Constructive Negation in Logic Programs

Clifford Walinsky
B.A., University of California, San Diego, 1978

A dissertation submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

August, 1987

The dissertation "Constructive Negation in Logic Programs" by Clifford Walinsky has been examined and approved by the following Examination Committee:

Richard Hamlet, Thesis Advisor
Professor

Richard Kieburtz
Professor and Chairman

David Maier
Associate Professor

Kamal Abdali
Adjunct Associate Professor

Dedication

I dedicate this work to my wife, who worked tirelessly to finance my education, and who provided endless encouragement. I must also acknowledge the fine work of Betsy Moore who raised our child while both parents were away at work. I thank Dick Hamlet for his consistently good advice and excellent reviewing. I also thank David Maier for his vast store of knowledge about logic programming, Mark Grossman for advising me about the value of monotonicity, and Dick Kieburtz for allowing me to attend the Oregon Graduate Center. Finally, I extend thanks to Vincent Sigillito at the Air Force Office of Scientific Research for the generous financial support of his agency.

Table of Contents

List of Lemmata and Theorems vii
Glossary ix
List of Notation xiii
Lists of Figures and Examples xv
Abstract xvi
Chapter 1. Introduction 1
1.1. Declarative Languages 3
1.2. Logic Programming 7
1.3. Constructive Negation 9
1.4. Overview 11
Chapter 2. Basic Concepts and Notation 14
2.1. Syntax of Terms and Formulas 14
2.2. Substitutions 18
2.3. Common Instances of Terms 21
2.4. Terms as Denotations for Sets 24
Chapter 3. Enhancing Expressiveness of Logic Programming 29
3.1. Underlying Model of Definite Clause Programs 29
3.2. SLD-Resolution 34
3.3. Negation by Failure 42
3.3.1. The Closed World Assumption 42
3.3.2. Program Completion 43
3.3.3. Incorrectness of Negation by Failure 45
3.3.4. Incompleteness of Negation by Failure 47
3.3.4.1. Canonical Programs 48
3.3.4.2. Inconsistency of a Program's Completion 48
3.4. Enhancing Expressiveness of Programs with Negation 50
3.4.1. Extended Programs 50
3.4.2. Model Elimination 52
Chapter 4. Constructive Negation 54
4.1. A Language of Logic Programs with Constructive Negation 55
4.2. Underlying Model of DIF-Programs 56
4.2.1. Three-Valued Logic 57
4.2.2. Constructive Interpretations of Formulas 58
4.2.3. Constructive Models of DIF-Programs 63

Chapter 5. Fixedpoint-Consistent DIF-Programs 74
5.1. Syntax of DEF-Programs 75
5.2. Compilation of DEF-Programs 76
Generating Dual DIFs 76
Non-Conflicting DEF-Programs 77
Fixedpoint-Consistency is Attained 78
5.3. Evaluation of DEF-Programs 80
5.4. Resolving Incorrectness of Universally Quantified Queries 87
Overly-General Generated Values 88
Insufficient Generated Values 89
No Generated Values 92
Summary 93
5.5. Correctness of the Evaluation System 94
5.6. Incompleteness of any Evaluation System for DEFs 99
Chapter 6. Enhancements 104
6.1. Controlling Explosion of DEFs 104
Evaluation of Equality 104
Database-Oriented Programs 108
Polymorphic Programs 110
Summary 113
6.2. Incorporating Type Information for Self-Coverage 113
6.3. Enhancing Evaluation of Universal Quantification 118
6.3.1. Term-Matching 119
6.3.2. Free Variables in Universally Quantified Queries 121
Free Variables in Only the Generator or the Tester 121
Free Variables in both Generator and Tester 122
Functional Generators 124
Summary 127
6.4. Summarizing the Enhancements 128
Chapter 7. Implementation Through Meta-Programming 129
7.1. Compilation of DEF-Programs 130
7.1.1. Variable Disambiguation 131
7.1.2. Overlap Checking 131
7.1.3. Typechecking 132
7.1.4. Self-Coverage Testing 134
7.1.5. Generating Dual DIFs 135
7.2. Evaluating Queries 136
7.3. Summary 140

Chapter 8. Summary and Future Work 141
8.1. Summary 141
8.2. Future Work 143
References. 145
Appendix A. Three-Valued Truth Tables 149
Appendix B. Three-Valued Valuations 151
Appendix C. DEF-Program Example 155

List of Lemmata and Theorems

Lemma 2.1: $\cong$ is an equivalence relation on terms. 21

Lemma 2.2: $[s] \gtrsim [t]$ iff t is an instance of s. 22

Lemma 2.3: $\gtrsim$ is a partial ordering on equivalence classes of terms. 22

Lemma 2.4: $\tilde{\sqcap}$ produces the mgci of terms. 23

Lemma 2.5: If terms s and t have no variables in common, then s and t are unifiable iff the terms have a common instance. 23

Lemma 2.6: If t is an instance of s, $(s \mathbin{@} \Sigma) \supseteq (t \mathbin{@} \Sigma)$. 25

Lemma 2.7: When Σ is non-trivial, for all terms s and t, $s@\Sigma = t@\Sigma$ iff $s \cong t$. 26

Lemma 2.8:
(1) If terms s and t have no common instance, then $s@\Sigma$ and $t@\Sigma$ are disjoint.
(2) If terms s and t have a common instance, then $((s \tilde{\sqcap} t)@\Sigma) = (s@\Sigma) \cap (t@\Sigma)$. 27

Lemma 2.9: For all terms s and t, $(s@\Sigma) \supseteq (t@\Sigma)$ iff $[s] \gtrsim [t]$. 27

Lemma 3.1: Any interpretation for a definite clause program has an equivalent Herbrand interpretation. 31

Lemma 3.2 (Model Intersection) 31

Lemma 3.3 (Logical Implication) 31

Lemma 3.4 (Characterization of Fixedpoints) 33

Lemma 3.5 (Characterization of Least Model by T) 34

Lifting Lemma 3.6: If query $Q\sigma$ has a success path, then Q has a success path. 39

Lemma 3.7 (Construction correctness) 40

Lemma 3.8 (Construction completeness) 40

Corollary 3.9 (Construction Completeness & Correctness) 41

Lemma 3.10 (Meaning of Failure for Ground Atoms) 45

Lemma 3.11 (Meaning of Failure) 45

Lemma 3.12 (Completeness for Canonical Programs) 48

Lemma 4.1 (Monotonicity of Interpretations) 61

Lemma 4.2 (Complement equivalent to Negation) 62

Lemma 4.3 (T monotonic): 65

Lemma 4.4 (Relating Models and Fixedpoints) 66

Lemma 4.5 (Relating Strong Models and Fixedpoints) 68

Lemma 4.6: Fixedpoint-consistency of an arbitrary DIF-program is undecidable. 71

Lemma 5.1 (Compilation is Meaning Preserving) 77
Lemma 5.2 (Non-Overlapping Programs are Non-Conflicting) 78
Lemma 5.3 (Non-conflicting DEF-programs are fixedpoint-consistent) 78
Lemma 5.4 (Correctness of Self-Coverage Test) 91
Theorem 5.5 (Correctness for Finite Full Search Trees) 95
Theorem 5.6 (Correctness for General Full Search Trees) 99
Lemma 5.7: For any TM m and configuration c, accept(m,c) is fixedpoint-implied from NonRE if a final configuration can be reached via the relation $\vdash_m{}^*$ from initial configuration c; otherwise, ~accept (m,c) is fixedpoint-implied. 100
Lemma 5.8 (NonRE accepts L_e) 102
Corollary 5.9: There is no complete evaluation system for DEF-programs. 102
Lemma 6.1 (Complement of Functionally-Determined Formula Equivalent to Negation) 126

Glossary

(In order of appearance)

underlying model: descriptive characterization of a declarative language program.	3
evaluation procedure: procedural characterization of a declarative language.	4
recognizer: an evaluation procedure that affirms or denies the presence of elements satisfying a query.	4
transducer: an evaluation procedure that enumerates elements of an underlying model satisfying a query.	4
definite clause: statement of implication.	7
head: conclusion of a clause.	7
body: premise of a clause.	7
assertion: an unconditional definite clause with no body.	7
arity: number of arguments of a function of predicate symbol.	14
constant: function symbol with arity zero.	14
ground: a term, atom or literal without variables.	14
proposition: predicate symbol with arity zero.	15
atom: an atomic formula.	15
logical operators: negation, and infinite disjunction and conjunction.	15
formula: a Boolean expression with atomic elements from $A(\Pi, \Sigma)$, for some arbitrary sets of predicate symbols Π and function symbols Σ.	16
connective: written forms of the logical operators.	16
free variable: a variable not bound by a quantifier.	17
bound variable: variable X is bound in $\exists X{:}F$ and $\forall X{:}F$.	17
closed formula: contains no occurrences of unbound variables.	18
sentence: a closed formula.	18
open formula: contains free occurrences of variables.	18
existential closure: e.c. of formula F, denoted $\exists F$, binds every free variable in F with an existential quantifier.	18
universal closure: u.c. of formula F, denoted $\forall F$, binds every free variable in F with a universal quantifier.	18
substitution: mapping from variables to terms. Later extended to mappings from formulas to formulas.	18
domain of a substitution: set of variables not mapped to themselves by a substitution.	18
identity substitution: ϵ maps every variable X to X.	21

renaming substitution: a bijective substitution. 21

variant terms: terms that differ only by a renaming. 21

instance: a term that can be produced from another by application of a substitution. 21

common instance: c.i. of terms S is (all variants of) a term that is an instance of all terms in S. 23

most general common instance (mgci): mgci of S is the most general term that is an instance of all terms in S. 23

mgci: most general common instance. 23

unifiability: terms s and t are unifiable if there is a substitution σ such that $s\sigma = t\sigma$. 23

unifying substitution: a u.s. of terms produces a common instance of the terms. 23

decomposition of mgci: when terms s and t have a mgci, and therefore a unifying substitution, the unifying substitution can be partitioned according to the variables in s and t. 24

template: t. of a set of ground terms T is a term t such that $T = t@\Sigma$. 25

non-trivial set of function symbols: a set Σ such that $|T(\Sigma)| > 1$. 25

interpretation: (see also constructive interpretation) a correspondence between syntactic elements of a formula, and actual objects, functions and relations. 29

Herbrand interpretation: a symbolic interpretation. 30

Herbrand Universe: set of all ground terms constructible from symbols in a program. 30

Herbrand Base: set of all ground atoms constructible from symbols in a program. 30

model: M is a model of program P if $M[P]$ is true. 31

query: a conjunction of atoms for a definite clause program, and a formula for a DIF-program. 31

logical implication: a query is l.i. by a program if some instance of the query is true in all models of a program. 31

immediate consequence functional: denoted by T_P; unique for each program, producing all atoms inferred by the program from an Herbrand interpretation. 32

powers (of T): multiple applications of the functional T. 32

fixedpoint (of T): an Herbrand interpretation $I = T(I)$. 33

least fixedpoint: intersection of all fixedpoints of T_P. 33

greatest fixedpoint: union of all fixedpoints of T_P. 33

answer substitution: derived by SLD-resolution and the evaluation system for DIF-programs that produces an instance of a query logically implied by a program. 35

full search tree: data structure used by SLD-resolution and the evaluation system for DIF-programs to obtain answer substitutions. 35

success node: a leaf of a full search tree indicating that a query and all subsequent subqueries have been resolved. 36

success path: a path from the root to a success node in a full search tree. 36

value of success path: composition of the labels of all edges along a success path. 36

general clause: a formula of the form $A \leftarrow L_1 \wedge \ldots \wedge L_n$, where A is an atom and each L_i is a literal. Used to express negation by failure. 42

Closed World Assumption: all facts not logically implied are assumed false. 42

failed full search tree: a full search tree without any success paths. 43

program completion: the "if-and-only-if" version of a definite clause program. 43

general form (of a clause): an implication produced from a definite clause used in generating a completed program. 43

completed form (of a predicate): the if-and-only-if definition of a predicate in a completed program. 43

completion of a program: the if-and-only-if version of a program. 44

correct selection rule: selection rule for SLD-resolution with negation by failure, selecting negated atoms only if ground. 46

floundered query: a query for which no selection rule is correct. 46

allowed query: all variables occurring in a negated atom occur within a positive atom elsewhere within the query. 46

allowed program: variables occurring within the head of every general clause occur within a positive atom within the body of the clause. 46

canonical program: a definite clause program for which $T{\downarrow}\omega = gfp$. 48

DIF: see Definite Inference Form. 55

Definite Inference Form: a statement of implication used to define positive and negative information, denoted $L \leftarrow F$, where L is a literal and F is a formula. 55

constructive interpretation: a mapping from the Herbrand Base of a program to the three logical constants. 58

complement: c. of a formula is negation of the formula with negation applied only to atoms. 62

strong model: s.m. of a program is a constructive interpretation that gives a valuation of true to the program. 64

weak model: w.m. of a program is a constructive interpretation that does not assign a false valuation to the program. 64

inconsistent program: in 2-valued logic a program without models; in 3-valued logic a program without strong models. 64

fixedpoint-inconsistent program: a DIF-program without fixedpoints. 67

closure ordinal: the least ordinal α such that $T\uparrow\alpha$ is a fixedpoint and $T\uparrow\alpha = lfp$. 70

fixedpoint-implied formula: a formula assigned a true valuation by every fixedpoint of a DIF-program. 70

DEF: see Definite Equivalence Form. 75

Definite Equivalence Form: a statement of equivalence used to compile into dual DIFs, denoted by $A \leftrightarrow F$, where A is an atom and F is a formula. 75

definition: all DEFs in a program with heads $p(x_1, \ldots, x_n)$ for some predicate p. 75

conflicting DEFs: DEFs $A_1 \leftrightarrow F_1$ and $A_2 \leftrightarrow F_2$ for which there are substitutions σ_1 and σ_2 such that $A_1\sigma_1 = A_2\sigma_2$. Conflicting DEFs in a DEF-program may cause fixedpoint-inconsistency. 77

overlap test: determines if a program contains distinct DEFs whose heads unify; used to prevent fixedpoint-inconsistency. 77

generator: the formula G of a bounded universally quantified formula $\forall X\colon G \rightarrow T$ used to generate values of X. 80

tester: the formula T of a bounded universally quantified formula $\forall X\colon G \rightarrow T$ used to test all generated X values. 80

term-matching: a procedure to determine if a term is an instance of another. 88

self-covering DEF-program: for each ground atom A there is a closed instance of some DEF in the program with A at its head; used to ensure correct evaluation of universally quantified formulas. 89

nesting depth: count of nested occurrences of subterms within a term. 90

configuration: a condition of a TM describing the contents of the tape and the current state. 99

transition relation: describes all possible ways for a TM to proceed in a single step. 99

type: a subset of the function symbols occurring in a program. 113

type partition: partition of all function symbols into types. 114

predicate type assignment: assignment of types to predicate symbols. 114

function type assignment: assignment of types to function symbols. 114

variable type assignment: assignment of types to variables. 115

functionally-determined quantifier: a logical formula denoted by $\boxminus X\colon F \bigwedge G$; true if whenever F determines a unique value x for F, $G(x)$ is true. 125

List of Notation

(In order of appearance)

Υ: set of all variable symbols. 14

$f^{(n)}$: function or predicate with arity n. 14

$T(\Sigma)$: set of finite terms freely-generated from Σ. 14

$\underline{T(\Sigma)}$: set of ground terms in $T(\Sigma)$. 15

$A(\Pi, \Sigma)$: set of atoms generated from predicate symbols in Π and terms in $T(\Sigma)$. 15

$\underline{A(\Pi, \Sigma)}$: set of ground atoms in $A(\Pi, \Sigma)$. 15

Ω_0: all logical operators. 15

$B(C)$: all Boolean expressions freely generated from Ω_0 and some carrier C. 15

$\exists F$: existential closure of formula F abbreviates $\exists X_1 \ldots \exists X_n : F$, where $X_1 \ldots X_n$ are all of the free variables occurring in F. 18

$\forall F$: universal closure of formula F abbreviates $\forall X_1 \ldots \forall X_n : F$, where $X_1 \ldots X_n$ are all of the free variables occurring in F. 18

$\{X_1 = t_1, \ldots, X_n = t_n\}$: finite denotation of a substitution. 18

$\sigma \circ \tau$ (σ and τ substitutions): $x(\sigma \circ \tau) = (x\sigma)\tau$, where x is a term or formula. 20

ϵ: identity substitution. 21

ρ^{-1}: inverse of a renaming substitution ρ. 21

$s \cong t$: terms s and t are variants. 21

$[t]$: set of all all variants of term t. 21

$[s] \gtrsim [t]$: all terms in $[t]$ are instances of all terms in $[s]$. 21

$\tilde{\sqcap} S$: greatest lower bound, w.r.t. $\gtrsim$, of all (variants of) terms S. 22

$\tilde{\sqcup} S$: least upper bound, w.r.t. $\gtrsim$, of all (variants of) terms S. 22

$s \tilde{\sqcap} t$: mgci of terms s and t. 23

$t@\Sigma$: set of ground instances of term t. 24

$(\mathbf{D}, \mathbf{F}_\Sigma, \mathbf{P}_\Pi)$: an interpretation with domain $\mathbf{D}$, mapping $\mathbf{F}_\Sigma$ of function symbols in Σ to functions, and mapping $\mathbf{P}_\Pi$ of predicate symbols in Π to relations. 29

$I[F]$: valuation of a formula F by an interpretation I. 30

HU_P (HU when P understood): The Herbrand Universe of program P. 30

HB_P (HB when P understood): the Herbrand Base of program P. 30

$\mathbf{M}_P$ ($\mathbf{M}$ when P understood): set of all Herbrand models of definite clause program P. 31

T_P (T when P understood): the immediate consequence functional for program P. Also defined for DIF-programs.	32
$\mathbf{X}_P$ ($\mathbf{X}$ when P understood): set of all fixedpoints of T_P.	33
lfp_P (lfp when P understood): least fixedpoint of program P.	33
gfp_P (gfp when P understood): greatest fixedpoint of program P.	33
$T\uparrow\alpha$: abbreviation for $T^{\alpha}(\varnothing)$.	33
$T\downarrow\alpha$: abbreviation for $T^{\alpha}(HB)$.	33
$\Box$: the empty conjunction; a vacuously true formula.	36
$\mathtt{not}\,A$: negation of an atom A to be proved with negation by failure.	42
$comp(P)$: completion of a program.	44
$\sim F$: constructive negation of a formula F.	55
$\mathbf{t}$, $\mathbf{u}$, $\mathbf{f}$: the three logical constants standing for true, undefined, and false, respectively.	57
$c \sqsubset d$: logical constant c is strictly less defined than logical constant d.	57
$c \sqsubseteq d$: logical constant c is less defined than logical constant d.	57
$\{L_1,\ldots,L_n\}$: denotation for a constructive interpretation, where each L_i is a ground literal.	58
$\mathtt{true}$: a distinguished proposition symbol for DEF-programs always assigned to the logical constant $\mathbf{t}$.	75
m_p: the maximum nesting depth of heads of all DEFs in a program defining predicate p.	90
$S\%d$: the set of all atoms from set S with maximum nesting depth at most d.	90
TM: abbreviation for Turing machine.	99
$\vdash_m$: the transition relation for a TM m.	99
$\vdash_m^*$: the reflexive and transitive closure of $\vdash_m$.	99
L_{ne}: all valid encoding of TMs that accept nonempty languages.	101
L_e: all strings that are not valid encodings of TMs, and all strings that are valid encodings of TMs accepting empty languages.	101
dif: used in correct evaluation of equality queries.	106
$\underline{TV}$: an assignment of the empty type to all variables.	115
TP_π (TP when π understood): a predicate type assignment.	115
TF_π (TF when π understood): a function type assignment.	115
TV_π (TV when π understood): a variable type assignment.	115
$\boxminus$: the functionally-determined quantifier.	125

Lists of Figures and Examples

Figure 3.1 36
Figure 3.2 36
Figure 3.3 38
Figure 4.1 67
Figure 4.2 71
Figure 5.1 85

Example 1.1 9
Example 3.1 30
Example 3.2 34
Example 3.3 49
Example 5.1 76
Example 5.2 77
Example 5.3 88
Example 6.1 107
Example 6.2 112
Example 6.3 114

Abstract

Constructive Negation in Logic Programs

Clifford Walinsky, Ph.D.
Oregon Graduate Center, 1987

Supervising Professor: Richard Hamlet

Logic programming languages such as Prolog possess a relatively efficient evaluation procedure but restrict the expressiveness of full predicate logic. Various implementations of negation within logic programming are directed at restoring expressiveness. Negation by failure, the predominant implementation, can be both incorrect and incomplete. Furthermore, negative queries solved by failure do not return answer substitutions as do positive queries. Another implementation of negation, model elimination, is complete but may be as inefficient as resolution. Other implementations have a similar tradeoff between completeness and efficiency.

Constructive negation is an effort to provide negation within logic programming based on ad hoc methods commonly used by programmers to obtain answer substitutions from negative queries. The ad hoc methods involve definition of both positive and negative information with definite clauses to retain efficient evaluation.

While logic programs are described with reference to classical logic, programs incorporating constructive negation must be described by a three-valued logic containing an additional undefined value. Programs with constructive negation may be inconsistent, and syntactic restrictions are needed to ensure consistency. The resulting programs may contain universal quantifiers. An evaluation procedure for universal quantifiers is proposed that under further weak syntactic conditions is correct though necessarily incomplete. Thus, programs incorporating constructive negation are assured to be consistent and have a relatively efficient evaluation procedure.

Chapter 1
Introduction

Logic programming languages such as "pure" Prolog [EK76] are declarative languages possessing a relatively efficient evaluation system. But efficient evaluation of logic programs comes at the cost of expressiveness. Certain logical forms such as negation and universal quantification are absent, so problems naturally containing negation and universal quantification must be transformed in order to conform to the restrictions of logic programming languages.

This dissertation describes a logic-based language that satisfies many concerns about expressiveness. The central new feature of this language is constructive proof of negation, called *constructive negation*. With certain syntactic constraints on programs, a correct evaluation system can be produced with a Prolog interpreter. Use of Prolog benefits from its efficiency and the presence of meta-logical primitives. The Prolog implementation of constructive negation is incomplete, as would be any evaluation system.

Constructive negation allows program definitions of negative as well as positive facts. This idea is already widely practiced on an ad hoc basis. In fact Clark, in an article describing negation by failure [Cl78], provided an excellent example of a negative definition in the following logical formula:

$$\texttt{non-maths-major(X)} \leftarrow \texttt{maths-course(Y)} \wedge \sim\!\texttt{takes(X,Y)}.$$

Our formalization of this practice ensures faithfulness to the logical foundation of logic programming. Formalization also provides guidelines for ensuring consistency and correctness.

This approach is compared with negation by failure, the predominant implementation of negation in logic programming. Constructive negation is not meant to supplant negation by failure entirely. Indeed, in many ways the two interpretations of negation complement each other. Negation by failure can be used to enhance the evaluation system of constructive negation. On the other hand, there are occasions where negation by failure is not sufficient to solve certain problems, leading programmers to adopt constructive negation on an ad hoc basis.

Syntactic constraints ensuring consistency and correctness are proposed for programs with constructive negation. By comparison, correctness of negation by failure is also ensured when certain syntactic constraints are obeyed; however, these constraints greatly reduce expressiveness.

No evaluation system for programs using constructive negation can be complete. But incompleteness exists in current logic programming systems. Prolog, the predominant logic programming language, is an implementation of a complete refutation procedure, called SLD-resolution, that uses an incomplete search strategy. Also, negation by failure is incomplete. Without completeness of an evaluation system, programs must contain "hints" for use by the evaluation system to ensure termination. This is an additional burden on programmers, but is viewed as the cost of providing an efficient evaluation system for an expressive language.

1.1. Declarative Languages

Logic programming languages, such as pure Prolog, fall into the class of declarative languages. These languages are radically different in character from conventional imperative languages, such as Pascal [WJ74]. The natural model for imperative languages is a state machine with addressable memory [HU79]. Input is converted into output through a series of state transformations. Therefore, full comprehension of an imperative program requires knowledge of all states reached during every computation. Techniques that attempt such analysis tend to be either informal, and capable of analysis of fairly large programs, or formal, and capable of analysis of rather small programs [Fa85 (Ch.8)]. The fundamental problem in any rigorous analysis of an imperative program is the vast size of the state-input space.

By contrast, the *underlying model* of a declarative language program is a description of elements from a domain satisfying properties specified in the program. The underlying model is not based on state transition. Context free grammars are an example of a declarative language [HU79]. All strings generated by a context-free grammar satisfy the grammar's specification independent of any notion of state transition.

Decomposability and non-sequentiality aid in determining general properties of an underlying model. Decomposability is apparent in the following grammar rules:

If_Statement → **if** *Expression* **then** *Statement* **else** *Statement*
If_Statement → **if** *Expression* **then** *Statement*

From these rules we can determine the structure of all *If_Statements*. And given strings representing an *Expression* and *Statement*, the grammar rules describe how to compose an *If_Statement*.

Elements of an underlying model reflect the non-sequentiality present in declarative programs. Sequentiality does exist; the above grammar rules describe a sequence of terminal and non-terminal symbols used to construct *If_Statements*. However, sequentiality is not imposed where it is not necessary. Again the grammar rules above demonstrate this fact, because changing the order of the rules has no effect on the underlying model.

The underlying model of a program often cannot be represented explicitly. For example, a context free language may be infinite. *Queries* are posed to an *evaluation procedure* to determine the content of a program's underlying model. Certain evaluation procedures are *recognizers*, used to decide if an element is present within the underlying model. Other evaluation procedures are *transducers*, used to generate elements of the underlying model that satisfy a query. When membership of an underlying model is decidable and the model is enumerable, a recognizer can act as a transducer by working in a generate-and-test manner. In the case of context-free grammars, pushdown automata (PDAs) are recognizers that decide if a given string is generated by a context-free grammar.

An evaluation system may rely on state-transition, as for PDAs. Nonetheless, detailed knowledge of the evaluation system is not essential to understanding its input-output behavior. This behavior is prescribed by the underlying model. Therefore, comprehension of a declarative program does not depend on knowledge of a state-input space or its evaluation procedure.

As increased knowledge about an evaluation system is garnered, evaluation system generators are usually devised. Such generators compile information from programs to produce efficient evaluators. As an example, a PDA generated by Yacc [AJ74] determines if an input string is a member of the language given by an LALR grammar.

Below, various representative examples of declarative languages, their underlying models, evaluation systems, and some notable implementations are listed:

Context-free grammars:

Underlying model: context-free language.

Evaluation system: PDA.

Evaluation system generators: Yacc [AJ74], Sac [Ro85].

Equational programming languages:

Underlying model: least congruence of the initial algebra [ADJ78].

Evaluation system: term-rewriting [Hu80].

Evaluation system generator: Ep [O85].

Implementation: OBJ [GM82].

Relational Algebra:

Underlying model: set of relations.

Evaluation system: operations on sets of tuples [Co70].

Implementations: see [Da86] for PRTV, SQL, Ingres, System R.

Predicate Logic:

Underlying model: all implied formulas [En72].

Evaluation system: resolution [Ro65]; tableaux [Sz69].

Implementation: MRPPS [N80 (Ch.5)].

All of the evaluation systems in this list are transducers, except for PDAs.

For the evaluation systems listed above, three important properties are apparent:

Correctness:

When a *correct* evaluation system terminates successfully in response to a query, it returns a member of the underlying model that satisfies the query or affirms that the query is a member of the underlying model.

Completeness:

In response to a query, a *complete* evaluation system can generate every member in a program's underlying model that satisfies the query. If a language is decidable and enumerable, correctness ensures completeness, since a generate-and-test strategy will generate all members of the underlying model.

Efficiency:

A measure of the evaluation system's speed and memory consumption compared to the length of a query or program.

To illustrate these properties of evaluation systems, consider the language of first-order predicate logic [En72]. From a slightly unconventional viewpoint, all formulas implied by the clausal form of a program (the program's *theory*) are contained in the underlying model. It has been shown that the resolution theorem-proving method [Ro65], the evaluation system for this language, is both correct and complete. However, this evaluation system is generally quite inefficient [Sh86]. At each step of the resolution procedure there may be a number of ways in which execution can proceed.

1.2. Logic Programming

Despite the poor performance of resolution theorem-proving methods, predicate logic remains a desirable language. As a declarative language, it lacks the state-input space problem of imperative languages. And a logic notation seems amenable to many forms of knowledge representation [Ni80], including of course mathematical knowledge [En72].

Efforts to enhance the performance of resolution theorem-proving continue. Impressive performance has been obtained for a restricted form of predicate logic. The restriction permits only *definite* clauses of the form $A_0 \leftarrow A_1 \wedge \cdots \wedge A_n$, where each A_i is an atomic formula and $n \geq 0$. This clause is a statement of implication: $\forall X_1 \cdots \forall X_m : (A_1 \wedge \cdots \wedge A_n \rightarrow A_0)$, where $X_1, \ldots, X_m$ are all of the variables occurring in the A_i. The *head* of a definite clause is the atom A_0, while the *body* is the conjunction $A_1 \wedge \cdots \wedge A_n$. A clause with an empty body is an *assertion.*

For a declarative language of definite clauses, the underlying model is still a program's theory. *SLD-resolution* [Ko74] can be used as an evaluation system for definite clause programs, and is much more efficient than resolution theorem-proving.

In fact the efficient evaluation system has lead to use of the term "logic programming," rather than theorem-proving, for definite clause programs.

The Prolog language uses SLD-resolution. As stated earlier, this implementation of SLD-resolution is incomplete. Therefore, programmers ensure termination only through detailed knowledge of the states of the evaluation system during execution.

The expressiveness of predicate logic is seriously curtailed in logic programming languages. It is not possible to express disjunctive information of the form $A \bigvee B$ with definite clauses. Also, negated atoms are prohibited everywhere within definite clauses.

Because all variables are universally quantified, existentially quantified variables cannot be expressed directly within definite clauses. This is exemplified by the predicate logic formula `∃X:p(X)`. The existential quantifier can be removed through Skolemization [Ni80], producing the formula `p(c)`, where constant `c` occurs nowhere else in the program. Thus, the domain of discourse may expand by an element outside of the program's original domain. Skolemization is justified when all possible domains are to be considered. However, addition of a new element may radically alter the programmer's intended underlying model, so in some applications Skolemization may not be desirable.

Limiting expressiveness can also compromise the non-procedural nature of logic programs, even though the underlying model remains isolated from state considerations. The following logic program defines a predicate `not_divp`(i, j) which is true if integer i does not divide integer j evenly. This definition makes no mention of recursion, yet the program does contain recursion.

Example 1.1

```
    % not_divp(I,J): true if I does not divide J evenly.
    not_divp(I,J) ← p(0,I,J).

    p(X,I,J) ← X > J.
    p(X,I,J) ← X×I≠J ∧ X1=X+1 ∧ p(X1,I,J).
```

Recourse is made to recursion in order to test that every value $x \le j$ when multiplied by i does not equal j. The recursion is artificial, for consider:

```
  % not_divp(I,J): true if I does not divide J evenly.
  not_divp(I,J) ← (∀X: X≤J → X×I≠J).
```

Of course this formula is not in clausal form.

Use of definite clauses requires certain logical statements to be encoded into algorithmic steps, even though a stepwise procedure is not manifested by the logical statements. A programmer must then determine if the encoding has been faithful to the original statement, a process we refer to as the *coding problem*.

1.3. Constructive Negation

Constructive negation is an approach for improving the expressiveness of logic programs. It originally began as an attempt to provide negation within logic programming languages. Gradually the work widened to incorporate all other connectives and quantifiers from standard predicate logic. The distinctive features of this work include:

(1) Negation is based on constructive proof, enabling the evaluation procedure to return answer substitutions for negated queries.

(2) Underlying models of programs with constructive negation may have formulas with neither true nor false valuations.

(3) Evaluation, based on SLD-resolution, is efficient when compared to resolution.

(4) Syntactic restrictions ensure consistency and correctness, but do not restrict expressiveness.

Programs using constructive negation are composed of *Definite Inference Forms* (DIFs). Each DIF is written as $L \leftarrow F$, where L is a literal (a positive or negative atom), and F is a well-formed formula. Constructive negation of a formula F is represented by the formula $\sim F$. DIFs still cannot express disjunctive statements, such as p⋁q, and all variables within the head are always universally quantified. In contrast with definite clauses, bodies of DIFs may contain negation, quantification, and all logical connectives. The following DIFs demonstrate this form:

```
% divp(I,J): true if I divides J evenly; otherwise, false.
  divp(I,J)  ←  ∃X: X≤J ⋀  X×I=J.
 ∼divp(I,J)  ←  ∀X: X≤J → X×I≠J.
```

Recursion need not be introduced artificially, in contrast to Example 1.1.

DIF-programs provide definitions of all positive and negative propositions. It is therefore possible that some propositions will not be assigned a truth value. For example, "nonsense" propositions, such as `divp(0,fred)`, are neither true nor false; they are undefined. To acknowledge this characteristic of DIF-programs, the underlying model is based on three-valued logic, containing *true*, *false*, and *undefined* logical values.

Three-valued logic is weaker than two-valued logic because formulas implied under two-valued logic may not be implied under three-valued logic. Consider the DIF-program below:

```
 p ← q.
~p.
```

For this program, ~q is implied under two-valued logic, but ~q is undefined in three-valued logic. On the other hand, any formula implied under three-valued logic is also implied under two-valued logic.

The evaluation system for DIF-programs is based on SLD-resolution, with a major enhancement to evaluate universally quantified queries. The system retains much of the efficiency of SLD-resolution.

1.4. Overview

Following Chapter 2, describing notation and basic concepts, Chapter 3 reviews current attempts at enhancing the expressiveness of logic programming. Much of this work concerns implementation of negation. The main implementation of negation is by "failure to prove." Negation by failure is not correct for all queries, and is not in general complete. Another implementation of negation, model elimination, is correct and complete, but can be much less efficient than SLD-resolution.

Chapter 4 describes the underlying models of DIF-programs. Strong models, weak models and fixedpoints of a semantic operator are compared. Strong and weak models have undesirable closure properties, while fixedpoints always contain a least element. Therefore, least fixedpoints are chosen as underlying models of programs. The least fixedpoint may be undefined. Programs with undefined least fixedpoint are called *fixedpoint-inconsistent*. To be efficient, an evaluation procedure cannot expend computation resources detecting fixedpoint-consistency. Therefore, fixedpoint-consistency must be detected from the text of a program. But, fixedpoint-consistency is undecidable.

To define a fixedpoint-consistent set of programs, syntactic constraints are explored in Chapter 5. ***DEF-programs*** are statements of equivalence that compile to fixedpoint-consistent DIF-programs. The compiled DIF-programs can, however, contain universal quantifiers, producing noncomputable least fixedpoints. Thus, preventing fixedpoint-inconsistency through syntactic constraints implies incompleteness for any evaluation procedure.

An evaluation system is described for DIF-programs. This system is similar to SLD-resolution. Evaluation of DIF-programs compiled from DEF-programs is correct only when an additional syntactic constraint, ***self-coverage***, is satisfied. Self-coverage implies that each proposition is defined by some DEF.

Chapter 6 describes several enhancements to compilation and evaluation:

(1) The syntactic constraints on DEF-programs may require an extremely large number of DEFs to describe base relations of database-oriented programs and polymorphic programs. The explosion of DEFs is controlled by implementing equality within the evaluation system.

(2) The self-coverage test does not accommodate well-typed programs, because self-coverage requires even ill-typed propositions to be described by DEFs. Self-coverage is therefore generalized to incorporate types, eliminating the need for ill-typed DEFs.

(3) Enhancements widen the scope of universally quantified queries that can be correctly evaluated.

The full implementation of the evaluation system uses C-Prolog, and is described in Chapter 7. The implementation encompasses tests to ensure consistency and correctness. Query evaluation uses meta-programming techniques that treat DIF-program elements as data structures.

Chapter 2
Basic Concepts and Notation

The syntactic structure of terms and predicate logic formulas is first described in the next section. Substitutions and the algebra of substitutions is then introduced. Any term can be used as the denotation for the set of all of its instances resulting in an inclusion ordering of terms, which naturally incorporates unification.

2.1. Syntax of Terms and Formulas

Variables will always be written as strings of alphanumerics, beginning with an uppercase letter, e.g., `Ans1`. Υ will be the set of all variables. Each function symbol will be written as a string of alphanumerics, beginning with a lowercase letter or a numeral, e.g., `f`, `0`. Every function symbol possesses a unique finite *arity*, the number of arguments taken by the function. When necessary, the arity of a function symbol is written as a parenthesized superscript, e.g., $\mathtt{f}^{(2)}$ means function `f` takes two arguments. *Constants* are treated as function symbols with arity zero.

Let Σ be a denumerable set of function symbols. The set $T(\Sigma)$ is the set of finite *terms* freely generated by the function symbols in Σ and the variables of Υ. Therefore, a term in $T(\Sigma)$ is either a constant from Σ, a variable from Υ, or is of the form $f^{(n)}(t_1, \ldots, t_n)$, where $f^{(n)} \in \Sigma$, and each t_i is a term over Σ. For example, if $\mathtt{0}^{(0)}, \mathtt{s}^{(1)}, \mathtt{f}^{(2)} \in \Sigma_1$, the following are terms in $T(\Sigma_1)$: `0`, `s(s(X))`, and `f(Y,s(0))`. When a term contains no variables, it is a *ground* term. The set of all ground terms constructible in $T(\Sigma)$ is written $\underline{T(\Sigma)}$.

Let Π be a denumerable set of predicate symbols. Usually, predicate symbols are distinct from function symbols. Each predicate symbol possesses a unique arity denoted by a superscript. *Propositions* are predicate symbols with arity zero.

The set $A(\Pi, \Sigma)$ is the set of atomic formulas (*atoms*) generated by the predicate symbols in Π and the terms in $T(\Sigma)$. An atom in $A(\Pi, \Sigma)$ is either a proposition symbol, or is of the form $p^{(n)}(t_1, \ldots, t_n)$, where $p^{(n)} \in \Pi$ and each term t_i is in $T(\Sigma)$. For example, with $\texttt{lt}^{(2)} \in \Pi_1$ and Σ_1 as given previously, `lt(X,s(0))` is in $A(\Pi_1, \Sigma_1)$. Those atoms from $A(\Pi, \Sigma)$ without variables are *ground atoms*, denoted $\underline{A(\Pi, \Sigma)}$.

The set Ω_0 consists of the logical operators $\neg$, $\bigwedge$ and $\bigvee$, designating negation, conjunction and disjunction, respectively. Given some set C, the carrier, a Boolean algebra is $B(C)$, the set of *Boolean expressions* formed from the elements of the carrier and the logical operators in Ω_0. Thus, any element $x \in C$ is a Boolean expression. If x is a Boolean expression, then $\neg x$ is also a Boolean expression. And if S is a set of Boolean expressions, both $\bigwedge S$ and $\bigvee S$ are themselves Boolean expressions. Use of an unordered set of Boolean expressions is justified because conjunction and disjunction are commutative. Conjunction of an empty set will be considered a vacuously true formula, and disjunction of an empty set will be considered false. (This inversion of the customary meaning attributed to conjunction and disjunction of empty sets comes about because such formulas will arise only in queries, which are implicitly negated.)

A logical *formula* is a Boolean expression in $B(A(\Pi, \Sigma))$, for some sets Π and Σ of predicate and function symbols, respectively.

Boolean expressions are not usually written with the logical operators. Instead, *connectives* are defined from abbreviations for Boolean expressions. The set of connectives and their meaning is presented in the following table:

Logical Connectives	
Form	Meaning
$F \bigwedge G$	Finite conjunction
$F \bigvee G$	Finite disjunction
$F \rightarrow G$	Implication
$F \leftrightarrow G$	Equivalence
$\exists X : F$	Existential quantification
$\forall X : F$	Universal quantification

When general conjunction and disjunction are formed over finite sets of expressions, conventional infix form can be used. For example,

```
lt(s(0),X) ⋀ lt(X,s(s(s(0))))
```

is an abbreviation for:

```
⋀( {lt(s(0),X), lt(X,s(s(s(0))))} ).
```

Implication and equivalence have the usual definitions, using finite conjunction and disjunction.

Implication: $F_1 \rightarrow F_2$ abbreviates $\neg F_1 \bigvee F_2$

Equivalence: $F_1 \leftrightarrow F_2$ abbreviates $(F_1 \rightarrow F_2) \bigwedge (F_2 \rightarrow F_1)$.

In writing definite clause forms, it is customary to reverse the direction of implication.

Quantifiers can be viewed as finite denotations for conjunction and disjunction of infinite sets of formulas. Existentially and universally quantified formulas are written as $\exists X{:}F$ and $\forall X{:}F$, respectively, where X is the quantified variable in both formulas.

The presence of quantified variables requires scoping rules. A variable X is *free* at any occurrence of X in an atom. If X occurs free in formula F then X also occurs free in $\neg F$, $\exists Y{:}F$, and $\forall Y{:}F$, where $X \neq Y$. If X occurs free in either formula F or G, X also occurs free in $F \bigwedge G$, $F \bigvee G$, $F \rightarrow G$, and $F \leftrightarrow G$. Every free occurrence of X in formula F is *bound* in quantified formulas $\exists X{:}F$ and $\forall X{:}F$. For example, `X` occurs free in `lt(s(0),X)` and:

`(lt(s(0),X)`$\bigwedge$`lt(X,P))` $\rightarrow$ $\neg$`divp(X,P)`,

but all occurrences of `X` are bound in the formula below:

$\forall$`X:((lt(s(0),X)`$\bigwedge$`lt(X,P))` $\rightarrow$ $\neg$`divp(X,P))`. (1)

Existentially and universally quantified formulas are finite denotations for infinite disjunctive and conjunctive Boolean expressions.

Existential quantification: $\exists X{:}F$ abbreviates $\bigvee\{F(t) \mid t \in \underline{T(\Sigma)}\}$

Universal quantification: $\forall X{:}F$ abbreviates $\bigwedge\{F(t) \mid t \in \underline{T(\Sigma)}\}$,

where $F(t)$ means term t replaces every free occurrence of variable X in F.

Each connective has a binding precedence that eliminates overuse of parentheses. Negation has highest precedence, followed by conjunction, disjunction, implication and equivalence, and finally the quantifiers. For example, the following formula:

```
∀X:lt(s(0),X)⋀lt(X,P)  →  ¬divp(X,P)
```

is equivalent to formula (1), above.

A *closed* formula, or *sentence*, contains no free variables. An *open* formula contains at least one free variable. Formula (1) is open, because it contains the free variable `P`.

The *existential closure* of a formula, denoted $\exists F$, binds every free variable in F with an existential quantifier. So $\exists F$ is an abbreviation for the sentence $\exists X_1 \cdots \exists X_n : F$, where $X_1 \cdots X_n$ are all of the free variables occurring in F. As seen by expanding the abbreviation, the meaning of $\exists X_1 \cdots \exists X_n : F$ is independent of the ordering chosen for existential quantifiers. Similarly, the *universal closure* of a formula $\forall F$ abbreviates $\forall X_1 \cdots \forall X_n : F$.

2.2. Substitutions

The following discussion uses notation and concepts from Eder [Ed85]. A *substitution* over Σ is a mapping from variables in Υ to terms in $T(\Sigma)$. The *domain* of a substitution is the set of variables that are not mapped to themselves. When the domain of a substitution is finite, the substitution can be fully expressed in written form as a set of pairs $X = t$, meaning that variable X is replaced by term t, where X is a variable from the domain.

Any substitution σ can be naturally extended to a mapping $\overline{\sigma}$ on formulas in $B(A(\Pi,\Sigma))$ and terms in $T(\Sigma)$, where Π and Σ are sets of predicate and function symbols, respectively. In this extension, conventional notation for application of substitutions is adopted. When x is a formula or term and $\overline{\sigma}$ a substitution, $x\,\overline{\sigma}$ denotes application of $\overline{\sigma}$ to x, defined as follows:

$$X\overline{\sigma} = \sigma(X) \text{ for } X \in \Upsilon$$

$$f^{(n)}(t_1, \ldots, t_n)\overline{\sigma} = f^{(n)}(t_1\overline{\sigma}, \ldots, t_n\overline{\sigma}) \text{ for } n \geq 0 \text{ and } f^{(n)} \in \Sigma$$

$$p^{(n)}(t_1, \ldots, t_n)\overline{\sigma} = p^{(n)}(t_1\overline{\sigma}, \ldots, t_n\overline{\sigma}) \text{ for } n \geq 0 \text{ and } p^{(n)} \in \Pi$$

$$(\neg x)\sigma = \neg(x\,\sigma)$$

$$(\bigwedge\{x_1, x_2, \cdots\})\sigma = \bigwedge\{x_1\sigma, x_2\sigma, \cdots\}$$

$$(\bigvee\{x_1, x_2, \cdots\})\sigma = \bigvee\{x_1\sigma, x_2\sigma, \cdots\}$$

Application of a substitution to an abbreviation of a formula can produce an abbreviation, but the resulting abbreviation must be equivalent to application of the substitution to the unabbreviated formula. Therefore, application of substitutions distribute over finite conjunction, disjunction, implication and equivalence:

$$(x \bigwedge y)\sigma = (x\,\sigma) \bigwedge (y\,\sigma)$$

$$(x \bigwedge y)\sigma = (x\,\sigma) \bigwedge (y\,\sigma)$$

$$(x \rightarrow y)\sigma = (x\,\sigma) \rightarrow (y\,\sigma)$$

$$(x \leftrightarrow y)\sigma = (x\,\sigma) \leftrightarrow (y\,\sigma)$$

Also, as the rules below describe, application of a substitution to a quantified formula cannot alter occurrences of bound variables.

$$(\exists X{:}F)\overline{\sigma} = \exists X{:}F\,\overline{\sigma'} \text{ where } \sigma'(Y) = \begin{cases} \sigma(Y) & \text{for } X \neq Y \\ Y & \text{for } X = Y \end{cases}$$

$$(\forall X{:}F)\overline{\sigma} = \forall X{:}F\,\overline{\sigma'} \text{ where } \sigma'(Y) = \begin{cases} \sigma(Y) & \text{for } X \neq Y \\ Y & \text{for } X = Y \end{cases}$$

For example, suppose that `X` and `U` are distinct variables; `0`, `a` and `c` are constants; `f` and `s` are function symbols; `lt`, `mult`, and `add` are predicate symbols; and $\sigma(\mathtt{X}) = \mathtt{a}$. Then the following equalities hold:

(1) $\mathtt{X}\overline{\sigma} = \mathtt{a}$.

(2) $\mathtt{U}\overline{\sigma} = \mathtt{U}$.

(3) $\mathtt{f(X,c,U)}\,\overline{\sigma} = \mathtt{f(a,c,U)}$.

(4) $\mathtt{lt(s(0),X)}\,\overline{\sigma} = \mathtt{lt(s(0),a)}$.

(5) $\mathtt{[(\exists X{:}mult(0,s(0),X))} \bigwedge \mathtt{add(X,s(0),s(s(0)))]}\overline{\sigma} =$

$\mathtt{(\exists X{:}mult(0,s(0),X))} \bigwedge \mathtt{add(a,s(0),s(s(0)))}$.

When no confusion results, σ will be used in place of $\overline{\sigma}$.

Substitutions σ and τ may be composed, forming a new substitution as follows: $x\,(\sigma \circ \tau) = (x\,\sigma)\,\tau$, for all formulas and terms x. For example, if $\sigma(\mathtt{X}) = \mathtt{f(Y)}$ and $\tau(\mathtt{Y}) = \mathtt{a}$, then $\mathtt{s(X)}(\sigma \circ \tau) = \mathtt{s(f(a))}$. When substitutions σ and τ are composed, the resulting function is always a substitution η such that $\eta(X) = X\,(\sigma \circ \tau)$ for all variables X.

Since composition always results in a substitution, composition of substitutions can be shown to be associative. Consider any term or formula x. For arbitrary substitutions ρ, σ and τ, $x\,(\rho \circ (\sigma \circ \tau)) = ((x\,\rho)\sigma)\tau$. Since $\rho \circ \sigma$ and $(\rho \circ \sigma) \circ \tau$ are substitutions, $((x\,\rho)\sigma)\tau = x\,((\rho \circ \sigma) \circ \tau)$.

There is a unique *identity* substitution ϵ, such that $\epsilon(X) = X$ for all variables X. Therefore, $x\,\overline{\epsilon} = x$ for any term or formula x. The identity substitution forms a left and right identity with respect to composition.

A *renaming* is a bijective substitution. A renaming cannot map any variable to a non-variable. A renaming also does not introduce additional constraints between variables. For example, {X=U, Y=V} is a renaming, while {X=U, Y=U} is not a renaming because it is not 1-1. Every renaming ρ has an inverse ρ^{-1}, such that $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = \epsilon$.

2.3. Common Instances of Terms

Terms s and t are *variants* if ρ is a renaming and $s\,\rho = t$. Denote by $s \cong t$ the fact that terms s and t are variants.

Lemma 2.1: $\cong$ is an equivalence relation on terms.

Proof: Trivial.

The equivalence class of a term t determined by the renaming relation $\cong$ will be denoted $[t]$. Note that $[t]$ contains only t when t is a ground term.

A term t is an *instance* of a term s if there is a substitution σ such that $s\,\sigma = t$. For example, f(a,Y,U) is an instance of f(X,Y,Z).

Now define a relation $[s] \gtrsim [t]$ on the equivalence classes of terms s and t, to mean that every term in $[t]$ is an instance of every term in $[s]$. Therefore, $[s] \gtrsim [t]$ if for every variant s' of s and t' of t there is a substitution σ' such that $s'\sigma' = t'$. The number of variants of a term is usually infinite, so directly deciding if $[s] \overset{\sim}{>} [t]$ may be difficult. Fortunately, testing for $[s] \overset{\sim}{>} [t]$ reduces to finding just one substitution σ such that $s\,\sigma = t$.

Lemma 2.2: For any terms s and t, $[s] \overset{\sim}{>} [t]$ iff there is a substitution σ such that $s\,\sigma = t$.

Proof: Trivial.

The equivalence relation $\cong$ is a subset of $\overset{\sim}{>}$. When terms s and t are variants, $[s] \overset{\sim}{>} [t]$ and $[t] \overset{\sim}{>} [s]$. Also, the following result holds.

Lemma 2.3: The relation $\overset{\sim}{>}$ is a partial ordering on equivalence classes of terms.

Proof:

<u>Reflexivity</u>: For all terms s, $[s] \overset{\sim}{>} [s]$, since $s\,\epsilon = s$.

<u>Asymmetry</u>: Suppose $[s] \overset{\sim}{>} [t]$ and $[t] \overset{\sim}{>} [s]$. Then $s\,\sigma = t$ and $t\,\tau = s$ for some substitutions σ and τ. If s and t are not variants, either (i) some variable X occurs in s and $X\sigma$ is not a variable or (ii) distinct variables X and Y occur in s and $X\sigma = Y\sigma$. In case (i), if $X\sigma$ is not a variable, there is no substitution τ such that $X(\sigma \circ \tau) = X$. In case (ii), if $X\sigma = Y\sigma$, there is no substitution τ such that $X(\sigma \circ \tau)$ is distinct from $Y(\sigma \circ \tau)$. Therefore, s and t must be variants, and $s \cong t$.

<u>Transitivity</u>: Suppose $[s] \overset{\sim}{>} [t]$ and $[t] \overset{\sim}{>} [u]$. Then $s\,\sigma = t$ and $t\,\tau = u$ for some substitutions σ and τ. Since $s\,(\sigma \circ \tau) = u$, $[s] \overset{\sim}{>} [u]$. □

The *greatest lower bound* $\overset{\sim}{\sqcap} S$ and *least upper bound* $\overset{\sim}{\sqcup} S$ of equivalence classes of terms S is defined with respect to the ordering $\overset{\sim}{>}$. When s is a variable, $[s] \overset{\sim}{>} [t]$

for all terms t, so $\widetilde{\bigsqcup} S$ is defined for every set S. However, consider two distinct constants c and d. There is no term l for which $[c] \gtrsim [l]$ and $[d] \gtrsim [l]$, so $\widetilde{\bigsqcap} S$ may not be defined for some sets S.

A *common instance* of terms s and t is an equivalence class $[u]$, where $[s] \gtrsim [u]$ and $[t] \gtrsim [u]$.

Lemma 2.4: Let $C(s,t)$ be the set of common instances for terms s and t. When $C(s,t)$ is nonempty, $\widetilde{\bigsqcup} C(s,t) \cong s \,\widetilde{\sqcap}\, t$.

Proof: Trivial.

Therefore, if s and t have a common instance, the *most general common instance* (mgci) of terms s and t is defined to be $s \,\widetilde{\sqcap}\, t$. Since $s \,\widetilde{\sqcap}\, t$ is in fact an equivalence class, the mgci is unique up to renaming.

Terms s and t are *unifiable* if there is a *unifying substitution* σ such that $s\,\sigma = t\,\sigma$. The next result demonstrates that unifiability of terms is equivalent to determining if the terms have a common instance.

Lemma 2.5: If terms s and t have no variables in common, then s and t are unifiable iff the terms have a common instance.

Proof:

($\rightarrow$) The term $s\,\sigma$ is a common instance of s and t, so when terms are unifiable they have a common instance.

($\leftarrow$) Next, assume terms s and t have a common instance. Then $s\,\sigma = t\,\tau$ for certain substitutions σ and τ, and since s and t have no variables in common $\sigma \circ \tau = \tau \circ \sigma$. The substitution $\sigma \circ \tau$ can serve as a unifying substitution for s and t. □

Robinson's unification algorithm [Ro65] is guaranteed to find a unifying substitution that produces an mgci of terms s and t if:

(1) s and t have no variables in common; and

(2) s and t have a common instance.

The first constraint can be readily achieved. Since the presence of a common instance of terms s and t is independent of the variables within the terms, terms s and t can be renamed to satisfy constraint (1).

Unification requires an occurs check to insure that infinite terms will not be mgci's. Due to efficiency concerns, Prolog implementations typically omit this check. Hence, determination of unifiability may be made by an implementation when in fact the terms are not unifiable.

When terms have a common instance, by Lemma 2.5 they are unifiable, and the unique mgci (modulo renaming) leads to a decomposition of the unifying substitution, also called a weak unifier [Ed85]. Consider terms s and t containing disjoint sets of variables V and W. If s and t have an mgci $[m]$, a *decomposition* of the mgci consists of substitutions σ and τ, where $s\,\sigma = m = t\,\tau$ and the domains of σ and τ are V and W, respectively.

2.4. Terms as Denotations for Sets

A term t containing variables will commonly be used to denote the set of ground instances of t. Define $t \,@\, \Sigma$ to be the set of terms $t\,\sigma$ that are ground instances of t. For example, `f(X,s(X))` $@ \{0^{(0)}, s^{(1)}\}$ contains `f(0,s(0))`, `f(s(0),s(s(0)))`, etc. Term t is the *template* of $t @ \Sigma$.

As special cases, $t @ \Sigma = \{t\}$ when t is a ground term, and $X @ \Sigma = \underline{T(\Sigma)}$ when X is a variable. If Σ contains at least one constant, then $t @ \Sigma \neq \emptyset$ for any term t.

A set of function symbols Σ is *non-trivial* if $|\underline{T(\Sigma)}| > 1$. Non-triviality of Σ is ensured if $|\Sigma| > 1$ and Σ contains at least one constant.

Use of non-trivial sets of function symbols will determine that variables occurring within template terms serve only as placeholders. The actual variable names used within a template term should not affect the set of ground instances of the template. For example, $(\texttt{f(X,Y)} @ \Sigma) = (\texttt{f(U,V)} @ \Sigma)$, for all "reasonable" sets Σ of function symbols. There are sets Σ for which terms s and t may not be variants, yet $s @ \Sigma = t @ \Sigma$. As Lemma 2.7 demonstrates, this occurs only when Σ is empty or contains only one constant. Consider, for example, $\Sigma = \{\texttt{c}\}$ with templates `X` and `c`. These templates are not variants, yet $\texttt{X} @ \Sigma = \texttt{c} @ \Sigma$. To avoid this anomaly, Σ should be non-trivial. We first demonstrate a result used several times within this section.

Lemma 2.6: For all terms s and t, if $[s] \gtrsim [t]$ then $(s @ \Sigma) \supseteq (t @ \Sigma)$.

Proof: Consider any term $t\tau \in t @ \Sigma$. There must be some substitution σ such that $s\sigma = t$, since $[s] \gtrsim [t]$. So $t\tau = s(\sigma \cdot \tau) \in (s @ \Sigma)$. □

The next result demonstrates that terms must be variants if they generate identical sets of ground terms.

Lemma 2.7: When Σ is non-trivial, for all terms s and t, $s @ \Sigma = t @ \Sigma$ iff $s \cong t$.

Proof:

($\rightarrow$) The contrapositive is demonstrated: $s \ncong t$ implies $s @ \Sigma \neq t @ \Sigma$. If s and t are not unifiable, they have no common instance so $(s @ \Sigma) \cap (t @ \Sigma) = \varnothing$. Since $|\underline{T(\Sigma)}| > 1$, $s @ \Sigma$ and $t @ \Sigma$ are nonempty, so $s @ \Sigma \neq t @ \Sigma$.

If s and t are unifiable, let $m \cong s \tilde{\sqcap} t$. Using decomposition, there are substitutions σ and τ such that $s\,\sigma = m = t\,\tau$, where either σ or τ is not a renaming substitution. Without loss of generality, assume σ is not a renaming. Either (i) s contains a variable X and $X\sigma$ is not a variable, or (ii) s contains variables X and Y and $X\sigma = Y\sigma$. In case (i) let $\sigma' = \{X = u\}$, and in case (ii) let $\sigma' = \{X = X\sigma, Y = u\}$, where $u \neq X\sigma$. Term u is guaranteed to exist because $|\underline{T(\Sigma)}| > 1$. By Lemma 2.6, $(s\,\sigma' @ \Sigma) \subseteq (s @ \Sigma)$, and $(m @ \Sigma) \subseteq (t @ \Sigma)$. But by the construction of σ', $(s\,\sigma' @ \Sigma) \cap (m @ \Sigma) = \varnothing$. Since this holds for all $m \cong s \tilde{\sqcap} t$, $s @ \Sigma \neq t @ \Sigma$.

($\leftarrow$) Suppose $s \cong t$. Then both $[s] \gtrsim [t]$ and $[t] \gtrsim [s]$. By Lemma 2.6, both $s @ \Sigma \supseteq t @ \Sigma$ and $t @ \Sigma \supseteq s @ \Sigma$. Hence, $s @ \Sigma = t @ \Sigma$. □

Under very loose restrictions, we have established a correspondence between sets of ground terms and their templates. When sets of ground instances of templates are equal, the templates must be variants. The converse is directly implied by Lemma 2.6. Henceforth, we will assume that every set of function symbols is non-trivial, so that Lemma 2.7 holds. As a consequence of this lemma, equivalence classes can be used as templates: $[t] @ \Sigma$ denotes $t' @ \Sigma$, where t' is any representative of $[t]$.

Next, we demonstrate a connection between the most general common instance and sets of ground terms.

Lemma 2.8:

(1) If terms s and t have no common instance, then $s @ \Sigma$ and $t @ \Sigma$ are disjoint.

(2) If terms s and t have a common instance, then $((s \tilde{\sqcap} t) @ \Sigma) = (s @ \Sigma) \cap (t @ \Sigma)$.

Proof:

(1) Trivial.

(2) ($\subseteq$) Consider a term $x \in ((s \tilde{\sqcap} t) @ \Sigma)$. Then $[s] \gtrsim [x]$ and $[t] \gtrsim [x]$, so $x \in (s @ \Sigma)$ and $x \in (t @ \Sigma)$.

($\supseteq$) Consider a term $x \in (s @ \Sigma)$ and $x \in (t @ \Sigma)$. Then $[x]$ is a unifier of s and t. By definition, $s \tilde{\sqcap} t \gtrsim [x]$. Hence, $x \in ((s \tilde{\sqcap} t) @ \Sigma)$, by Lemma 2.6. □

When two terms have no variables in common, by Lemma 2.5 the terms have a common instance if and only if they are unifiable. So Lemma 2.8 also states that if terms s and t have no variables in common and the terms are unifiable, then $((s \tilde{\sqcap} t) @ \Sigma) = (s @ \Sigma) \cap (t @ \Sigma)$.

Finally, an inclusion relationship can be drawn between sets of ground terms and the relation $\gtrsim$ between equivalence classes of template terms.

Lemma 2.9: For all terms s and t, $(s @ \Sigma) \supseteq (t @ \Sigma)$ iff $[s] \gtrsim [t]$.

Proof:

($\rightarrow$) Since $(t @ \Sigma) \subseteq (s @ \Sigma)$, $(s @ \Sigma) \cap (t @ \Sigma) = t @ \Sigma$, which is nonempty.

Therefore, $s\,@\,\Sigma$ and $t\,@\,\Sigma$ are not disjoint, and s and t are unifiable, by Lemma 2.8. Lemma 2.4 ensures that $s\,\tilde{\sqcap}\,t$ exists. By Lemma 2.8, $(s\,\tilde{\sqcap}\,t)\,@\,\Sigma =$ $(s\,@\,\Sigma) \cap (t\,@\,\Sigma) = t\,@\,\Sigma$. Therefore, $s\,\tilde{\sqcap}\,t \cong t$, by Lemma 2.7, and $[s] \gtrsim [t]$.

($\leftarrow$) Demonstrated in Lemma 2.6. □

Chapter 3
Enhancing Expressiveness of Logic Programming

Many approaches toward enhancing the expressiveness of logic programming have been proposed. Almost all center on implementing negation. To appreciate the implementation strategies, it is necessary first to review SLD-resolution, an efficient evaluation system for definite clause programs. Negation by failure, the most predominant implementation of negation, is based on detecting failure of query evaluation.

3.1. Underlying Model of Definite Clause Programs

To recall the discussion of Chapter 1, the underlying model of a definite clause program is the program's theory, i.e., the set of all conjunctive formulas implied by the program. For definite clause programs, the underlying model can be derived from a unique minimal model. The results below are presented in more detail in [L82].

Consider a program P, a member of Boolean algebra $B(A(\Pi,\Sigma))$. An *interpretation* of P is a triple $(\mathbf{D},\mathbf{F}_\Sigma,\mathbf{P}_\Pi)$, where $\mathbf{D}$ is a nonempty domain, $\mathbf{F}_\Sigma$ is a mapping of function symbols and constants from Σ into functions and constants over the domain $\mathbf{D}$, and $\mathbf{P}_\Pi$ is a mapping of predicate and proposition symbols from Π into relations on the domain $\mathbf{D}$.

Consider the program below:

Example 3.1

```
% lt(I,J): true if I < J.
lt(0,s(J)).
lt(s(I),s(J)) ← lt(I,J).
```

One interpretation for this program could be $I_1 = (\mathbf{N}, \mathbf{F}_{\Sigma_1}, \mathbf{P}_{\Sigma_1})$, where $\mathbf{N}$ is the set of natural numbers, $\mathbf{F}_{\Sigma_1}$ translates `0` to the number 0 and `s` to the successor function, and $\mathbf{P}_{\Sigma_1}$ translates `lt` to the binary relation $\{<x,y> \mid x < y\}$.

Interpretations are used to assign a truth valuation to formulas. The valuation is based on classical logic. If I is an interpretation and F a formula, $I[F]$ denotes the valuation of F by I. Computation of this valuation is described by Enderton [En72], for example. The valuation of interpretation I on program P is $I[P]$. I_1 places a true valuation on the program of Example 3.1.

Assume that a program P is from $B(A(\Pi, \Sigma))$. A *Herbrand interpretation* is a symbolic interpretation with $\mathbf{D}$ and $\mathbf{F}_\Sigma$ fixed. The domain $\mathbf{D} = \underline{T(\Sigma)}$ is called the *Herbrand Universe* (HU_P). $\mathbf{F}$ maps every function symbol $f^{(n)} \in \Sigma$ and n-tuple of terms $(t_1, \ldots, t_n) \in HU^n$ to the individual $f(t_1, \ldots, t_n)$ of $\mathbf{D}$. The *Herbrand Base* (HB_P) of program P is equivalent to $\underline{A(\Pi, \Sigma)}$. When program P is understood, subscripts on HU and HB will be omitted. Since Herbrand interpretations are symbolic, any Herbrand interpretation can be written by listing only the subset of ground atoms in HB that are true in the interpretation. For example, interpretation I_1, above, can be represented with an Herbrand interpretation HI_1 consisting of all atoms `lt`$(s^i(0), s^j(0))$ such that $i < j$.

Any two interpretations I and J are *equivalent* if $I[F] = J[F]$ for all sentences F. The following result demonstrates that Herbrand interpretations are adequate to represent all interpretations. Within this chapter, if a lemma or theorem has a reference, its proof is contained in the referenced source.

Lemma 3.1: Any interpretation for a definite clause program has an equivalent Herbrand interpretation [EK76].

Interpretation M is a *model* for a program P if $M[P]$ is true. For definite clause programs, a *query* is a conjunction of atoms. A query Q is *logically implied* by a program if $\exists Q$ is true in all models. The following property reduces this test to a single model. Let $\mathbf{M}_P$ (or just $\mathbf{M}$ when P is understood from context) be the class of all Herbrand models for a program P.

Lemma 3.2 (Model Intersection [EK76]): $\mathbf{M}$ is closed with respect to $\cap$, i.e., $\cap\mathbf{M} \in \mathbf{M}$.

For example, it can be shown that HI_1 is the least Herbrand model for the program of Example 3.1.

According to the following result, the least Herbrand model of a program can derive the program's theory, the set of all logically implied queries.

Lemma 3.3 (Logical Implication [L82 (Thm. 7.1)]): A query Q is logically implied by a definite clause program P iff $\exists Q$ is true in P's least Herbrand model.

The query:

$Q =$ `lt(s(0),X)` $\bigwedge$ `lt(X,s(s(s(0))))`

is logically implied by Example 3.1, since `lt(s(0),s(s(0)))` and `lt(s(s(0)),s(s(s(0))))` are contained in the program's least model, HI_1. Thus `X = s(s(0))` establishes $\exists X: Q$.

For any particular definite clause program, there is a lattice of Herbrand interpretations, ordered by set inclusion. This lattice is complete: every set of Herbrand interpretations has a least upper bound and a greatest lower bound, where these elements are computed by set union and intersection, respectively. The minimal element of the lattice is the empty interpretation, $\varnothing$. The maximal element is HB.

The *immediate consequence* functional T_P (or just T when P is understood from context) is a mapping from Herbrand interpretations to Herbrand interpretations, defined as follows:

$A \in T(I)$ iff

A is a ground instance of an assertion,

or there is a ground instance $A \leftarrow A_1 \bigwedge \cdots \bigwedge A_n$ of a clause and

$I[A_1 \bigwedge \cdots \bigwedge A_n]$ is true, i.e., $\{A_1, \ldots, A_n\} \subseteq I$.

For any Herbrand interpretation I, powers of T can be computed as follows:

$T^0(I) = I$

$T^{k+1}(I) = T(T^k(I))$ for all successor ordinals $k+1$

$$T^{\beta}(I) = \bigcup_{\alpha<\beta} T^{\alpha}(I) \text{ for all limit ordinals } \beta$$

A fixedpoint of T is an Herbrand interpretation I for which $I = T(I)$. Let $\mathbf{X}_P$ be the class of fixedpoints of T_P. Since T is a monotonic mapping with respect to the inclusion ordering of Herbrand interpretations, the Knaster-Tarski Theorem [Ta55] ensures completeness for the lattice of this class of fixedpoints. In particular unique minimal and maximal elements exist. Let $lfp_P = \sqcap\mathbf{X}_P$ be the *least* fixedpoint of P, and let $gfp_P = \sqcup\mathbf{X}_P$ be the *greatest* fixedpoint. As usual, when program P is understood from context, the subscripts on $\mathbf{X}$, lfp and gfp are omitted. To compute fixedpoints of programs, define the following distinguished interpretations for all ordinals α:

$$T\uparrow\alpha = T^{\alpha}(\varnothing)$$
$$T\downarrow\alpha = T^{\alpha}(HB)$$

Lemma 3.4 (Characterization of Fixedpoints [L82 (Thm. 5.2)]): When P is a definite clause program:

(1) There exist ordinals n_1 and n_2 such that $p \geq n_1$ implies $T\uparrow p = lfp$ and $q \geq n_2$ implies $T\downarrow q = gfp$.

(2) $lfp = T\uparrow\omega$, where ω is the cardinality of the natural numbers.

(3) $gfp \subseteq T\downarrow\omega$.

The following example (from [AE82]) provides a program for which the inclusion of (3) above is proper.

Example 3.2

```
p(a) ← p(X) ∧ q(X).
p(s(X)) ← p(X).
q(b).
q(s(X)) ← q(X).
```

In this program $gfp = T\downarrow(\omega+\omega) = \{q(s^i(b)) \mid i \geq 0\}$, while $T\downarrow\omega =$ $\{q(s^i(b)) \mid i \geq 0\} \cup \{p(s^i(a)) \mid i \geq 0\}$.

Lemma 3.5 (Characterization of Least Model by T [EK76]): When P is a definite clause program, $lfp = \cap\mathbf{M}$.

In summary, Herbrand interpretations are symbolic representations of all interpretations. For definite clause programs, logical implication of all queries by the least Herbrand model of a program is equivalent to logical implication by the program (Lemma 3.3). And determination of the least model requires a finite number of iterations of the T functional (Lemma 3.4). However, infinite iterations may be necessary to produce the greatest fixedpoint. Since negation by failure will rely on the greatest fixedpoint for meaning (Section 3.3.2), negation by failure is generally an incomplete evaluation procedure.

3.2. SLD-Resolution

SLD-resolution is a relatively efficient evaluation procedure for definite clause programs. It is a specialization of resolution [Ro65]. But because of the restriction to definite clauses, many efficiencies are attained by SLD-resolution over resolution. SLD-resolution is a linear-input resolution strategy [Ni80]. Linear-input resolution resolves the initial query with a program clause to form a new query. Each new query is again resolved with a program clause. Because queries are never resolved with previously obtained queries, evaluation is focused on a deduction from the initial query and not on other unrelated proofs. For general clause programs, this strategy is incomplete. But as stated in Lemma 3.8, below, this strategy is complete for definite clause programs. Finally, resolvents can be formed in a last-in first-out manner, permitting efficient construction and access of data structures representing resolvents.

Given a query Q, SLD-resolution determines an *answer substitution* of values for variables occurring within Q. The nature of the procedure ensures correctness: whenever α is an answer substitution for a query Q, $\forall(Q\,\alpha)$ is logically implied by the program. Further, SLD-resolution is complete: if $\exists Q$ is logically implied by the program, Q will execute successfully. On the other hand, if $\exists Q$ is not logically implied, termination of the procedure is not guaranteed.

Implicitly, SLD-resolution constructs a full search tree for a query and then a success path is found within the tree. In fact it is not necessary to represent the entire search tree within any actual implementation. The search tree is developed while searching for a success path, so that the procedure can terminate even if its full search tree is infinite. The manner in which full search trees are traversed affects completeness of the implementation. Prolog implementations are incomplete, since full search trees are developed in depth-first order. This may lead to non-termination, though success paths would be present in other branches of the full tree. On the other hand, no other search strategy can find a success path faster than depth-first search.

Description of a well-formed full search tree is based on the structure of a query. A full search tree can be depicted as nodes labeled by queries (conjunctions of atoms), and directed edges labeled by substitutions. The full search tree for a query Q is the full search tree whose root is labeled Q.

A full search tree consisting only of a node labeled by the empty conjunction, denoted $\square$, is well-formed. The empty conjunction designates a vacuously true formula. Otherwise, the root node of the tree is labeled by a nonempty conjunction of atoms $A \bigwedge C$, where C is a conjunction of atoms. A *variant* of a clause is the result of applying a renaming to the clause so that variables in the clause will be distinct from all others in use. Collect variants of all clauses from the program $A_1 \leftarrow C_1, \ldots, A_n \leftarrow C_n$ for which the selected atom A unifies with each A_i $(1 \leq i \leq n)$. There is a unique mgci (modulo renaming) for each pair A and A_i. Section 2.3 describes a unique decomposition of the mgci producing substitutions σ_i and τ_i such that $A\,\sigma_i = A_i\,\tau_i$. Suppose each query $(C_i\,\tau_i) \bigwedge (C\,\sigma_i)$ $(1 \leq i \leq n)$ has a well-formed full search tree. Then the full search tree of Figure 3.1 is well-formed.

As an example of this evaluation procedure, consider the program of Example 3.1 and the query `lt(s(0),X)`$\bigwedge$`lt(X,s(s(0))))`. A full search tree for this query is presented in Figure 3.2:

A *success* node in a full search tree is any leaf labeled $\square$, indicating that no atoms are left to be resolved. A *success path* is a path from the root of the tree to a success node. The *value* of a success path is derived from composition of the labels of

Well-Formed Full Search Tree

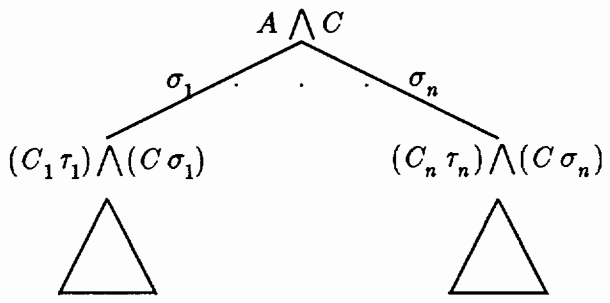

$A \wedge C$
σ_1
σ_n
$(C_1 \tau_1) \wedge (C \sigma_1)$
$(C_n \tau_n) \wedge (C \sigma_n)$

Figure 3.1

Example of the Full Search Tree Construction

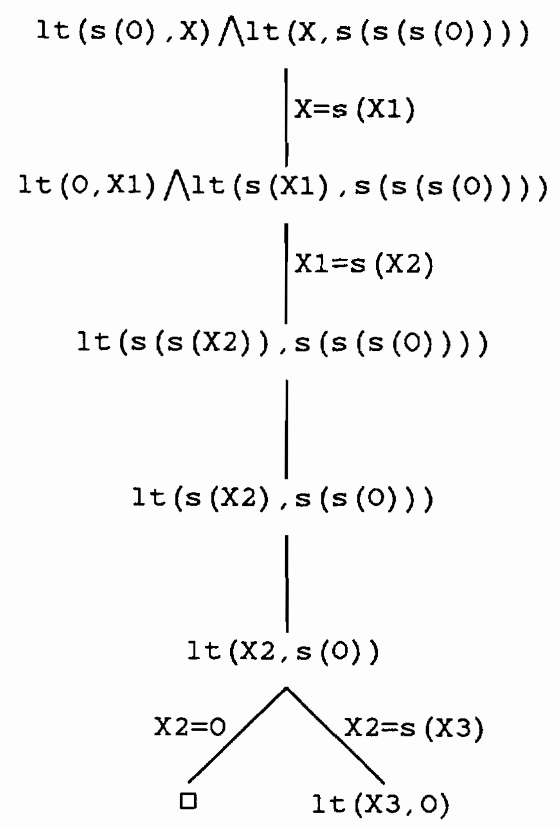


Figure 3.2

edges along the path. The value of the path in Figure 3.3 is $\sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_{n-2} \circ \sigma_{n-1}$. In Figure 3.2, the value of the only success path is:

$\{$X=s(s(0)), X1=s(0), X2=0$\}$.

An *answer substitution* is the value of a success path restricted to those variables

A Success Path

$$\begin{tikzcd}
C_1 \arrow[d, dash, "\sigma_1"] \\
C_2 \arrow[d, dash, "\sigma_2"] \\
\vdots \arrow[d, dash, "\sigma_{n-2}"] \\
C_{n-1} \arrow[d, dash, "\sigma_{n-1}"] \\
C_n = \square
\end{tikzcd}$$

Figure 3.3

occurring in the original query. Thus the answer substitution for Figure 3.2 is {`X=s(s(0))`}.

A "Lifting Lemma" applies to the construction of full search trees. This lemma ensures a success path in a full search tree for a query Q when some instance $Q\sigma$ of Q has a full search tree with a success path.

Lifting Lemma 3.6 [L82 (Thm. 8.2)]: If query $Q\sigma$ has a success path, then Q has a success path.

A *selection rule* used to obtain an atom from a conjunction of atoms is implicit in the construction of full search trees. SLD-resolution selects the first atom from a conjunction, reflecting efficiencies in adding and removing the first element from a data structure representing a conjunction. This selection rule may not be desirable for use with negation by failure. Other selection rules could be employed. Any particular selection rule only affects the size of the tree, and does not affect answer substitutions obtained from the tree, as will be seen in Lemma 3.8 providing for construction completeness.

In addition to construction of a full search tree, an implementation of SLD-resolution must search for a success node starting from the root. As stated earlier, usually this search is performed in conjunction with construction of the full search tree, because full search trees can be infinite. So the procedure for traversing a full search tree is the main factor in determining completeness of an SLD-resolution implementation.

The following properties concerning the construction of full search trees hold:

Lemma 3.7 (Construction correctness [L82 (Thm. 7.4)]): For every answer substitution α in the full search tree for Q (constructed using any selection rule), $\forall Q\,\alpha$ is logically implied by the program.

Lemma 3.8 (Construction completeness): For any ground query Q logically implied from the program, the full search tree for Q constructed using any selection rule has a success path.

A corollary to completeness of the construction is slightly stronger:

Corollary 3.9 (Construction Completeness & Correctness): $\exists Q$ is logically implied from the program iff the full search tree for Q (constructed with any selection rule) contains a success path.

Proof:

($\rightarrow$) If $\exists Q$ is logically implied, there is a substitution σ such that $Q\,\sigma$ is a closed formula and is logically implied. By Lemma 3.8, the full search tree for $Q\,\sigma$ has a success path. Lifting Lemma 3.6 provides that the full search tree for Q has a success path.

($\leftarrow$) If Q contains a success path with value σ, then $\forall Q\,\sigma$ is logically implied from the program, according to Lemma 3.7. Thus any closed instance of $Q\,\sigma$ is logically implied. Therefore, $\exists Q$ is also logically implied. $\square$

Though the full search tree construction provides for correctness and completeness, another component in the implementation of SLD-resolution dilutes these properties. This component is the procedure by which the full search tree is traversed from the root node to a success node. Correctness of the search procedure provides that success is declared only when a success node is found. Completeness of the search procedure ensures that every success node can eventually be located. Prolog relies on depth-first search. As noted previously, depth-first search is incomplete because an infinite non-success path could be followed, while success paths remain unexplored. Another search technique, called *staged* depth-first search [St84], aborts a depth-first search as soon as a certain depth is reached. When no success node is encountered, and nodes remain to be explored at greater depths, the maximum depth is incremented by some amount, and the staged depth-first search resumed. Thus, this search technique is complete.

3.3. Negation by Failure

In order to obtain efficiency in the resolution theorem-proving procedure, the notation of predicate logic is restricted. This restriction dramatically reduces the expressiveness of the notation, leading to the coding problem described in Section 1.2.

Negation within definite clause programs can overcome many of the difficulties associated with the coding problem. Definite clause programs utilizing *negation by failure* permit negated atoms within the bodies of clauses. Such clauses are referred to as *general* clauses. A negated atom appearing in a program for execution under negation by failure is expressed as not A. Use of negation fundamentally alters the underlying model of the language, and requires revisions in the evaluation procedure.

3.3.1. The Closed World Assumption

Negated atom not A is logically implied from a program if A is false in all models. It may be, however, that A is true in some models and false in others, permitting no valuation of A. For example, if a program contains only the clause p $\leftarrow$ q, Herbrand models of this program are $M_1 = \varnothing$ and $M_2 = \{p, q\}$. Hence, neither p nor not p are logically implied (similarly for q).

This problem can be resolved using the *Closed World Assumption* (CWA) [Re78]: whatever is not logically implied is assumed false in all models. For programs without negation, SLD-resolution is capable of determining logical implication. It is thus possible that SLD-resolution could be used to implement CWA. A full search tree without any success paths is a *failed* full search tree. Lemma 3.8 (construction completeness) implies that a closed atom A is not logically implied from a program if A has a failed full search tree. Thus, if A is a closed atom and A has a failed full search tree, `not` A is implied by the CWA. This correctness result is not really justified for general clause programs. Determination of negation for such programs requires a full resolution theorem-proving system.

3.3.2. Program Completion

In contrast to the CWA, Clark has suggested *program completion* [Cl78] to provide a basis for negation. The assumption underlying program completion is that a program embodies complete knowledge about a domain. Rather than implicational statements, a program is taken to provide definitions. The following algorithm transforms a definite clause program into a completed program:

(1) The *general form* of each clause $p(t_1, \ldots, t_n) \leftarrow C$ is $p(X_1, \ldots, X_n) \leftarrow \exists Y_1 \cdots Y_m : X_1 = t_1 \wedge \cdots \wedge X_n = t_n \wedge C$, where $X_1 \cdots X_n$ are variables not occurring in the clause and $Y_1 \cdots Y_m$ are all variables occurring in the original clause. Recall that C is a conjunction of atoms.

(2) Let the general forms of all clauses defining predicate p be $p(X_1, \ldots, X_n) \leftarrow E_1, \ldots, p(X_1, \ldots, X_n) \leftarrow E_k$. Then the *completed form* of predicate p is $\forall X_1 \cdots X_n : p(X_1, \ldots, X_n) \leftrightarrow E_1 \vee \cdots \vee E_k$. If there are no clauses defining a predicate $q^{(n)}$ in the program, the completed form is $\forall X_1 \cdots X_n : q(X_1, \ldots, X_n) \leftrightarrow$ `false`.

(3) The *completion* of a program P, *comp*(P), is the conjunction of the completed form of all predicate symbols in P.

Step (1) of this algorithm is a transformation of clauses that is meaning-preserving when equality is interpreted in a manner consistent with unification. Step (2) provides that all facts not logically implied by a program will be false in all models.

As an example of the completion procedure, consider the program below:

```
% add(I,J,K): true if I+J=K.
add(0,J,J).
add(s(I),J,s(K)) ← add(I,J,K).

% mult(I,J,K): true if I×J=K.
mult(0,J,0).
mult(s(I),J,K) ← mult(I,J,X) ∧ add(J,X,K).
```

The completion consists of the two formulas:

```
add(X1,X2,X3) ↔
     ∃J: [X1=0 ∧ X2=J ∧ X3=J]
     ∨ ∃I,J,K: [X1=s(I) ∧ X2=J ∧ X3=s(K) ∧ add(I,J,K)].

mult(Y1,Y2,Y3) ↔
     ∃J: [Y1=0 ∧ Y2=J ∧ Y3=0]
     ∨ ∃I,J,K,X: [Y1=s(I) ∧ Y2=J ∧ Y3=K ∧
          mult(I,J,X) ∧ add(J,X,K)].
```

Since equality is introduced into the completion, equality axioms must be added to the theory. The axioms [Cl78] will not be reproduced here; they provide for completeness and correctness of unification.

The following result characterizes negation by failure in terms of completed programs.

Lemma 3.10 (Meaning of Failure for Ground Atoms [L82 (Thm. 13.2)]): Let P be a definite clause program. Ground atom $A \notin gfp$ iff $\texttt{not}\ A$ is logically implied from $comp(P)$.

Whenever an atom A is not contained in the greatest fixedpoint, the full search tree for query A does not have a success path [AE82]. Therefore, $\texttt{not}\ A$ is logically implied from $comp(P)$. Lemma 3.10 is also generalized to non-ground queries.

Lemma 3.11 (Meaning of Failure): Let P be a definite clause program. A query Q has a failed full search tree iff $comp(P)$ logically implies $\forall(\texttt{not}\ Q)$.

Proof: If Q is a closed conjunction of atoms, the result follows: Q has a failed full search tree iff some atom A in Q has a failed full search tree, which holds iff $comp(P)$ logically implies $\texttt{not}\ A$. And this holds iff $comp(P)$ logically implies $\texttt{not}\ Q$. Now if Q contains variables, the general case, Q has a failed full search tree iff every ground instance $Q\sigma$ has a failed full search tree. This holds iff $comp(P)$ logically implies $\texttt{not}\ (Q\sigma)$, which holds iff $comp(P)$ logically implies $\forall(\texttt{not}\ Q)$. □

3.3.3. Incorrectness of Negation by Failure

Lemma 3.11 points out the incorrectness inherent in use of negation by failure. Suppose a query Q succeeds. By Lemma 3.9, $\exists Q$ is logically implied by the program and its completion. Also, the query $\texttt{not}\ Q$ fails, and Lemma 3.11 provides that $\forall(\texttt{not}(\texttt{not}\ Q))$ is logically implied by the program's completion. But $\forall(\texttt{not}(\texttt{not}\ Q))$ is logically equivalent to $\forall Q$, which is not implied by $\exists Q$.

This confusion of quantifiers surfaces in Example 3.1. The query `lt(X,s(0))` succeeds with answer substitution `X = 0`, and so `not lt(X,s(0))` fails using negation by failure. By Lemma 3.11, $\forall$`X:lt(X,s(0))` is logically implied by the program's completion, which is clearly not true due to all contradictory values of `X` $=$ `s`^{n}`(0)` $(n \geq 1)$.

Correctness is assured only when ground negated atoms are evaluated. A *correct* selection rule thus selects a negated atom only if it is ground. Under correct selection rules, it is possible for the query containing non-ground negated atoms to *flounder*. For example, the query `p(X)` $\wedge$ `not q(X)` flounders on a program containing only the assertion `p(Y)` under every correct selection rule.

To eliminate floundering, only *allowed* queries are permitted on *allowed* programs [Cl78]. A query is allowed if every variable occurring in a negated atom occurs somewhere else within a positive atom. A clause is allowed if the body of the clause constitutes an allowed query, and every variable occurring in the head of the clause occurs within a positive atom within the body of the clause. The restriction to allowed clauses ensures that termination of any query composed of positive atoms will instantiate all variables to ground terms. Evaluation of the positive atoms of an allowed query can then instantiate the variables occurring within negative atoms of the query, and any correct selection rule will never flounder under terminating evaluations.

It should be clear that allowed queries on allowed programs do not flounder. The result, however, is a severe restriction on the expressiveness of programs. For example, the assertion in Example 3.1, `lt(O,s(J))`, is not allowed. This assertion is a concise statement that zero is less than every positive number. The restriction to allowed clauses has restricted the ability to state universal properties within clauses.

It can be argued that allowed programs do not restrict the expressiveness of database-oriented programs [Cl78] (Section 6.1). Logic programs for such applications typically have a large number of assertions and a small number of rules. Each assertion is conventionally represented by a record in a relational database, and will contain neither variables nor structured terms. Clauses act as database views [U80], capable of generating additional relations, and will not be able to introduce variables or structured terms into records. Under these constraints, restriction to allowed programs seems reasonable.

3.3.4. Incompleteness of Negation by Failure

There are also several problems with respect to completeness of negation by failure. In order to determine that a full search tree is failed, it is necessary to traverse the entire tree searching for a success node. When a full search tree is infinite, this search is impossible. Hence, evaluation of a negated query using negation by failure must ensure finiteness of the failed full search tree. It may be necessary to construct various full search trees using alternate selection rules to find one that is finite and failed. Fortunately, there are maximal selection rules [Sh84], essentially those that are fair, that can obtain finite failed full search trees if any exist. Still, there are examples of programs producing infinite full search trees that no maximal selection rule can make finite:

```
    p ← p.
```

For this program and query p, the only full search tree produced under every selection rule, including maximal selection rules, is infinite.

3.3.4.1. Canonical Programs

Completeness of negation as failure is achieved for a certain class of programs. Definite clause program P is *canonical* if $T \downarrow \omega = gfp$.

Lemma 3.12 (Completeness for Canonical Programs [JLM84]): When P is a canonical program and a ground negated atom not A is logically implied from $comp(P)$, the query A has a finitely failed full search tree.

As for Lemma 3.11, this generalizes to non-ground conjunctions of atoms. Example 3.2 is non-canonical. Canonical programs are obtained only with stringent syntactic constraints, for example permitting only constant terms in programs [AE82]. Jaffar and Stuckey have shown that for every definite clause program there is an equivalent canonical program [JS86]. Their proof is not useful in deciding if a logic program is canonical, however, because they produce a canonical program from the description of a partial recursive function, rather than from another definite clause program.

3.3.4.2. Inconsistency of a Program's Completion

Completeness of negation by failure also depends on the consistency of the completion of a program. If a program's completion is inconsistent, its greatest fixedpoint will not exist. Any query is logically implied from an inconsistent program, but SLD-resolution may not succeed for every query. Consider the general clause program below:

Example 3.3

```
    p ← not p.
```

This program has an inconsistent completion: `p ↔ not p`. Therefore, `p` is logically implied by the program, but the query `p` fails with an infinite full search tree.

Inconsistency can be prevented by requiring *stratified* programs [ABW85]. A program from $B(A(\Pi, \Sigma))$ is stratified if there is a well-founded ordering $<_\Pi$ over Π^2 such that $p <_\Pi q$ whenever $q(x) \leftarrow C$ is contained in the program and C contains the negated atom not $p(y)$. In effect stratification prevents recursive references by negated atoms, as in Example 3.3. Stratification is sufficient to guarantee consistency of the program's completion.

It is not clear that general programming tasks fit well within the requirements set down by stratification. However, negation for relational database applications is accomplished by the relative complement operation [Co70], which requires full definition of its operands prior to evaluation. Thus there are no recursive references by negated atoms. But strictly speaking, relational algebra has no capacity at all to express unbounded recursion.

Negation by failure is an efficient implementation of negation within logic programming. With respect to the completion of a program, ground negative queries can be evaluated correctly. Only strict syntactic restrictions can ensure completeness. When negation is permitted within programs, stringent syntactic restrictions ensure consistency and correctness. Consistency of a program's completion can be assured by stratifying the program. To prevent incorrectness of negation by failure, negated queries flounder if all selection rules cannot instantiate variables of these queries. Floundering is eliminated by evaluating only allowed queries on allowed programs.

3.4. Enhancing Expressiveness of Programs with Negation

Any implementation of negation is sufficient to significantly reduce the coding problem described in Section 1.2. Lloyd and Topor suggest a logic language of *extended programs* based on negation by failure. Implementation of this language therefore suffers from incorrectness and incompleteness. The *model elimination* procedure permits full predicate logic. Its implementation is a significant enhancement to SLD-resolution, and is complete for negative queries, unlike negation by failure. However, the implementation may be far less efficient than SLD-resolution.

3.4.1. Extended Programs

Implementation of negation within logic programs permits implementation of all other logical connectives within the bodies of clauses. This increased expressiveness reduces, though does not eliminate, the coding program suffered by logic programs. To demonstrate the expressiveness obtained when negation is implemented, *extended programs* are defined [LT84]. An extended program from $B(A(\Pi, \Sigma))$ is composed of *extended clauses*. An extended clause is of the form $A \leftarrow F$, where F is a formula using all logical connectives and A is an atom.

Every extended program P can be converted algorithmically into a general clause program P' such that the set of sentences implied by *comp*(P') is equivalent to the set of sentences implied by *comp*(P). The conversion algorithm is applied to every extended clause of an extended program until every clause is just a general clause.

The conversion rules preserve stratification of the original extended program (Section 3.3.4.2). It is not assured that the general clause program produced by the transformation will be allowed (Section 3.3.3). For example, the extended clause `p ← ∀X:q(X)` is converted into two general clauses:

```
p ← not aux.
aux ← not q(X).
```

This program is not allowed because the variable `X` in the second clause does not appear within a positive atom elsewhere within the body of the same clause. This leads to floundering of the query `p`, because the non-ground negative query `not q(X)` ensues.

Extended programs partially resolve the coding problem, described in Section 1.2. Using a conversion procedure and an implementation of negation, all logical connectives can be included within the bodies of clauses. Nonetheless, requiring that programs will be allowed and stratified impedes these additional expressive capabilities. The model elimination procedure, discussed next, permits the full expressiveness of logic, though the procedure is not as efficient as SLD-resolution.

3.4.2. Model Elimination

Negation by failure relies on SLD-resolution to provide an implementation of negation within logic programs. This strategy retains the efficiency of SLD-resolution, though completeness of the evaluation system is sacrificed. The *model elimination* procedure [Lo78] is an evaluation system for full predicate logic. The expressiveness of the language is therefore equivalent to the expressiveness of predicate logic. But efficiency of the evaluation system is now in question.

Model elimination is an enhancement to the linear-input resolution strategy, called ancestry-filtered resolution [Ni80]. While linear-input resolution is incomplete for general clauses, an ancestor search component restores completeness to linear-input resolution. Model elimination therefore enhances SLD-resolution. Efficiency of model elimination is somewhere between SLD-resolution and resolution.

When using resolution, statements of predicate logic are converted into clauses of the form:

$$A_1 \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n,$$

where each A_i and B_j is an atom. For model elimination, each clause is further converted into *contrapositive* forms. As an example, the following contrapositives:

$$\begin{array}{rcl} p & \leftarrow & r \wedge s \wedge \neg q \\ q & \leftarrow & r \wedge s \wedge \neg p \\ \neg r & \leftarrow & s \wedge \neg p \wedge \neg q \\ \neg s & \leftarrow & r \wedge p \wedge \neg q \end{array}$$

are obtained from the clause $p \vee q \leftarrow r \wedge s$. Every contrapositive obtained from a clause is logically equivalent to the clause. SLD-resolution can be used on programs consisting of contrapositives by generalizing the procedure to permit unification of a query $\neg A$ with the head of a contrapositive $\neg A' \leftarrow L_1 \bigwedge \cdots \bigwedge L_n$.

Model elimination constructs a full search tree with the same construction rules used by SLD-resolution. An additional *reduction* rule also applies. If $A \bigwedge C$ labels a node in the full search tree with a descendent node labeled $\neg A' \bigwedge C'$, where $A'\sigma = A$ for some substitution σ, and $\neg A' \bigwedge C'$ arises from solving A, then $\neg A'$ can be eliminated, resulting in the descendent $C'\sigma$ of $\neg A' \bigwedge C'$. This implementation of negation effectively duplicates reasoning through *reductio ad absurdum*.

To detect reduction the full search tree to the root is traversed, though certain nodes along the way can be disregarded. Indexing schemes reduce the number and length of such searches [PG86]. As demonstrated in [MW87], it is not always advantageous to perform a reduction when instantiation of variables would occur. Thus, each step of model elimination involves more choices and more processing than each step of SLD-resolution. Experience of an actual implementation on actual programs will demonstrate if model elimination used in practice is as efficient as SLD-resolution [St84].

Chapter 4
Constructive Negation

Negation by failure, described in Chapter 3, is the predominant implementation of negation with logic programming. This procedure can be both incorrect and incomplete. And because negation by failure does not produce answer substitutions, it does not fulfill important programming requirements. For this reason, programmers often use ad hoc negative definitions of predicates within programs to produce answer substitutions.

Constructive negation is a formalization of this approach. Its negated queries can produce answer substitutions. To derive answers, programs incorporating constructive negation, DIF-programs, contain definitions for both positive and negative facts.

The first section of this chapter describes the syntax of DIF-programs. Section 4.2 describes the underlying model of DIF-programs, based on 3-valued logic. DIF-programs may be inconsistent. While resolution can always detect inconsistency, detection imposes inefficiency on the evaluation system. Lemma 4.6 demonstrates that inconsistency is not decidable. Sufficient syntactic conditions ensuring consistency are described in Chapter 5.

4.1. A Language of Logic Programs with Constructive Negation

Constructive negation requires definitions of both true and false information. In fact, the definitions can be completely disjoint. A *Definite Inference Form* (DIF) is used to express such definitions. Every DIF is either an assertion L or is of the form $L \leftarrow F$ where L is a literal, and F is a formula containing all logical connectives. All negated formulas that are to be evaluated under constructive negation are expressed as $\sim F$. Only variables occurring in the head of a DIF are permitted to occur free in its body. A DIF-program consists of a collection of DIFs. A fragment of a DIF-program follows:

```
% mult(I,J,K): true if I×J=K; otherwise, false.
 mult(0,J,0).
 mult(s(I),J,K) ←
      ∃X: mult(I,J,X) ∧  add(J,X,K).
~mult(0,J,s(K)).
~mult(s(I),J,K) ←
      ∀X: mult(I,J,X) → ~add(J,X,K).
```

Because all logical connectives can be present within the body of a DIF, DIF-programs are as expressive as as the extended programs of Lloyd and Topor (Section 3.4.1); treatment of negation is a key difference.

In this language all literals are treated equally regardless of sign. An interpretation of a DIF-program is a *constructive* interpretation. Just as an interpretation of a definite clause program can be given by a set of ground atoms, a constructive interpretation is represented by a set of ground *literals*. A ground atom A is constructively true (respectively, constructively false) in constructive interpretation I if A (respectively, $\sim A$) is a member of I.

It is essential to differentiate logical negation from constructive negation. Temporarily define the *standard part* of a constructive interpretation to be the set of all ground *atoms* within the interpretation; hence, the standard part of a constructive interpretation is an interpretation. For constructive interpretation $I = \{\sim p\}$, proposition p is constructively false and p is logically false in the standard part of I. On the other hand, for $I = \emptyset$, proposition p is logically false in the standard part of I, but p is neither constructively true nor constructively false. Thus, the law of the excluded middle does not hold. This fact leads naturally to development of a three-valued logic for DIF-programs.

4.2. Underlying Model of DIF-Programs

The ability to define both true and false propositions within programs also entails the possibility that the truth value of some propositions may not be defined. To cope with this possibility, an undefined logical value can be assigned to formulas by an interpretation. Models of DIF-programs may be strong or weak (Section 4.2.3). Strong models assign true valuations to programs, while weak models assign either true or undefined valuations. Unlike definite clause programs, the least strong model of a DIF-program may not exist, while the least weak model assigns to every formula the undefined logical value. Thus, we take fixedpoints of the T functional as the basis for a DIF-program's meaning. The set of fixedpoints may be empty, but absence of fixedpoints cannot be detected by efficient evaluation systems or by any decision procedure.

4.2.1. Three-Valued Logic

The logical constants are:

Logical Constants	
Symbol	Intended Meaning
t	true
u	undefined
f	false

There are several proposals for 3-valued truth tables of the Boolean operators [Tu84]. To some extent the content of truth tables for the logic is arbitrary, though a monotonicity property should hold for all logical operators. The information ordering on truth values is defined with **u** as the least informative element:

$$\mathbf{u} \sqsubset \mathbf{t} \text{ and } \mathbf{u} \sqsubset \mathbf{f}.$$

Monotonicity ensures that:

Negation:
if $x \sqsubseteq y$ then $(\sim x) \sqsubseteq (\sim y)$

Sets of expressions:
if there is a bijection $o: S \rightarrow T$, such that $x \sqsubseteq o(x)$ for all $x \in S$, then $S \sqsubseteq T$

Disjunction and Conjunction:
if $S \sqsubseteq T$, then $(\bigwedge S) \sqsubseteq (\bigwedge T)$ and $(\bigvee S) \sqsubseteq (\bigvee T)$.

For example, the truth table of implication follows:

Truth Table for Implication				
			y	
	$x \rightarrow y$	**t**	**u**	**f**
x	**t**	**t**	**u**	**f**
	u	**t**	**u**	**u**
	f	**t**	**t**	**t**

Appendix A contains 3-valued truth tables for all logical connectives. With respect to the ordering $\sqsubseteq$, these 3-valued truth tables are the strongest extension of the usual 2-valued truth tables; Appendix B demonstrates this assertion. Appendix B also demonstrates that all laws observed by the usual truth tables are observed by the extension.

4.2.2. Constructive Interpretations of Formulas

Suppose a DIF-program is from $B(A(\Pi,\Sigma))$. It possesses an associated Herbrand Universe and Base, $HU = \underline{T(\Sigma)}$ and $HB = \underline{A(\Pi,\Sigma)}$, respectively. A *constructive interpretation* is a mapping from HB to the three logical constants **t**, **u**, and **f**. Valuation of a formula F by a constructive interpretation I is denoted $I[F]$. Whenever the meaning is clear from context, constructive interpretations will henceforth be referred to only as interpretations.

As for Herbrand interpretations, set notation is used to denote constructive interpretations. A set of ground literals *qualifies* as an interpretation if it contains no occurrence of an atom and its negation. If a set S of ground literals qualifies as an interpretation, then an interpretation I_S may be constructed from S as follows. For every atom $A \in HB$:

If $A \in S$, then $I_S[A] = \mathbf{t}$.

If $\sim A \in S$, then $I_S[A] = \mathbf{f}$.

If neither A nor $\sim A$ are in S, then $I_S[A] = \mathbf{u}$.

For example, suppose $HB_1 = \{\mathrm{p}, \mathrm{q}, \mathrm{r}\}$, and $S_1 = \{\mathrm{p}, \sim\mathrm{q}\}$. Then I_{S_1} contains the following mappings:

$I_{S_1}[\mathrm{p}] = \mathbf{t}$

$I_{S_1}[\mathrm{q}] = \mathbf{f}$

$I_{S_1}[\mathrm{r}] = \mathbf{u}$.

When L is a ground literal and I is an interpretation, $L \in I$ is taken to mean that $I[L] = \mathbf{t}$. $I \subseteq J$, where I and J are interpretations, means that for all ground literals L $J[L] = \mathbf{t}$ whenever $I[L] = \mathbf{t}$. As examples of this notation: $\mathrm{p} \in I_{S_1}$, and if $S_2 = \{\mathrm{p}, \sim\mathrm{q}, \mathrm{r}\}$, then $I_{S_1} \subseteq I_{S_2}$. Henceforth, sets of ground literals qualifying as interpretations will be used freely to designate interpretations without the unnecessary step of designating the unique interpretations associated with the sets.

An interpretation I can be naturally extended to a mapping $\hat{I}$ over all formulas from $B(A(\Pi, \Sigma))$ as follows:

$\hat{I}[\sim F_1] = \sim\hat{I}[F_1]$

$\hat{I}[\bigwedge\{F_1, F_2, \cdots\}] = \bigwedge\{\hat{I}[F_1], \hat{I}[F_2], \cdots\}$

$\hat{I}[\bigvee\{F_1, F_2, \cdots\}] = \bigvee\{\hat{I}[F_1], \hat{I}[F_2], \cdots\}$

The extension $\hat{I}$ also distributes over abbreviations for formulas:

$$\hat{I}[F_1 \rightarrow F_2] = \hat{I}[F_1] \rightarrow \hat{I}[F_2]$$

$$\hat{I}[F_1 \leftrightarrow F_2] = \hat{I}[F_1] \leftrightarrow \hat{I}[F_2]$$

$$\hat{I}[F_1 \bigwedge F_2] = \hat{I}[F_1] \bigwedge \hat{I}[F_2]$$

$$\hat{I}[F_1 \bigvee F_2] = \hat{I}[F_1] \bigvee \hat{I}[F_2]$$

Since $\forall X{:}F_1$ is an abbreviation for $\bigwedge_{t \in HU} F_1(t)$,

$$\hat{I}[\forall X{:}F_1] = \bigwedge_{t \in HU} \hat{I}[F_1(t)].$$

Similarly, $\exists X{:}F_1$ is an abbreviation for $\bigvee_{t \in HU} F_1(t)$, and therefore:

$$\hat{I}[\exists X{:}F_1] = \bigvee_{t \in HU} \hat{I}[F_1(t)].$$

Henceforth, an interpretation I will be used in place of its extension $\hat{I}$ when no confusion can result.

As an example of evaluation of a formula by an interpretation:

Let $HU = \{\mathtt{a}, \mathtt{b}\}$,

$HB = \{\mathtt{p(a)}, \mathtt{p(b)}, \mathtt{q(a)}, \mathtt{q(b)}\}$,

$I_1 = \{\mathtt{p(a)}, \mathtt{\sim p(b)}, \mathtt{q(a)}\}$.

Then $I_1[\forall \mathtt{X}{:}\mathtt{p(X)} \rightarrow \mathtt{q(X)}] = (I_1[\mathtt{p(a)}] \rightarrow I_1[\mathtt{q(a)}]) \bigwedge (I_1[\mathtt{p(b)}] \rightarrow I_1[\mathtt{q(b)}])$

$$= (\mathbf{t} \rightarrow \mathbf{t}) \bigwedge (\mathbf{f} \rightarrow \mathbf{u})$$

$$= \mathbf{t} \bigwedge \mathbf{t}$$

$$= \mathbf{t}.$$

The next result demonstrates a form of monotonicity maintained by formulas.

Lemma 4.1 (Monotonicity of Interpretations): Let I and J be interpretations. Then $I \subseteq J$ iff $I[F] \sqsubseteq J[F]$, for all sentences F.

Proof:

($\rightarrow$) Suppose $I \subseteq J$. The proof proceeds by induction on the nesting depth of operators in F. In the basis case, the nesting depth is zero, so F is a ground atom. There are three subcases:

(a) If $F \in I$, then $F \in J$, so $I[F] = J[F] = \mathbf{t}$.

(b) If $\sim F \in I$, then $\sim F \in J$, so $I[F] = J[F] = \mathbf{f}$.

(c) Otherwise, $I[F] = \mathbf{u}$, and $J[F] \in \{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$.

In all three cases, $I[F] \sqsubseteq J[F]$.

For the induction hypothesis, assume that $I \subseteq J$ implies $I[F] \sqsubseteq J[F]$ for all sentences F with nesting depth at most d. Assume that $I \subseteq J$, and consider a sentence F with nesting depth $d+1$. Let $F = c\ S$, where c is a logical operator in Ω_0 and S is a set of formulas of nesting depth at most d. (When c is the negation operator, S will be a singleton set.) The induction hypothesis holds for each subformula in S, so $I[F_i] \sqsubseteq J[F_i]$ for all $F_i \in S$. Consequently, $c\ \{I[F_1], I[F_2], \cdots\} \sqsubseteq c\ \{J[F_1], J[F_2], \cdots\}$. But I denotes $\hat{I}$ and J denotes $\hat{J}$, so $I[c\ S] \sqsubseteq J[c\ S]$. □

It is easily shown that interpretations preserve any laws observed by the usual truth tables. By applying De Morgan's laws, the logical connectives are expressive enough to dispense with negation applied to any non-atomic formula. The *comple-*

ment of a formula F, denoted $\overline{F}$, produces a new formula with negation innermost, applied only to atoms. Complement is defined as follows:

$\overline{A} = \sim A$, where A is an atom

$\overline{\sim F} = F$

$\overline{\bigwedge\{F_1, F_2, \cdots\}} = \bigvee\{\overline{F_1}, \overline{F_2}, \cdots\}$

$\overline{\bigvee\{F_1, F_2, \cdots\}} = \bigwedge\{\overline{F_1}, \overline{F_2}, \cdots\}$

The following result ensures that complement is meaning-preserving.

Lemma 4.2 (Complement equivalent to Negation): For every interpretation I and sentence F, $I[\sim F] = I[\overline{F}]$.

Proof: This is just application of De Morgan's laws.

Thus the syntactic procedure for complementing a formula is equivalent to the semantic notion of constructive negation.

Extending complement to the logical connectives provides the following rules:

$\overline{F_1 \bigwedge F_2} = \overline{F_1} \bigvee \overline{F_2}$

$\overline{F_1 \bigvee F_2} = \overline{F_1} \bigwedge \overline{F_2}$

$\overline{F_1 \rightarrow F_2} = F_1 \bigwedge \overline{F_2}$

$\overline{F_1 \leftrightarrow F_2} = \overline{F_1 \rightarrow F_2} \bigwedge \overline{F_2 \rightarrow F_1}$

$\overline{\exists X : F} = \forall X : \overline{F}$

$\overline{\forall X : F} = \exists X : \overline{F}$

In order to evaluate DIF-programs, Chapter 5 requires all quantifiers to be bounded, of the form $\exists X{:}F \bigwedge G$ and $\forall X{:}F \rightarrow G$. Thus, the rules above can be refined to produce only bounded quantifiers from formulas with bounded quantifiers:

$$\overline{\exists X{:}F_1 \bigwedge F_2} = \forall X{:}F_1 \rightarrow \overline{F_2}$$
$$\overline{\forall X{:}F_1 \rightarrow F_2} = \exists X{:}F_1 \bigwedge \overline{F_2}.$$

Because the syntactic complement of a formula is equivalent to evaluating its negation, the complement form of a formula can always be used whenever negation is applied to a non-atomic formula. The resulting formula is logically equivalent to the original negated formula. For example, the following DIFs are logically equivalent:

```
~mult(s(I),J,K)  ←  ~∃X: mult(I,J,X)  ⋀   add(J,X,K).
~mult(s(I),J,K)  ←   ∀X: mult(I,J,X)  →  ~add(J,X,K).
```

Henceforth, only DIF-programs with negation applied to atomic formulas will be considered. DIF-programs with negation applied to arbitrary formulas can be converted in a meaning-preserving manner to programs with negation applied only to atoms. As the above example demonstrates, having negation applied only to atoms manifests occurrences of universal quantification. It is important to detect implicit occurrences of universal quantification because universal quantification can make any evaluation procedure incomplete.

4.2.3. Constructive Models of DIF-Programs

Two notions of model are possible under 3-valued logic, as discussed in [LM85]. An interpretation M is a *strong* model of a DIF-program if $M[\forall F] = \mathbf{t}$ for every DIF

F in the program. Therefore, $M[F\sigma] = \mathbf{t}$ for every closed instance $F\sigma$ of F. Recall from Lemma 3.2 that the least model $\bigcap\mathbf{M}$ is a model of any definite clause program. This is not generally the case for strong models of DIF-programs. A program containing only the DIF `p ← q` has strong models:

```
{p, q},
{p},
{p, ~q},
{~q},
{~p, ~q}.
```

The intersection of this collection is $\emptyset$, which is not a strong model. It is important to obtain a unique least model. Without a least model the evaluation system must cope with indefinite information, which is not possible for an evaluation system based on SLD-resolution.

As an alternative, an interpretation M is a *weak* model of a DIF-program if $M[\forall F] \neq \mathbf{f}$ for every DIF F in the program. Hence, $M[F\sigma] \neq \mathbf{f}$ for every closed instance $F\sigma$ of F. Weak models for the program containing only the DIF `p ← q` are those strong models listed above and the following:

```
{q},
∅,
{~p}.
```

The intersection $\emptyset$ is now a weak model. The situation in the example is general — $\emptyset$ is always the least weak model of any DIF-program.

Within the framework of 3-valued logic, a program is *inconsistent* if it has no strong models. For example, a program consisting of two assertions `p` and `~p` has no strong models, but $\emptyset$ is a weak model. So consideration of weak models is too permissive.

Thus neither strong nor weak models are suitable for assigning underlying models to programs. The set of strong models of a DIF-program may not possess a least element, while the least weak model is $\varnothing$ even in the event of inconsistency. Instead we turn again to the immediate consequence functional T of Section 3.1. The definition is broadened to accommodate DIF-programs as follows:

Closed literal $L \in T(I)$ iff:

L is a ground instance of an assertion,

or $L \leftarrow F$ is a closed instance of a DIF in the program and $I[F] = \mathbf{t}$.

Lemma 4.3 (T monotonic): $T(I) \subseteq T(J)$ whenever $I \subseteq J$.

Proof: Suppose $I \subseteq J$ and closed literal $L \in T(I)$. If L is a closed instance of an assertion, then $L \in T(J)$. Otherwise, there must be a closed instance $L \leftarrow F$ of a DIF with $I[F] = \mathbf{t}$. By monotonicity of interpretations (Lemma 4.1), $J[F] = \mathbf{t}$, so $L \in T(J)$. □

Unlike definite clause programs, it is possible at some iteration α for $T \uparrow \alpha$ to become undefined, as in the following DIF-program:

```
~p.
 p ← q.
 q ← r.
 r.
```

For this program, iterations of T are the following:

$T\uparrow 0 = \varnothing$

$T\uparrow 1 = \{{\sim}\mathrm{p}, \mathrm{r}\}$

$T\uparrow 2 = \{{\sim}\mathrm{p}, \mathrm{q}, \mathrm{r}\}$

$T\uparrow 3$ undefined.

In general, $T\uparrow$ becomes undefined at iteration α when $T\uparrow\alpha$ attempts to include both a ground atom A and its negation ${\sim}A$.

As for definite clause programs, we will be interested in fixedpoints of T. Often, we state that X is a fixedpoint of a program P, meaning that X is a fixedpoint of T_P. The following result places an inclusion ordering on the classes of fixedpoints of T, strong models, and weak models.

Lemma 4.4 (Relating Models and Fixedpoints)

(1) If M is a strong model then M is a weak model.

(2) Every fixedpoint of T is a weak model.

Proof:

(1) Immediate from the definitions.

(2) We will show that if $M = T(M)$ then M is a weak model. Consider any closed instance L of an assertion. It must be that $L \in T(M)$. But then $L \in M$, so $M[L] \neq \mathbf{f}$. Next consider any closed instance $L \leftarrow F$ of a DIF. If $M[F] = \mathbf{t}$ then $L \in T(M)$, so $L \in M$, and $M[L \leftarrow F] \neq \mathbf{f}$. Otherwise, $M[F] \neq \mathbf{t}$, so $M[L \leftarrow F] \neq \mathbf{f}$. Hence, M is a weak model. □

This result only provides that the collections of fixedpoints and strong models are contained in the collection of weak models. In general, the inclusions are strict, as shown in Figure 4.1. Lemma 4.5 will demonstrate that the intersecting region of fixedpoints and strong models is nonempty.

There is a connection between strong models and fixedpoints. The following lemma demonstrates that every strong model contains a fixedpoint. A DIF-program is *fixedpoint-inconsistent* if it has no fixedpoints. The lemma also shows that

Relationships Between Fixedpoints and Models

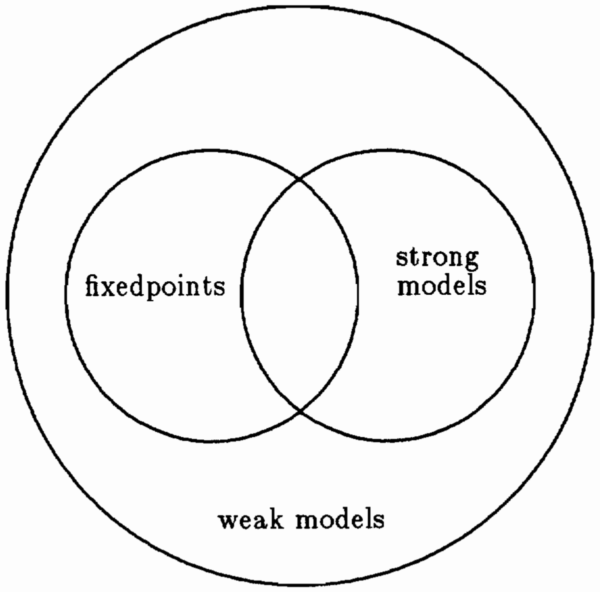


Figure 4.1

fixedpoint-inconsistency implies inconsistency.

Lemma 4.5 (Relating Strong Models and Fixedpoints):

(1) If M is a strong model, then $T^{\alpha}(M) \subseteq M$ is a fixedpoint for some ordinal α.

(2) If a DIF-program is fixedpoint-inconsistent, then it is inconsistent.

Proof: Point (2) follows from (1), since every strong model M contains a fixedpoint $T^{\alpha}(M)$. The existence of any strong model implies the existence of a fixedpoint.

The proof of point (1) is in two parts. First, we show that $M \supseteq T(M)$ whenever M is a strong model. If there is a closed literal $L \in T(M)$, there are two cases:

(a) If L is a closed instance of an assertion, then $L \in M$ because M is a strong model.

(b) There is a closed instance $L \longleftarrow F$ of a DIF, where $M[F] = \mathbf{t}$. Since M is a strong model, $L \in M$ also.

Since T is monotonic, $T^{\alpha}(M) \supseteq T^{\alpha+1}(M)$ for all ordinals α. No chain $M \supset T(M) \supset \cdots \supset T^{\alpha}(M)$ can be ever-decreasing. At worst, α can be the cardinality of M and $T^{\alpha}(M) = \varnothing$. Therefore, for some ordinal α, $T^{\alpha}(M) \subseteq T^{\alpha+1}(M)$, so $T^{\alpha}(M) = T^{\alpha+1}(M) = T(T^{\alpha}(M))$. □

Lemma 4.5 demonstrates that there is a decreasing chain from each strong model to a fixedpoint. The converse of point (2) in Lemma 4.5 does not hold, as the following program demonstrates:

```
  p ←  ~p
 ~p ←   p.
```

This DIF-program is fixedpoint-consistent ($lfp = \emptyset$), but is inconsistent.

Let $\mathbf{X}_P$ be the set of fixedpoints of T_P, where P is a DIF-program. P is fixedpoint-inconsistent if $\mathbf{X}_P$ is empty. When P is fixedpoint-consistent, the Knaster-Tarski Theorem [Ta55] guarantees existence of a least fixedpoint $lfp_P = \bigcap \mathbf{X}_P$. As usual, when program P is understood, the subscripts will be omitted.

Lemma 4.6: For every fixedpoint-consistent program, $lfp = T\uparrow\alpha$ for some ordinal α.

Proof: The proof first demonstrates that $T\uparrow\alpha$ is contained in every fixedpoint, for all ordinals α. In particular, $T\uparrow\alpha \subseteq lfp$. Next, we show that $T\uparrow\alpha \subseteq T\uparrow\beta$ for all ordinals $\alpha \leq \beta$. Therefore, there is a "maximal" ordinal γ such that $T\uparrow\alpha \subseteq T\uparrow\gamma$ for all ordinals α; otherwise, there would be ordinals α such that $HB \subset T\uparrow\alpha$. $T\uparrow\gamma$ is a fixedpoint, so $lfp \subseteq T\uparrow\gamma$.

To demonstrate the first part, induction is performed on all ordinals α. Since $T\uparrow 0 = \emptyset$, the basis case holds. For the induction hypothesis, assume $T\uparrow\alpha$ is contained in every fixedpoint. Consider the successor ordinal $\alpha+1$. Suppose L is a ground instance of an assertion in P. Then $L \in T\uparrow(\alpha+1)$; also, L is true in every fixedpoint. If $L \leftarrow F$ is a closed instance of a DIF in P, and $T\uparrow\alpha[F] = \mathbf{t}$, then $L \in T\uparrow(\alpha+1)$. Also, by the induction hypothesis, $T\uparrow\alpha$ is contained in every fixedpoint, and due to monotonicity of interpretations, Lemma 4.1, F is true in every fixedpoint. Therefore, L is true in every fixedpoint. The induction follows for limit ordinals also.

To show that $T\uparrow\alpha \subseteq T\uparrow\beta$ whenever $\alpha \leq \beta$, for all ordinals α and β, let ordinal δ be such that $\alpha+\delta=\beta$. Since $\varnothing \subseteq T\uparrow\delta$, $T^{\alpha}(\varnothing) \subseteq T^{\alpha}(T\uparrow\delta)$ by monotonicity of T, Lemma 4.3. Using the definition of $\uparrow$, $T\uparrow\alpha \subseteq T\uparrow\beta$. □

The least ordinal α for which $T_P\uparrow\alpha = T_P\uparrow(\alpha+1) = lfp_P$ is called the *closure ordinal* of program P.

Fixedpoints of T will represent strong models, since the set of strong models of a program may not have a least element. Three characteristics of fixedpoints justify their use as representatives:

(1) For any fixedpoint-consistent program, the set of fixedpoints is closed under $\cap$.

(2) Fixedpoint-inconsistency implies inconsistency.

(3) Every strong model contains a fixedpoint.

Fixedpoints are therefore chosen as the basis for the underlying model of DIF-programs. A formula F is *fixedpoint-implied* by a program P if and only if $X[F]=\mathbf{t}$ for every fixedpoint X of T_P. When program P is fixedpoint-consistent, F is fixedpoint-implied by P if and only if $lfp[F]=\mathbf{t}$.

Programs containing universal quantifiers and with infinite domains may have infinite closure ordinals. Consider the following program:

```
p ← ∀X: nat(X)→nat(s(X)).
nat(0).
nat(s(N)) ← nat(N).
```

For this program, $lfp = T\uparrow\omega{+}1$. Since the completeness proof of the SLD-resolution procedure relies on a correspondence between the iterations of the T functional and the depth of a success path in the full search tree, there will be queries that cannot execute to completion on programs with transfinite closure ordinals. Consequently, completeness is sacrificed with DIF-programs. Chapter 5 will demonstrate the necessity of this incompleteness result through Turing-reducibility.

Eliminating the universal quantifier from formulas, thereby reducing the expressiveness of the language, necessarily results in completeness. However, DIF-programs may be fixedpoint-inconsistent, which will not be detected by efficient evaluation system such as SLD-resolution.

The resolution procedure is correct and complete even in the event of inconsistency, because resolution need not use a program clause to form a resolvent. SLD-resolution attains efficiency over resolution by forming each resolvent only from the previous resolvent and a program clause. SLD-resolution is correct and complete even with inconsistency because definite clause programs cannot be inconsistent. With introduction of negation in DIF-programs, fixedpoint-inconsistency can arise.

To retain the efficiency of SLD-resolution for evaluation of DIF-programs, fixedpoint-consistency should be decided before evaluation. But, as the following lemma demonstrates, fixedpoint-consistency is undecidable.

Lemma 4.6: Fixedpoint-consistency of an arbitrary DIF-program is undecidable.

Proof: It is easy to produce a contradiction. Consider Hilbert's tenth problem: producing integer solutions for polynomial equations in several variables. Matijasevic has shown this problem undecidable. Consider program `Hilbert` of Figure 4.2. Definitions of certain predicates used for lists and integers are not included in this

Program Hilbert

```
% hilbert(Vars,Exprs): true if there are bindings of integers
%     to constants standing for variables in Vars that evaluate
%     each expression in Exprs to 0.
hilbert(Vars,Exprs) ←
      ∃Bindings: (makeBindings(Vars,Bindings) ⋀
            (∀Expr: in(Expr,Exprs) → eval(Bindings,Expr,0))).

% makeBindings(Vars,Bindings): true if Bindings contains
%     bindings b(Var,Value) for each variable Var in Vars
%     and some integer Value.
makeBindings(Vars,Bindings) ←
      (∃N: length(Vars,N) ⋀ length(Bindings,N))
      ⋀ (∀Var: in(Var,Vars) →
            (∃Value: integer(Value) ⋀ in(b(Var,Value),Bindings)).

% eval(Bindings,Expr,Value): true if the value of
%     Expr is Value.
eval(Bindings,Expr,Value) ←
                  in(b(Expr,Value),Bindings).
eval(Bindings,add(Expr1,Expr2),Value) ←
      ∃V1,V2:     (eval(Bindings,Expr1,V1) ⋀
                   eval(Bindings,Expr2,V2) ⋀
                   add(V1,V2,Value)).
eval(Bindings,mult(Expr1,Expr2),Value) ←
      ∃V1,V2:     (eval(Bindings,Expr1,V1) ⋀
                   eval(Bindings,Expr2,V2) ⋀
                   mult(V1,V2,Value)).
eval(Bindings,power(Expr,N),Value) ←
            ∃V:   (eval(Bindings,Expr,V) ⋀
                   power(V,N,Value)).
```

Figure 4.2

program. Their definitions should be self-explanatory from the program's text. Any other undecidable problem can be used in place of this one.

If for some set of variables v and expressions e, `hilbert`(v, e) is not fixedpoint-implied by `Hilbert`, then any automated procedure for deciding fixedpoint-consistency should find that `Hilbert` augmented with the assertion `~hilbert(X,Y)` is fixedpoint-consistent. Hence, Hilbert's tenth problem becomes decidable, while it has been shown undecidable. $\square$

Because fixedpoint-consistency cannot be decided, Chapter 5 introduces syntactic restrictions on programs to ensure fixedpoint-consistency.

Chapter 5
Fixedpoint-Consistent DIF-Programs

Efficient evaluation of DIF-programs cannot detect fixedpoint-inconsistency. Since fixedpoint-inconsistency is undecidable, syntactic restrictions must be placed on DIF-programs to ensure fixedpoint-consistency. The syntactic restrictions are easily ensured by using a new language containing statements of equivalence, called DEFs (Definite Equivalence Forms), rather than statements of implication (DIFs). DEF-programs are then compiled into DIF-programs that are guaranteed to be fixedpoint-consistent.

An evaluation system for DIF-programs, based on SLD-resolution, is presented. Unlike SLD-resolution, the evaluation system must be able to evaluate universally quantified formulas. Such formulas arise naturally from DEF-programs. For evaluation, universally quantified formulas must be bounded, of the form $\forall X\colon G \rightarrow F$. Evaluation utilizes G as a generator of values, and F as a tester.

Correctness for evaluation of universally quantified formulas may not be attained in certain cases. A combination of enhancements to the evaluation procedure and syntactic requirements are used to ensure correctness.

Theorems 5.5 and 5.6 verify correctness of the evaluation procedure for compiled DIF-programs. Completeness of the evaluation system cannot be obtained for queries on these programs, however.

5.1. Syntax of DEF-Programs

Syntactic restrictions on DIF-programs will ensure fixedpoint-consistency. Two basic ideas underlie the syntactic restrictions; informally they are:

Dual DIFs:

Every DIF $L \leftarrow F$ in a program has a *dual* $\overline{L} \leftarrow \overline{F}$.

Non-conflicting DIFs:

There is at most one closed instance $A \leftarrow F$ of a DIF for each ground atom $A \in HB$.

Each pair of dual DIFs essentially produces a statement of equivalence. The resulting DIF-program is similar to the Clark completion [Cl78], though Clark's approach *implicitly* produces equivalence.

To make these restrictions more easily verified by an automated procedure, a new language is introduced. Programs in this new language are "compiled" into DIF-programs. A *Definite Equivalence Form* (DEF) from $B(A(\Pi,\Sigma))$ is of the form $A \leftrightarrow F$, where A is an atom from $A(\Pi,\Sigma)$ and F is a formula from $B(A(\Pi,\Sigma))$. A *DEF-program* is a finite collection of DEFs. The set of predicate symbols Π must contain a distinguished proposition symbol `true`. Underlying models of DEF-programs always assign the logical constant **t** to `true`. Assertions in DEF-programs take the form $A \leftrightarrow$ `true` or $A \leftrightarrow \sim$`true`. As with DIFs, only variables occurring within the head of a DEF may occur free in the body. All DEFs of the form $p(x_1, \ldots, x_n) \leftrightarrow F$ *define* predicate p. Finally, no DEF is permitted to (re)-define the distinguished proposition `true`.

5.2. Compilation of DEF-Programs

The compilation procedure that produces a compiled DIF-program from a DEF-program generates dual DIFs from each DEF. Compilation also ensures that the compiled DIF-program will be free of conflicting definitions.

Generating Dual DIFs

If $A \leftrightarrow F$ is a DEF in program P_{DEF}, then compilation generates the DIFs $A \leftarrow F$ and $\overline{A} \leftarrow \overline{F}$. Notice that the complement $\overline{A} \leftarrow \overline{F}$ is generated, rather than $\sim A \leftarrow \sim F$. Lemma 4.2 has demonstrated that $I[\overline{F}] = I[\sim F]$ for any sentence F and interpretation I. Using the complement form enables efficient evaluation of compiled DIF-programs. To demonstrate compilation, consider the fragment of a DEF-program below:

Example 5.1

```
% mult(I,J,K): true if IXJ=K; otherwise, false.
mult(0,J,0) ↔ true.
mult(0,J,s(K)) ↔ ~true.
mult(s(I),J,K) ↔ ∃X: mult(I,J,X) ∧ add(X,J,K).
```

From this program fragment, the following DIFs are generated:

```
 mult(0,J,0) ← true.
 mult(0,J,s(K)) ← ~true.
 mult(s(I),J,K) ← ∃X: mult(I,J,X) ∧  add(X,J,K).
~mult(0,J,0) ← ~true.
~mult(0,J,s(K)) ← true.
~mult(s(I),J,K) ← ∀X: mult(I,J,X) → ~add(X,J,K).
```

The meaning preservation lemma, below, formally supports compilation.

Lemma 5.1 (Compilation is Meaning Preserving): Let P_{DEF} be a DEF-program and P_{DIF} the compiled DIF-program. For every interpretation I, $I[P_{DEF}] = I[P_{DIF}]$.

Proof: Logical equivalence of the DEF-program and its compiled form results from the logical equivalence of $A \leftrightarrow F$ and $(A \leftarrow F) \bigwedge (\sim A \leftarrow \sim F)$. □

Non-Conflicting DEF-Programs

A DEF-program has *non-conflicting* DEFs if there is at most one closed instance $A \leftrightarrow F$ of all DEFs for each atom A in the Herbrand Base of the program. For example, the program below is fixedpoint-inconsistent and has DEFs that conflict:

Example 5.2

```
p(X,b)  ↔  q.
p(a,Y)  ↔  r.
q  ↔   true.
r  ↔  ~true.
```

The compiled program contains the DIFs:

```
 p(X,b)  ←   q.
~p(a,Y)  ←  ~r.
  q  ← true.
 ~r  ← true.
```

When `true` is assigned **t** in all interpretations, the conflict arises for the atom `p(a,b)`. Without this conflict, the program would be fixedpoint-consistent.

Conflicting DEFs can be found with a syntactic test. DEFs $A \leftrightarrow F$ and $B \leftrightarrow G$ *overlap* if:

(1) the DEFs contain disjoint sets of variables, and

(2) A and B unify.

A DEF-program is *overlapping* if it contains distinct DEFs whose variants unify.

Lemma 5.2 (Non-Overlapping Programs are Non-Conflicting): If a DEF-program is non-overlapping, it is non-conflicting.

Proof: We will show that a conflicting DEF-program is overlapping. Suppose the DEF-program is from $B(A(\Pi, \Sigma))$. If the program is conflicting, there are distinct closed instances $(A_1 \leftrightarrow F_1)\sigma_1$ and $(A_2 \leftrightarrow F_2)\sigma_2$ of DEFs such that $A_1\sigma_1 = A_2\sigma_2 = A$. Since $(A_1 @ \Sigma) \cap (A_2 @ \Sigma)$ is nonempty, Lemma 2.8 provides that A_1 and A_2 are unifiable. Therefore, the program is overlapping. □

In Example 5.2, the program is overlapping and has conflicting definitions.

Fixedpoint-Consistency is Attained

Combination of dual DIFs and non-conflicting DEFs ensures fixedpoint-consistency.

Lemma 5.3 (Non-conflicting DEF-programs are fixedpoint-consistent): If P_{DEF} is a non-conflicting DEF-program, then $P_{DEF} \cup \{\texttt{true}\}$ is fixedpoint-consistent.

Proof: Notice that addition of the assertion `true` to the DEF-program P_{DEF} assures assignment of **t** to `true`. By Lemma 5.1, compilation of P_{DEF} to a DIF-program P_{DIF} is meaning-preserving. Therefore, if $P_{DIF} \cup \{\texttt{true}\}$ is fixedpoint-consistent, so is $P_{DEF} \cup \{\texttt{true}\}$. Let $P = P_{DIF} \cup \{\texttt{true}\}$. We now prove by induction that $T_P \uparrow \alpha$ is defined for all ordinals α. In the basis case, $T \uparrow 0 = \varnothing$. For the induction hypothesis, assume that $T \uparrow \alpha$ is defined. Consider a successor ordinal $\alpha + 1$.

$T\uparrow(\alpha+1)$ is undefined in the following cases:

(a) There are closed instances $A \leftarrow F_1$ and $\sim A \leftarrow F_2$, and $T\uparrow\alpha[F_1] = T\uparrow\alpha[F_2] = \mathbf{t}$. Since the DEF-program is non-conflicting, there is only one closed instance $A \leftrightarrow F_1$ of all DEFs. Therefore, $\sim A \leftarrow F_2$ is the dual of $A \leftarrow F_1$, and $F_2 = \overline{F_1}$. By Lemma 4.2, $T\uparrow\alpha[\sim F_1] = T\uparrow\alpha[F_2]$. Hence, $T\uparrow\alpha[F_1] = T\uparrow\alpha[F_2]$ only when both are undefined.

(b) There is a closed instance $\sim$`true`$\leftarrow F$, and $T\uparrow\alpha[F] = \mathbf{t}$. (Recall that P contains the assertion `true`.) However, P_{DEF} cannot contain definitions for the proposition `true`.

Therefore, $T\uparrow(\alpha+1)$ is defined. The induction holds for limit ordinals also. Using the contrapositive form of Lemma 4.6, since $T\uparrow\alpha$ is defined for every ordinal α, $T\uparrow\alpha = lfp$ for some ordinal α. □

From the statement of Lemma 5.3, formula F is *fixedpoint-implied* from a DEF-program P_{DEF} if compilation produces DIF-program P_{DIF} and $\forall F$ is fixedpoint-implied from $P_{DIF} \cup \{$`true`$\}$.

By Lemma 5.3, non-conflicting DEF-programs are always fixedpoint-consistent. And Lemma 5.2 assures that the overlap test can detect conflicting DEFs. Finally, compilation of DEF-programs is meaning preserving, by Lemma 5.1. So any compiled DIF-program produced from a fixedpoint-consistent DEF-program is also fixedpoint-consistent. We have therefore ensured fixedpoint-consistency of DIF-programs through syntactic conditions on DEF-programs. These conditions are only sufficient to ensure fixedpoint-consistency. As shown in Lemma 4.6, necessary and sufficient syntactic conditions do not exist. Finally, as discussed in Chapter 4, because compiled DIF-programs may contain universal quantifiers in bodies of DIFs, such programs may possess infinite closure ordinals. The presence of infinite closure ordinals eliminates possibilities for completeness of the evaluation system.

5.3. Evaluation of DEF-Programs

Having guaranteed fixedpoint-consistency of DEF-programs, we now discuss the evaluation system for DEF-programs. Given that a DEF-program can be compiled into a DIF-program, queries on the DEF-program are evaluated against the DIF-program.

In order to provide a feasible evaluation system, all universally quantified formulas within a generated DIF-program are required to be "bounded," of the form $\forall X\colon G \rightarrow F$. In essence, this restriction permits computation within the Herbrand Universe. Since the bounded formula $\forall X\colon \texttt{true} \rightarrow F$ is logically equivalent to $\forall X\colon F$, requiring bounded formulas is not a restriction on expressiveness.

The bounded universal quantifier is amenable to computation. Essentially, bounded universally quantified queries of the form $\forall X\colon G \rightarrow F$ are interpreted as having a *generator* G of X-values and a *tester* F of the generated X-values. Generation and testing of values may be conducted sequentially or in parallel. Evaluation of universally quantified formulas with a generate-and-test procedure is limited because when the set of values satisfying the generator is infinite, the computation may not terminate. Computability results prohibit completeness of any procedure, as seen in Section 5.6.

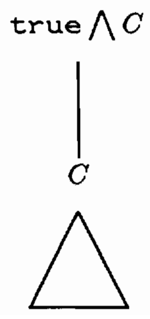
The evaluation system for DIF-programs is based on construction of full search trees and fair traversal of these trees. Many attributes of full search trees for DIF-programs are similar to full search trees constructed by SLD-resolution. Every node of any well-formed full search tree is labeled by a conjunction of formulas. Edges in the tree are labeled by substitutions. A success node is labeled by the empty conjunction, denoted □. The empty conjunction is assigned the logical value **t**. The value of a success path is the composition of all substitutions along edges from the root to a success node.

To define the *well-formed* full search trees, consider all possible structures of the label at the root (C is a conjunction of formulas):

□: A full search tree consisting only of the node labeled by the empty conjunction is well-formed.

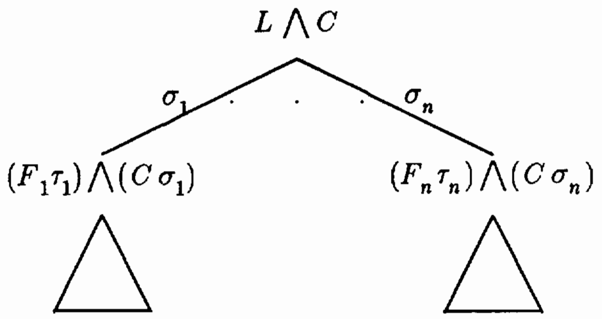
$\texttt{true} \bigwedge C$:

If the full search tree C is well-formed, the following tree is well-formed:



$L \bigwedge C$ (L a literal):

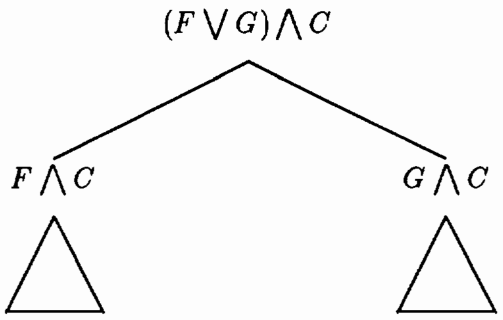
When L is a literal distinct from `true`, and all trees $(F_i \tau_i) \bigwedge (C \sigma_i)$ for $1 \leq i \leq n$ are well-formed, the following tree is well-formed:


$L \bigwedge C$
σ_1
σ_n
$(F_1 \tau_1) \bigwedge (C \sigma_1)$
$(F_n \tau_n) \bigwedge (C \sigma_n)$

For this diagram, $L_1 \leftarrow F_1 \cdots L_n \leftarrow F_n$ are variants of all clauses in program P for which L unifies with each L_i $(1 \leq i \leq n)$. There is a unique (modulo renaming) mgci for each pair of literals L and L_i. Section 2.3 describes a decomposition producing substitutions σ_i and τ_i such that $L \sigma_i = L_i \tau_i$. When there are no variants of clauses whose heads unify with L, $n = 0$, and the tree consists of only the node $L \bigwedge C$.

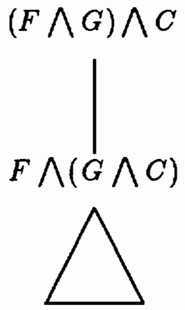
$(F \bigvee G) \bigwedge C$:

When trees $F \bigwedge C$ and $G \bigwedge C$ are well-formed, the following tree is well-formed:



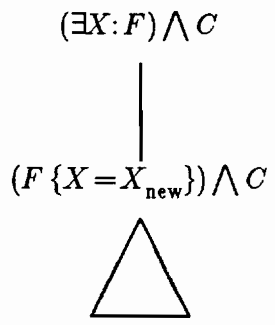
$(F \bigwedge G) \bigwedge C$:

When the tree labeled by $F \bigwedge (G \bigwedge C)$ is well-formed, the following tree is well-formed:



$(\exists X : F) \bigwedge C$:

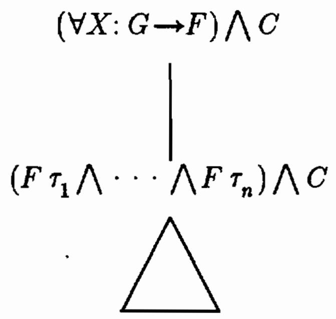
When X_{new} is a variable occurring nowhere else, and $(F\{X = X_{\text{new}}\}) \bigwedge C$ has a well-formed tree, the following tree is well-formed:


$(\exists X{:}F)\bigwedge C$
$(F\{X=X_{\text{new}}\})\bigwedge C$

The substitution $\{X=X_{\text{new}}\}$ disambiguates multiple occurrences of X in different binding scopes by setting all free occurrences of X to a unique variable X_{new}.

$(\forall X{:}\,G\rightarrow F)\bigwedge C$:

When the only free variable in G is X, and the tree $(F\,\tau_1\bigwedge\cdots\bigwedge F\,\tau_n)\bigwedge C$ is well-formed, the following tree is well-formed:


$(\forall X{:}\,G\rightarrow F)\bigwedge C$
$(F\,\tau_1\bigwedge\cdots\bigwedge F\,\tau_n)\bigwedge C$

where $\tau_1, \ldots, \tau_n$ are the values of all success paths in the full search tree for G. Section 6.3.2 will suggest methods to evaluate universally quantified formulas with occurrences of free variables other than the universally quantified variable in the generator.

To demonstrate the evaluation procedure, consider the DEF-program below:

```
% divp(I,J): true if I divides J evenly;
%    otherwise, false.
divp(I,J) ↔ ∃X: le(X,J) ∧ mult(X,I,J).

% le(I,J): true if I≤J; otherwise, false.
le(0,J) ↔ true.
le(s(I),0) ↔ ~true.
le(s(I),s(J)) ↔ le(I,J).
```

The DIFs present in the compiled DIF-program of interest for this demonstration are the following:

```
~divp(I,J) ← ∀X: le(X,J) → ~mult(X,I,J).
  le(0,J) ← true.
  le(s(I),0) ← ~true.
  le(s(I),s(J)) ← le(I,J).
```

Consider the query:

```
~divp(s(s(0)),s(s(s(0)))).
```

The only direct descendent of this query in its full search tree is a node labeled by the query:

```
∀X:le(X,s(s(s(0)))) → ~mult(X,s(s(0)),s(s(s(0)))).
```

To produce the full search tree for this query, a full search tree for the generator `le(X,s(s(s(0))))` is produced. This is presented in Figure 5.1. As expected, the answer substitutions obtained from the generator are:

Full Search Tree for `le(X,s(s(s(0))))`

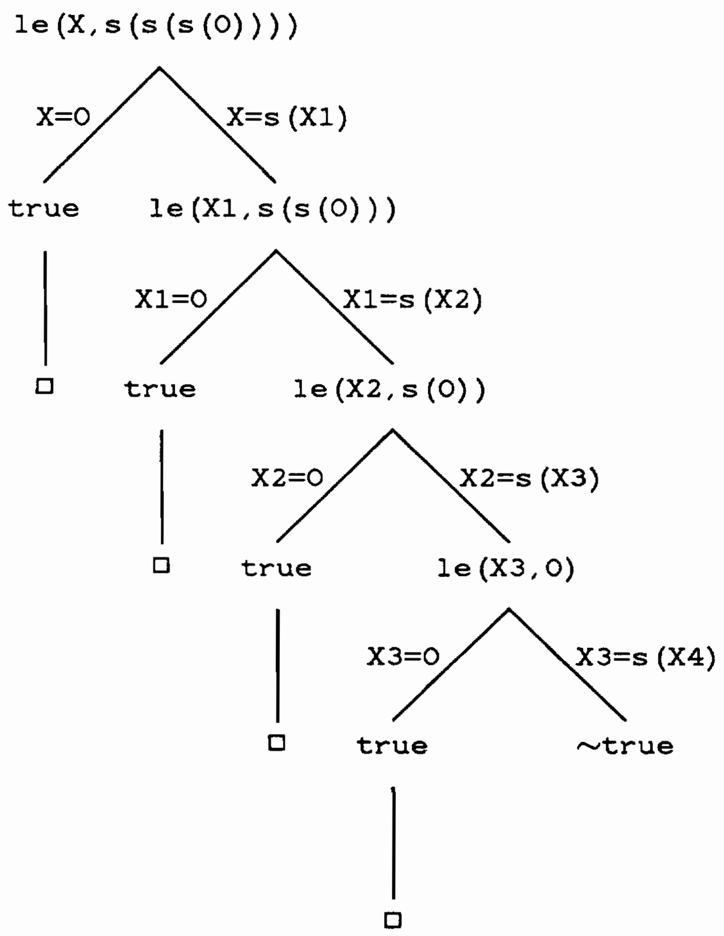


Figure 5.1

```
X = 0,

X = s(0),
```

`X = s(s(0))`, and

`X = s(s(s(0)))`.

The direct descendent of the universally quantified query is then:

```
    ~mult(0,s(s(0)),s(s(s(0))))
 ∧ ~mult(s(0),s(s(0)),s(s(s(0))))
 ∧ ~mult(s(s(0)),s(s(0)),s(s(s(0))))
 ∧ ~mult(s(s(s(0))),s(s(0)),s(s(s(0))))
```

Using the definitions provided in Example 5.1, this conjunction produces a full search tree with a success path. Hence, the entire tree for the original query has a success path, which is to be expected since 2 does not evenly divide 3.

5.4. Resolving Incorrectness of Universally Quantified Queries

As specified here, the full search tree construction is not generally correct for universally quantified queries. There are three cases where the incorrectness arises:

(1) The generator produces a value that is more general than some value satisfying the tester. Utilization of term-matching within the tester, rather than unification, ensures that generated values will not be too general.

(2) The generator produces too few values. A self-coverage requirement ensures that every ground atom can be described by a program. Self-coverage has a syntactic test.

(3) The generator produces no values, because the universally quantified variable does not occur free in the generator. The entire universally quantified formula

can be rewritten in a meaning-preserving manner to resolve this problem.

These three problems and their solutions are discussed in the next three sections.

Overly-General Generated Values

Consider the DEF-program below:

Example 5.3

```
p(X) ↔  true.
q(a) ↔  true.
q(b) ↔ ~true.
```

The query ∀X: p(X)→q(X) succeeds but is not fixedpoint-implied by the program. The problem is that the tester q(X) performs full unification on the X-value generated by p(X). Instead, for any generated X-value t, the tester should be satisfied by a value t' more general than t, i.e. $[t'] \gtrsim [t]$. To determine when $[t'] \gtrsim [t]$ holds, a form of one-sided unification, commonly called *term-matching* is used. Implementation of this enhancement will be discussed in Section 6.3.1.

This correctness problem can be avoided by ensuring that every generated X-value is ground using a "type predicate" within each generator. The type predicate will be true for every ground term in the Herbrand Universe. For Example 5.3, the definition of the type predicate is:

```
hu(a).
hu(b).
```

Now the query ∀X: (p(X)∧hu(X))→q(X) fails, because the generator produces values a and b for X, and the query q(a)∧q(b) fails.

Recall that negated queries solved through negation by failure are also required to be ground. Requiring generated X-values to be ground is far less stringent. Negated queries still yield answer substitutions under constructive negation. And a type predicate can be easily added to the generator to produce ground values during evaluation. Nonetheless, this solution is less desirable from the standpoint of performance than term-matching. In the extreme, generation of only ground values could result in an infinite stream, when the stream generated for non-ground values would have been finite.

Insufficient Generated Values

A more difficult problem is posed by the following program:

```
p(a) ↔  true.
q(a) ↔  true.
q(b) ↔ ~true.
```

The query `∀X: p(X)→q(X)` is assigned the undefined value by the least fixed-point, since `p(b)` is undefined. However, the evaluation procedure produces a full search tree with a success path.

The problem here is that definitions for the p predicate did not describe all elements of the Herbrand Universe. To resolve this problem, a new requirement is placed on all predicates defined within a program. A DEF-program is *self-covering* if there is some closed instance $A \leftrightarrow F$ of a DEF for each closed atom $A \in HB$. As described below, self-coverage can be decided, and therefore incorporated into compilation of DEF-programs into DIF-programs. Since compilation also checks for non-overlapping DEFs, conjunction of the two properties requires the existence of exactly one closed instance $A \leftrightarrow F$ of some DEF for each atom $A \in HB$.

The test for self-coverage of a program is performed for each predicate symbol occurring in the program. A predicate p is self-covering if there is a closed instance $p(x) \leftrightarrow F$ of some DEF for each ground atom $p(x) \in HB$. A program is therefore self-covering iff every predicate occurring in the program is self-covering.

When a term or atom is represented as a directed acyclic graph, the *nesting depth* of a term is the length of the longest path from the root node to a leaf. For example, if `p` is a proposition, the nesting depth is 0. Also, the atom `p(f(c),b)` has nesting depth 2. Suppose predicate p is defined by the DEFs $p(x_1) \leftrightarrow F_1, \ldots, p(x_n) \leftrightarrow F_n$. If each atom $p(x_i)$ occurring in the head of some DEF defining predicate p has nesting depth d_i $(1 \leq i \leq n)$ the *maximum nesting depth* for predicate p is $m_p = \max_{1 \leq i \leq n} d_i$. In Example 5.1, $m_{\texttt{mult}} = 2$. Next, we define a special operation that selects from a set of atoms those of limited nesting depth. If S is a set of atoms, $S \% d = \{A \in S \mid \text{the nesting depth of } A \text{ is at most } d\}$.

Predicate p satisfies the *self-coverage test* if there is a closed instance $p(x) \leftrightarrow F$ of some DEF in the program for every atom $p(x) \in HB \% (m_p + 1)$. In Example 5.1 the self-coverage test mandates that all ground atoms $\texttt{mult}(x_1, x_2, x_3)$ with nesting depth at most 3 shall be matched against DEFs in the program. Example 5.1 satisfies the self-coverage test because every such atom matches with the head of some DEF defining `mult`. This test always terminates, since $HB \% (m_p + 1)$ is always finite.

Lemma 5.4 (Correctness of Self-Coverage Test): Suppose a program P is from $B(A(\Pi,\Sigma))$ and predicate $p \in \Pi$. The self-coverage test for predicate p succeeds iff predicate p is self-covering in program P.

Proof:

($\rightarrow$) Suppose the self-coverage test for predicate p is satisfied. Consider an atom $p(z) \in HB$ with nesting depth d. If $d \leq m_p + 1$ then the self-coverage test verifies that $p(z)$ is a ground instance of the head of some DEF in P.

Otherwise, $d > m_p + 1$. We now define a *strip* function that produces an atom of depth $m_p + 1$ from $p(z)$ by replacing every subterm at depth $m_p + 1$ by a unique variable. This function is defined recursively as follows:

$strip(c, d) = c$ for all constants c and strip depths d.

$strip(f(x_1, \ldots, x_n), 0) = X_{\text{new}}$ where X_{new} is a unique variable.

$strip(f(x_1, \ldots, x_n), d+1) = f(strip(x_1, d), \ldots, strip(x_n, d))$.

For example, *strip*(`f(f(f(a,b),b),a)`,2) = `f(f(X,b),a)`.

Let $p(y) = strip(p(z), m_p + 1)$. Consider a ground atom $p(y_0) \in (p(y) @ \Sigma) \% (m_p + 1)$. This ground atom $p(y_0)$ is obtained from $p(z)$ by replacing every non-constant subterm at depth $m_p + 1$ by a constant from Σ. Since predicate p satisfies the self-coverage test and $p(y_0)$ is a ground atom with depth $m_p + 1$, there is a DEF $p(x) \leftrightarrow F$ such that $[p(x)] \gtrsim [p(y_0)]$. Since $p(x)$ has depth at most m_p, matching of $p(x)$ with $p(y_0)$ does not depend on the particular constants chosen to replace variables occurring at depth $m_p + 1$ of $p(y)$. So $[p(x)] \gtrsim [p(y)]$. By Lemma 2.9,

$(p(x) @ \Sigma) \supseteq (p(y) @ \Sigma)$. Since $p(z) \in (p(y) @ \Sigma)$, $p(z) \in (p(x) @ \Sigma)$, and therefore a closed instance of $p(x) \leftrightarrow F$ has head $p(z)$.

($\leftarrow$) Trivial. $\square$

The self-coverage test terminates in all cases, even when the Herbrand Universe is infinite. Section 6.2, will present improvements incorporating data types.

DEFs must usually be added to a program to satisfy the self-coverage test. The extreme case occurs with a program containing just the DEF $p(c_1, \ldots, c_n) \leftrightarrow \texttt{true}$, where the c_i are all distinct constants. Satisfaction of the self-coverage and overlap tests would require $n^n - 1$ additional DEFs. However, with an implementation of inequality these additional DEFs would not be required. Inequality is discussed further in Section 6.1.

The self-coverage property also affects the evaluation system presented earlier in this section. Suppose a DEF-program compiles successfully to a DIF-program, and satisfies the self-coverage test. When a literal distinct from `true` and `~true` is selected as the root label of a full search tree, it will always unify with the head of some DIF. The only nodes without descendents are labeled with the empty conjunction $\square$, or with $\sim\texttt{true} \bigwedge C$, for some conjunction C.

No Generated Values

The final instance where the evaluation procedure for universally quantified queries is incorrect occurs when there are no generated values, due to the absence of free occurrences of the universally quantified variable in the generator. Consider, for example, a program consisting only of the DEF `p(c)` $\leftrightarrow$ `~true` and the query `∀X: true→p(X)`, equivalent to `∀X:p(X)`. The evaluation procedure produces no values for `X`, so the query succeeds even though there is a value `c` that disputes the query.

The universally quantified variable must occur free in the generator of any universally quantified formula, which is a decidable property. When this property is violated by a formula, as in the example above, there are two remedies.

When the tester contains a free occurrence of the universally quantified variable, convert the original formula $\forall X\colon G \rightarrow F$ to the logically equivalent $\forall X\colon \overline{F} \rightarrow \overline{G}$. The generator of the new formula now contains a free occurrence of the universally quantified variable. In the example above, the query is converted to `∀X: ~p(X) →~true`. When *evaluated, the new generator produces the value* `c`, *giving* rise to the conjunction `~true`, which correctly fails. This strategy is not a complete remedy, since the new generator may produce an infinite stream of values.

When the tester does not contain a free occurrence of the universally quantified variable, the universal quantifier is superfluous. The quantified variable occurs neither in the generator nor tester. Therefore, the original formula $\forall X\colon G \rightarrow F$ can be rewritten to the logically equivalent formula $\overline{G} \vee F$.

Summary

Evaluation of universally quantified queries poses three different correctness problems. First, it is possible to generate overly-general values. This problem can be avoided by utilizing term-matching, or requiring generation only of ground terms. Second, it is possible to generate an insufficient number of values. This problem is avoided by requiring programs to be self-covering, a decidable property. Finally, it is possible that no values are generated, due to the absence of a free occurrence of the universally quantified variable in the generator. This problem can be detected, and the violating formula rewritten to resolve this problem. Resolution of these concerns is sufficient to demonstrate correctness of the evaluation procedure. We assume at this point that compilation invokes the following syntactic tests:

(1) Overlapping DEFs.

(2) Self-coverage.

(3) All universal quantifiers bounded.

(4) Generators of universally quantified formulas contain free occurrences of the universally quantified variable.

Having remedied faults with the evaluation procedure for universally quantified queries, the next section presents proof that the entire evaluation system is correct.

5.5. Correctness of the Evaluation System

Evaluation of a universally quantified query requires searching the generator's full search tree for all answer substitutions. The search examines every leaf of the full search tree; therefore, the full search tree of the generator must be finite. All paths of a finite full search tree are traversed to detect the presence of success nodes, thereby producing all generated values. Thus, correctness of the evaluation system relies on correctness with respect to finite full search trees.

For the following two correctness theorems, we temporarily make some definitions. Consider a DEF-program from $B(A(\Pi, \Sigma))$. Let $Ans(F) = \bigcup\{F\,\alpha @ \Sigma \mid \alpha \text{ is an answer for } F\}$, and let $\overline{Ans(F)} = (F @ \Sigma) - Ans(F)$. Informally, $Ans(F)$ is the set of closed instances of all answers obtained for query F. $\overline{Ans(F)}$ is the complement of $Ans(F)$, relative to all closed instances of F. For example, if $\Sigma = \{\mathtt{0}^{(0)}, \mathtt{s}^{(1)}\}$, $F =$ `~lt(X,s(0))`, and the only answer to query F is `X = s(X1)`, then $Ans(F) = \{$`~lt(s(0),s(0)),~lt(s(s(0)),s(0)),...`$\}$. Also $\overline{Ans(F)} = \{$`~lt(0,s(0))`$\}$.

Theorem 5.5 (Correctness for Finite Full Search Trees): For any DEF-program from $B(A(\Pi, \Sigma))$ that compiles successfully to a DIF-program P, if a query F has a finite full search tree, then:

(1) $lfp[F\,\sigma] = \mathbf{t}$ iff $F\,\sigma \in Ans(F)$.

(2) $lfp[F\,\sigma] = \mathbf{f}$ iff $F\,\sigma \in \overline{Ans(F)}$.

Proof: By induction on the height of all full search trees. We define the height of a full search tree for the formula $\forall X\colon G \rightarrow F$ to be the maximum height of the trees for G and the ensuing conjunction produced from F.

A tree of height 0 can be labeled by `~true` $\bigwedge C$ or $\Box$. In the former case, $Ans(F) = \varnothing$, and so $lfp[F\,\sigma] = \mathbf{f}$ for all $F\,\sigma \in \overline{Ans(F)}$, since $lfp[$`~true` $\bigwedge C] = \mathbf{f}$. In the latter case, the answer substitution is ϵ. The theorem holds vacuously, since the empty conjunction is assigned value $\mathbf{t}$.

For the induction hypothesis, assume the theorem is true for all trees of height at most h. Consider a full search tree for a query F with height $h+1$.

$\underline{F = \mathtt{true}}$:

The direct descendent of the root is a leaf labeled by the empty conjunction. This leaf is a success node, and the edge from the root is a success path with value ϵ. Therefore, $Ans(F) = \{\mathtt{true}\}$ and $lfp[\mathtt{true}] = \mathbf{t}$. Also, $\overline{Ans(F)} = \emptyset$.

$\underline{F = \forall X{:}F_1 \rightarrow F_2}$:

The direct descendent of the root is a full search tree for $G = F_2\alpha_1 \bigwedge \cdots \bigwedge F_2\alpha_n$, where $\alpha_1, \ldots, \alpha_n$ are all answer substitutions in the full search tree for F_1. Every answer substitution for G is an answer substitution for F. So $G\sigma \in Ans(G)$ iff $F\sigma \in Ans(F)$. We are assuming that the only free variable in F_1 is X. This assumption is weakened in Section 6.3. For convenience, let $F_i(t)\sigma$ denote the formula $F_i(\{X=t\} \circ \sigma)$, for $i = 1,2$. By the definition of height for universally quantified queries, both F_1 and G have full search trees of height at most h. Therefore, the theorem holds for F_1 and G.

Let σ be a substitution such that $F\sigma \in Ans(F)$. Then $G\sigma \in Ans(G)$. Since $G\sigma = F_2(\alpha_1 \circ \sigma) \bigwedge \cdots \bigwedge F_2(\alpha_n \circ \sigma)$, let $G\sigma = F_2(t_1)\sigma \bigwedge \cdots \bigwedge F_2(t_n)\sigma$, where each t_i $(1 \leq i \leq n)$ is a ground term. Specific terms t_i can be selected because only term-matching is employed in obtaining answers for F_2. Consider any ground term t. If $t = t_i$ for some $1 \leq i \leq n$, then by the induction hypothesis, $lfp[G\sigma] = \mathbf{t}$, and $lfp[F_2(t)\sigma] = \mathbf{t}$. Also, $F_1(t_i) \in (F_1\alpha_i @ \Sigma)$, so $F_1(t) \in Ans(F_1)$, and again by the induction hypothesis, $lfp[F_1(t)\sigma] = \mathbf{t}$. Therefore, $lfp[(F_1 \rightarrow F_2)(t)\sigma] = \mathbf{t}$. If $t \neq t_i$ for all

$1 \leq i \leq n$, then $F_1(t) \in \overline{Ans(F_1)}$, so $lfp[F_1(t)\sigma] = \mathbf{f}$, by the induction hypothesis. Therefore, $lfp[(F_1 \rightarrow F_2)(t)\sigma] = \mathbf{t}$. This holds for all terms t, so $lfp[F\sigma] = \mathbf{t}$.

Alternatively, $F\sigma \in \overline{Ans(F)}$, so $G\sigma \in \overline{Ans(G)}$. By the induction hypothesis, $lfp[G\sigma] = \mathbf{f}$, so $lfp[F_2(t_i)\sigma] = \mathbf{f}$, for some $1 \leq i \leq n$. As has been shown, $lfp[F_1(t_i)\sigma] = \mathbf{t}$. Therefore, $lfp[(F_1 \rightarrow F_2)(t_i)\sigma] = \mathbf{f}$, and $lfp[F\sigma] = \mathbf{f}$.

<u>F is a literal</u>:

When literal F is distinct from `true` and `~true`, the direct descendents of the root are trees labeled $F_1\tau_1, \ldots, F_n\tau_n$. There must be variants $L_1 \leftarrow F_1, \ldots, L_n \leftarrow F_n$ in the compiled DIF-program, where $F\sigma_i = L_i\tau_i$ for all $1 \leq i \leq n$. Then $F(\sigma_i \circ \sigma) \in Ans(F)$ iff $F_i(\tau_i \circ \sigma) \in Ans(F_i\tau_i)$. The induction hypothesis holds for each direct descendent $F_i\tau_i$ of F. Therefore, $lfp[F_i(\tau_i \circ \sigma)] = \mathbf{t}$ for all $F_i(\tau_i \circ \sigma) \in Ans(F_i\tau_i)$. Since lfp is a fixedpoint, $lfp[L_i(\tau_i \circ \sigma)] = \mathbf{t}$. $F\sigma_i = L_i\tau_i$, so $lfp[F(\sigma_i \circ \sigma)] = \mathbf{t}$, and $F(\sigma_i \circ \sigma) \in Ans(F)$.

On the other hand, suppose $F_i(\tau_i \circ \sigma) \in \overline{Ans(F_i\tau_i)}$. By the induction hypothesis, $lfp[F_i(\tau_i \circ \sigma)] = \mathbf{f}$. Consequently $lfp[\overline{F_i}(\tau_i \circ \sigma)] = \mathbf{t}$. Because program P is non-conflicting and self-covering, there is exactly one closed instance $(\overline{L_i} \leftarrow \overline{F_i})(\tau_i \circ \sigma)$ of a DIF in the compiled program. Since lfp is a fixedpoint, $lfp[\overline{L_i}(\tau_i \circ \sigma)] = \mathbf{t}$. Therefore, $lfp[L_i(\tau_i \circ \sigma)] = \mathbf{f}$, and $F(\sigma_i \circ \sigma) \in \overline{Ans(F)}$.

<u>$F = F_1 \wedge F_2$</u>:

For every answer α of F, $\alpha = \alpha_1 \circ \alpha_2$, where α_1 is an answer for F_1 and α_2 is an answer for F_2. The full search tree can be divided into an upper prefix solving the query F_1, and lower subtrees solving queries of the form $F_2\alpha_1$, where α_1 is an answer to F_1. The induction hypothesis holds for the upper and lower segments of the full search tree.

Consider $F\sigma \in Ans(F)$. Then $F\sigma \in (F(\alpha_1 \cdot \alpha_2) @ \Sigma)$. From the tree construction, $F_1\sigma \in Ans(F_1)$ and $F_2\sigma \in Ans(F_2\alpha_1)$. By the induction hypothesis, $lfp[F_1\sigma] = \mathbf{t}$ and $lfp[F_2\sigma] = \mathbf{t}$, so $lfp[F\sigma] = \mathbf{t}$.

Alternatively, consider $F\sigma \in \overline{Ans(F)}$. Then, from the tree construction, either (i) $F_1\sigma \in \overline{Ans(F_1)}$, or (ii) $F_2\sigma \in \overline{Ans(F_2\alpha_1)}$ for some answer α_1 of F_1. In case (i), $lfp[F_1\sigma] = \mathbf{f}$. In case (ii), $lfp[F_2\sigma] = \mathbf{f}$. Therefore, in both cases $lfp[F\sigma] = \mathbf{f}$.

$\underline{F = F_1 \vee F_2}$:

The direct descendents of F are full search trees for F_1 and F_2. The induction hypothesis holds for these subtrees. Any answer substitution for F_1 is an answer substitution for F, and similarly for F_2. Suppose $F_1\sigma \in Ans(F_1)$. By the induction hypothesis, $lfp[F_1\sigma] = \mathbf{t}$, so $lfp[F\sigma] = \mathbf{t}$. Also, $lfp[F\sigma] = \mathbf{t}$ if $F_2\sigma \in Ans(F_2)$. Alternatively, suppose $F_1\sigma \in \overline{Ans(F_1)}$ and $F_2\sigma \in \overline{Ans(F_2)}$. By the induction hypothesis, $lfp[F_1\sigma] = lfp[F_2\sigma] = \mathbf{f}$. So $lfp[F\sigma] = \mathbf{f}$.

$\underline{F = \exists X : F_1}$:

The direct descendent of F is a full search tree for F_1 with X renamed to a new unique variable. Any answer substitution for F_1 is an answer substitution for F. Consider any substitution σ such that $F\sigma \in (F @ \Sigma)$. If there is a ground term t where $F_1(t)\sigma \in Ans(F_1)$, the induction hypothesis provides $lfp[F_1(t)\sigma] = \mathbf{t}$. Therefore, $lfp[(\exists X : F_1)\sigma] = \mathbf{t}$. On the other hand, suppose there is no ground term t where

$F_1(t)\sigma \in Ans(F_1)$. Then $F_1(t)\sigma \in \overline{Ans(F_1)}$ for all ground terms t. By the induction hypothesis, $lfp[F_1(t)\sigma] = \mathbf{f}$ for all ground terms t. Therefore, $lfp[(\exists X{:}F_1)\sigma] = \mathbf{f}$. □

The correctness theorem for finite full search trees expands to general full search trees, but must be weakened because of the possibility of infinite paths.

Theorem 5.6 (Correctness for General Full Search Trees): For any DEF-program from $B(A(\Pi, \Sigma))$ that compiles successfully to a DIF-program P, $lfp[F\sigma] = \mathbf{t}$ for all $F\sigma \in Ans(F)$.

Proof: Similar to the correctness proof for finite full search trees, except induction is now over the length of a success path.

5.6. Incompleteness of any Evaluation System for DEFs

There is no possibility of finding a complete execution system for DEF-programs. To prove this point, a program that defines a non-r.e. relation is presented. Any complete evaluation system would therefore accept a non-r.e. language, which is not possible.

A *configuration* $x\ q\ y$ of a Turing machine (TM) is a situation where the TM is in state q, string x precedes the tape head, and string y follows the tape head. The *transition* relation $c \vdash_m c'$ indicates that TM m can move from configuration c to configuration c' in one step. The reflexive and transitive closure of the transition relation $c \vdash_m^* c'$ indicates that TM m can proceed from configuration c to configuration c' in any number of steps (possibly zero). An initial configuration for a TM m is $q_0 w$, where q_0 is m's initial state and w is the input string. An accepting configuration for a TM m is $x\ q_f\ y$, where q_f is an accepting (final) state and x and y are strings. A TM m *accepts* a string w if $q_0 w \vdash_m{}^* x\ q_f\ y$, where q_0 and q_f are m's initial and accepting states, respectively. A TM m accepts a language L if and only if m accepts only the strings in L.

Consider the following DEFs taken from a DEF-program called NonRE:

```
% accept(M,C): true iff TM M can accept from configuration C.
accept(M,C) ↔
     final(M,C) ⋁ (∃C': transit(C,M,C') ⋀ accept(M,C')).
```

The DEF-program NonRE also contains definitions of predicates final and transit. Informally, final(m,c) is true if c is an accepting configuration for TM m; otherwise, final(m,c) is false. Also, transit(c,m,c') is true if $c \vdash_m c'$; otherwise, transit(c,m,c') is false. When compiled, the DEF defining accept produces the following DIFs:

```
  accept(M,C) ←
       final(M,C) ⋁ (∃C': transit(C,M,C') ⋀  accept(M,C')).
~accept(M,C) ←
     ~final(M,C) ⋀ (∀C': transit(C,M,C') → ~accept(M,C')).
```

Notice that the second generated DIF above contains a universal quantifier. This is essential in demonstrating incompleteness.

The following lemma demonstrates that accept defines the intended relation:

Lemma 5.7: For any TM m and configuration c, accept(m,c) is fixedpoint-implied by NonRE if a final configuration can be reached via the relation $\vdash_m{}^*$ from initial configuration c; otherwise, ~accept(m,c) is fixedpoint-implied.

Proof:

There are two cases, depending on whether a final configuration can be reached.

(1) Suppose that a final configuration can be reached by TM m from configuration c. Either c is an accepting configuration, or $c \vdash_m c'$ and m can accept from c'.

(2) Suppose that a final configuration cannot be reached by TM m from configuration c. Then c is not an accepting configuration and whenever $c \vdash_m c'$, m cannot accept from c'. Generally, this argument requires infinite transitions by m and infinite closure ordinals. □

This lemma will be used to demonstrate that DEF-programs can describe non-r.e. languages. As in [HU79], let $<m>$ be the string encoding TM m. Consider the following languages:

$$L_{ne} = \{<m> \mid L(m) \neq \varnothing\}$$
$$L_e = \{<m> \mid L(m) = \varnothing\}$$

It has been shown [HU79] that L_{ne} is r.e. and not recursive, and L_e is not r.e. To describe these languages, program NonRE is augmented with the following DEF:

```
% ne(W): true if string W is a valid TM and L(W) nonempty;
%     otherwise, false.
ne(W) ↔ ∃C: (∃X: initial(W,X,C)) ∧ accept(W,C).
```

This DEF is compiled into the following DIFs:

```
 ne(W) ← ∃C: (∃X: initial(W,X,C)) ∧  accept(W,C).
~ne(W) ← ∀C: (∃X: initial(W,X,C)) → ~accept(W,C).
```

Again, note that a generated DIF includes a universal quantifier, making completeness of any evaluation procedure doubtful. Program NonRE also contains a definition for initial: initial(w,x,c) is true if $w = <m>$ is a valid encoding of a TM and $c = q_0 x$ is an initial configuration for TM m; otherwise, initial(w,x,c) is false.

As the next lemma demonstrates, this program describes both languages L_{ne} and L_e.

Lemma 5.8 (NonRE accepts L_e):

(1) Any string $w \in L_{ne}$ iff ne(w) is fixedpoint-implied by NonRE.

(2) Any string $w \in L_e$ iff ~ne(w) is fixedpoint-implied by NonRE.

Proof:

(1) $<m> \in L_{ne}$ iff there is an initial configuration c that can reach a final configuration for TM m. By the previous lemma, this holds iff accept(m,c) is fixedpoint-implied by NonRE. Using the DEF defining ne, accept(m,c) is fixedpoint-implied from the program iff ne(m) is fixedpoint-implied.

(2) $<m> \in L_e$ iff there is no initial configuration capable of reaching a final configuration. By Lemma 5.7 ~accept(m,c) is fixedpoint-implied by NonRE for all initial configurations c. Therefore, ~ne(w) is fixedpoint-implied by NonRE. □

Corollary 5.9: There is no complete evaluation system for DEF-programs.

To summarize, a DEF-program NonRE has been constructed that describes a non-r.e. language, L_e. Any complete evaluation system for DEF-programs must be capable of succeeding only for those queries ~ne(w) on program NonRE where

$w \in L_e$. Such an evaluation system would therefore accept non-r.e. language L_e. If Church's Thesis is to be believed, no evaluation system can accept a non-r.e. language, so no evaluation system can be complete for DEF-programs.

How important is completeness of an evaluation system? The most efficient implementation of an evaluation system for definite clause programs currently available is Prolog. Chapter 3 demonstrated that Prolog's depth-first search is incomplete. Yet the evaluation system is still used. Evidently, concern for completeness is subordinate to concerns for efficiency and correctness. Finally, the alternative implementation of negation within logic programs, negation by failure, is also incomplete. Despite this fact, negation by failure is used as the predominant implementation of negation in logic programming languages, primarily due to its ease of implementation.

Chapter 6
Enhancements

A number of topics have been deferred in Chapter 5 for further exploration. These topics fall under three main areas:

(1) Use of equality to control the explosion of DEFs required to satisfy the self-coverage test.

(2) Incorporating type information within the self-coverage test.

(3) Permitting free variables in universally quantified queries.

6.1. Controlling Explosion of DEFs

Self-coverage of programs is necessary for correctness of the evaluation procedure (Section 5.4). A program is self-covering if there is a closed instance $A \leftrightarrow F$ of a DEF for every ground atom A in the program's Herbrand Base. This requirement can lead to an explosion in the number of DEFs in database-oriented and polymorphic programs. The explosion is controlled by providing an equality predicate within the evaluation system.

Evaluation of Equality

Definite clause programs can define equality of finite terms succinctly with the following assertion:

```
% equal(X,Y): true if X=Y.
equal(X,X).
```

This succinctness cannot be attained by DEF-programs that are self-covering and non-conflicting. Within a DEF-program from $B(A(\Pi,\Sigma))$, the following groups of DEFs are needed to define the `equal` predicate:

For all constants $\texttt{c} \in \Sigma$:
```
    equal(c,c) ↔ true.
```

For all distinct constants $\texttt{c},\texttt{d} \in \Sigma$:
```
    equal(c,d) ↔ ~true.
```

For all function symbols $\texttt{f} \in \Sigma$:
```
    equal(f(X1,...,Xn),f(Y1,...,Yn)) ↔
            equal(X1,Y1)⋀...⋀equal(Xn,Yn).
```

For all distinct function symbols $\texttt{f}^{(m)},\texttt{g}^{(n)} \in \Sigma$:
```
    equal(f(X1,...,Xm),g(Y1,...,Yn)) ↔ ~true.
```

For $m \neq n$ and $\texttt{f}^{(m)},\texttt{f}^{(n)} \in \Sigma$:
```
    equal(f(X1,...,Xm),f(Y1,...,Yn)) ↔ ~true.
```

With n constants in Σ, on the order of n^n DEFs are required to define `equal`. Furthermore, any query $Q = \texttt{equal}(x,y)$ will be evaluated far less efficiently using the DEF-program than with the definite clause version. Evaluation of Q with the definite clause program produces a full search tree consisting only of Q and its direct descendent, the empty conjunction. Evaluation of Q with the DEF-program produces a tree proportional in size to the number of subterms in Q. While unification is performed only once in constructing the full search tree for the definite clause program, the number of unifications for the DEF-program is proportional to the number of subterms in Q.

Both the excessive number of DEFs and poor performance of the evaluation system with the DEF-program argue for a special case of equality. Therefore, definition of `equal` is embodied within the evaluation system.

To incorporate the definition of `equal`, a binary function *dif* is defined returning one of the three logical constants.

$$dif(x,y) = \begin{cases} \mathbf{t} \text{ if } x \text{ and } y \text{ are not unifiable} \\ \mathbf{f} \text{ if } x \text{ and } y \text{ are syntactically identical} \\ \mathbf{u} \text{ otherwise} \end{cases}$$

This function is available to the programmer as a system-defined predicate within the Prolog-II system [Co82]. It provides a correct implementation of inequality of terms, distinct from non-unifiability. When $dif(x,y) = \mathbf{t}$, terms x and y have no common instances. On the other hand, $dif(x,y) = \mathbf{f}$, when there is no way to differentiate x and y. Finally, $dif(x,y) = \mathbf{u}$ when x and y have common instances, but x and y are not equal; further instantiation of variables within x and y can either equate or differentiate the terms. As an example, $dif($`f(X,c)`, `f(b,c)`$) = \mathbf{u}$ because `f(X,c)`$\{$X=b$\}$ = `f(b,c)` and `f(X,c)`$\{$X=a$\} \neq$ `f(b,c)`.

With the `equal` predicate implemented by the evaluation system, DEF-programs cannot contain definitions for `equal`. Evaluation of equality and inequality queries is through construction of full search trees, as follows:

Root node is `equal`(x,y):

When x and y are unifiable, m is a variant of the mgci of x and y, and $x\,\mu = y\,\mu = m$, then the following full search tree is well-formed:

$$\begin{tikzcd}\texttt{equal}(x,y) \arrow[d, dash, "\mu"] \\ \square\end{tikzcd}$$

When x and y are not unifiable, the full search tree consists only of the node `equal`(x,y).

Root node is $\sim$`equal`(x,y):

When $dif(x,y) = \mathbf{t}$, then the following full search tree is well-formed:

$$\begin{tikzcd}\sim\texttt{equal}(x,y) \arrow[d, dash] \\ \square\end{tikzcd}$$

When $dif(x,y) = \mathbf{f}$, the full search tree consists only of the node $\sim$`equal`(x,y).

When $dif(x,y) = \mathbf{u}$, x and y are unifiable but they are not equal. Consider the query of Example 6.1, below:

Example 6.1

```
∀X: ~equal(X,c) → ~true.
```

If ~`equal(X,c)` fails because `X` and `c` are unifiable, then Example 6.1 will succeed. This is incorrect, since the query of Example 6.1 is logically equivalent to `∀X:equal(X,c)`, which is false in any domain containing more than one element.

In fact *dif*(X, c) = **u**, so the evaluation system must abort further construction of the full search tree for Example 6.1, due to possible incorrectness.

Prolog-II permits delay in evaluation of *dif* queries until variables within the query are instantiated to ground values sufficiently for *dif*$(x, y) \neq$ **u**. Delaying queries is analogous to use of a fair selection rule (see Section 3.2). There are still instances, as in Example 6.1, where the evaluation system must halt to avoid an incorrect action.

Since *dif*(f(X), f(X)) = **f**, the query `~equal(f(X), f(X))` fails without abnormal termination. This action is acceptable. For example, the query:

`∀X: ~equal(f(X), f(X)) → ~true`

succeeds and is true, because the query is equivalent to the formula `∀X:equal(f(X), f(X))`.

The evaluation system with equality enhancement is correct: When a query `equal`(x, y) succeeds, x and y are unifiable, so $\exists(x = y)$. When a query `~equal`(x, y) succeeds, *dif*$(x, y) =$ **t**, and $\forall(x \neq y)$ implying that $\exists(x \neq y)$.

Database-Oriented Programs

Provision of equality resolves problems in satisfying self-coverage and non-conflict requirements for certain categories of DEF-programs. The explosion in the number of DEFs for database-oriented programs occurs when defining base relations. Definition of a base relation $r^{(n)}$ within a definite clause program is typically achieved with assertions. There is usually one assertion r$(c_1, \ldots, c_n)$ for each tuple

$(c_1, \ldots, c_n) \in r$, where each c_i is a constant.

A DEF-program defining base relation r contains either `r`$(c_1, \ldots, c_n) \leftrightarrow$ `true` if $(c_1, \ldots, c_n) \in r$, or `r`$(c_1, \ldots, c_n) \leftrightarrow \sim$`true` if $(c_1, \ldots, c_n) \notin r$. The resulting DEF-program contains $n^{|T(\Sigma)|}$ DEFs.

Using the `equal` predicate, all of the DEFs defining a single base relation can be reduced to a single DEF whose length is approximately the size of the base relation. Suppose a base relation $r^{(n)}$ contains t tuples:

$$(c_{1,1}, \ldots, c_{1,n}), \ldots, (c_{t,1}, \ldots, c_{t,n})$$

The following DEF defines a corresponding predicate `r`:

$$\begin{aligned} \mathtt{r(X1,\ldots,Xn)} \leftrightarrow & \,\mathtt{equal}(\mathtt{f}(c_{1,1}, \ldots, c_{1,n}), \mathtt{f(X1,\ldots,Xn)}) \\ & \vee \mathtt{equal}(\mathtt{f}(c_{2,1}, \ldots, c_{2,n}), \mathtt{f(X1,\ldots,Xn)}) \\ & \vee \cdots \vee \\ & \vee \mathtt{equal}(\mathtt{f}(c_{t,1}, \ldots, c_{t,n}), \mathtt{f(X1,\ldots,Xn)}). \end{aligned}$$

In this scheme, the function symbol `f` is unique to the DEF.

As an example, suppose base relation r contains the following tuples:

```
(a,b),
(c,d).
```

Representation of relation r within a definite clause program requires only the two assertions:

```
r(a,b).
r(c,d).
```

Representation of relation r within a DEF-program, using the `equal` predicate, employs the following DEF:

```
r(X1,X2) ↔
      equal(f(a,b),f(X1,X2)) ∨
      equal(f(c,d),f(X1,X2)).
```

The following properties of this encoding are easily verified:

$(c_1, \ldots, c_n) \in r$ iff $\mathrm{r}(c_1, \ldots, c_n)$ is fixedpoint-implied.

$(c_1, \ldots, c_n) \notin r$ iff $\sim\mathrm{r}(c_1, \ldots, c_n)$ is fixedpoint-implied.

Reflecting on the evaluation of `~equal`, it is also clear that any non-ground query $\sim\mathrm{r}(x_1, \ldots, x_n)$ cannot produce a correct full search tree. Similarly, negation by failure is incorrect for non-ground queries. But negation by failure requires *all* negative queries to be ground, even for programs that are not database-oriented.

Polymorphic Programs

Often in definite clause programs, predicates are defined for arbitrary data types. As an example, consider the definite clause program below:

```
% prefix(L,P): true if P is a prefix of list L.
prefix(L,nil) ← true.
prefix(cons(X,L),cons(X,P)) ← prefix(L,P).
```

In this program any list whose first element is x and whose tail is l is represented by a term $\mathrm{cons}(x,l)$. An empty list is denoted by the constant `nil`. Using this program, the query:

```
prefix(cons(c,cons(d,nil)),X)
```

has answers:

```
X = nil,
X = cons(c,nil),
X = cons(c,cons(d,nil)).
```

This program is polymorphic because it can be used for lists of integers, characters, names, etc.

Use of polymorphism has the following benefits:

Succinct clauses:

The same group of clauses can be used for different data types, instead of having different groups of clauses providing identical definitions for different data types.

Independence from change:

If data types change in a clause that invokes a polymorphic predicate, it may be possible to continue using the polymorphic predicate without change.

Adaptability:

Polymorphic predicates are applicable to arbitrary data types. So when new data types are used within a program, the polymorphic predicates can be reused in new roles.

These benefits provide strong reason to support polymorphism within DEF-programs. However, self-coverage and non-conflict requirements on DEF-programs make it difficult to provide polymorphism. For example, a possible self-covering non-conflicting representation of the `prefix` program above within a DEF-program is the following:

Example 6.2

```
% prefix(L,P): true if P is a prefix of list L;
%    otherwise, false.
prefix(L,nil) ↔ true.
prefix(nil,cons(Y,P)) ↔ ~true.
prefix(cons(X,L),cons(Y,P)) ↔
     (eqNats(X,Y) ⋁ eqChars(X,Y) ⋁ eqNames(X,Y))
     ⋀ prefix(L,P).
```

The self-coverage and non-conflict requirements force use of distinct equality tests for each data type that can constitute a list.

Example 6.2 is not polymorphic, as it is specialized only to lists of natural numbers, lists of characters, and lists of names; however, it suggests that use of the `equal` predicate can restore polymorphism, since `equal` is defined over arbitrary data types. Thus, the third DEF of Example 6.2 can be replaced by the following DEF, attaining polymorphism for the `prefix` predicate:

```
prefix(cons(X,L),cons(Y,P)) ↔
     equal(X,Y) ⋀ prefix(L,P).
```

In fact this technique is general. To satisfy self-coverage and non-conflict, heads of DEFs do not contain multiple occurrences of any single variable. Without multiple occurrences of variables in heads of DEFs, equality testing cannot be achieved in a polymorphic manner. The `equal` predicate provides for polymorphic equality testing. Since evaluation of an `equal` query is just as efficient as testing for unifiability, use of the `equal` predicate does not compromise efficiency of the evaluation system.

Summary

Evaluation of an `equal` predicate has been described. Its definition is embodied within the evaluation system for DIF-programs, providing conciseness and efficiency. Evaluation of `equal` queries may abort to avoid any possible incorrectness. The `equal` predicate has applications within database-oriented and polymorphic programs, increasing the range of programs that can practically use constructive negation. Programs utilizing data types also require special accommodation for practical use of constructive negation.

6.2. Incorporating Type Information for Self-Coverage

It is proper to invoke certain predicates with only certain types of arguments. For example, suppose a predicate `length`(l, n) is true if the length of list l is number n. Arguments other than a list and a number are improper. By extension `~length` is properly invoked only with terms denoting a list and integer. For example, it is proper to query:

```
~length(cons(a,cons(b,nil)),0),
```

and it is improper to query `~length(0,nil)`.

A *type* of a DEF-program is a subset of its function symbols. All types of a program must be disjoint. For example, suppose a program consists only of the following DEFs:

Example 6.3

```
% length(L,N): true if list L has length N;
%    otherwise, false.
length(nil,0) ↔ true.
length(nil,s(N)) ↔ ~true.
length(cons(X,L),0) ↔ ~true.
length(cons(X,L),s(N)) ↔ length(L,N).
```

The function symbols are:

$$\Sigma_0 = \{\texttt{0}^{(0)}, \texttt{s}^{(1)}, \texttt{nil}^{(0)}, \texttt{cons}^{(2)}\}.$$

One partition of Σ_0 is π_0:

$$\texttt{lists} = \{\texttt{cons}, \texttt{nil}\}$$
$$\texttt{nats} = \{\texttt{s}, \texttt{0}\}.$$

Suppose a DEF-program is from $B(A(\Pi, \Sigma))$, and π is a partition of Σ. A *predicate type assignment* under π is an assignment of a list of types from π to each predicate symbol in Π. For example:

$$TP_{\pi_0}(\texttt{length}) = \texttt{[lists,nats]}$$

Similarly, a *function type assignment* under π is an assignment of a list of types from π to each function symbol in Σ. For example:

$$TF_{\pi_0}(\texttt{0}) = \texttt{[]}$$

$$TF_{\pi_0}(\texttt{s}) = \texttt{[nats]}$$

$$TF_{\pi_0}(\texttt{nil}) = \texttt{[]}$$

$$TF_{\pi_0}(\texttt{cons}) = \texttt{[nats,lists]}$$

Finally, a *variable type assignment* under π is an assignment to each variable in Υ

either a type from π or the empty type $\emptyset$. When a variable type assignment maps a variable to the empty type, the variable's type is unassigned. The distinguished type assignment $\underline{TV}$ maps every variable to $\emptyset$. Usually TP_π, TF_π and TV_π will denote predicate, function and variable type assignments, respectively, under type partition π. When π is understood from context, it will be omitted.

The following algorithm determines if various syntactic parts of a DEF-program are well-typed. The algorithm is "top-down" so that variable type assignments can differentiate between different binding scopes.

(1) A DEF-program is well-typed by (TP, TF) if every DEF in the program is well-typed by (TP, TF).

(2) A DEF $A \leftrightarrow F$ is well-typed by (TP, TF) if $\forall(A \leftrightarrow F)$ is well-typed by $(TP, TF, \underline{TV})$.

(3) A quantified formula $\forall X{:}F$ or $\exists X{:}F$ is well-typed by (TP, TF, TV) if π contains a type τ and F is well-typed by (TP, TF, TV'), where:

$$TV'(Y) = \begin{cases} \tau & \text{if } Y = X \\ TV(Y) & \text{if } Y \neq X \end{cases}$$

(4) Any formula $F \leftrightarrow G$, $F \rightarrow G$, $F \bigwedge G$, $F \bigvee G$, or $\sim F$ is well-typed by (TP, TF, TV) if F and G are both well-typed by (TP, TF, TV).

(5) Every proposition p is well-typed.

(6) Any atom $p(x_1, \ldots, x_n)$ $(n \geq 1)$ is well-typed by (TP, TF, TV) if:

$TP(p^{(n)}) = [\tau_1, \ldots, \tau_n]$, and each x_i $(1 \leq i \leq n)$ is assigned τ_i by (TP, TF, TV).

(7) Any constant $c \in \tau$ is assigned type τ.

(8) Any term $f(x_1, \ldots, x_n)$ $(n \geq 1)$ is assigned type τ by (TP, TF, TV) if:

$f \in \tau$, and

$TF(f^{(n)}) = [\tau_1, \ldots, \tau_n]$, and

each x_i $(1 \leq i \leq n)$ is assigned τ_i by (TP, TF, TV).

(9) Any variable X is assigned type τ by (TP, TF, TV) if $TV(X) = \tau$.

As an example, suppose $TV_{\pi_0}(\text{N}) =$ `nats`. Then the following observations hold:

N is assigned type `nats` by $(TP_{\pi_0}, TF_{\pi_0}, TV_{\pi_0})$; hence, `s(N)` is assigned type `nats` by $(TP_{\pi_0}, TF_{\pi_0}, TV_{\pi_0})$;

`nil` is assigned type `lists`; hence, `length(nil,s(N))` is well-typed by $(TP_{\pi_0}, TF_{\pi_0}, TV_{\pi_0})$;

`true` is well-typed by $(TP_{\pi_0}, TF_{\pi_0}, TV_{\pi_0})$; hence:

`~true` is well-typed by $(TP_{\pi_0}, TF_{\pi_0}, TV_{\pi_0})$;

`length(nil,s(N))` $\leftrightarrow$ `~true` is well-typed by $(TP_{\pi_0}, TF_{\pi_0}, TV_{\pi_0})$;

DEF `length(nil,s(N))` $\leftrightarrow$ `~true` is well-typed by (TP_{π_0}, TF_{π_0}).

In fact the program of Example 6.3 is well-typed by (TP_{π_0}, TF_{π_0}).

Systems to infer a minimal type partition and type assignment for a given definite clause program have been suggested [Mi84,MK84]. It is not difficult to extend these systems to DEF-programs.

Unless the self-coverage test is modified to observe type restrictions, well-typed programs will not be self-covering. Example 6.3 is not self-covering, because (among others) `length(0,nil)` is not a ground instance of the head of any DEF in the program.

A DEF-program well-typed by (TP, TF) is *self-covering* for (TP, TF) if there is a closed instance $A \leftrightarrow F$ of a DEF in the program for every ground atom A well-typed by $(TP, TF, \underline{TV})$. Under this refined criterion, the `length` program is self-covering for type assignments (TP_{π_0}, TF_{π_0}).

The self-coverage test is similarly altered. Suppose a DEF-program is well-typed by (TP, TF). Define $T(TF, \tau)$ to be the set of all ground terms assigned type τ. For example, $T(TF_{\pi_0},$ `lists`$)$ contains among other elements:

```
nil,
cons(0,nil),
cons(s(0),cons(0,nil)).
```

Similarly, let $A(TP, TF)$ be the set of all ground atoms well-typed by $(TP, TF, \underline{TV})$. Thus, if $TP(p) = [\tau_1, \ldots, \tau_n]$, then $p(x_1, \ldots, x_n) \in A(TP, TF)$ if and only if each $x_i \in T(TF, \tau_i)$. For example, `length(cons(0,nil),0)` is a member of $A(TP_{\pi_0}, TF_{\pi_0})$.

Recall that m_p is the maximum nesting depth of heads of all DEFs defining predicate p (Section 5.4). Also, if S is a set of atoms, $S \% d$ is the subset of S consisting of only those atoms with nesting depth at most d. A predicate p occurring in a program satisfies the self-coverage test with respect to type assignments (TP, TF) if:

(1) All DEFs defining p are well-typed by (TP, TF) and

(2) There is a closed instance $p(x_1, \ldots, x_n) \leftrightarrow F$ of a DEF for every atom $p(x_1, \ldots, x_n) \in A(TP, TF) \% (m_p + 1)$.

Under this new criterion, Example 6.3 satisfies the self-coverage test with respect to type assignments (TP_{π_0}, TF_{π_0}). With an argument similar to Lemma 5.4, it can be shown that if a program P is well-typed by type assignment $TA = (TP, TF)$, then P satisfies the self-coverage test with respect to TA if and only if P is self-covering. Correctness of the evaluation procedure for well-typed programs follows automatically.

Utilizing typed programs can aid in writing programs that perform as intended [MK84]. Accommodating typed DEF-programs mandates a simple revision to the self-coverage test. Typed programs also reduce the number of DEFs needed to satisfy self-coverage.

6.3. Enhancing Evaluation of Universal Quantification

The implementation of universally quantified queries within the evaluation system (Section 5.3) left several issues outstanding. Correctness of the procedure

(Theorem 5.5) relied on use of term-matching within the filter for every generated term. Implementing term-matching efficiently within a logic programming evaluation system is straightforward. Also, the correctness proof for the evaluation procedure makes the assumption that every universally quantified query is closed. Permitting free variables within universally quantified queries increases the flexibility of the evaluation procedure. Queries that could flounder under a correct implementation of negation by failure can be efficiently evaluated with the evaluation procedure for DEF-programs.

6.3.1. Term-Matching

In Section 5.4, term-matching was required to ensure correctness of certain universally quantified queries. While unifiability of terms s and t determines if an mgci $s \tilde{\sqcap} t$ exists, term s *matches* term t if $[s] \gtrsim [t]$ (Section 2.3). The evaluation system already has a unification component. As suggested in [Dw84], term-matching is a special case of unification. Matching term s against term t is achieved by unifying terms s and t', where t' is a ground instance of t with every variable of t set to a unique constant.

Instead of binding every variable in a term to a unique constant before attempting term matching, an actual implementation could associate a tag with each variable. The tag is set if the variable should not be further instantiated. The unification procedure must be revised to check the tag of a variable whenever an attempt is made to set a variable to a value. If a variable is tagged and unification attempts to set the variable equal to another untagged variable, the untagged variable should be set equal to the tagged variable. If a variable is tagged and the variable will be set equal to a non-variable term or another tagged variable, unification should fail.

When term-matching is incorporated within the procedure for evaluating universally quantified queries, all variables occurring within a generated term will be tagged. Any attempt by the filter to further instantiate a generated value should terminate abnormally, in order to notify the user of an incorrect condition. Consider Example 5.3 reproduced below:

```
p(X) ↔  true.
q(a) ↔  true.
q(b) ↔ ~true.
```

The generator `p(X)` of the query $\forall X{:}p(X) \rightarrow q(X)$ produces a tagged value `X`. The evaluation procedure then produces a query `q(X)`. Any attempt to further instantiate `X` will meet with failure, eliminating the possibility of incorrectness caused by overly-general generated values.

Implementation of term-matching requires checking a variable's tag any time it is to be set to a value, and an initial sweep through every generated value tagging all variables. Tag checking of terms during unification is performed anyway, for other purposes. Term-matching can also be used when free variables are included within universally quantified queries, as described in the next section.

Term-matching has not yet been implemented within a logic programming evaluation system. In its absence, correctness is assured by generating only ground values. Ground terms can be generated with a type predicate such as `hu`, as described in Section 5.4. The NU-Prolog system [TZ87] is capable of delaying evaluation of queries until certain variables are instantiated to ground values.

6.3.2. Free Variables in Universally Quantified Queries

The correctness proof of the evaluation system for DEF-programs (Theorem 5.5) makes the assumption that free variables do not occur within universally quantified queries. This section will discuss how this restriction can be weakened while retaining correctness of the evaluation procedure.

Free Variables in Only the Generator or the Tester

In fact the evaluation procedure and proof accommodates free variables within the tester of a universally quantified query. Consider the program below:

```
p(a)  ↔  true.
p(b)  ↔  true.

q(a,1)  ↔  true.
q(b,1)  ↔  true.
```

The query $\forall$X: p(X) → q(X,Y) contains the free variable Y. The answers obtained from the generator are X=a and X=b. The evaluation procedure then creates the conjunction q(a,Y) ∧ q(b,Y). Evaluation of the conjunction produces an answer Y=1 to the full query.

Extended programs (Section 3.4.1) cannot correctly evaluate query $\forall$X: p(X) → q(X,Y). The extended query produces the general clause:

```
aux(Y)  ←  p(X)  ∧  not  q(X,Y).
```

and query `not aux(Y)`. Since negation by failure is incorrect for negated queries containing free variables, this query will flounder. As just demonstrated, under constructive negation the query can be evaluated.

Thus, when free variables occur only in the tester of a universally quantified query, the evaluation procedure can correctly produce answer substitutions for these variables. When free variables occur only within the generator G of a query $\forall X\colon G \rightarrow F$, the query can be rewritten to $\forall X\colon \overline{F} \rightarrow \overline{G}$ without changing the query's meaning. Now variables occur only within the tester, and the evaluation procedure can proceed correctly.

Free Variables in both Generator and Tester

Free variables occurring within both the generator and tester of a universally quantified query pose the greatest challenge to the evaluation procedure. Universally quantified queries are transformed to instantiate free variables occurring within generators prior to evaluation. Following the transformation, if free variables remain, term-matching can ensure that they are not instantiated within the universally quantified formula. This transformation scheme has also been proposed for negation by failure [D87].

The following identity is used to rewrite the original universally quantified query:

$$I[\forall X\colon G \rightarrow F] = I[\{(\exists X\colon G) \bigwedge (\forall X\colon G \rightarrow F)\} \bigvee \{(\exists X\colon \overline{G}) \bigwedge \sim(\exists X\colon G)\}].$$

This identity holds for any interpretation I such that $I[G\sigma] \neq \mathbf{u}$ for every closed instance $G\sigma$ of G, which holds whenever $\forall X\colon G \rightarrow F$ terminates. If the identity does not hold, evaluation will not terminate in any case.

The second disjunct $(\exists X\colon \overline{G}) \wedge \sim(\exists X\colon G)$ can be evaluated with negation by failure. The query not G succeeds with negation by failure if G has a finitely failed full search tree. By Theorem 5.5, if a query $\exists X\colon G$ has a finitely failed full search tree, $\sim\exists X\colon G$ is fixedpoint-implied by the program. Consequently, when not $\exists X\colon G$ succeeds with negation by failure, $\sim\exists X\colon G$ is fixedpoint-implied. The second disjunct can therefore be rewritten to $[(\exists X\colon \overline{G}) \wedge (\texttt{not}\ \exists X\colon G)]$.

As an illustration of the transformation, consider the query

`~mult(s(0),J,K)` with the program fragment of Example 5.1, repeated here:

```
% mult(I,J,K): true if I×J=K; otherwise, false.
mult(0,J,0) ↔ true.
mult(0,J,s(K)) ↔ ~true.
mult(s(I),J,K) ↔ ∃X: mult(I,J,X) ∧ add(X,J,K).
```

The evaluation procedure produces the subquery:

```
∀X: mult(0,J,X) → ~add(X,J,K).
```

The following new subquery results from the transformation:

```
(1)   [(∃X:mult(0,J,X)) ∧ (∀X:mult(0,J,X) → ~add(X,J,K))] ∨
(2)   [(∃X:~mult(0,J,X)) ∧ (not ∃X:mult(0,J,X))].
```

In solving disjunct (1) of this query, its first conjunct `∃X:mult(0,J,X)` is true for `X=0` without further instantiating `J`. With a term-matching implementation, variable `J` is tagged indicating no further instantiation can occur. Without term-matching, when the universally quantified subquery is entered, presence of a non-ground free variable should bring the evaluation procedure to abnormal termination. Termination avoids incorrect full search tree construction. Evaluation of the universal quantifier in disjunct (1) produces the subquery `~add(O,J,K)`.

In solving disjunct (2) of this query, its first conjunct `∃X:~mult(O,J,X)` is true for `X=s(X')` without further instantiating `J`. Again, the variable `J` should be tagged to prevent further instantiation. Negation by failure is now used for the query `not mult(O,J,X)`. The query correctly fails due to the contradictory value `O` for `X`. So disjunct (2) produces no answers to the original query.

Functional Generators

Frequently, a universally quantified variable is functionally determined by the generator of its universally quantified formula. This occurs in Example 5.1, repeated in the previous section. This program contains the DEF:

```
mult(s(I),J,K) ↔
      ∃X: mult(I,J,X) ∧ add(X,J,K).
```

The associated object program then contains the DIF:

```
~mult(s(I),J,K) ← ∀X: mult(I,J,X) → ~add(X,J,K).
```

Given values i and j, mult(i,j,X) yields exactly one value for X, because `mult` behaves as a total function. Therefore, the universal quantifier can be replaced by an existential quantifier, and the DIF:

```
~mult(s(I),J,K) ← ∃X: mult(I,J,X) ∧ ~add(X,J,K)
```

is equivalent to the previous DIF.

A *functionally-determined* quantifier $\boxminus$ is used within DEF-programs to mean that a predicate such as `mult` behaves as a total function. A formula containing the functionally-determined quantifier $\boxminus X{:}F \bigwedge G$ is an abbreviation for the formula $(\exists! X{:}F) \rightarrow (\exists X{:}F \bigwedge G)$. Within this definition, $\exists! X{:}F$ abbreviates $\exists X{:}(F \bigwedge (\forall Y{:}F(Y) \rightarrow X=Y))$: "there exists a unique X satisfying F." Functionally-determined formulas are always *bounded*, containing a conjunction of formulas. The DEF defining `mult` can be rewritten with a functionally-determined quantifier as:

```
mult(s(I),J,K) ↔
      ⊟X: mult(I,J,X) ⋀ add(X,J,K).
```

Use of the functionally-determined quantifier within a program may arise from syntactic analysis or semantic knowledge possessed by a programmer. It is undecidable in general if a predicate behaves as a total function. Certain classes of programs, such as primitive recursive programs, always define predicates to be total functions. If a programmer's semantic knowledge is erroneous, compilation will no longer preserve meaning.

The complement of a functionally determined formula is defined to be:

$$\overline{\boxminus X{:}F \bigwedge G} = \boxminus X{:}F \bigwedge \overline{G}$$

If there is exactly one value x_0 such that $F(x_0)$ is fixedpoint-implied, then the complement is logically equivalent to negation.

Lemma 6.1 (Complement of Functionally-Determined Formula Equivalent to Negation): For any interpretation I and closed formula $\boxminus X{:}F \bigwedge G$, if $I[\exists! X{:}F] = \mathbf{t}$, then $I[\sim\boxminus X{:}F \bigwedge G] = I[\boxminus X{:}F \bigwedge \sim G]$.

Proof: The proof first demonstrates that if $I[\exists! X{:}F] = \mathbf{t}$, then $I[F(x)] \neq \mathbf{u}$ for all x. Suppose that $I[F(x_0)] = \mathbf{t}$. Then $I[\exists! X{:}F] = \mathbf{t}$ implies $I[F(x) \rightarrow x = x_0] = \mathbf{t}$ for all terms x. Since $I[x = x_0] = \mathbf{f}$ for all terms $x \neq x_0$, $I[F(x)] = \mathbf{f}$.

For the main part of the proof, note that:

$$\begin{aligned} I[\sim\boxminus X{:}F \bigwedge G] &= I[(\exists! X{:}F) \bigwedge (\forall X{:}F \rightarrow \sim G)] \\ &= I[\forall X{:}F \rightarrow \sim G], \text{ because } I[\exists! X{:}F] = \mathbf{t}. \end{aligned}$$

As above, suppose $I[F(x_0)] = \mathbf{t}$, and hence, $I[F(x)] = \mathbf{f}$ for all $x \neq x_0$.

$$\begin{aligned} I[\forall X{:}F \rightarrow \sim G] &= I[(F \rightarrow \sim G)(x_0)] \bigwedge (\bigwedge\{I[(F \rightarrow \sim G)(x)] \mid x \neq x_0\}) \\ &= I[(\sim G)(x_0)] \\ &= I[(\mathbf{t} \bigwedge \sim G)(x_0)] \bigvee (\bigvee\{I[(\mathbf{f} \bigwedge \sim G)(x) \mid x \neq x_0\}) \\ &= I[(F \bigwedge \sim G)(x_0)] \bigvee (\bigvee\{I[(F \bigwedge \sim G)(x)] \mid x \neq x_0\}) \\ &= I[\exists X{:}F \bigwedge \sim G] \\ &= I[\boxminus X{:}F \bigwedge \sim G], \text{ because } I[\exists! X{:}F] = \mathbf{t}. \end{aligned}$$

□

As a result, if $\boxminus X{:}F \bigwedge G$ is a closed formula and $\exists! X{:}F$ is fixedpoint-implied, then $\overline{\boxminus X{:}F \bigwedge G}$ is logically equivalent to $\sim\boxminus Y{:}F \bigwedge G$ in all fixedpoints. Definition of the complement form of functionally-determined formulas enables compilation of these formulas. For example, the DEF below:

```
mult(s(I),J,K) ↔
      ƎX: mult(I,J,X) ⋀ add(X,J,K).
```

compiles to the dual DIFs:

```
 mult(s(I),J,K) ← ƎX: mult(I,J,X) ⋀  add(X,J,K).
~mult(s(I),J,K) ← ƎX: mult(I,J,X) ⋀ ~add(X,J,K).
```

A functionally-determined formula $ƎX\!:F \bigwedge G$ is equivalent to an existentially quantified formula $\exists X\!:F \bigwedge G$ along with a uniqueness hypothesis. Therefore, evaluation of a functionally-determined formula is identical to evaluation of an existentially quantified formula: $ƎX\!:F \bigwedge G$ is evaluated with the same procedure as $\exists X\!:F \bigwedge G$.

Use of the functionally-determined quantifier eliminates occurrences of universal quantification within compiled DIF-programs. Hence, the appearance of free variables within universally quantified formulas are also limited. For example, the original DIF defining `~mult` contained a universal quantifier, so a query such as `~mult(s(s(0)),J,K)` produces a universally quantified query with a free variable in its generator. When `mult` is defined with a functionally-determined quantifier, this query can be evaluated, returning answers:

```
{J=0,  K=0},
{J=s(0),  K=s(s(0))},
{J=s(s(0)),K=s(s(s(s(0))))}, etc.
```

Summary

The presence of free variables within universally quantified queries can be handled in a number of ways. If free variables occur only in the tester, the evaluation procedure will produce correct values for the free variables. If the free variables occur only in the generator, the contrapositive form of the query brings the free variables into the tester, where correct values can be obtained. If the free variables occur both in the tester and generator, the query can be altered to instantiate free variables prior to evaluation of the universally quantified query. Any attempt to further instantiate the free variables within evaluation of the universally quantified query should result in abnormal termination. Finally, the number of universal quantifiers in a program can sometimes be reduced drastically by using a functionally determined quantifier.

6.4. Summarizing the Enhancements

The topics of this chapter have involved practical implementation and coding issues. Satisfying the self-coverage and non-conflict requirements can lead to an explosion in the number of DEFs. The explosion occurs especially for database-oriented and polymorphic programs. An equality predicate resolves concerns about database-oriented and polymorphic programs. Self-coverage is also generalized to accommodate data types. Introduction of the equality predicate and the enhancement to self-coverage broadens the scope of applications for DEF-programs.

Correctness of universally quantified queries depends on the absence of free variables (Theorem 5.5). Strategies for evaluating universally quantified queries with free variables have been developed. Finally, a functionally-determined quantifier reduces occurrences of universal quantifiers, thereby reducing the number of universally quantified queries containing free variables.

Chapter 7
Implementation Through Meta-Programming

This chapter will discuss actual implementation of the evaluation system for DEF-programs. A sample DEF-program is contained in Appendix C. This implementation includes:

(1) Disambiguation of variables by scoping.

(2) Typechecking.

(3) Overlap-checking.

(4) Self-coverage testing.

(5) Generation of dual DIFs from every DEF.

(6) Evaluation of queries on object DIF-programs.

The implementation was performed entirely in C-Prolog [P85]. Prolog has many facilities for self-reference, providing an excellent prototyping environment, and the pattern-matching facilities of Prolog are also useful in the implementation, especially for typechecking and obtaining the complement of formulas.

The Prolog program fragments presented in this chapter obey the syntax conventions required by C-Prolog. In particular, clauses are written as A_0:- $A_1, \ldots, A_n$ where the A_i are all atomic formulas. This clause is equivalent to a clause of the form $A_0 \leftarrow A_1 \wedge \cdots \wedge A_n$ in the notation of this dissertation. The different notation will be helpful in differentiating the Prolog program from formulas that are to be compiled or evaluated.

All logical connectives, except equivalence (↔) and implication (←), are treated within Prolog as uninterpreted function symbols. All of the predicate symbols present in DEF-programs are also treated as uninterpreted function symbols. The equivalence and implication connectives are treated as uninterpreted predicate symbols. For example, a DEF:

```
prime(P) ↔
      ∀X:  (lt(s(0),X)⋀lt(X,P))  →  ~divp(X,P)
```

is treated as the assertion:

```
↔(prime(P),
      ∀(X,→(⋀(lt(s(0),X),lt(X,P)),~(divp(X,P))))).
```

When necessary, the more readable infix form of the logical connectives will appear within program fragments.

7.1. Compilation of DEF-Programs

Compilation involves a number of steps. Variables produced in different scopes are disambiguated first. Next, the overlap test is performed to eliminate fixedpoint-inconsistent DEF-programs. The DEF-program is then typechecked. The self-coverage test is performed next, ensuring correctness of the evaluation system. Finally, dual DIFs are generated from each DEF in a program.

7.1.1. Variable Disambiguation

Disambiguation of variables is necessary when using C-Prolog, because this language has no facility for variable scoping. For example, if the first conjunct `∃X:p(X)` of:

`(∃X:p(X))` $\bigwedge$ `(∃X:q(X))`

succeeds during query evaluation, the answer for `X` will be incorrectly passed to the second conjunct, `∃X:q(X)` (Section 5.3). This possibility is eliminated by introducing a new quantified variable for each new scope. Disambiguation creates the formula:

`(∃X1:p(X1))` $\bigwedge$ `(∃X2:q(X2))`

for the preceding formula. Disambiguating nested scopes requires disambiguating sub-formulas first, and then generating a new variable in the outermost scope.

NU-Prolog [TZ87] has variable scoping, thus requiring no variable disambiguation.

7.1.2. Overlap Checking

Section 5.1 introduced the overlap test, ensuring fixedpoint-consistency (Lemma 5.2). The test fails if there are distinct DEFs $A \leftrightarrow F$ and $A' \leftrightarrow F'$ such that A and A' unify. This test is achieved with the following definite clause:

```
% overlap: true if there are distinct DEFs A↔F and A'↔F'
%     such that A and A' unify.
overlap :-
      A↔F1,
      A↔F2,
      distinct(F1,F2).
```

The predicate distinct($F1$,$F2$) determines if formulas $F1$ and $F2$ are syntactically distinct.

7.1.3. Typechecking

Types are assigned to function symbols and predicates appearing in the DEF-program. The type assignments are declared as assertions within a DEF-program. A declaration $f:[\tau_1, \ldots, \tau_n] \rightarrow \tau$ indicates that function symbol f is a member of type τ, and its arguments $1, \ldots, n$ are of types $\tau_1, \ldots, \tau_n$. Similarly, for predicate symbols the declaration $p:[\tau_1, \ldots, \tau_n]$ indicates the argument types. Type partitions of programs are not declared; however, the programmer must ensure that every function symbol appearing within a program is a member of a type. The type declarations are viewed by the C-Prolog system as assertions defining a binary infix predicate ":". For example, the declaration s: [nats]→nats is actually an assertion :(s, [nats]→nats). Again, the readable form will be used.

Typechecking terms utilizes a typecheckTerm predicate that is true of arguments x and τ if term x can be assigned type τ. If x is a variable, x is instantiated to τ ensuring that each occurrence of a variable is assigned only one type. If $x = f(x_1, \ldots, x_n)$ $(n \geq 0)$, typecheckTerm obtains a type assignment $f:[\tau_1, \ldots, \tau_n] \rightarrow \tau$ from the program, and typecheckTerm is recursively invoked to assign type τ_i to subterm x_i for all $1 \le i \le n$.

As an example, consider the type declarations below:

```
0: []→nats.
s: [nats]→nats.

nil: []→lists.
cons: [nats,lists]→lists.
```

The term `cons(0,X)` is successfully assigned type `lists` by `typecheckTerm` using the following informal derivation:

`typecheckTerm(cons(0,X),lists)` is true if:

  `typecheckTerm(0,nats)` and `typecheckTerm(X,lists)` are true.

    `typecheckTerm(0,nats)` is true.

    `typecheckTerm(X,lists)` is true with `X` instantiated to `lists`.

The following unsuccessful derivation demonstrates that `typecheckTerm` cannot assign type `lists` to term `cons(X,X)`:

`typecheckTerm(cons(X,X),lists)` is true if:

  `typecheckTerm(X,nats)` and `typecheckTerm(X,lists)` are true.

    `typecheckTerm(X,nats)` is true with `X` instantiated to `nats`.

    `typecheckTerm(X,lists)` is not true because `X` has been previously instantiated to `nats` and cannot be instantiated to `lists`.

Instantiation of variables to type names by the `typecheckTerm` predicate is correct only if type names are distinct from function symbols.

The predicate `typecheckPred` is true of an atomic formula A if A is well-typed. When $A = p(x_1, \ldots, x_n)$ $(n \ge 0)$, `typecheckPred` obtains a type assignment $p:[\tau_1, \ldots, \tau_n]$ and typechecks the subterms by invoking `typecheckTerm`(x_i, τ_i) for all $1 \leq i \leq n$. For example, suppose the predicate `length` is declared to have the following type:

```
length: [lists,nats].
```

Then `typecheckPred(length(cons(X,L),s(N)))` is true if:

`typecheckTerm(cons(X,L),lists)` and

`typecheckTerm(s(N),nats)` are true.

These subqueries are true with variables instantiated as follows:

`X = nats, L = lists, N = nats.`

The predicate `typecheckFormula` determines if a formula is well-typed by recursively determining if each non-atomic subformula is well-typed; atomic formulas are well-typed using the `typecheckPred` predicate. Disambiguation of variables is important for this task, since variables are instantiated to type names. Consider the well-typed formula:

$(\exists$`X:length(nil,X)`$) \wedge (\exists$`X:length(X,0)`$)$.

The `typecheckFormula` predicate instantiates `X` in the first conjunct to `nats` and in the second conjunct to `lists`. Without explicit disambiguation, typechecking would fail for the formula above.

7.1.4. Self-Coverage Testing

The test for self-coverage makes use of the type declarations. For prototyping purposes, the self-coverage test is slightly simplified from the test specified in Section

5.4. A maximum depth d_{max} is determined for all heads of DEFs, and is computed using system predicates. Then all well-typed ground atoms with depth at most $d_{max}+1$ are generated non-deterministically. Prolog is again a good language choice for this task. Every generated atom must match the head of some DEF in the program for the self-coverage test to succeed.

To generate all well-typed ground atoms $p(x_1, \ldots, x_n)$ of maximum depth $d_{max}+1$, the type declaration of predicate p, $p:[\tau_1, \ldots, \tau_n]$, is obtained. All well-typed ground terms $x_1 \cdots x_n$ of maximum depth d_{max} are then generated from types $\tau_1 \cdots \tau_n$.

To generate all well-typed ground terms of type τ and maximum depth d, each type declaration $f:[\tau_1, \ldots, \tau_n] \rightarrow \tau$ for type τ is obtained. Recursively, all well-typed ground terms $x_1 \cdots x_n$ of types $\tau_1 \cdots \tau_n$ and maximum depth $d-1$ are generated. These are combined to form a term $f(x_1, \ldots, x_n)$ whose maximum depth is d. In the basis case, all constants c with type declarations c: [] $\rightarrow \tau$ produce all terms of nesting depth 0 and type τ.

7.1.5. Generating Dual DIFs

When a DEF-program has passed all tests, an object DIF-program can be generated. Two tasks are performed. Negation applied to non-atomic formulas is moved inward. And the dual DIFs $A \leftarrow F$ and $\overline{A} \leftarrow \overline{F}$ are generated from each DEF $A \leftrightarrow F$.

Negation is moved inward by computing the complement of non-atomic formulas. In the process, only bounded universal and existential quantifiers, of the form $\forall X{:}\, G \rightarrow F$ and $\exists X{:} F \bigwedge G$, are generated. Since negation of a formula is equivalent to its complement (Lemma 4.8), this transformation is meaning-preserving.

The predicate compformula(F, CF) is true if formula F has complement CF in negation-innermost form. The definition of this predicate is taken almost directly from the rules for producing the complement. Symbolic manipulation capabilities of Prolog make this especially easy. Some of the clauses defining compformula are:

```
compformula(⋀(F1,F2),⋁(CF1,CF2)) :-
     compformula(F1,CF1), compformula(F2,CF2).
compformula(∃(X,⋀(F1,F2)),∀(X,→(F1,CF2)) :-
     compformula(F2,CF2).
```

The compformula predicate is used within the following clause to generate dual DIFs:

```
% compile(F1,F2,F3): true if F2 and F3 are dual DEFs
%     for the DEF F1.
compile(A↔F,A←F,~A←CF) :-
     compformula(F,CF).
```

7.2. Evaluating Queries

Queries can now be evaluated against compiled DIF-programs. Evaluation is based on enhancements to the usual implementation of SLD-resolution through meta-programming [SB86]. The usual implementation contains the following definitions of a predicate sld:

```
% sld(Q): true if query Q succeeds through SLD-resolution.
sld(true).
sld(Q1∧Q2) :- sld(Q1), sld(Q2).
sld(A) :- clause(A,Q), sld(Q).   % A is an atom.
```

The system predicate `clause` obtains an instance of a clause from the program whose head unifies with the first argument. The enhancements to this program cover the additional logical connectives, and obtain instances of DIFs from assertions defining the ← predicate.

```
% sld(Q): true if query Q succeeds through SLD-resolution.
sld(true).
sld(equal(X,X)).
sld(~equal(X,Y)) :- dif(X,Y).
sld(Q1∧Q2) :- sld(Q1), sld(Q2).
sld(Q1∨Q2) :- sld(Q1).
sld(Q1∨Q2) :- sld(Q2).
sld(∃X:Q) :- sld(Q).
sld(⊟X:Q) :- sld(Q).
sld(L) :- L←Q, sld(Q).    % L is a literal.
```

The system-defined predicate `dif` implements the *dif* function described in Section 6.1.1.

Evaluation of the universal quantifier has various cases depending on occurrences of free variables in the generator and tester. In the first case, the generator is a closed formula.

```
sld(∀X:G→F) :-
      closed(∀X:G),
      forall(∀X:G→F).
```

where `forall` is defined with the following clause:

```
forall(∀X:G→F) :-
      bagof(F,sld(G),Bag),
      makeConj(Bag,Conj),
      sld(Conj).
```

Upon evaluation of a query `forall`($\forall X : G \rightarrow F$), the following steps are taken:

(1) The variable `Bag` is instantiated to a list $[F\alpha_1, \ldots, F\alpha_n]$ composed of instances of F such that each α_i is an answer substitution for `sld`(G).

(2) Given an instantiation for `Bag`, variable `Conj` is instantiated to the conjunction $F\alpha_1 \bigwedge \cdots \bigwedge F\alpha_n$.

(3) `sld` is invoked recursively on query `Conj`.

If free variables occur in the tester, but not in the generator, these two formulas can be swapped within the implication:

```
sld(∀X:G→F) :-
      open(∀X:G),
      closed(∀X:F),
      compformula(G,CG),
      compformula(F,CF),
      forall(∀X:CF→CG).
```

In this clause, the predicate open(F) is true if formula F contains free variables.

Finally, if free variables occur both in the generator and tester, a new query that attempts to instantiate the free variables is created. This query makes use of negation by failure.

```
sld(∀X:G→F) :-
      open(∀X:G),
      open(∀X:F),
      disambiguate(∃X:G,G1),
      sld(G1),
      sld(∀X:G→F).
sld(∀X:G→F) :-
      open(∀X:G),
      open(∀X:F),
      compformula(G,CG),
      disambiguate(∃X:CG,CG1),
      sld(CG1),
      not sld(G).
```

These clauses make use of the predicate `disambiguate`($F1$,$F2$) which disambiguates variable scoping in formula $F1$, creating a formula $F2$ with distinct names for variables in different scopes. Because C-Prolog does not utilize a correct selection rule for negation by failure, incorrectness may result from use of the last clause above. When this deficiency is corrected in the C-Prolog system, the full implementation will be correct.

The clauses defining `sld` are evaluated with SLD-resolution in conjunction with the encoding of a DIF-program. For example, using the program of Appendix C, this implementation produces the following results:

```
?- sld( ~divp(Ans,s(s(s(0)))) ).

      Ans = 0;

      Ans = s(s(0));

      Ans = s(s(s(s(0))));

      Ans = s(s(s(s(s(0)))))

?- sld( ~divp(s(0),s(s(s(0)))) ).

      no
```

In this example, one answer substitution is produced at a time by the C-Prolog system; an additional answer is obtained by typing a semi-colon. The first query would continue to enumerate all representations of natural numbers greater than 3. The second query returned `no` indicating finite failure.

7.3. Summary

Compilation and the actual implementation of the evaluation system have been described. The implementation uses C-Prolog. The main shortcoming in using C-Prolog is that variables must be disambiguated. Its advantages include pattern-matching, self-reference, and non-determinism. These facilities were used extensively in typechecking, overlap-checking, self-coverage testing, and generation of dual DIFs. Since the evaluation system for DIF-programs is similar to SLD-resolution, implementation of the evaluation system was also eased.

Chapter 8
Summary and Future Work

8.1. Summary

This dissertation proposes an enhancement, called constructive negation, to the expressiveness of logic programming languages. The enhancement is based on formalizing the ad hoc methods of defining negative as well as positive facts. A three-valued logic is required, because some facts will inevitably be assigned neither true nor false.

Fixedpoints are chosen as the underlying model of DIF-programs. Certain programs are fixedpoint-inconsistent: no fixedpoints exist. Fixedpoint-consistency is undecidable and efficient evaluation systems cannot detect fixedpoint-inconsistency. Therefore, syntactic constraints are imposed on programs to ensure fixedpoint-consistency.

A set of consistency constraints are proposed involving dual definitions and absence of conflicting definitions. The resulting programs are DEF-programs. Underlying models of DEF-programs can be non-computable; hence, any evaluation system is necessarily incomplete. An evaluation system for DEF-programs is proposed based on enhancements to SLD-resolution. This evaluation system is correct only if a self-coverage test is satisfied.

Enhancements are needed to enable practical use of DEF-programs. An equality predicate is incorporated into the evaluation system, data types are introduced, and evaluation of universally quantified formulas is made more flexible. A prototype implementation of this system has been achieved with the C-Prolog language.

Other strategies for enhancing the expressiveness of logic programming languages also involve implementation of negation. The predominant implementation is by failure. But answer substitutions are not returned after evaluation of negated queries, and correctness of negation by failure is ensured only if negated queries are variable-free. Negation by failure is also incomplete, and cannot detect inconsistent programs. Consistency is ensured only through stratification.

Model elimination is a complete evaluation system for programs with negation. This system converts clauses (not necessarily definite) into contrapositive forms. These forms are similar to DIFs, in that negated atoms may occur in the heads of contrapositives. The evaluation system uses SLD-resolution and searches at ancestor nodes in the full search tree. Unless the ancestor search can be controlled through indexing, model elimination can suffer from the same inefficiencies as resolution.

Examples have demonstrated the use of constructive negation. In many cases, constructive negation is more flexible than negation by failure, because answer substitutions can be returned from evaluation of negative queries. Increased expressiveness is achieved by DEF-programs, because all logical connectives may be present within the bodies of DEFs.

8.2. Future Work

The weakest part of the evaluation system for DEF-programs is evaluation of universal quantifiers. The current implementation uses a system-defined construct that stores generated values within heap memory. A large conjunction is then formed from the generated values and tester, again in heap memory.

Tamaki and Sato [TS83,KH87] have investigated transforming universally quantified formulas into recursive clauses. Recursion effectively stores generated values on a stack. In fact, if the recursive clauses are tail recursive, only a fixed amount of space is required from the stack. Also, explicit storage of a conjunction is not needed. The transformation rules are not incorporated into a compiler, because the search space of transformations is too large. The transformations must be guided step-by-step by a programmer. The transformation rules are applicable to DEF-programs, but further work is needed to perform the transformation automatically for some interesting class of programs.

Use of non-conflicting DEFs to produce dual DIFs ensures fixedpoint-consistency. Since these syntactic constraints on programs are merely sufficient, better constraints may exist to ensure fixedpoint-consistency.

Ad hoc techniques for describing negation may also be used in equational programming languages. Constructive negation within logic programs relies on three-valued logic and quantifiers. Replacement of the ad hoc techniques for describing negation within equational programming languages by a constructive negation may require analogs to three-valued logic and quantifiers. The non-conflict and self-coverage properties could also be important in satisfying the Church-Rosser property for term-rewriting systems.

The non-conflict and self-coverage properties of programs are applicable to functional programming languages. Non-conflict ensures that each function is well-defined. Self-coverage ensures that all functions have definitions for all possible arguments. Self-coverage does not ensure total functions, but constitutes a useful precondition for defining total functions.

References

[ABW85] Apt, Blair & Walker, "Towards a Theory of Declarative Knowledge," Technical Report, IBM Corp., Yorktown Heights, 1985.

[ADJ78] Goguen, Thatcher & Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in *Current Trends in Programming Methodology*, vol. 4, Yeh (ed.), Prentice-Hall, 1978, pp. 80-149.

[AE82] Apt & van Emden, "Contributions to the Theory of Logic Programming," *JACM*, 29(3), July 1982, pp. 841-862.

[AJ74] Aho & Johnson, "LR Parsing," *Computing Surveys*, June 1974.

[Cl78] Clark, "Negation as Failure", in *Logic and Data Bases*, Gallaire & Minker (eds.), Plenum Press, New York, 1978, pp. 293-322.

[Co70] Codd, "A Relational Model for Large Shared Data Banks," *CACM*, 13(6), June 1970, pp. 377-387.

[Co82] Colmerauer, "Prolog and Infinite Trees," in *Logic Programming*, Clark & Tarnlund (eds), Academic Press, New York, 1982, pp. 324-340.

[D87] Decker, "The Range Form of Database Clauses: Or How to Avoid Floundering," Technical Report, European Computer-Industry Research Centre, 1987.

[Da86] Date, *An Introduction to Database Systems*, Addison-Wesley, Reading, MA, 1986.

[Dw84] Dwork, et al, "On the Sequential Nature of Unification," *J. of Logic Programming*, 1(1), 1984, pp. 35-50.

[Ed85] Eder, "Properties of Substitutions and Unifications," *J. of Symbolic Computation*, vol. 1, 1985, pp. 31-46.

[EK76] van Emden & Kowalski, "The Semantics of Predicate Logic as a Programming Language," *JACM*, 23(4), October 1976, pp. 733-742.

[En72] Enderton, *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[Fa85] Fairley, *Software Engineering Techniques*, McGraw-Hill, New York, 1985.

[GM82] Goguen & Meseguer, "Rapid Prototyping in the OBJ Executable Specification Language," *ACM SigSoft Software Engineering Notes*, 7(5), December 1982, pp. 75-84.

[GMW79] Gordon, Milner & Wadsworth, "Edinburgh LCF," *Lecture Notes in Computer Science*, vol. 78, Springer-Verlag, Berlin, 1979.

[He80] Henderson, *Functional Programming*, Prentice-Hall, Englewood-Cliffs, 1980.

[HU79] Hopcroft & Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, Reading, MA, 1979.

[Hu80] Huet, "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems," *JACM*, 27(4), October 1980, pp. 797-821.

[JLM84] Jaffar, Lassez & Maher, "A Theory of Complete Logic Programs with Equality," *J. of Logic Programming*, 1(3), 1984, pp. 211-223.

[JS86] Jaffar & Stuckey, "Canonical Logic Programs," *J. of Logic Programming*, 2(2), 1986, pp. 143-155.

[KH87] Kanamori & Horiuchi, "Construction of Logic Programs Based on Generalized Unfold/Fold Rules," *Proceedings of the 4th International Conference on Logic Programming*, MIT Press, 1987, pp. 744-768.

[Ko74] Kowalski, "Predicate Logic as a Programming Language," in *Information Processing '74*, Rosenfeld (ed.), North-Holland, Amsterdam, 1974, pp. 556-574.

[L82] Lloyd, "Foundations of Logic Programming," TR-82/7, University of Melbourne, 1982. Also see *Foundations of Logic Programming*, Springer-Verlag, New York, 1984.

[LM85] Lassez & Maher, "Optimal Fixedpoints of Logic Programs," *Theoretical Computer Science*, vol. 39, no. 1, 1985, pp. 15-25.

[Lo78] Loveland, *Automated Theorem Proving: A Logical Basis*, North-Holland, Amsterdam, 1978.

[LT84] Lloyd & Topor, "Making Prolog More Expressive," *J. of Logic Programming*, 3(3), 1984, pp. 225-240.

[Mi84] Mishra, "Towards a Theory of Types in Prolog," *Proc. 1st International IEEE Symposium on Logic Programming,* Atlantic City, 1984, pp. 289-298.

[MK84] Mycroft & O'Keefe, "A Polymorphic Type System for Prolog," *Artificial Intelligence,* vol. 23, 1984, pp. 295-307.

[MW87] Maier & Warren, *Computing with Logic,* Benjamin-Cummings, Menlo Park, CA, 1987.

[Na85] Naish, "All Solutions Predicates in Prolog," *Proc. 2nd IEEE Symposium on Logic Programming,* 1985, pp. 73-77.

[Ni80] Nilsson, *Principles of Artificial Intelligence,* Tioga, Palo Alto, 1980.

[O85] O'Donnell, *Equational Logic as a Programming Language,* MIT Press, 1985.

[P85] Pereira, et al., *C-Prolog User's Manual,* edCAAD, Dept. of Architecture, University of Edinburgh, 1985.

[PG86] Poole & Goebel, "Gracefully Adding Negation and Disjunction to Prolog," *Proceedings 3rd International Conf. on Logic Programming,* London, 1986, also in *Lecture Notes in Computer Science,* Shapiro (ed.), vol. 225, Springer-Verlag, Berlin, pp. 635-641.

[Re78] Reiter, "On Closed World Databases," in *Logic and Data Bases,* Gallaire & Minker (eds.), Plenum Press, New York, 1978, pp. 55-76.

[Ro65] Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *JACM,* 12(1), January 1965, pp. 23-41.

[Ro85] Rollins, *A Syntax-Directed Compiler Constructor,* TR-85/005, Oregon Graduate Center, 1985.

[SB86] Sterling & Beer, "Incremental Flavor-Mixing of Meta-Interpreters for Expert System Construction," *Proc. 3rd IEEE Symposium on Logic Programming,* 1986, pp. 20-27.

[Sh84] Shepherdson, "Negation as Failure," *J. of Logic Programming,* 1(1), 1984, pp. 51-79.

[Sh85] Shepherdson, "Negation as Failure. II," *J. of Logic Programming,* 2(3), 1985, pp. 185-202.

[Sh86] Shepherdson, "Negation in Logic Programming: A Survey," *Foundations of*

Deductive Databases and Logic Programming, Minker (ed.), Washington, DC, 1986.

[SS82] Sebelik & Stepanek, "Horn Clause Programs for Recursive Functions," in *Logic Programming,* Clark & Tarnlund (eds), Academic Press, New York, 1982, pp. 324-340.

[St77] Stoy, *Denotational Semantics,* MIT Press, Cambridge, MA, 1977.

[St84] Stickel, "A Prolog Technology Theorem Prover," *Proceedings 1st International IEEE Symposium on Logic Programming,* Atlantic City, 1984, pp. 211-217.

[Sz69] Szabo (ed.), *Collected Papers of Gerhard Gentzen,* North-Holland, Amsterdam, 1969.

[Ta55] Tarski, "A Lattice-Theoretical Fixpoint Theorem and its Applications," *Pacific Journal of Mathematics,* 1955, pp. 285-309.

[TS83] Tamaki & Sato, "A Transformation System for Logic Programs with Preserves Equivalence," TR-018, ICOT, 1983.

[Tu76] Turner, *SASL Language Manual,* Computer Laboratory, University of Kent, 1976.

[Tu84] Turner, *Logics for Artificial Intelligence,* Halsted Press, New York, 1984.

[TZ87] Thom & Zobel (eds.), *NU-Prolog Reference Manual,* TR-86/10, University of Melbourne, 1987.

[U80] Ullman, *Principles of Database Systems,* Computer Science Press, Potomac, MD, 1980.

[WJ74] Wirth & Jensen, *Pascal User Manual and Report,* Springer-Verlag, Berlin, 1974.

Appendix A
Three-Valued Truth Tables

This appendix contains three-valued truth tables for all of the logical connectives in Ω_0:

Negation:

Truth Table for Negation

x	$\neg x$
t	**f**
u	**u**
f	**t**

Conjunction:

Truth Table for Conjunction

		y		
	$x \wedge y$	**t**	**u**	**f**
x	**t**	**t**	**u**	**f**
	u	**u**	**u**	**f**
	f	**f**	**f**	**f**

Disjunction:

Truth Table for Disjunction

		y		
	$x \vee y$	**t**	**u**	**f**
x	**t**	**t**	**u**	**t**
	u	**t**	**u**	**u**
	f	**t**	**u**	**f**

Implication:

Truth Table for Implication				
			y	
	$x \rightarrow y$	**t**	**u**	**f**
x	**t**	**t**	**u**	**f**
	u	**t**	**u**	**u**
	f	**t**	**t**	**t**

Equivalence:

Truth Table for Equivalence				
			y	
	$x \leftrightarrow y$	**t**	**u**	**f**
x	**t**	**t**	**u**	**f**
	u	**u**	**u**	**u**
	f	**f**	**u**	**t**

Appendix B
Three-Valued Valuations

This appendix contains justification for the three-valued truth tables of Appendix A. The assignment of a logical constant to a Boolean expression is called a *valuation*. The truth tables of Appendix A are the strongest extension of the classical 2-valued truth tables. The 3-valued truth tables agree with the usual truth tables on Boolean expressions that do not contain **u**. When **u** is viewed as containing less information than **t** and **f**, the truth tables are monotonic. Monotonicity ensures that a better defined valuation always results from a more informative Boolean expression. As a result, all laws, such as De Morgan's, are observed by the 3-valued truth tables.

As in Chapter 4, the relation $\sqsubset$ on the logical constants is defined, based on their information content.

$$\mathbf{u} \sqsubset \mathbf{f} \text{ and } \mathbf{u} \sqsubset \mathbf{t}$$

This relation may be extended to Boolean expressions constructed from algebra $B(\{\mathbf{t}, \mathbf{u}, \mathbf{f}\})$, as follows:

Logical constants:

$x \sqsubseteq y$ if $x = y$ or $x \sqsubset y$.

Set of expressions:

$S \sqsubseteq T$ if there is a bijection $o: S \rightarrow T$, such that $x \sqsubseteq o(x)$ for all $x \in S$.

Logical operators:

$(\sim x) \sqsubseteq (\sim y)$ if $x \sqsubseteq y$

$(\bigwedge S) \sqsubseteq (\bigwedge T)$ if $S \sqsubseteq T$

$(\bigvee S) \sqsubseteq (\bigvee T)$ if $S \sqsubseteq T$.

Through induction on the structure of Boolean expressions, $\sqsubseteq$ is a partial ordering.

As particular Boolean algebras, let:

$B_2 = B(\{\mathbf{t}, \mathbf{f}\})$, and

$B_3 = B(\{\mathbf{t}, \mathbf{u}, \mathbf{f}\})$.

B_2 is the set of Boolean expressions containing only the logical constants **t** and **f**, while B_3 is the set of expressions containing all logical constants. Note that the maximal expressions in B_3 with respect to the partial ordering $\sqsubseteq$ are expressions from B_2.

A 2-valued *valuation* is a mapping from B_2 to $\{\mathbf{t}, \mathbf{f}\}$. The *classical valuation* v_c is a particular valuation (there could be others). For example, $v_c : \mathbf{t} \bigwedge \mathbf{f} \mapsto \mathbf{f}$. Similarly, a 3-valued valuation is a mapping from B_3 to $\{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$.

A 3-valued valuation v' is an *extension* of a 2-valued valuation v if $v'(x) \sqsubseteq v(x)$ for all $x \in B_2$. That is, v' is no better defined than v on any 2-valued expression. We will be describing a particular 3-valued extension $v_c{}'$ of v_c. In this extension $v_c{}'(x) = v_c(x)$ for all $x \in B_2$. For example, since $v_c{}'$ is an extension of v_c, it must be that $v_c{}': \mathbf{t} \wedge \mathbf{f} \mapsto \mathbf{f}$.

According to the usual definition, 3-valued valuation v' is *monotonic* if $x \sqsubseteq y$ implies $v'(x) \sqsubseteq v'(y)$ for all expressions x and y from B_3. This property ensures that increased information will enhance the information provided by the valuation.

If S is a set of expressions, as shorthand, let $v(S) = \{v(x) \mid x \in S\}$. As a particular class of 3-valued valuations, V is a functional producing a 3-valued valuation from its 2-valued input. Define $V(v)(x) = \sqcap\, v(M_x)$, where $M_x = \{y \in B_2 \mid x \sqsubseteq y\}$. Each 3-valued valuation $V(v)$ is an extension of 2-valued valuation v. For example, the classical extension is defined as $v_c{}' = V(v_c)$. Appendix A provides truth tables for the operations of Ω_0 using this definition of $v_c{}'$. It is necessary to show that the resulting 3-valued valuation conforms to its 2-valued component, and is well-behaved.

Lemma B.1: For all 2-valued valuations v, $V(v)$ is a monotonic extension of v.

Proof: Suppose expressions x and y are in B_3, and $x \sqsubseteq y$. Then $M_x \supseteq M_y$, so $v(M_x) \supseteq v(M_y)$, and $\sqcap\, v(M_x) \sqsubseteq \sqcap\, v(M_y)$. Hence, $V(v)(x) \sqsubseteq V(v)(y)$, and $V(v)$ is monotonic. □

We now show that the functional V is as strong as any other method for producing monotonic extensions of 2-valued valuations. Valuation w is *stronger* than valuation v, denoted $v \sqsubseteq w$, if $v(x) \sqsubseteq w(x)$ for all $x \in B_3$. Since this relation on valuations is a simple extension of the partial ordering $\sqsubseteq$ on Boolean expressions, it is easy to show that the relation on valuations is also a partial ordering.

Lemma B.2: If valuation w is a monotonic extension of 2-valued valuation v, then $w \sqsubseteq V(v)$.

Proof: If $w(x) = \mathbf{u}$, then $w(x) \sqsubseteq V(v)(x)$ regardless of the actual valuation $V(v)$. If $w(x) \neq \mathbf{u}$, then $w(x) = w(y)$ for all y such that $x \sqsubseteq y$. In particular, for any maximal element $m \in M_x$, $w(x) = w(m)$. Since w is an extension of v, $w(m) \sqsubseteq v(m)$. And by the definition of V, $v(m) = V(v)(m)$. Since $w(x) \sqsubseteq V(v)(m)$ for all $m \in M_x$, $w(x) \sqsubseteq \bigsqcap v(M_x)$. □

We have therefore established that $v_c{}'$ is the strongest monotonic extension of the classical 2-valued valuation v_c. Within Chapter 4, it is important to determine that certain properties, including De Morgan's laws, still hold for the valuation $v_c{}'$. Because the associative and commutative operation $\bigsqcap$ is used in the construction of $V(v)$, the 3-valued valuation $v_c{}'$ is indeed associative and commutative for conjunction and disjunction. De Morgan's laws are strengthened for infinite conjunctions and disjunctions, as follows:

$$\sim\!(\bigwedge\{x_1, x_2, \cdots\}) = \bigvee\{\sim\!x_1, \sim\!x_2, \cdots\}$$
$$\sim\!(\bigvee\{x_1, x_2, \cdots\}) = \bigwedge\{\sim\!x_1, \sim\!x_2, \cdots\}$$

It is not difficult to show that De Morgan's laws are also observed for $v_c{}'$.

Having justified the construction of $v_c{}'$, its use is implicit within the dissertation. Thus, a Boolean expression x will stand for its valuation under $v_c{}'$.

Appendix C
DEF-Program Example

The following program is used as an example of a DEF-program. The program defines a `prime` predicate, among others. The program is not especially efficient, but does clearly represent certain relations about natural numbers. Included in the program are assertions utilized for typechecking, discussed in Chapter 6, Section 2.

```
0:[]→nats.
s:[nats]→nats.

% lt(I,J): true iff I<J.
lt:[nats,nats].
lt(0,s(J)) ↔ true.
lt(I,0) ↔ ~true.
lt(s(I),s(J)) ↔ lt(I,J).

% le(I,J): true iff I≤J.
le:[nats,nats].
le(I,J) ↔ ~lt(J,I).

% ge(I,J): true iff I≥J.
ge:[nats,nats].
ge(I,J) ↔ ~lt(I,J).

% gt(I,J): true iff I>J.
gt:[nats,nats].
gt(I,J) ↔ lt(J,I).

% eq(I,J): true iff I=J.
eq:[nats,nats].
eq(I,J) ↔ le(I,J) ∧ le(J,I).

% add(I,J,K): true iff I+J=K.
add:[nats,nats,nats].
add(0,J,K) ↔ eq(J,K).
add(s(I),J,0) ↔ ~true.
add(s(I),J,s(K)) ↔ add(I,J,K).
```

```
% mult(I,J,K): true iff I*J=K.
mult:[nats,nats,nats].
mult(0,J,0) ↔ true.
mult(0,J,s(K)) ↔ ~true.
mult(s(I),J,K) ↔ ∃X: mult(I,J,X) ∧ add(X,J,K).

% divp(I,J): true iff I divides J evenly.
divp:[nats,nats].
divp(I,J) ↔ ∃X: le(X,J) ∧ mult(X,I,J).

% prime(P): true iff P is prime.
prime:[nats].
prime(P) ↔
      gt(P,s(0))
      ∧ (∀X: lt(s(0),X) ∧ lt(X,P) → ~divp(X,P)).
```

Biographical Note

The author was born 25 October 1957, in London, England. In 1961 he moved to the San Francisco Bay Area where he attended public schools and graduated from San Carlos High School in 1974. He received a Bachelor of Arts degree from the University of California at San Diego, majoring in the Management Science specialty of the Economics department.

After graduation, the author began work as a commercial programmer; first, at Freightliner Corporation in Portland, Oregon; and then at Microsoft of Redmond Washington. After four years, the author moved back to Portland to begin computer science studies at the Oregon Graduate Center. While at the Graduate Center, the author was awarded first prize at the first annual Computer Science Research Symposium, and during the academic year 1986-1987, his last year at the Graduate Center, obtained a generous grant from Vincent Sigilito at the Air Force Office of Scientific Research.

The author has been married for nine years to the former Deborah Lynn Morrison and they have one daughter, Sarah Elizabeth, age 2.

The author is leaving the Graduate Center to accept a tenure-track position in the Mathematics and Computer Science department of Dartmouth College, Hanover, New Hampshire. The author will continue work in logic programming and declarative languages.