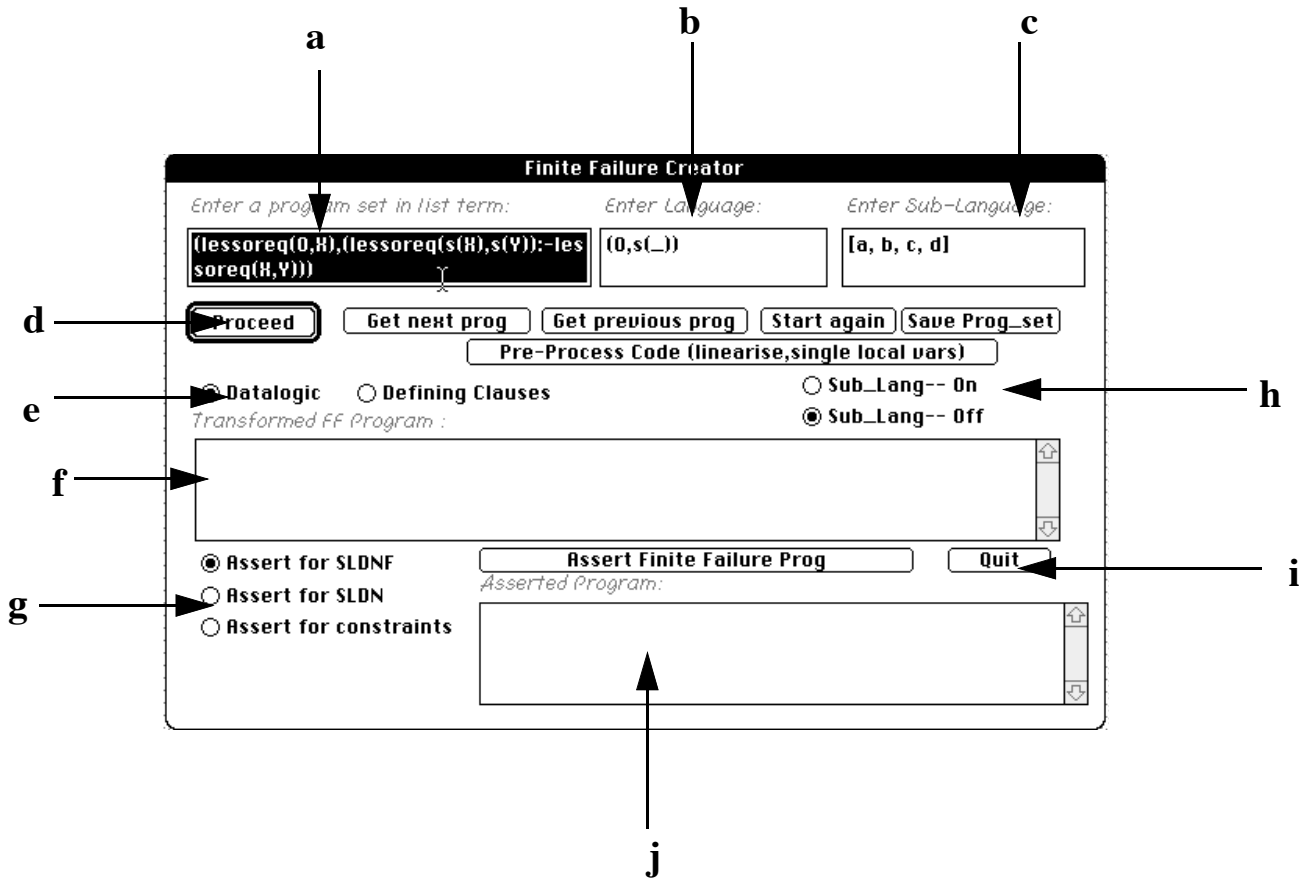


Figure A-3 Transformation Display Guide

The following utilities are available on the **'Finite Failure Creator'.**

Entering a program (a, b & c)

- **Clauses** - Clauses are entered as a complete term with different clause instances separated by commas and the whole structure needs to be placed within brackets so that it may by stored and accessed by the program for future use.
- **Language** - The language is entered as for clauses.
- **Sub-Language**- The sub-language is entered as for clauses.

Entering a non left-linear program

- **Clause conversion** - Non-left linear programs may be entered so that they can be pre-processed into a left-linear form with the required **eq** clauses inserted into program bodies.

Clause Storing

- **Save (program clause set)** - Once a program structure has been entered it may be stored. This places the program in a data window and allows the user to see the clauses if so desired.
- **Get Next (clause)** - Each program and language set is stored with a key (integer) and this is used to select any clause in the data window for transformation.

- **Get Previous (clause)** - The user has the ability to search the stored program structures by clicking on **Get Next** and **Get Previous**.
- **Start Again** - The first clause in the database is displayed.

Assertion of Transformed clauses (g & j)

The program set, once transformed, may be asserted by the user for future use. This also provides the function of transforming from the notation of **nil** and **cons(_,_)** to the more standard **[]** and **[X|Xs]** if the program is selected as a **defining clause set** and uses this format.

- **SLDN** - The local variable is solved by creating SLDN clauses which solve the universally quantified variable problem in a generate and test mode (cf. §4 ,§7).
- **SLDNF** - Nothing is changed under the assumption that there are no local variables and so SLDNF is satisfactory resolution mechanism.
- **Constraints** - The universal quantified variables are solved by the constraints meta-interpreter (cf. §8) and so the 'forall' predicate can remain in the code.

Transforming a Clause (d with e & h - display in f)

- **Datalogic** - The program set consists of database clauses and so all clauses are instantiated to ground terms for successful negative non-ground querying.
- **Defining Clause** - The defining clause set are the produce of unification and variables are intentional.

and

- **Sub-Language: On** - The sub-language as given by the user is used to instantiate further ground clauses for successful querying in a language that is 'outside' of the **Herbrand Universe** of the program set. These terms would fail for the original program set and so they should theoretically succeed for the transform. This is of use for all possible query input.
- **Sub-Language: Off** - The sub-language is ignored and the program only uses the language for transformation.

Exiting the System (i)

- **Quit** The System is exited after clicking this button.

A2
References

[Apt94] Apt, K.R. and Bol, R.N. , "Logic Programming and Negation: A Survey", *Journal of Logic Programming* 20, 1994, 9 - 71.

[Bal93] Balbiani, P. , "Modal Logic and Negation as Failure", *Journal of Automated Theorem Proving,* 1993, 331 - 356

[Bar87] Barbuti, R., Mancarella, P., Pedreschi, D. and Turini, F., "Intensional Negation of Logic Programs: examples and implementation techniques" , *Proceedings of International Joint Conference on Theory and Practice of Software Development*, TAPSOFT '87, Lecture Notes in Computer Science 250, 1987, 96 - 110.

[Bar90] Barbuti, R., Mancarella, P., Pedreschi, D. and Turini, F., " A Transformational Approach to Negation in Logic Programming." *The Journal of Logic Programming*, Vol. 13, 1990.

[Bru89] Bruffaerts, A. and Henin, E., " Negation as Failure: Proofs, Inference Rules and Meta-interpreters", in Abramson, H. and Rogers, M., editors in *Proceedings of the Workshop on Meta-programming in Logic Programming*, Bristol, 1989, 169 - 190.

[Cha88] Chan, D., "Constructive Negation Based on the Completed Database", in Kowalski, R. A. and Bowen, K.A., editors in *Logic Programming: Proceedings of the Fifth International Conference and Symposium,* Seattle, USA, 1988, 111 - 125.

[Cha89] Chan, D. and Wallace, M., " A Treatment of Negation during Partial Evaluation", in Abramson, H. and Rogers, M., editors in *Proceedings of the Workshop on Meta-programming in Logic Programming*, Bristol, 1989, 299 - 314.

[Dra91] Drabent,W. and Martelli,M., "Strict Completion of Logic Programs", *New Generation Computing*, 1991, 69 - 80.

[Hog90] Hogger, C.J. , Essentials of Logic Programming, Oxford University Press, Oxford, 1990

[Kan87] Kanamori, T. and Fujita, H. , "Formulation of Induction Formulas in Verification of Prolog Programs" ICOT Technical Report, TR - 094, 1987

[Llo87] Lloyd, J.W., Foundations of Logic Programming, Second, Extended Edition,

Springer Verlag, Berlin, 1987.

[Man88] Mancarella, P., Martini, S. and Pedreschi, D., "Complete Logic Programs with Domain-Closure Axiom", *Journal of Logic Programming* Vol. 5, 1988, 263 - 276.

[Ng90] Ng, B.T. and Broda, K., "Partially Evaluating the Completed Program (Revised)", *Technical Report,* Dept. of Computing, Imperial College.

[Sek93] Seki, H. , " Unfold/Fold Transformation of General Logic Programs for the Well-Founded Semantics", in *Journal of Logic Programming*, 1993, Vol. 16, 5 - 24.

[Ser90] Sergot, M. , "Negation in Logic Programs and Deductive Databases", *unpublished lecture notes*, Dept. of Computer Science, Imperial College, 1990.

[Shep84] Shepherdson, J.C., "Negation as Failure: A Comparison of Clark's Completed Data Base and Reiter's Closed World Assumption", *Journal of Logic Programming* Vol. 1, 1984, 51 - 79.

[Shep85] Shepherdson, J.C., "Negation as Failure II", *Journal of Logic Programming* Vol. 2, 1985, 185 - 202.

[Ster86] Sterling, L. and Shapiro, E. , The Art of Prolog: Advanced Programming Techniques, MIT Press, Massachusetts, 1986.

[Vor92] Voronkov, A. , "Logic Programming with Bounded Quantifiers", in *Lecture Notes in Artificial Intelligence 592: Proceedings of the Second Russian Conference in Logic Programming,* St. Petersburg, 1991, 486 - 514.

[Vas86] Vasey, P. , "Qualified Answers and Their Applications to Transformation", in *Lecture Notes in Computer Science 225: Proceedings of the Third International Conference on Logic Programming*, London, 1986, 425 - 432.

[Van89] Van Hentenryck, P. Constraint Satisfaction in Logic Programming. MIT Press, 1989.

A3
Naïve Reverse Transformation

The following is the prolog program for naive reversal:-

```
reverse([],[]).
reverse([X|L],M):-
        reverse(L,N),
        append(N,[X],M).
append([],X,X).
append([X|Y],V,[X|Z]):-
        append(Y,V,Z).
```

STEP 1.

The program is rewritten in left-linear form. This is achieved in the implementation using a simple linearisation pre-processing algorithm which inserts the unifying 'eq' predicate to ensure all clause heads are unrestricted.

```
reverse([],[]).
reverse([X|L],M):-
        reverse(L,N),
        append(N,[X],M).
append([],X,Y):- eq(X,Y).
append([X|Y],V,[Y|Z]):-
        eq(X,Y),
        append(Y,V,Z).
```

Together with the definition of 'eq':-

```
eq(X,X).
```

STEP 2.

Now we form the completed program:-

$$\forall X \forall Y \; \text{reverse}(X,Y) \Leftrightarrow (X = [] \wedge Y = []) \vee$$
$$\exists V \exists L \exists N \, (X = [V|L] \wedge \text{reverse}(L,N) \wedge \text{append}(N,[V],Y))$$
$$\forall X \forall Y \forall Z \; \text{append}(X,Y,Z) \Leftrightarrow (X = [] \wedge \text{eq}(Y,Z)) \vee$$
$$\exists V \exists V2 \exists Vs \exists Zs (X = [V|Vs] \wedge Z = [V2|Zs] \wedge \text{eq}(V,V2) \wedge \text{append}(Vs,Y,Zs))$$

The completed definition for 'eq' is:-

$$\forall X \forall Y \ (eq(X,Y) \Leftrightarrow X = Y)$$

The completion includes in this case the axioms which define the '=' predicate and the domain closure axiom which constrains the interpretation domain.

STEP 3.

We apply logical negation so that negative occurrences of axioms may be treated positively. This gives us:-

$$\forall X \forall Y \ \overline{reverse}(X,Y) \Leftrightarrow \neg (X = [] \wedge Y = []) \wedge$$
$$\neg \exists V \exists L \exists N \ (X = [V|L] \wedge reverse(L,N) \wedge append(N,[V],Y))$$
$$\forall X \forall Y \forall Z \ \overline{append}(X,Y,Z) \Leftrightarrow \neg (X = [] \wedge eq(Y,Z)) \wedge$$
$$\neg \exists V \exists V2 \exists Vs \exists Zs (X = [V|Vs] \wedge Z = [V2|Zs] \wedge eq(V,V2) \wedge append(Vs,Y,Zs))$$

STEP 3. Part 2

The negation operators are applied for further transformation.

$$\forall X \forall Y \ \overline{reverse}(X,Y) \Leftrightarrow (X \neq [] \vee Y \neq []) \wedge$$
$$\forall V \forall L \forall N \ (X \neq [V|L] \vee \overline{reverse}(L,N) \vee \overline{append}(N,[V],Y))$$
$$\forall X \forall Y \forall Z \ \overline{append}(X,Y,Z) \Leftrightarrow (X \neq [] \vee \overline{eq}(Y,Z)) \wedge$$
$$\forall V \forall V2 \forall Vs \forall Zs \ (X \neq [V|Vs] \vee Z \neq [V2|Zs] \vee \overline{eq}(V,V2) \vee \overline{append}(Vs,Y,Zs))$$

STEP 4.

We continue the transformation using the equivalence lemma in [Cha89].

$$\forall X \forall Y \ \overline{reverse}(X,Y) \Leftrightarrow$$
$$(X \neq [] \wedge \forall V \forall L \forall N \ (X \neq [V|L] \vee \overline{reverse}(L,N) \vee \overline{append}(N,[V],Y))) \vee$$
$$(Y \neq [] \wedge \forall V \forall L \forall N \ (X \neq [V|L] \vee \overline{reverse}(L,N) \vee \overline{append}(N,[V],Y)))$$
$$\forall X \forall Y \forall Z \ \overline{append}(X,Y,Z) \Leftrightarrow$$
$$\left(\begin{array}{l} X \neq [] \wedge \\ \forall V \forall V2 \forall Vs \forall Zs \ (X \neq [V|Vs] \vee Z \neq [V2|Zs] \vee \overline{eq}(V,V2) \vee \overline{append}(Vs,Y,Zs)) \end{array}\right) \vee$$
$$\left(\begin{array}{l} X = [] \wedge \overline{eq}(Y,Z) \wedge \\ \forall V \forall V2 \forall Vs \forall Zs \ (X \neq [V|Vs] \vee Z \neq [V2|Zs] \vee \overline{eq}(V,V2) \vee \overline{append}(Vs,Y,Zs)) \end{array}\right)$$

Continuing the transformation with ^-distribution and application of the lemma.

$\forall X \forall Y\ \overline{\text{reverse}}(X,Y) \Leftrightarrow$

$(X \neq [] \wedge \forall V \forall L \forall N\ X \neq [V|L]\ ) \vee$

$(X \neq [] \wedge \exists V \exists L \forall N\ (X = [V|L] \wedge \overline{\text{reverse}}(L,N) \vee \overline{\text{append}}(N,[V],Y))\ ) \vee$

$(X = [] \wedge Y \neq [] \wedge \forall V \forall L \forall N\ (X \neq [V|L] \vee \overline{\text{reverse}}(L,N) \vee \overline{\text{append}}(N,[V],Y))) \vee$

$(X = [] \wedge Y \neq [] \wedge \forall V \forall L \forall N\ X = [V|L] \wedge \overline{\text{reverse}}(L,N)) \vee$

$(X = [] \wedge Y \neq [] \wedge \forall V \forall L \forall N\ X = [V|L] \wedge \overline{\text{append}}(N,[V],Y))$

$\forall X \forall Y \forall Z\ \overline{\text{append}}(X,Y,Z) \Leftrightarrow$

$(X \neq [] \wedge \forall V \forall V2\ X \neq [V|Vs]) \vee$

$(X \neq [] \wedge \forall V1 \forall Zs \exists V \exists Vs\ X = [V|Vs] \wedge Z \neq [V1|Zs]) \vee$

$(X \neq [] \wedge \exists V1 \exists Zs \exists V \exists Vs\ X = [V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{eq}}(V, V1)) \vee$

$(X \neq [] \wedge \exists V1 \exists Zs \exists V \exists Vs\ X = [V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{append}}(Vs, Y, Zs)) \vee$

$(X = [] \wedge \overline{\text{eq}}(Y, Z) \wedge \forall V \forall V2\ X \neq [V|Vs]) \vee$

$(X = [] \wedge \overline{\text{eq}}(Y, Z) \wedge \forall V1 \forall Zs \exists V \exists Vs\ X = [V|Vs] \wedge Z \neq [V1|Zs]) \vee$

$(X = [] \wedge \overline{\text{eq}}(Y, Z) \wedge \exists V1 \exists Zs \exists V \exists Vs\ X = [V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{eq}}(V, V1)) \vee$

$(X = [] \wedge \overline{\text{eq}}(Y, Z) \wedge \exists V1 \exists Zs \exists V \exists Vs\ X = [V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{append}}(Vs, Y, Zs))$

STEP 5.

We now apply the set theoretic complements to the negative guards. It relies on the theory provided by the DCA.

$\forall X \forall Y\ \overline{\text{reverse}}(X,Y) \Leftrightarrow$

$(X = \exists V \exists Vs\,[V|Vs] \wedge \forall N \exists V \exists L\,(X = [V|L] \wedge \overline{\text{reverse}}(L,N) \vee \overline{\text{append}}(N,[V],Y))) \vee$

$(X = \exists V \exists Vs\,[V|Vs] \wedge \exists V \exists L \forall N X = [V|L] \wedge) \vee$

$(X = [] \wedge Y = \exists V \exists Vs\,[V|Vs] \wedge (X = [])) \vee$

$(X = [] \wedge Y = \exists V \exists Vs\,[V|Vs] \wedge \forall V \forall L \forall N\, X = [V|L] \wedge \overline{\text{reverse}}(L,N)) \vee$

$(X = [] \wedge Y = \exists V \exists Vs\,[V|Vs] \wedge \forall V \forall L \forall N\, X = [V|L] \wedge \overline{\text{append}}(N,[V],Y))$

$\forall X \forall Y \forall Z\ \overline{\text{append}}(X,Y,Z) \Leftrightarrow$

$(X = \exists V \exists Vs\,[V|Vs] \wedge X = []) \vee$

$(X = \exists V \exists Vs\,[V|Vs] \wedge \forall V1 \forall Zs \exists V \exists Vs\, X = [V|Vs] \wedge Z = [\,]) \vee$

$(X = \exists V \exists Vs\,[V|Vs] \wedge \exists V1 \exists Zs \exists V \exists Vs\, X = [V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{eq}}(V, V1)) \vee$

$(X = \exists V \exists Vs\,[V|Vs] \wedge \exists V1 \exists Zs \exists V \exists Vs\, X = [V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{append}}(Vs,Y,Zs)) \vee$

$(X = [] \wedge \overline{\text{eq}}(Y,Z) \wedge X = []) \vee$

$(X = [] \wedge \overline{\text{eq}}(Y,Z) \wedge \forall V1 \forall Zs \exists V \exists Vs\, X = [V|Vs] \wedge Z = []) \vee$

$(X = [] \wedge \overline{\text{eq}}(Y,Z) \wedge \exists V1 \exists Zs \exists V \exists Vs\, X = [V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{eq}}(V, V1)) \vee$

$(X = [] \wedge \overline{\text{eq}}(Y,Z) \wedge \exists V1 \exists Zs \exists V \exists Vs\, X = [V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{append}}(Vs,Y,Zs))$

STEP 6.

We turn conjunctions of formulae for the same variable with positive guards into single formula by finding the most general unifiers. The result of this process is the completed definition of the desired negative predicate.

$\forall X \forall Y\ \overline{\text{reverse}}(X,Y) \Leftrightarrow$

$(\forall N\,[X = \exists V \exists L\,[V|L] \wedge \overline{\text{reverse}}(L,N) \vee \overline{\text{append}}(N,[V],Y]) \vee$

$(X = [] \wedge Y = \exists V \exists Vs\,[V|Vs])$

$\forall X \forall Y \forall Z\ \overline{\text{append}}(X,Y,Z) \Leftrightarrow$

$(X = \exists V \exists Vs\,[V|Vs] \wedge Z = []) \vee$

$(X = \exists V \exists Vs\,[V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{eq}}(V, V1)) \vee$

$(X = \exists V \exists Vs\,[V|Vs] \wedge Z = [V1|Zs] \wedge \overline{\text{append}}(Vs,Y,Zs)) \vee$

$(X = [] \wedge \overline{\text{eq}}(Y,Z))$

STEP 7.

It is now possible to extract the logic program from the completed definition. The final program for the negation of naïve reverse is:-

```
reverse([],[Y|Ys]).
reverse([X|Xs],Ys):- ∀N π([Ξ|Ξσ],Ψς,N).
π([Ξ|Ξσ],Ψς,N):- ρεϖερφεσ(Ξσ,N).
π([Ξ|Ξσ],Ψς,N):- απενδ(N,[Ξ],Ψς).
απενδ([ς|ςσ],Ψ,[]).
απενδ([],Ξ,Ψ):- εθ(Ξ,Ψ).
απενδ([ς|ςσ],Ψ,[Ξ|Ξσ]):- εθ(ς,Ξ).
απενδ([ς|ςσ],Ψ,[Ξ|Ξσ]):- απενδ(ςσ,Ψ,Ξσ).
```

The definition of $\overline{\text{eq}}$ has to be created in an ad hoc fashion as detailed in §4. Its clausal definition is satisfied whenever the two arguments are different.

A4
Transformation Code Listing

Notation - Each clause is commented using the following prefixes

? - input term

^ - output term

so that, for example, sumvars(?Varlist,^Num) denotes that Varlist is an input term and Num an output term.

```
/*      comp(?Term,?Language,?Language,^Complement_set):-
        Creates the Complement_set using the given Language for the
        input Term. It directly implements the Not algorithm of [Bar90] */

comp(Item,[Y|Tail],L,Tail):-        % includes integers in constant test.
        constant(Item),
        Item == Y.

comp(Item,[T|Tail],L,[T|Tail2]):-
        constant(Item),
        Item \= T,
        comp(Item,Tail,L,Tail2).

comp(Item,Tails,L,[]):-
        var(Item).

comp(Item,[Y|Tail],L,Result):-
        not var(Item),
        not constant(Item),
        Item =.. [Fname|FArgs],
        checkit(Item,Y),
        numstruct(FArgs,1,Argnum),
        list_comp(Argnum,L,CompArgs),
        length(FArgs,Flen),
        process(Fname,CompArgs,Flen,Result).  % apply complement to each argument

comp(Item,[T|Tail],L,[T|Tail2]):-
        not var(Item),
        Item =.. [Fname|FArgs],
        comp(Item,Tail,L,Tail2).

/* checkit(?Function1,?Function2):-
         Holds if Function1 and Function2 have the same name and
         number of arguments regardless of any variable values   */

checkit(X,Y):-
        Y =.. [Fname|Fargs],
        X =.. [F2|Fargs2],
        Fname == F2,
        length(Fargs,T),
```

```
        length(Fargs2,T).

/*      list_comp(?Arguments,?Language,^Term_List):-
        Takes the Arguments and for each member forms the complement
        using the language provided and places each elements in Term_List.
        All input elements in Arguments are postfixed with a number (/Num) and
        all output in Term_List corresponding to the complement of a term has
        the same number postfixed. This is for placing in the result easily. */

list_comp([],_,[]).
list_comp([X/N|T],Lang,TermC):-
        comp(X,Lang,Lang,X1),
        assign(X1,N,Xnum),
        list_comp(T,Lang,T1),
        append(Xnum,T1,TermC).

/* assign(?List,?Num,^List_Num):-
        Postfixes a number to all item of List so that all items of List_Num are X/N */

assign([],_,[]).
assign([X|T],N,[X/N|T2]):-
        assign(T,N,T2).

/* constant(?X):- Holds if X is a constant */
constant(X):-
        (integer(X);
                atom(X)).

/* process(?FunctionName, ?Complement_List,?ArgLength,^Result_List):-
        Each element X/N in Complement_List has a clause of name FunctionName
        and ArgLength number of arguments with X in position N created for it
        and is placed in the Result_List. It creates fresh variables automatically
        in each clause.  */

process(F,[],N,[]).
process(FuncName,[C1/Parentpos|CTail],Len,[R|Res2]):-
        P2 is Parentpos - 1,
        createvars(P2,Varlist),
        K is Len - Parentpos,
        createvars(K,Varlist2),
        makefun(FuncName,C1,Varlist,Varlist2,R),
        process(FuncName,CTail,Len,Res2).

/* makefun(?FunctionName,?Compitem,?VarsBefore,?VarsAft,^Result):-
        Result is clause FunctionName(VarBefore,...,Compitem,...,VarsAft) */

makefun(F,Compitem,V1,V2,R):-
        append(V1,[Compitem],V3),
        append(V3,V2,Args),
        R =.. [F|Args].

/* process(?FuncName, ?Complement_List, ?ArgLength, ?Lang_items, ^Result_List ):-
        Each element X/N in Complement_List has a clause of name FuncName
        and ArgLength number of arguments with X in position N created for it
        and is placed in the Result_List. It places all permutations of Lang_items
        in all argument positions of the clause list.  */

process(F,[],N,L,[]):-
        makefun2(FuncName,[],0,0,L,R). % Note that L is not used here.

process(FuncName,[C1/Parentpos|CTail],Len,Langvars,[R|Res2]):-
        P2 is Parentpos - 1,
```

```
        K is Len - Parentpos,
        makefun2(FuncName,C1,P2,K,Langvars,R),
        process(FuncName,CTail,Len,Langvars,Res2).

/* makefun2(?FuncName,?Compitem,?VarsBefore,?VarsAft,?Lang_perm,^Result):-
        Result is clause FuncName(langargs,...,Compitem,...,langaftargs)
        where all permutation of language items for the clause of length
        clausearglength - 1 are inserted into the function */

makefun2(Fname,Term,LBef,LAft,[Vars|VarTermList],[R|R1]):-
        (LBef > 0;LAft > 0),
        obtain(LBef,LAft,Vars,V1,V2),
        append(V1,[Term],Vterm),
        append(Vterm,V2,Vres),
        R =.. [Fname|Vres],
        makefun2(Fname,Term,LBef,LAft,VarTermList,R1).

makefun2(Fname,Term,LBef,LAft,[Vars|VarTermList],[R]):-
        LBef == 0,
        LAft == 0,
        R =.. [Fname|[Term]].

makefun2(Fname,Term,LBef,LAft,[],[]).

/*      obtain(?Num1,?Num2,?Items,^ItemsBefore,^ItemsAfter):-
        Return Num1 of the Items in ItemsBefore and Num2 of the items
        in ItemsAfter.  */

obtain(N1,N2,[X|Xs],[X|Ys],Vs):-
        N1 > 0,
        Nk is N1 -1,
        obtain(Nk,N2,Xs,Ys,Vs).
obtain(0,_,Xs,[],Xs).

/* createvars(?N,^NVars):- forms a list of N variables */

createvars(0,[]).
createvars(N,[X|T]):-
        K is N - 1,
        createvars(K,T).

/*      numstruct(?ArgList,?Num,^Res_List):-
        Numbers each element in Arglist in increments starting at N */

numstruct([],_,[]).
numstruct([X|T],N,[X/N|T2]):-
        N1 is N + 1,
        numstruct(T,N1,T2).

/*      doit(?Initlist,?Lang,?Length,?_,^Result):-
        Successively adds each language item in Lang to structure Initlist until
        Length is decremented to 1 and places all results in Result. A special
        instance is for length 0 when it is called for items with no arguments.
        This forms all permutations of length Length from the language Lang */

doit(Initlist,Lang,Len,L3,R):-
        Len > 1,
        add_elements(Initlist,Lang,Res1),
        Len2 is Len - 1,
        doit(Res1,Lang,Len2,L3,R).
doit(I,_,1,_,I).
doit(I,_,0,_,I).
```

```
/* add_elements(?Argument_List,?Lang,^Res1):-
        Res1 is a list of list structures formed by adding all language items in Lang
        to all elements in the Argument_List */

add_elements([X|Xs],Lang,R1):-
        run_all(X,Lang,Rfirst),
        add_elements(Xs,Lang,R2),
        append(Rfirst,R2,R1).
add_elements([],Lang,[]).

/* run_all(?Item,?Language,^Result):-
        Result is a list of list structures with all elements of the language
        added to Item. */

run_all(X,[[L]|Ls],[X2|Rfirst]):-
        append(X,[L],X2),
        run_all(X,Ls,Rfirst).
run_all(X,[],[]).

/*      sift_append(?List1,?List2,^Result1,?Length)
        appends all elements of List1 and List2 whilst sifting them for
        repeated items which are not included ensuring that all items in List1
        are of length Length                                                    */

sift_append([X|Ys],Zs,[X|Zss],Len):-
        length(X,Num),
        Num = Len,
        sift_append(Ys,Zs,Zss,Len).
sift_append([X|Ys],Zs,Zss,Len):-
        length(X,Num),
        Num < Len,
        sift_append(Ys,Zs,Zss,Len).
sift_append([],Xs,Xs,Len):-
        length(Xs,N2),
        N2 = Len.
sift_append([],Xs,[],Len):-
        length(Xs,N2),
        N2 < Len.


/*      sub_list(?List1,^List2):-
        List2 is a list of lists, each list containing one item of List1 */

sub_list([X|Xs],[[X]|Ys]):-
        sub_list(Xs,Ys).
sub_list([],[]).

/* negC(?(Clause:-Body),?Language,^Res):-
        Res contains all negated Clauses for the input clause for Language. It
        operates exactly in accordance with the negC definition in [Bar90]      */

negC((C:-B),Language,R):-
        negC1(C,Language,R1),!,
        negC2((C:-B),Language,R2),!,
        append(R1,R2,R).

/*      negC2(?(Clause:-Body),?Language,^Res1):-
        Res1 is the complement for clauses of the form (Clause:-Body) */

negC2((Clause:-Body),Language,R1_List):-
```

```
        Clause =.. [Pname|Args],
        concat(Pname,dash,P2),
        Head =.. [P2|Args],
        to_list(Body,Blist),
        obtain_inv(Head,Blist,R1_List).

/*      negC1(?Clause,?Language,^Res1):-
        Res1 is the complement structure of Clause for  Language. */

negC1(Clause,Language,R1):-
        Clause =.. [Pname|Args],
        length(Args,Arglen),
        numstruct(Args,1,Argstruct),
        list_comp(Argstruct,Language,Compargs),
        concat(Pname,dash,P1),
        process(P1,Compargs,Arglen,R1).

/* obtain_inv(?Head,?BodyLiterals,^Res_List):-
        Res_List is the  list containing the disjunctive clauses
        corresponding to a conjunctive original clause of the form (Head:-      BodyLiterals)
                                        */

obtain_inv(Head,[Body1|Body2_n],[R|Rest]):-
        Body1 =.. [P2|Argsbod],
        concat(P2,dash,P3),
        Body1d =.. [P3|Argsbod],
        find_local_vars(Head,Body1d,Local_Vars),
        create_complement(Head,Body1d,Local_Vars,R),
        obtain_inv(Head,Body2_n,Rest).

obtain_inv(_,[],[]).

/*      create_complement(H,B,?Varlist,(H:-(forall(Varlist,B)))).:-
        Varlist are the local variables that must be universally quantified in the body. */

create_complement(H,B,[X|Xs],(H:-(forall([X|Xs],B)))).
create_complement(H,B,[],(H:-B)).

/* find_local_vars(Head,Bodyitem,Local):-
        Local is the list of Local variables peculiar to Bodyitem and not in the Head */

find_local_vars(H,B,Local):-
        vars_in(H,Hvars),
        vars_in(B,Bvars),
        rem(Hvars,Bvars,Local).

/*      vars_in(?X,^Vars):- Holds if Vars are all the variables in function X      */

vars_in(X,Varlist):-
        X =.. [Fname|Args],
        get_vars(Args,Varlist).

/*      get_vars(?Argument_List,^Variable_List):-
        Variable_List are all the variables present in Argument_List     */

get_vars([X|Xs],[X|Ys]):-
        var(X),
        get_vars(Xs,Ys).

get_vars([X|Xs],Ys2):-
        not var(X),
        X =.. [Funcname|Args],
```

```
        get_vars(Args,ArgVars),
        get_vars(Xs,Ys),
        append(ArgVars,Ys,Ys2).

get_vars([X|Xs],Ys):-
        not var(X),not X =.. [Funcname|Args],
        get_vars(Xs,Ys).

get_vars([],[]).


/* rem(?Args,?Args2,^Res):-
        Res consists of all arguments solely present in Args and not in Args2     */

rem(Xs,[Y|Ys],Ys2):-
        in(Y,Xs),
        rem(Xs,Ys,Ys2).
rem(Xs,[Y|Ys],[Y|Ys2]):-
        not in(Y,Xs),
        rem(Xs,Ys,Ys2).
rem(Xs,[],[]).

/*      in(?element,?List):-
        Holds if element is a member of List. Will not incur any of the
        potential unifying properties of in */
in(Y,[X|Xs]):-
        Y == X.
in(Y,[X|Xs]):-
        Y \== X,
        in(Y,Xs).

/*      to_list(?A_struct,^A_list):-
        A_list is the list form of bracketed structure A_struct              */

to_list((A,B),[A|Bs]):-
        to_list(B,Bs).
to_list((A),[A]).

/* list(?L):- Holds if L is a list.                                          */
list([X|Xs]).
list([]).

/* flatten(?ListofLists,^Output_List):-
        ListofLists may be any structure containing elements or lists of elements
        which are processed into one list Output_List with all duplicate elements
        removed through a filtering process.                                  */

flatten([L|List],F2):-
        list(L),
        flatten(L,Fl),
        flatten(List,Fl1),
        filt_append(Fl,Fl1,F2).

flatten([L|List],[L|Ls2]):-
        not list(L),
        flatten(List,Ls2).
flatten([],[]).

/*      filt_append(?List1,?List2,^Result_List):-
        Result_List is List1 and List2 appended together with duplicate elements
        removed. */
```

```
filt_append([X|Xs],Zs,[X|Ys]):-
        not on_it(X,Xs),
        not on_it(X,Zs),
        filt_append(Xs,Zs,Ys).

filt_append([X|Xs],Zs,Ys):-
        on_it(X,Xs),
        on_it(X,Zs),
        filt_append(Xs,Zs,Ys).

filt_append([X|Xs],Zs,Ys):-
        on_it(X,Xs),
        not on_it(X,Zs),
        filt_append(Xs,Zs,Ys).

filt_append([X|Xs],Zs,Ys):-
        not on_it(X,Xs),
        on_it(X,Zs),
        filt_append(Xs,Zs,Ys).

filt_append([],Zs,Zs2):-
        set(Zs,Zs2).

/*      set(?List1,^Reslist):-
        Reslist is the set structure of List1.  */

set([X|Xs],[X|Xs2]):-
        not on_it(X,Xs),
        set(Xs,Xs2).
set([X|Xs],Xs2):-
        on_it(X,Xs),
        set(Xs,Xs2).
set([],[]).

/*      on_it(?Element,?List):-
        Holds if Element is a member of List. Used only for functions. */
on_it(X,[Y|Xs]):-
        equiv(X,Y),!.
on_it(X,[Y|Xs]):-
        on_it(X,Xs).

/*      equiv(?X,?Y):- Holds if X and Y are the same functions with the same args. */
equiv(X,Y):-
        X =.. [P|Args],
        Y =.. [P|Args2],
        same(Args,Args2).

/* same(?Argslist1,?Argslist2):-
        Holds if Argslist1 and Argslist2 are the same  including any functions in the
        argument lists. */
same([X|Xs],[Y|Ys]):-
        var(X),var(Y),
        same(Xs,Ys).
same([X|Xs],[Y|Ys]):-
        not var(X), not var(Y),X == Y,
        same(Xs,Ys).
same([],[]).

/*  prog_trans(?Program_List,?Language,^Clause_set):-
        Clause_set is the set of all transformed clauses for the given Program_List
        using Language.  */
```

```
prog_trans([(H:-C)|Ps],Language,[Clause_set1|Cs]):-
        negC((H:-C),Language,Unabridge_Cset),
        flatten2(Unabridge_Cset,Clause_set1),
        prog_trans(Ps,Language,Cs).
prog_trans([C|Ps],Language,[Clause_set1|Cs]):-
        C \= (A :- B),
        negC1(C,Language,Un_Cset),!,
        flatten2(Un_Cset,Clause_set1),
        prog_trans(Ps,Language,Cs).
prog_trans([],_,[]).


%-----------------------------------------------------
% Section for DATALOGIC implementation.
%-----------------------------------------------------

/*      instan(?ListofPredTransformations,?Language_combinations,^Result):-
        All predicate combinations in Language_combinations are inserted into
        ListofPredTransformations. */

instan([F|Fs],L2,[Flist|Fs2]):-
        varprocess(F,L2,Flist),
        instan(Fs,L2,Fs2).
instan([],_,[]).

/* varprocess(?PredList,?Language_items,^Reslist):-
        The Reslist consists of all predicates in Predlist with all permutations
        of language_items inserted into the variable argument  positions */

varprocess([F|Fs],Lang2,[Reslist|R]):-
        varcheck(F),
        F =.. [Pname|Args],
        checkpositions(Args,Varamount),
        doit(Lang2,Lang2,Varamount,_,Result),
        plug_in(Args,Result,Pname,Reslist),
        varprocess(Fs,Lang2,R).
varprocess([F|Fs],Lang2,[F|R]):-
        not varcheck(F),
        varprocess(Fs,Lang2,R).
varprocess([],_,[]).

/* varcheck(?Function):- Holds if Function has at least one variable argument */

varcheck(F):-
        F =.. [Fname|Args],
        varmem(Args).

/* varmem(?Args):- Holds if Args contains at least one variable */

varmem([F|Fs]):-
        var(F).
varmem([F|Fs]):-
        not var(F),
        varmem(Fs).

/* checkpositions(?List,^Num):- Holds if Num is the number of variables in List */

checkpositions([X|Xs],Varnum):-
        sumvars([X|Xs],0,Varnum).

/* sumvars(?Varlist,?Num,^Num2):-
        Calculates the number of variables in Varlist incrementing from Num */
```

```
sumvars([X|Xs],N,V):-
        var(X),
        N1 is N+1,
        sumvars(Xs,N1,V).
sumvars([X|Xs],N,V):-
        not var(X),
        sumvars(Xs,N,V).
sumvars([],X,X).

/* plug_in(?Arglist,?Lang_Permlist,?FunctionName,^Result_List):-
        All permutation of language terms are inserted from Lang_Permlist,
        replacing  Arglist in its original variable positions and then placing them
        as the arguments for FunctionName and inserting in Result_List          */

plug_in(Args,[List1|Ls],Pname,[R|Esults]):-
        apply(Args,List1,ArgRes),
        R =.. [Pname|ArgRes],
        plug_in(Args,Ls,Pname,Esults).
plug_in(_,[],_,[]).

/* apply(?Args,?List1,^Result_List):-
        Processes Args and List1 simultaneously adding either an element from
        List1 or Args to Result_List depending on which is not a variable */

apply([X|Xs],[L1|Ls],[L1|Ls2]):-
        var(X),
        apply(Xs,Ls,Ls2).
apply([X|Xs],[L1|Ls],[X|Ls2]):-
        not var(X),
        apply(Xs,[L1|Ls],Ls2).
apply(Xs,[],Xs).
apply([],Xs,Xs).
apply([],[],[]).

/*

This version that negates a ground clause  was not included in the final
working version. It is included for interest as in the clause negation process
it instantiates all arguments directly during program transformation for the
datalogic version.

negC1(Clause,Language,R1):- % Just for db clauses.
        Clause =.. [Pname|Args],
        length(Args,Arglen),
        A2 is Arglen - 1,
        numstruct(Args,1,Argstruct),
        list_comp(Argstruct,Language,Compargs),
        sub_list(Language,L2),
        doit(L2,L2,A2,_,Res),
        concat(Pname,dash,P1),
        process(P1,Compargs,Arglen,Res,R1).

*/
```

The code for the Unification Procedure is now presented:-

```
%-----------------------------------------------------------------------
% Extended unification algorithm for transformed clause sets.
%-----------------------------------------------------------------------
```

```
/*      pred_name(?Clause,^Name):-
        Holds if Name is the predicate of the head of Clause. */

pred_name((L1:-V),Name):-
        !,
        L1 =.. [Name|_].
pred_name(L1,Name):-
        L1 =.. [Name|_].

/*      extract(?Clause_List,^Ordered_Clause_List):-
        Ordered_Clause_List is a List of lists for each predicate name from
        Clause_List. This enables unification to be achieved via a simple
        processing operation. */

extract([[L1|Ls]|Lss],[[[L1|Ls]|NLss]|Rs]):-
        pred_name(L1,N),
        remove_same_items(N,Lss,NLss,Rest),
        extract(Rest,Rs).
extract([],[]).


/*      remove_same_items(?N,?List1,?OutList,^Result):-
        All of the predicates in  List1 are added to Outlist if their
        name is N, otherwise they are added to Result  */

remove_same_items(N,[[L1|Ls]|Lss],[[L1|Ls]|Ls2],Rs):-
        pred_name(L1,N),
        remove_same_items(N,Lss,Ls2,Rs).

remove_same_items(N,[[L1|Ls]|Lss],Ls2,[[L1|Ls]|Rs]):-
        not pred_name(L1,N),
        remove_same_items(N,Lss,Ls2,Rs).

remove_same_items(N,[],[],[]).

/*      main_unif(?Clauses,^Result):-
        Result of the list of successful unifications for Clauses.                  */

main_unif(Clauses,Result):-
        extract(Clauses,SeparateClauselist),
        rec_unif(SeparateClauselist,Result).

/* rec_unif(?ClauseList,^Reslist):-
        Reslist is the result of unifications for a clause set ClauseList          */

rec_unif([C|Cs],[R|Rs]):-
        unif(C,R),
        rec_unif(Cs,Rs).
rec_unif([],[]).

/*      unif(?ClauseList,^Result):-
        Result is the successful unification for a singular clause ClauseList */

unif([C1|C2_n],Rlist):-
        list(C1),
        prog_unif(C1,C2_n,[],Rlist).

/*      prog_unif(?Clauses,?Clauses2,^PartialRs,^FinalRes):-
        Each item in Clauses is attempted at unification with Clauses2. The
        successful unifications given in FinalRes.  */

prog_unif([C1|Cs],[Clause1|Clauses],Rs,F):-
```

```
        copy_term(Clause1,Clause1copy),
        list_unification(C1,Clause1copy,Res1),
        prog_unif(Cs,[Clause1|Clauses],[Res1|Rs],F).

prog_unif([],[C|Cs],R,F):-
        flatten2(R,R1), % flatten2 simply flattens the input list.
        prog_unif(R1,Cs,[],F).
prog_unif(F,[],[],F).

/*      flatten2(?Input,^Flattened_List):-
        Flattened_List is Input list moulded into a singular list */

flatten2([L|List],F2):-
        list(L),
        flatten2(L,Fl),
        flatten2(List,Fl1),
        append(Fl,Fl1,F2).

flatten2([L|List],[L|Ls2]):-
        not list(L),
        flatten2(List,Ls2).
flatten2([],[]).

/*      list_unification(?Clause,?Clause_set,^Result):-
        Result consists of all successful unifications of Clause with
        all of the  clauses in Clause_set */

list_unification(X,[Y1|Ys],[Y1|Ys2]):-  % unified clause is Y1.
        copy_term(X,X2),
        unification(X2,Y1),
        not multiple(X2,_,_),
        not multiple(Y1,_,_),
        list_unification(X,Ys,Ys2).

list_unification(X,[Y1|Ys],[X2|Ys2]):-  % unified clause is Y1.
        copy_term(X,X2),
        unification(X2,Y1),
        multiple(X2,_,_),
        not multiple(Y1,_,_),
        list_unification(X,Ys,Ys2).

list_unification(X,[Y1|Ys],[Y1|Ys2]):-  % unified clause is Y1.
        copy_term(X,X2),
        unification(X2,Y1),
        multiple(Y1,_,_),
        not multiple(X2,_,_),
        list_unification(X,Ys,Ys2).

list_unification(X,[Y1|Ys],[(V:-V2,V3)|Ys2]):-  % unified clause is Y1.
        copy_term(X,X2),
        unification(X2,Y1),
        multiple(X2,V,V2),  % Extra cases are for different body situations.
        multiple(Y1,V,V3),
        list_unification(X,Ys,Ys2).

list_unification(X,[Y1|Ys],Ys2):-
        not unification(X,Y1),
        list_unification(X,Ys,Ys2).

list_unification(X,[],[]).

/* unification(?X,?Y):- Holds if X and Y unify together. */
```

```
unification(X,Y):-
        var(X),
        var(Y),
        X=Y.
unification(X,Y):-
        var(X),
        nonvar(Y),
        not_occurs(X,Y),
        X = Y.
unification(X,Y):-
        var(Y),
        nonvar(X),
        not_occurs(Y,X),
        Y = X.
unification(X,Y):-
        nonvar(X),
        nonvar(Y),
        constant(X),
        constant(Y),X = Y.
unification(X,Y):-
        nonvar(X),
        nonvar(Y),
        compound(X),
        compound(Y),
        termunif(X,Y).

/*      not_occurs(?X,?Y):- is an implementation of the occur check,
        not present in standard prolog systems for its computational
        expense. Succeeds if X does not occur in Y. */

not_occurs(X,Y):-
        var(Y),
        X \== Y.
not_occurs(X,Y):-
        nonvar(Y),
        constant(Y).
not_occurs(X,Y):-
        nonvar(Y),
        compound(Y),
        functor(Y,F,N),
        not_occurs(N,X,Y).
not_occurs(N,X,Y):-
        N > 0,
        arg(N,Y,Arg),
        not_occurs(X,Arg),
        N1 is N - 1,
        not_occurs(N1,X,Y).
not_occurs(0,X,Y).


% The following two clauses create what
% the transformational approach describes as the '@' operator in [Bar90]

/*      termunif(?X,?Y):- holds if X and Y are both functors and unifiable
        X and Y may consist of a clause of the form (Head:- Body) in which
        the heads are unified and the bodies conjuncted at a higher level.      */

termunif(X,Y):-
        functor(X,F,N),
        functor(Y,F,N),
        unifargs(N,X,Y).
```

```
termunif(X,Y):-    %  multiple((V:-Y),V,Y).
        multiple(X,V1,V2),
        functor(V1,F,N),
        functor(V2,F,N),
        functor(Y,F,N),
        unifargs(N,V1,Y),
        unifargs(N,V2,Y).

termunif(X,Y):-
        multiple(Y,V1,V2),
        functor(X,F,N),
        functor(V1,F,N),
        functor(V1,F,N),
        unifargs(N,X,V1).

termunif(X,Y):-    %  multiple((V:-Y),V,Y).
        multiple(Y,V1,V2),
        multiple(X,V3,V4),
        functor(V1,F,N),
        functor(V3,F,N),
        unifargs(N,V1,V3).

termunif(X,Y):-    %  multiple((V:-Y),V,Y).
        not multiple(Y,_,_),
        multiple(X,V3,V4),
        functor(Y,F,N),
        functor(V3,F,N),
        unifargs(N,Y,V3).

/*      unifargs(?Num_of_Args,?X,?Y):-
        Holds if X and Y unify for all of their constituent arguments as they
        consist of functors.  */

unifargs(N,X,Y):-
        N > 0,
        unifarg(N,X,Y),
        N1 is (N - 1),
        unifargs(N1,X,Y).
unifargs(0,X,Y).

unifarg(N,X,Y):-
        arg(N,X,ArgX),
        arg(N,Y,ArgY),
        unification(ArgX,ArgY).


multiple((V:-Y),V,Y).
```

The linearization code is:

```
/*      linearization(?Restricted_Program_Set,^Unres_prog_set)
        :-calls linearize for each program element in the
        Restricted_Program_Set list structure.                          */

linearization([P|Ps],[LC|LCs]):-
        linearize(P,LC),
        linearization(Ps,LCs).

linearization([],[]).
```

```
/*      linearize(?(Clause:-Body),^(New_Clause :- New_Body)):-
        Takes any non-left linear clause and recreates the
        clause in left linear form. Any type of 'pure' Prolog
        clause can be given as input and the Body can be null. */

linearize((Clause:-Body),(New_Clause :- Eqlist2,Body)):-
        Clause =.. [Fname|Args],
        equality(Args,NewArgs,Eqlist),
        not empty(Eqlist),
        New_Clause =.. [Fname|NewArgs],
        tostruct(Eqlist,Eqlist2),!.

linearize(Clause,(New_Clause :- Eqlist2)):-
        Clause \= (G:-V),
        Clause =.. [Fname|Args],
        equality(Args,NewArgs,Eqlist),
        not empty(Eqlist),
        New_Clause =.. [Fname|NewArgs],
        tostruct(Eqlist,Eqlist2),!.

linearize((Clause:-Body),(Clause :-Body)):-
        Clause =.. [Fname|Args],
        equality(Args,NewArgs,Eqlist),!,
        empty(Eqlist).

linearize(Clause,Clause):-
        Clause \= (G:-V),
        Clause =.. [Fname|Args],
        equality(Args,NewArgs,Eqlist),!,
        empty(Eqlist).

/*      equality(?Arguments,^Left_LinearArguments,^Eqs):-
        Arguments is a list of arguments taken from the clause
        structure that is returned in Left_LinearArguments.
        The Eqs is used to return any 'eq'/2 clauses that are formed      */

equality([X|Xs],X3,Eqs):-
        var(X),
        compare_elements(X,Xs,NewXs,Eqlist),
        equality(Xs,NewXs2,Eqlist2),
        append([X],NewXs,X3),
        append(Eqlist,Eqlist2,Eqs).

equality([X|Xs],[X|Xs3],Es):-
        constant(X),
        equality(Xs,Xs3,Es).

equality([X|Xs],[X1|Xs3],Eqout):-
        not var(X),
        not constant(X),
        X =.. [Xname|XArgs],
        equality(XArgs,NewXargs,Eq1),
        X1 =.. [Xname|NewXargs],
        compare_arguments(NewXargs,Xs,XsOut,Eqlist),
        equality(XsOut,Xs3,Eq2),
        append(Eq1,Eqlist,Eqdash),
        append(Eqdash,Eq2,Eqout).
equality([],[],[]).

/*      compare_elements(?Element,?Args,^Res_Args,^Eqs):-
        Element is compared for equality with the Args list. If any
        of the Args elements are functions the predicate is
```

```
        recursively called. Eqs contains any 'eq'/2 clauses that are
        required for left-linear form of the Res_Args list.                 */

compare_elements(X,[],[],[]).
compare_elements(X,[X1|Xs],[X1|Xs2],Es):-
        var(X1),
        X \== X1,
        compare_elements(X,Xs,Xs2,Es).
compare_elements(X,[X1|Xs],[Y2|Xs2],[eq(X,Y2)|Es]):-
        var(X1),
        X == X1,
        compare_elements(X,Xs,Xs2,Es).
compare_elements(X,[X1|Xs],[X1|Xs2],Es):-
        constant(X1),
        compare_elements(X,Xs,Xs2,Es).
compare_elements(X,[X1|Xs],[X1dash|Xs2],Es):-
        not var(X1),
        not  constant(X1),
        X1 =.. [X1name|Xargs],
        compare_elements(X,Xargs,NewXargs,Eq),
        X1dash =.. [X1name|NewXargs],
        compare_elements(X,Xs,Xs2,Eq2),
        append(Eq,Eq2,Es).

/*      compare_arguments(?Arg_List,?Arg2_List,^Res_Args,^Eqlist):-
        Sequentially runs through Arg_List comparing all of it elements
        with the whole of Arg2_List. The left-linear Res_Args is the
        output structure for the arguments along with any 'eq'/2 clauses
        needed in Eqlist                                                    */

compare_arguments([X|Xs],OtherArgs,ArgsOut,Eqlist):-
        compare_elements(X,OtherArgs,OutArgs,Eq2),
        compare_arguments(Xs,OutArgs,ArgsOut,Eq3),
        append(Eq2,Eq3,Eqlist).
compare_arguments([],A,A,[]).

/*      empty(List):- holds if List contains no elements                    */

empty([]).
```

Sample Clause Test Cases are given:

```
/* Data window.
        clauses(?Num,^ProgList) - Returns the desired program set according
        to the given number Num.
        signature(?Num,^Signature) - Does the same for the signature. */

clauses(1,[lessoreq(0,X),(lessoreq(s(X),s(Y)):-lessoreq(X,Y))]).
clauses(2,[even(0),(even(s(s(Y))):-even(Y))]).
clauses(3,[(member(X,cons(Y,Ys)):-eq(X,Y)),(member(X,cons(Y,Ys)):-member(X,Ys))]).
clauses(5,[(append(nil,X,Y):-eq(X,Y)),(append(cons(X,Xs),Ys,cons(X2,Zs)):-eq(X,X2),append(Xs,Ys,Zs))]).
clauses(6,[rev(nil,nil),(rev(cons(X,L),M):-rev(L,N),append(N,cons(X,nil),M))]).
clauses(7,[rev(nil,nil),(rev(cons(X,L),M):-rev(L,N),append(N,cons(X,nil),M)),(append(nil,X,Y):-eq(X,Y)),
(append(cons(X,Xs),Ys,cons(Y,Zs)):-eq(X,Y),append(Xs,Ys,Zs))]).
clauses(8,[(drev(nil,X-Y):-eq(X,Y)),(drev(cons(H,T),M-N):-drev(T,M-cons(H,N)))]).
clauses(4,[r(a,b),r(c,d),r(c,b),(q(X):-r(X,Y))]).   %
clauses(9,[r(b,c),r(b,a)]).
clauses(10,[(q(X):-r(X,Y),re(Y))]).
clauses(11,[q(b,X,Y),q(X,Y,b),r(X,Y,a),r(X,b,Y),(p(X,Y,Z):- q(X,Y,Z),r(X,Y,Z)),(s(X,Y):-p(X,Y,Z))]).
clauses(12,[(sad(X):- not happy2(X)),(happy2(X):- not happy(X)),happy(chris),happy(colin)]).
sig(12,[chris,colin,deborah]).
sig(11,[a,b]).
sig(10,[a]).
sig(9,[a,b,c]).
sig(5,[nil,cons(_,_)]).
sig(6,[nil,cons(_,_)]).
sig(7,[nil,cons(_,_)]).
sig(8,[nil,cons(_,_)]).
sig(4,[a,b,c,d]).
sig(3,[nil,cons(_,_)]).
sig(2,[0,s(_)]).
sig(1,[0,s(_)]).
sub_sig(1,[a,b,c,d]).

/* Sample transformed programs */
rev(nil,nil).
rev(cons(X,L),M):-rev(L,N),app2(N,cons(X,nil),M).
app2(nil,X,X).
app2(cons(X,Xs),Ys,cons(X,Zs)):-app2(Xs,Ys,Zs).
rev2([],[]).
rev2([X|L],M):-
        rev2(L,N),
        app2(N,[X],M).
app2([],X,X).
app2([X|Xs],Ys,[X|Zs]):-app2(Xs,Ys,Zs).
drev([],X-X).
drev([H|T],M-N):-
        drev(T,M-[H|N]).
eff_rev(X,Y):-
        drev(X,Y-[]).
happy(deb).
happy(jaya).
saddash(_1206):- happy2dash(_1206).  % transform is notdash (happy2(X)).
happy2dash(_1206):-notdash(happy(_1206)).
happydash(deborah).
notdash(X):- not X,!,fail.   %Needs local var check.
notdash(X):- X.

/*      go accesses the program and signature structures and transforms
        in the following manner:
                1. Transform whole clause set.
```

```
                2. Unify all arguments.
                3. Flatten final structure into one list structure.          */

go(Num,NiceFF):-
        clauses(Num,P),
        sig(Num,S),
        prog_trans(P,S,Res2),
        main_unif(Res2,FFset),
        flatten(FFset,NiceFF).

/*      godata accesses the program and signature structures and transforms
        in the following manner:
                1. Transform whole clause set.
                2. Unify all arguments.
                3. Instantiate all unbound clauses with all language item permutations
                4. Flatten final structure into one list structure.          */

godata(Num,NiceFF):-
        clauses(Num,P),
        sig(Num,S),
        prog_trans(P,S,Res2),
        main_unif(Res2,FFset),
        sub_list(S,S2),
        instan(FFset,S2,FFRes),
                flatten(FFRes,NiceFF).
```

The Front End Code is not included in this report due to its system specific nature. A copy of the code has been presented to Dr. K. Broda.

A5
Constraints Meta-Interpreter Code

```
% For final operation ^ was used as the forall operator.

:- op(140,xfy,^).

/* solve(?Goal,?Initial_constraints,^Result_constraints)
        A goal is solved according to restrictions imposed by the Initial constraints set     */

solve(true,_,_).

solve(X^Goal,Init_cons,Rcons):-
        solve(Goal,Init_cons,Rcons2),
        obtain_Q_cons(X,Rcons2,Quant_cons,Rc2),
        obtain_Q_cons(X,Init_cons,Quant_cons2,Icons),
        not equiv(Quant_cons,Quant_cons2),
        negate(Quant_cons,Neg_Quant_cons),
        append(Rc2,Neg_Quant_cons,Conslist),
        solve(X^Goal,Conslist,Rcons).

solve(X^Goal,Init_cons,Rc2):- % Universal Quantifier Evaluation.
        solve(Goal,Init_cons,Rcons2),
        obtain_Q_cons(X,Rcons2,XCons,Rc2),
        obtain_Q_cons(X,Init_cons,XC2,Icons),
        equiv(XCons,XC2).

solve((Goal1,Goal2),I1,Rcons):-
        solve(Goal1,I1,R1),
        solve(Goal2,R1,Rcons).

solve(Goal,Initial_Inequalities,Rcons):-
        Goal \= (_^_),   % for ground clauses.
        copy_term(Goal,Goal_Clone),
        clause(Goal_Clone,true),
        obtain_constraints(Goal,Goal_Clone,Cons),
        reconcile(Initial_Inequalities,Cons,Rcons).

solve(Goal,Initial_Inequalities,Rcons):-  % solves goals with constraints first.
        Goal \= (_^_),   % for ground clauses.
        clause(Goal,GoalBody),   % put a copy term in here?????????
        GoalBody \= true,
        separate(GoalBody,Body_Clauses,Body_Constraints),
        tostruct(Body_Clauses,BCstruct),
        reconcile(Initial_Inequalities,Body_Constraints,Relations),
        solve(BCstruct,Relations,Rcons).

solve(neg(Goal),Initial_Inequalities,Rcons):-
        solve(Goal,[],Relations),
        negate(Relations,Inv_relations),
        reconcile(Initial_Inequalities,Inv_relations,Rcons).
```

```
solve(token,R,R).

/* equiv(X,Y) succeeds if X and Y are the same  */
equiv([X|Xs],Ys):-
        on_it(X,Ys),
        equiv(Xs,Ys).
equiv([],_).

/* obtain_Q_cons(?Var,?Cons_set,^Quant_set,^Non-quant_set) takes Var
                        and a constraints set Cons_set and separataes the constraints
                        according to the variable value.   */

obtain_Q_cons(X,[T(V1,U)|Ys],[T(V1,U)|Ys2],Rs):-
        X == V1,
        obtain_Q_cons(X,Ys,Ys2,Rs).

obtain_Q_cons(X,[T(Y,U)|Ys],Ys2,[T(Y,U)|Rs]):-
        X \== Y,
        obtain_Q_cons(X,Ys,Ys2,Rs).

obtain_Q_cons(X,[],[],[]).


/* separate(?X,^V,^Y) takes the body of a clause, namely X, and separates
                the clause between literals and constraints, V and Y respectively.  */

separate((A,B),[A|Xs],Ys):-
        A \= ineq(_,_),
        A \= eq(_,_),
        separate(B,Xs,Ys).

separate((A,B),Xs,[A|Ys]):-
        (A = ineq(_,_);
        A = eq(_,_)),
        separate(B,Xs,Ys).

separate(A,[],[A]):-
        A = ineq(_,_);
        A = eq(_,_).

separate(A,[A],[]):-
        A \= ineq(_,_);
        A \= eq(_,_).

/* tostruct(?X,^Y) converts list X to structure Y  */

tostruct([X],X):- !.
tostruct([X|Xs],(X,Ys)):-!,
        tostruct(Xs,Ys).
tostruct([],token).

% Test clause
job((p(U,f(U),X):- ineq(U,4),ineq(U,2),q(2),eq(X,3),r(X))).

/* negate is at present a basic clause negation system */

negate([eq(U,V)|Xs],[ineq(U,V)|Ys]):-
        negate(Xs,Ys).
negate([ineq(U,V)|Xs],[eq(U,V)|Ys]):-
        negate(Xs,Ys).
negate([],[]).
% extend to handle disjuncts, conjuncts etc.
```

```
/* reconcile(?X,?X2,^Y) succeeds if X and X2 can be reconciled to
                                form Y  */
reconcile([],Xs,Xs):-
        check_cons(Xs),!.

reconcile(Xs,[],Xs):-
        check_cons(Xs),!.

reconcile(Xs,[Y|Ys],[Y|Xs2]):-
        check_cons(Xs),
        check_constituent(Y),
        not on_it(Y,Xs),!,
        resolve_conflicts(Y,Xs),
        reconcile(Xs,Ys,Xs2).

reconcile(Xs,[Y|Ys],Xs2):-
        check_cons(Xs),
        check_constituent(Y),
        on_it(Y,Xs),!,
        reconcile(Xs,Ys,Xs2).


/* check_cons(?X) checks a single constraint set X to ensure there
                are no conflicts   */
check_cons([X|Xs]):-
        check_constituent(X),
        check_cons(Xs).

check_cons([]).

check_constituent(Y):-
        Y =.. [Yname,A1,A2],
        var(A1),not var(A2).
check_constituent(Y):-
        Y =.. [Yname,A1,A2],
        Yname == eq,
        not var(A1),
        not var(A2),
        A1 == A2.

/* on_it(?X,?Xs) is a non-unifying member check to discover if X
                                is on Xs.   */
on_it(X,[Y|Ys]):-
        X == Y.
on_it(X,[Y|Ys]):-
        X \== Y,
        on_it(X,Ys).

/* resolve_conflicts(?Constraint,?Cons_set) checks if a constraint
                                is satisfiable with a constraint set. It requires enhancement. */
resolve_conflicts(eq(X,Y),Xs):-
        not on_it(ineq(X,Y),Xs).
resolve_conflicts(ineq(X,Y),Xs):-
        not on_it(eq(X,Y),Xs).

/* obtain_constraints(?T,?T2,Cons_set) forms constraints from
                                two clauses.                                    */

obtain_constraints(Term1,Term2,Conslist):-
        Term1 =.. [Pname|Args],
        Term2 =.. [Pname|Args2],
```

```
        createcons(Args,Args2,Conslist).

/* createcons(?S1,?S2,^Constraints_set) creates the constraint
                set from value set S1 and variable set S2  */

createcons([X|Xs],[Y|Ys],[eq(X,Y)|Rs]):-
        createcons(Xs,Ys,Rs).
createcons([],[],[]).

/* uninstanciate(?C,^Res,^Constraint) takes a clause and unbinds the
                bound arguments forming constraints out of them    */

uninstanciate(Clause,C2,Constraints):-
        Clause =.. [Pname|Args],
        filter(Args,VArgs,Constraints),
        C2 =.. [Pname|VArgs].

/* filter(?In,^Out,^Constraints) takes an argument set In and replaces
                constants with variables in structure Out and creates Constraints
                from the constants.                                              */

filter([X|Xs],[X|Ys],Cs):-
        var(X),
        filter(Xs,Ys,Cs).
filter([X|Xs],[W|Ys],[eq(W,X)|Cs]):-
        constant(X),
        filter(Xs,Ys,Cs).
filter([],[],[]).
ineq(_,_).
eq(_,_).

constant(X):-
        integer(X).
constant(X):-
        atom(X).


/*  A negative goal variant that was not expanded.                    */

solve(neg Goal,Initial_Inequalities,Rcons2):-
        not groundcheck(Goal),
        uninstanciate(Goal,Goal2,Constraints),
        solve(Goal2,[],Relations),
        negate(Relations,Inv_relations),
        reconcile(Initial_Inequalities,Inv_relations,Rcons),
        reconcile(Rcons,Constraints,Rcons2).

solve(neg Goal ,Initial_Inequalities,Rcons):-
        groundcheck(Goal),
            solve(Goal,Initial_Inequalities,Rcons).
```