Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

LOG(F): AN OPTIMAL COMBINATION OF LOGIC PROGRAMMING, REWRITING AND LAZY EVALUATION

Sanjai Narain

August 1988
CSD-880040

The dissertation of Sanjai Narain is approved.

Milos D. Ercegovac

David Jefferson

Yiannis N. Moschovakis

Srinivasan Balakrishnan

D. Stott Parker, Committee Chair

University of California, Los Angeles

1988

Dedicated to my parents

Table of Contents

Figures and Table

ACKNOWLEDGEMENTS

I am very grateful to my advisor, Professor D.S. Parker, for his sustained guidance, encouragement, and criticism. His generosity with his time, especially in the pleasant surroundings of the Kerchoff cafeteria, greatly accelerated the development of my ideas. His meticulousness has tremendously improved the clarity of the thesis. Where it is still obscure, it is because I did not heed his suggestions.

I thank my other committee members Professors Y. Moschovakis, M. Ercegovac, D. Jefferson, and S. Balakrishnan for their continued interest, and enthusiasm in my work.

Professor J.A. Robinson, by his example, showed me what research is, and how to do it. His influence upon my attitude towards many aspects of life has been profound.

I greatly appreciate Hassan Ait-Kaci's persistence in asking me questions. In the process of answering these, I was led to a stronger version of the reduction-completeness theorem, and eventually to a proof of confluence for DF*.

I have benefitted enormously from the fertile research environment at RAND Corporation. I am especially grateful to Anthony Hearn, Philip Klahr, Randall Steeb, Louis Miller, Iris Kameny, and Jeffrey Rothenberg for encouraging me to enter the doctoral program, and actively supporting me throughout it. I have received tremendous inspiration from the examples of David McArthur, Norman Shapiro, and Gary Martins. I am also grateful to Jim Guyton, and Terry West for keeping the RAND computing facilities first rate. Because of them I was spared from being just another graduate student at UCLA. Stephanie Cammarata's experience at UCLA was invaluable in streamlining my journey through it.

I am greatly indebted to the excellent teachers I have had since childhood, especially Ms. V. Donald, Ms. Collinson, Mr. McLeod, Mr. Flynn, Mr. G. Bose, Mr. W. Gardener, Mr. Gupta, Mr. Jha, Mrs. S. Varma in high school, Professor B. Bhat at Indian Institute of Technology, Professors R. Kowalski, and K. Clark, at Syracuse University, and Professor A. Church at UCLA.

Finally, I am very grateful to my wife Renu, who cheerfully let me spend most of the first few years of our marriage on my dissertation, even though it did not leave much time for her to get to know me well. Her support has been crucial in my endeavors.

VITA

Born, Uttar Pradesh, India

1979	B.Tech., Electrical Engineering Indian Institute of Technology New Delhi, India
1981	M.S., Computer Science Syracuse University Syracuse, New York
1981-Present	Computer Scientist, Rand Corporation Santa Monica, California

PUBLICATIONS AND PRESENTATIONS

Narain, S. [1987]. Why is it difficult to program in von Neumann languages? N-2660, RAND Corporation, Santa Monica, November.

Narain, S. [1986]. A technique for doing lazy evaluation in logic. *The Journal of Logic Programming*, vol. 3, no. 3, October.

Narain. S. [1986]. MYCIN: The expert system and its implementation in LOGLISP. In *Logic Programming and its Applications*, eds. D.H.D. Warren, Michel van Caneghem, Ablex Publishing.

Klahr, P., McArthur, D., Narain, S. [1985]. The ROSS Language Manual. N-1854-1, RAND Corporation, Santa Monica, July.

Steeb, R., Cammarata, S., Narain, S., Rothenberg, J., Giarla, W. [1985]. Cooperative Intelligence for RPV Fleet Control: Analysis and Simulation. R-3408, RAND Corporation, Santa Monica, April.

Klahr, P., Ellis, J., Giarla, W., Narain, S., Cesar, E., Turner, S. [1984]. TWIRL: Tactical Warfare in the ROSS Language. R-3158, RAND Corporation, Santa Monica, October.

Narain, S., McArthur, D., Klahr, P. [1982]. Large Scale System Development in Several Lisp Environments. *Proceedings of IJCAI,* Karlsruhe, West Germany.

Klahr, P., McArthur, D., Narain, S. [1982]. SWIRL: An Object-oriented Air Battle Simulator. *Proceedings of AAAI*, Pittsburgh, PA.

ABSTRACT OF THE DISSERTATION

LOG(F): An optimal combination of logic programming,

rewriting, and lazy evaluation

by

Sanjai Narain

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1988

Professor D. Stott Parker, Chair

A new approach for combining logic programming, rewriting, and lazy evaluation is described. It rests upon *subsuming* within logic programming, instead of upon *extending* it with, rewriting, and lazy evaluation.

A non-terminating, non-deterministic rewrite rule system, F* and a reduction strategy for it, select, are defined. F* is shown to be reduction-complete in that select simplifies terms whenever possible. A class of F* programs called Deterministic F* is defined and shown to satisfy confluence, directedness, and minimality. Confluence ensures that every term can be simplified in at most one way. Directedness eliminates searching in simplification of terms. Minimality ensures that select simplifies terms in a minimum number of steps. Completeness and minimality enable select to exhibit, respectively, weak and strong forms of laziness.

F* can be compiled into Horn clauses in such a way that when SLD-resolution interprets these, it directly simulates the behavior of select. Thus, SLD-resolution is made to exhibit laziness. LOG(F) is defined to be a logic programming system augmented with an F* compiler, and the equality axiom X=X. LOG(F) can be used to do lazy functional programming in logic, implement useful cases of the rule of substitution of equals for equals, and obtain a new proof of confluence for combinatory logic.

CHAPTER I

INTRODUCTION

1.0 THE PROBLEM

Logic programming [Kowalski 1979], is the use of statements of logic as computer programs. It has led to new insights into computing as well as logic. Rewriting is synonymous with reduction, as described, for example, in [Knuth & Bendix 1970]. It is simplification of an expression by successive application of some collection of rewrite rules. Its usefulness is evident from its appearance in many branches of mathematics. Lazy evaluation, e.g. [Vuillemin 1974], is a method of computing which ensures that a computation step is performed only when there is need to perform it. Thus, not only does it enable certain computations to terminate more quickly, it also enables computation with infinite data structures.

A system in which logic programming, rewriting, and lazy evaluation were combined could put considerable programming power at our disposal. In particular, it would simultaneously afford the expressive power of both functions, and relations.

Furthermore, such a system could be used to implement instances of the rule of substitution of equals for equals in logical statements. This is a very important rule, as witness its use in the simplest of mathematical derivations, e.g. solution of trigonometric identities. Logical statements could be expressed using logic programs, while equality theories could be expressed using rewrite rules.

We propose a new approach for building the above system which is rigorous, as well as computationally efficient. It rests upon *subsuming* within logic programming, instead of *extending* it with, rewriting and lazy evaluation. This means that SLD-resolution, the proof procedure used for logic programming, is not changed. Instead, Horn clauses, or *pure* Prolog clauses are written in such a way, that when SLD-resolution interprets them, it directly simulates lazy rewriting. The resulting system is called LOG(F). It can be said to make contributions to the following three areas:

1. Rewriting. Simple, syntactic conditions are defined under which non-terminating, non-deterministic rewrite rules, with pattern matching, satisfy quite useful computational properties. These are regarding demand-driven reduction, confluence, elimination of search during reduction, and lengths of reductions.

2. Combination of logic programming and rewriting. It is shown how rewriting can be *subsumed* within logic programming. In particular, the above computational properties of rewriting are realized within logic programming, without changing it, and without sacrificing logical rigor. This has two important consequences.

First, a satisfactory combination of logic programming, and rewriting is achieved, *without* developing a new computational model of which the two are instances. Developing such a model is quite difficult, particularly if it is to have satisfactory declarative, *and* satisfactory procedural semantics. Second, full advantage is taken of the very efficient implementations of Prologs. Thus, formidable problems that implementation of the new model would very likely pose, are avoided.

3. Lazy evaluation. By 1 and 2, it is shown how lazy evaluation can be done *efficiently, within* the normally eager framework of logic programming. Thus a basis is established for understanding lazy evaluation purely in terms of well understood ideas in first order logic. Also, a new, and powerful use is found for an old, and widely used tool, namely, Prolog.

2.0 SUMMARY OF MAIN RESULTS

2.1 A rewrite rule system F*

A first-order rewrite rule system F*, with pattern matching, is defined. The function symbols are partitioned *in advance* into constructors, and non-constructors. Simplification in F* means reducing ground terms to simplified forms i.e. terms of the form c(t1,..,tn) where c is a constructor symbol, and each of t1,..,tn is a ground term. Simplified forms can be used to represent finite approximations to infinite structures, and are analogous to head-normal forms in the lambda-calculus [Wadsworth 1976]. In contrast, a normal form is defined to be a term in which all function symbols are constructors.

Now, an important point is that a method for computing simplified forms can be used repeatedly to compute normal forms. Moreover, it would terminate more often than would a method which directly computes normal forms. Hence it is sufficient to develop, and study properties of, a method for computing simplified forms.

An F* program is a finite set of rules, each of the form LHS=>RHS, satisfying the following restrictions: (1) LHS is of the form f(L1,..,Lm), m>=0, f a non-constructor function symbol, and each Li either a variable, or of the form c(T1,..,Tn), n>=0, c a constructor symbol, and each Ti a variable, (2) a variable occurs at most once in LHS, and (3) all variables of RHS occur in LHS. Note that *non-terminating, non-deterministic* sets of rewrite rules are permissible. Also any rule with left hand side of depth greater than two can easily be expressed in terms of rules with left hand sides of depth at most two, as required by (1).

Where P is an F* program, and f(T1,..,Tn) a ground term, a reduction strategy for P, select_P, is defined by the following pseudo-Horn clauses:

> select_P(f(T1,..,Tn),f(T1,..,Tn)) if f(T1,..,Tn)$=>_P$X.
>
> select_P(f(T1,..,Ti,..,Tn),X) if
>
> > there is a rule f(L1,..,Li,..,Ln)=>RHS in P, and
> >
> > there is no substitution σ such that Ti=Liσ, and
> >
> > select_P(Ti,X).

Here A=>B means there is a rule LHS=>RHS such that A matches LHS with substitution σ, and B is RHSσ. Select is shown to be *reduction-complete*, in that if a term can be simplified, it can be simplified by reducing it via select. *Thus select exhibits a weak form of laziness.* An example of an F* program is:

```
perm([])=>[].
perm([A|V])=>insert(U,perm(V)).
insert(U,X)=>[U|X].
```

insert(U,[A|B])=>[A|insert(U,B)].

Here [], | are constructors, while perm, insert are non-constructors. The term perm([1,2,3]) is now reduced by select to [1|perm([2,3])], [2|insert(1,perm([3]))], and [3|insert(1,insert(2,perm([])))]. If further reduction is desired, select may be called recursively upon the arguments of | to yield each of [1,2,3], [1,3,2], [2,3,1], [2,1,3], [3,1,2], [3,2,1].

2.2 Deterministic F*

An F* program P is a DF* program if (1) left hand sides of no two rules in P unify, and (2) where f(L1,..,Li,..,Lm)=>RHS is a rule in P, and Li is not a variable, then in every other rule f(K1,..,Ki,..,Km)=>RHS1 in P, Ki is not a variable. These restrictions are very reasonable, and as examples throughout this thesis show, it is possible to adhere to these, yet write quite expressive programs.

DF* is shown to satisfy *confluence, directedness*, and *minimality*. Confluence ensures that a term can be simplified in at most one way. Directedness ensures that to simplify a term it is sufficient to compute *any* reduction computable by select. Moreover, all reductions computable by select are of equal length. Thus, during reduction, no searching is necessary. Provided, whenever a term is reduced, all copies of it are simultaneously reduced, minimality ensures that select simplifies terms in a minimum number of steps. *Thus, select exhibits a strong form of laziness.* An example of a DF* program is:

append([],X)=>X.

```
append([U|V],W)=>[U|append(V,W)].
interleave([U|V],X)=>[U|interleave(X,V)].
a=>[1|a].
b=>[2|b].
```

However, the F* program above to insert an element non-deterministically into a list is not in DF*.

2.3 Compiling F* into Horn clauses

F* programs can be compiled into Horn clauses in such a way that when SLD-resolution interprets these, it directly simulates the behavior of (the interpreter based upon) select. This means that there is, essentially, a one-to-one correspondence between steps executed by (the interpreter based upon) select, and steps executed by SLD-resolution. This is accomplished by translating each F* rule into a distinct Horn clause, and *simultaneously* embodying in that clause, information about the logic of the rule, as well as information about the control of select when interpreting that rule.

Thus, SLD-resolution is made to exhibit laziness. If the F* program is also in DF*, clauses can be further transformed to eliminate all backtracking. Finally, clauses can be compiled into machine code by Prolog compilers. The compilation algorithm consists of two steps:

Step 1. For each n-ary, n>=0, constructor symbol c in P, and where X1,..,Xn are distinct variables, generate the clause:

```
reduce(c(X1,..,Xn),c(X1,..,Xn))
```

Step 2. Let f(L1,..,Lm)=>RHS be a rule in P. Let A1,..,Am,Out be distinct Prolog variables not occurring in th rule. If Li is a variable let Qi be Ai=Li. If Li is c(X1,..,Xn) where c is a constructor symbol, and each Xi a variable, let Qi be reduce(Ai,c(X1,..,Xn)). Generate the clause:

```
reduce(f(A1,..,Am),Out):-Q1,..,Qm,reduce(RHS,Out).
```

In practice, if Li is a variable, Qi can be dropped, provided Ai is replaced by Li in f(A1,..,Am). For example, the above F*, and DF* programs are compiled into:

```
reduce([],[]).
reduce([U|V],[U|V]).

reduce(perm(X),Z):-reduce(X,[]),reduce([],Z).
reduce(perm(X),Z):-reduce(X,[FX|RX]),reduce(insert(FX,perm(RX)),Z).
reduce(insert(A,X),Z):-reduce([A|X],Z).
reduce(insert(A,X),Z):-reduce(X,[FX|RX]),reduce([FX|insert(A,RX)],Z).
reduce(append(X,Y),Z):-reduce(X,[]),reduce(Y,Z).
reduce(append(X,Y),Z):-reduce(X,[U|V]),reduce([U|append(V,Y)],Z).
reduce(interleave(X,Y),Z):-reduce(X,[U|V]),reduce([U|interleave(Y,V)],Z).
reduce(a,Z):-reduce([1|a],Z).
reduce(b,Z):-reduce([2|b],Z).
```

If we now type, in Prolog, reduce(perm([1,2,3]),Z), we obtain Z=[1|perm([2,3])], Z=[2|insert(1,perm([3]))], and Z=[3|insert(1,insert(2,perm([])))]. Note the following: First, perm([1,2,3]) is only partially reduced, and *directly* by Prolog, not by some lazy interpreter implemented in Prolog. Second, the terms to which Z is bound are exactly those to which perm([1,2,3]) is reduced by select. This illustrates Prolog simulating behavior of select. If we now define:

```
first(0,X,[]).
first(N,X,[FX|Z]):-not(N=0),reduce(X,[FX|RX]),N1 is N-1,first(N1,RX,Z).
make_list(E,[]):-reduce(E,[]).
make_list(E,[FE|Z]):-reduce(E,[FE|RE]),make_list(RE,Z).
print_list(X):-reduce(X,[FX|RX]),write(FX),write(','),print_list(RX).
```

and then type, make_list(perm([1,2,3]),Z), we obtain Z=[1,2,3], Z=[1,3,2], Z=[2,1,3], Z=[2,3,1], Z=[3,1,2], Z=[3,2,1]. As further examples, if we type the queries on the left-hand side, we obtain the answers on the right-hand side:

```
reduce(append(a,b),Z) ---> Z=[1|append(a,b)]
reduce(interleave(a,b),Z) ---> Z=[1|interleave(b,a)]
first(5,interleave(a,b),Z) ---> Z=[1,2,1,2,1]
print_list(interleave(a,b)) ---> 1,2,1,2,1,2,.....
```

2.4 LOG(F)

LOG(F) is defined to be a logic programming system augmented with an F* compiler, and the equality axiom X=X. The result of compilation is to add to a logic programming system, a primitive for lazily simplifying F* terms. This primitive can be called from other Horn clauses, so LOG(F) is proposed as a combination of logic programming, rewriting and lazy evaluation.

For problems for which lazy evaluation does not reduce lengths of computation, e.g. sorting, or all permutations, LOG(F) is empirically found to be about five times slower than Prolog. For problems for which lazy evaluation does reduce lengths of computation, e.g. N-queens, LOG(F) is faster than Prolog by unbounded, even infinite, amounts.

In the literature, [Vuillemin 1974, Berry & Levy 1979], optimality is used synonymously with minimality. Due to minimality of DF*, LOG(F) can also be said to be optimal. It can also be said to be so in a weaker sense, because of its desirable computational properties, and their economical realization in Prolog.

2.5 Applications of LOG(F)

LOG(F) can be used to do lazy functional programming in logic. In particular, it can be used to manipulate representations of infinite structures, such as in real analysis, exact real arithmetic, graphics, or networks of communicating processes.

The SKI rules of combinatory logic can be expressed as a DF* program. From confluence of DF*, a new proof is obtained of the confluence of combinatory logic.

DF* seems to offer a reasonable compromise between sequential execution and unbounded parallelism. Due to directedness of DF*, arguments of f in f(t1,..,tm) can be simplified in parallel, however, they would be simplified lazily. Thus, DF* seems to be a good candidate for implementation on parallel machines.

Finally, if a DF* program is interpreted as an equality theory, reduce clauses can be thought of as implementing an equality theory in Prolog with the restriction that it be used only for simplification of terms. Now, given a clause of the form p(c(X1,..,Xm)):-Body, where c is a constructor symbol, we can add another clause stating a rule of substitution of equals:

```
p(X):-reduce(X,c(X1,..,Xm)),p(c(X1,..,Xm)).
```

Now, even when a term E is not of the form c(X1,..,Xm), p can still be inferred for E, provided E is reducible to a term of the form c(X1,..,Xm). For example, with the Prolog rule for computing perimeters of regular polygons, peri(reg_poly(N,S),Z):-Z is N*S, we can infer peri(reg_poly(3,10),30). We can now add the clause:

```
peri(X,Y):-reduce(X,Z),peri(Z,Y).
```

Where reg_poly is a constructor, and equi, square, and hexagon are non-constructors, an equality theory among polygons, expressed in F*, is:

```
equi(S)=>reg_poly(3,S).
square(S)=>reg_poly(4,S).
hexagon(S)=>reg_poly(6,S).
```

This is compiled into:

```
reduce(reg_poly(A,B),reg_poly(A,B)).
reduce(equi(S),Z):-reduce(reg_poly(3,S),Z).
reduce(square(S),Z):-reduce(reg_poly(4,S),Z).
reduce(hexagon(S),Z):-reduce(reg_poly(6,S),Z).
```

The Prolog query peri(equi(10),30), now succeeds. Thus Prolog automatically infers the result of substituting equi(10) for reg_poly(3,10), in peri(reg_poly(3,10),30). Of course, if we type peri(square(3),Z), we obtain Z=12.

3.0 RELATIONSHIP WITH PREVIOUS WORK

There seem to be two major approaches to combining logic programming, and rewriting. The first consists of implementing logic programming in rewriting, e.g. LOGLISP [Robinson & Sibert 1982], or QLOG [Komorowski 1982]. However, it seems difficult for such an approach to lead to an efficient system since logic programs must pass through two high-level layers of interpretation.

The second approach consists of developing a new computational model of which both rewriting, and logic programming are instances. Examples of such models include those based upon upon semantic- or T-unification, [Goguen & Meseguer 1986], [Subrahmanyam & You 1984], [Kornfeld 1983], sets, [Robinson 1987], [Darlington et al. 1986], narrowing, [Reddy 1985], the Knuth-Bendix completion procedure, [Dershowitz & Josephson 1984], oriented equational clauses, [Fribourg 1984], residuation, [Ait-Kaci & Nasr 1987], extension of SLD-resolution with narrowing, [Yamamoto 1987], or extension of SLD-resolution with atom-elimination rule, [Barbuti et al. 1986].

In order for a new computational model to be satisfactory, it must posssess not only good declarative semantics, but also good procedural semantics. The former is essential for reasoning about programs in the model. The latter means that the behavior of the model is simple enough that it can be visualized, predicted, and controlled. It is essential if the model is to be used for programming, i.e. for expressing algorithms. In this regard, we also quote Robinson [1984]:

> ...one guiding principle must surely be that logic programming, however narrowly or broadly construed, essentially involves the ingredient of practicality. The underlying deductive processes should have enough directness and predictability to permit the planning of efficient logical computations. Herein probably lies the important distinction, difficult to make precise but nonetheless real, between logic programming proper and automatic deduction in general.

However, developing a satisfactory computational model, more general than logic programming and rewriting, is a very ambitious undertaking, particularly if the model is also to exhibit laziness. In particular, it appears that each of the above models, with the possible exception of [Robinson 1987], and [Darlington et al. 1986], either has complex declarative semantics, or complex procedural semantics.

Of course, even if a satisfactory computational model is developed, its efficient implementation on concrete machines can still pose a considerable software engineering challenge, requiring several person-years of effort. In particular, it appears that efficient implementation of the above proposals is still an ongoing effort.

Lazy evaluation itself does not seem to be easy to implement efficiently. Several implementations of lazy evaluation for functional, and logic-based languages have been proposed e.g. [Friedman & Wise 1976, Henderson 1980, Turner 1979, O'Donnell 1985, Clark & McCabe 1979, Hansson et al. 1982, Shapiro 1983, Barbuti et al. 1986]. However, only a few of these systems, e.g. Turner's, or O'Donnell's, seem to be efficient enough for practical programming.

In view of such difficulties with developing, and implementing a new computational model of which logic programming, rewriting, and lazy evaluation, are instances, we ask whether it is possible to *subsume* the last two within the first. In other words, we ask whether it is possible to keep SLD-resolution fixed, but use it in such a way that it performs, *in a computationally feasible manner*, rewriting, and lazy evaluation? If such an attempt were to succeed, we would not only obtain a declarative semantics of rewriting, and lazy evaluation using purely logical ideas, we would also have a very efficient implementation of these, in, say, Prolog.

Important precedents in this direction have already been established with the subsumption, within logic programming, of grammars, and relational databases. Definite clause grammar rules [Pereira & Warren 1980] can be expressed as Horn clauses in such a way that their interpretation using Prolog, directly simulates top-down parsing. Relational databases can be expressed directly as ground Horn clauses [Gallaire & Minker 1978]. Prolog enables inference with them in ways (e.g. using recursion) not possible with conventional data retrieval operators.

An important step towards subsuming rewriting within logic programming, has recently been taken by van Emden & Yukawa [1987], whose motivations are very similar to, but independent, of ours. They show how to derive logical consequences of the standard equality axioms which result in a small SLD-search space. They also show how to compile an equality theory into equality free Horn clauses, which also result in a small SLD-search space. *However, their approach is restricted only to terminating equality theories.* These are insufficient for representing infinite structures.

As pointed out in [Narain 1986], the compactness theorem of first order logic [Robinson 1979] suggests that lazy evaluation is already present in first order logic. It states that if an infinite set of clauses is unsatisfiable then it has a finite subset which is also unsatisfiable. Moreover, a complete proof procedure, such as SLD-resolution for Horn clauses would find this set in finite time. Thus, as with lazy evaluation, one could get termination in finite time even with an infinite input.

This idea was investigated further, and led to a method in [Narain 1986], for defining functions by Horn clauses in such a way that when SLD-resolution interprets these, it behaves lazily. However, the discussion is limited mainly to lists, although a generalization to other data structures is hinted.

The current system, LOG(F), is an attempt to generalize, and develop a purely syntactic explanation of the above method. It appears to subsume within logic programming, in a rigorous yet computationally efficient fashion, non-terminating, non-deterministic rewriting, and lazy evaluation.

Minimality of DF* appears to be a generalization of similar results by Vuillemin [1974], and Berry & Levy [1979]. Both derive it only for rewrite rules whose left hand sides are of the form f(X1,..,Xm), where each Xi is a variable. Thus, they must assume existence of a finite number of primitive functions such as if-then-else, which are not definable using such rules alone. In contrast, F* admits rewrite rules in which the Xi can be patterns. Thus, in F*, as in logic programming, it is not necessary to assume existence of any primitive functions.

Restrictions on rewrite rules in F*, and the reduction strategy select, seem to be substantially simpler than their counterparts in the system of O'Donnell [1985]. Select also seems to be substantially simpler than its counterpart in the system of Huet & Levy [1979]. Furthermore, since F* can be compiled into efficient Horn clauses, and Prolog can be used, implementation of F* is straightforward. However, implementation of the other two systems seems to be quite a major undertaking.

Confluence of DF* is anticipated by Huet [1980] who derives sufficient conditions for confluence for rewrite rule systems more general than DF*. However, our proof, being specialized for DF*, is very simple.

4.0 OUTLINE OF THESIS

Chapter II reviews relevant previous work in some depth. Chapter III defines F*, and the reduction strategy select, and establishes its reduction-completeness. Chapter IV defines DF*, and shows its confluence and directedness. Chapter V defines Labeled DF*, a subset of DF*, for the purpose of formalizing the notion of a copy of a term, and then establishes its minimality. Chapter VI describes an algorithm for compiling F* into Horn clauses, and proves its correctness. Chapter VII describes examples of programming in LOG(F). Chapter VIII compares performance of LOG(F) with that of Prolog. Chapter IX contains a summary and conclusions. APPENDICES 1-3 contain a listing of an F* compiler written in Prolog, and instructions on how to use it.

CHAPTER II

REVIEW OF PREVIOUS WORK

1.0 INTRODUCTION

This chapter discusses logic programming, rewriting, lazy evaluation, and their benefits. It then reviews previous attempts at combining the first two, and implementing the third. Most previous attempts at combining the first two, have focussed on developing advanced computational models of which the two are instances. Ensuring that such models have satisfactory declarative, and satisfactory procedural semantics is quite difficult. Moreover, their efficient implementation poses a formidable software engineering challenge.

The question arises, whether rewriting, and lazy evaluation can be *subsumed* within logic programming. If so, not only would an understanding of the first two be obtained in terms of purely logical ideas, a very efficient implementation of these would be obtained using Prolog.

2.0 LOGIC PROGRAMMING

An important step in the history of logic was the invention of the *Begriffsschrift* by Frege [1879]. It was a system consisting of two parts: a language for expressing logical ideas, and a set of rules for inferring, from statements in this language, other statements in this language. One of the most important aspects of this system was the extreme precision with which it was laid out. It would later prove crucial for enabling a computer to perform inference.

With the advent of the digital computer, logicians turned their attention to mechanizing, or automating the process of inference. A breakthrough was achieved with the discovery of the *resolution principle*, and the idea of *unification* [Robinson 1965]. Resolution was a considerable improvement over methods contemporary to it e.g. of Davis & Putnam [1960]. One application of it was to programming, following the idea that deduction could legitimately be called computation. This idea was later called *logic programming*. Unfortunately, Green [1969] showed that even resolution was too inefficient for practical programming.

In 1974, Kowalski [1979] proposed SLD-resolution, a refinement of resolution, for proving theorems in the Horn clausal subset of first-order logic. Furthermore, he refined the idea of logic programming, by proposing the *procedural interpretation* of Horn clauses. Under it, Horn clauses were to be regarded as procedures in a conventional programming language, and SLD-resolution as their interpreter. Thus, clauses could be used not only to specify relations but also, simultaneously, to specify *algorithms* for computing them.

For example, not only could one write clauses expressing the sorting relation, one could do so in such a way that sequences were sorted in a number of steps proportional to that required by quicksort (or mergesort, or bubblesort). Not only could one write clauses expressing grammar rules, one could do so in such a way that phrases were parsed in a top-down fashion.

The procedural interpretation is possible due to the simplicity of SLD-resolution. It is simple enough that its behavior, as it is interpreting Horn clauses, can be visualized and predicted. Hence one can write clauses in such a way that when SLD-resolution interprets them, it behaves, in most cases, in whatever fashion one would like it to behave. In contrast, resolution does not enjoy this property of programmability. Its behavior is substantially more difficult to visualize and predict, and hence to control. As noted in Chapter I, Robinson [1984] expresses similar views.

An important advantage of expressing algorithms in Horn clauses is that they can be analyzed as statements of logic. This has led to new insights into issues in algorithm design such as correctness, termination, composition, semantics, or concurrency. Moreover, the programmer also has at his disposal powerful concepts from logic such as logical variable, unification, inference step, sets, or non-determinism.

SLD-resolution is *not* necessarily more efficient than resolution. It is easy to conceive of a set of Horn clauses which can be refuted in a smaller number of steps by a bottom up proof procedure such as hyper-resolution, than by SLD-resolution. Worse, Rabin cites a result by himself and Fischer that decision problems for even rather simple logical systems, such as Presburger Arithmetic, are of super-exponential complexity. Since such problems can be coded up in Horn clauses, even SLD-resolution would be hopelessly inefficient for these. What then is the point of SLD-resolution? As discussed above, its point is not that it is efficient, but that it is *programmable*.

A *logic program* [Lloyd 1984] consists of a set of statements, called *Horn clauses*, each of the form:

$$A \leftarrow B_1,..,B_n,\ n \geq 0$$

where A and each Bi are predications, each of the form R(t1,..,tm), m>=0, R is a relation symbol and each ti is a term. A term is either a variable, or a function application of the form f(s1,..,sq), q>=0, where f is a q-ary function symbol, and each si is a term.

The reading of a clause A<-B1,..,Bn, n>=0 is that for all values of variables in the clause, the value of A is true if the value of each of B1,..,Bn is true. The clause R(t1,..,tm)<-B1,..,Bn can be thought of as part of definition of relation denoted by R.

A *substitution* is a set $\{<y1,t1>,..,<ym,tm>\}$, m>=0, where y1,..,ym are distinct variables. For each i, ti is a term, and yi is said to be bound to ti in the substitution. Where E is a term and θ is a substitution, $E\theta$ represents the result of replacing each variable in E by the term to which it is bound in θ. Where $\sigma=\{<x1,t1>,..,<xm,tm>\}$, and $\tau=\{<y1,s1>,..,<yk,sk>\}$, $\sigma\tau$ is defined to be the substitution $\{<x1,(x1\sigma)\tau>,..,<xm,(xm\sigma)\tau>, <y1,(y1\sigma)\tau>,..,<yk,(yk\sigma)\tau>\}$.

A substitution σ is said to be a *unifier* of terms E and F, if $E\sigma=F\sigma$. A substitution σ is said to be *more general* than a substitution τ, iff there exists a substitution δ such that $\sigma\delta=\tau$. A unifier σ of E and F is said to be *most general*, if for every other unifier θ of E and F, there exists a substitution δ such that $\sigma\delta=\theta$.

A term is called *ground* if it does not contain any variables. A term E is a called a *ground instance* of a term F, if there exists a substitution σ such that $E=F\sigma$, and E is ground.

Let S be a logic program. A query on S is a conjunction of predications D1,..,Dm, m>=0. Let the variables in this query be x1,..,xn, n>=0. The central problem in logic programming is to determine whether there exists a substitution θ associating terms with x1,..,xn, such that every ground instance of (D1,..,Dm)θ is a logical consequence of S, and furthermore, to determine the most general such substitution. The values of x1,..,xn can thus be "computed".

An attempt is made to solve this problem using the *SLD-resolution* proof procedure. This has been shown to be sound and complete for Horn clauses [Hill 1974], [Apt & van Emden 1982]. Given a query D1,..,Di-1,Di,Di+1,..,Dm, m>=0, and a clause A<-B1,..,Bn, n>=0, where A and Di unify with most general unifier θ, the SLD-proof procedure derives the new query:

(D1,..,Di-1,B1,..,Bn,Di+1,..,Dm)θ

An SLD-derivation consists of a sequence of queries Q0,Q1,Q2,.., and a sequence of substitutions θ1,θ2,.., such that Qi+1 is derived from Qi, and θi is the associated most general unifier. If for some i, Qi is empty, the SLD-derivation is called successful, and the composition of substitutions θ1,θ2,..,θi is determined as one answer to Q0.

To see an example of a logic program, consider first an algorithm for appending two lists: to append lists x and y, if x is empty, output y, otherwise output the result of attaching the head of x to the result of appending the tail of x to y. A logic program such that when the SLD-resolution proof procedure interprets it, it simulates the execution of this algorithm, is:

```
append([],X,X).
```

append([U|V],W,[U|Z])<-append(V,W,Z).

Here [] represents the empty list while | represents the list constructor function. [A1,A2,..,Am] is an abbreviation for [A1|[A2|..|[Am|[]..]. Now if we query whether there exists a B such that append([1,2],[3],B) the SLD-resolution procedure answers B=[1,2,3].

Note also that the above clauses are also true statements about the append relation, and there are other interesting consequences of these statements. For example, if we query whether there exist A and B such that append(A,B,[1,2]), we get the following pairs of answers: A=[],B=[1,2], A=[1],B=[2], A=[1,2],B=[]. We obtain these answers due to the completeness property of SLD-resolution. Such use of the same logic program in more than one way is one of the most powerful features of logic programming. Note that these extra answers cannot be obtained by the original algorithm.

The programming language *Prolog* [Warren et al. 1977] is an approximate implementation of logic programming. For example, it can sometimes fail to find an answer even though the SLD-resolution procedure would find it. Sometimes it can even compute a wrong answer. However, for many practical purposes Prolog can be regarded as an exact implementation of logic programming.

There is already an impressive number of applications of Prolog [Warren & van Caneghem 1986], in areas such as databases [Gallaire & Minker 1978], natural language analysis [Pereira & Warren 1980], expert systems [Narain 1986], [Clark & McCabe 1982], symbolic algebra [Bundy & Welham 1981], and circuit analysis

[Barrow 1983].

3.0 REWRITING

The idea of reduction or simplification is an old and useful one. Objects are reduced, rewritten, or simplified using some set of rules to other objects. A set of such rules is called a rewrite rule system. Examples of rewrite rule systems include formal grammars [Hopcroft & Ullman 1979], combinatory logic [Curry & Feys 1958], rules for minimization of combinational logic expressions [Kohavi 1978], rules for converting sentences of first order logic into clausal form [Kowalski 1979], the lambda calculus [Church 1941], Lisp [McCarthy 1960], [Henderson 1980], SASL [Turner 1979], HOPE [Burstall et al. 1980].

We now formally define rewrite rule systems and discuss some major issues which arise in them. We restrict attention to first order rewrite rule systems. These are sufficient for representing all computable functions. A first order rewrite rule system is a collection of *rewrite rules*, each of the form:

$$A \Rightarrow B$$

where each of A and B are terms. A term is either a variable, or of the form f(t1,..,tn), n>=0, where f is an n-ary function symbol and each ti is a term.

A term E is said to *reduce* to a term F, (in symbols E->F), if there is a subterm G in E, a rule A=>B, and a substitution θ such that $G=A\theta$, and F is the result of replacing G in E by $B\theta$. The step of reducing E to F is called a reduction step.

A term E is said to *narrow* to a term F if there is a non-variable subterm G in E, a rule A=>B, and a most general unifier θ of G and A, such that F is obtained by applying θ to the result of replacing G in E by B. Thus, in reduction only variables of A can be bound whereas in narrowing variables of both A and E can be bound.

Given a term E0, a *reduction* is a, possibly infinite, sequence E0,E1,.. such that for all i, whenever Ei and Ei+1 both exist, Ei->Ei+1. -*> is defined to be the reflexive-transitive closure of ->. Given terms E0 and En, if E0-*>En and there is no term F such that En->F then En is called a *normal form* of E0. If for all terms M,N,P, M-*>N and M-*>P, implies there exists a term Q such that N-*>Q and P-*>Q, then the rewrite rule system is called *confluent*. An important consequence of confluence is that every term has at most one normal form. Methods for checking whether rewrite rule systems are confluent, and if not, how to make them so, are studied in [Knuth & Bendix 1970] and [Huet 1980]. If no infinite reductions are possible, the set of rewrite rules is called *terminating*. A terminating, confluent system is called *canonical*.

Given a term, there can in general be many reductions starting with it, some of which end in normal forms while others are infinite. Precisely which one is generated is determined by a *reduction strategy*. A reduction strategy is a mapping which takes a term E as input, and returns a subterm G of E as output, such that there is some rule A=>B and substitution α, such that G=Aα. We can now replace G in E by Bα, and then repeat this step to obtain a single reduction starting at E. This reduction is said to be computed by the reduction strategy.

The choice of a reduction strategy has an important bearing upon two issues, *reduction-completeness*, and *efficiency*. A strategy, or the associated rewrite rule system, is reduction-complete if for each term, each of its normal forms can be computed exclusively by use of this strategy. A reduction strategy R1 is more efficient than another one R2, if normal forms can always be computed in a smaller number of steps using R1 than using R2. For example, it is well known that for the lambda calculus, the *normal-order*, or leftmost strategy is reduction-complete, while the *applicative-order* strategy is not, even though it is usually more efficient. However, the normal-order strategy is not always sufficient to guarantee reduction-completeness, as the following example shows.

```
f(X)=>f(X).
f([])=>[].
a=>[].
```

Even though f(a) has [] as normal form, the only normal-order reduction starting at it is f(a),f(a),..... Ironically, there is an innermost, terminating reduction f(a),f([]),[]. Rewrite rule systems are useful in at least three ways and these are now discussed.

3.1 Rewriting and solution of identities

Equality, like partial ordering, is a useful relation. A set of statements each of the form A=B, where A and B are terms is called an *equality theory*. Examples of equality theories are identities in trigonometry, in the differential and integral calculus, or in polynomial arithmetic. In contrast, *equality axioms* comprise a fixed set containing the following statements:

```
X=X.
X=Y<-Y=X.
X=Y<-X=Z,Z=Y
```

and for each n-ary function symbol f, and integer i, the statement:

```
f(X1,..,Xi,..,Xn)=f(X1,..,Yi,..,Xn)<-Xi=Yi.
```

and for each n-ary predicate symbol p, and integer i, the statement:

```
p(X1,..,Xi,..,Xn)<-Xi=Yi,p(X1,..,Yi,..,Xn).
```

The last two axioms are called substitutivity axioms and express the very important rule that equals can be substituted for equals in expressions without changing their values. Examples of use of this rule can be found in the simplest of mathematical derivations, e.g. in showing that log(a*b)=log(a)+log(b).

Given an equality theory T, the equality axioms E, and terms A and B, an important problem which arises is whether A=B is a logical consequence of $T \cup E$. This problem is also called the *word problem*. Here A=B is interpreted as an *identity* so that all its variables are universally quantified. For example, where T consists of elementary trigonometric identities, the problem may be to determine whether sin(2*x)=2*sin(x)*cos(x) is an identity.

In principle, this problem can be tackled simply by submitting T, E and A=B to some complete proof procedure such as SLD-resolution. However, the resulting search space is prohibitive [Robinson & Wos 1969], [van Emden & Yukawa 1986]. For example, for the simplest of problems, there would be an infinite branch in the search space due to the symmetricity axiom.

Rewrite rule systems offer a more computationally feasible approach to solving this problem. Let T* be the rewrite rule system obtained by converting = to => in each rule L=R in T. Now, if in the context of T*, C-*>D, it is easily verified that C=D is a logical consequence of T ∪ E. Now, *if every term has at most one normal form*, a sufficient condition for checking equality can be obtained. To check whether A and B are equal, obtain the normal forms of A and B, and check whether these are syntactically identical.

This approach is computationally feasible for three reasons. First, there is no infinite branch corresponding to the symmetricity axiom. Second, if T* is terminating, then any reduction strategy will compute the normal form of A and B. Third, one can reduce A and B independently of each other. Equality can also be checked by reducing both A and B to the same term not necessarily in normal form. However, then, A cannot be reduced independently of B, since the term to which A must be reduced depends upon the term to which B must be reduced, and vice versa.

Note that in general, this condition for equality is only sufficient, not necessary. It is entirely possible that A=B be a logical consequence of T U E, but that A or B not have normal forms. For example, where T is {int(N)=[N|int(s(N))]}, int(0)=[0|int(s(0))] is a logical consequence of T ∪ E, however, the normal form of int(0) will never be obtained.

However, if T* is also terminating, the condition for equality is also necessary [Knuth & Bendix 1970]. This is a major reason for the importance of confluent, terminating, or canonical theories.

3.2 Rewriting and functional programming

A rewrite rule system T can be thought of as a functional programming system, provided every term in T has at most one normal form, => is interpreted as equality, operators as (partial) functions, and ground terms as objects in the domain and range of these functions. The reading of a rule A=>B is now that for all values of variables in the rule, the value of A is the same as the value of B.

A rule f(t1,..,tn)=>B can be thought of as part of definition of function denoted by f. A reduction step can be thought of as a step in which equals are substituted for equals. The normal form of a term f(t1,..,tn), if it exists, is unique, and can be taken to represent the value of function denoted by f for arguments denoted by t1,..,tn. Thus, reduction can be thought of as computation. For example, the rules:

```
append([],X)=>X.
append([U|V],W)=>[U|append(V,W)].
```

can be thought of as defining the list concatenation function. The value of append([1],[2]) is taken to be the value of its normal form [1,2]. Suitable versions of the lambda calculus [Church 1941], combinatory logic [Turner 1979], or Lisp [Henderson 1980], are rewrite rule systems used for functional programming.

3.3 Rewriting and computation with infinite structures

Computation with infinite structures, such as power series, streams, or real numbers, can be interpreted more elegantly in the context of rewriting than in that of logic programming. For example, suppose we wish to determine the first n elements of sequences, where these sequences can be infinite, such as an infinite list of 1s. In Prolog we could write:

```
first(0,X,[]).
first(s(X),[U|V],[U|Z]):-first(X,V,Z).
p([1|Z]):-p(Z).
```

Now, if we wanted to compute the first element of the list computed by p, we would type:

```
p(X),first(s(0),X,Z).
```

We could arrange, as in Parlog [Clark & Gregory 1986] or Concurrent Prolog [Shapiro 1983] that p and first coroutine, i.e. whenever a new element is generated by p, control transfers to first. Then, even though p computes an infinite list, first would still terminate with Z bound to [1]. However, since p would never terminate, no theorem would ever be proved, so *within the framework of SLD-resolution we would not be entitled to infer anything.*

In a rewrite rule system, however, we would express first and p as follows:

first(0,X)=>[].

first(s(X),[U|V])=>[U|first(X,V)].

p=>[1|p].

Now, the normal form of first(s(0),p) is [1], *well within the framework of reduction.*

4.0 LAZY EVALUATION

Consider two situations. First, suppose we wish to determine whether two sequences A and B are identical, i.e. whether for every i, A[i]=B[i], where X[i] represents the ith element of sequence X. We can generate A completely, then generate B completely, and then compare their elements from left to right. However, A and B may be very long but may differ at, say, the third position. The effort of generating A and B beyond the third position would then be wasted.

Second, suppose the sequences A and B above are infinite, but differ, as before, at the third position. An attempt to generate A or B completely would never terminate, and so we would never know that A and B differ.

To deal with such situations, the idea of lazy evaluation has been developed. It is a method of computing which ensures, roughly, that a computation step is performed only when there is need to perform it. Thus, in both situations above, lazy evaluation would generate the ith elements of A and B only when it was known that for all j, j<i, A[j]=B[j]. In the first case the answer would be produced more efficiently. In the second case the answer would be computed in finite time, without computation getting trapped in an infinite loop.

Thus, lazy evaluation has two advantages. *First, it allows certain computations to terminate more quickly. Second, it allows computation with infinite structures.* These arise in areas such as exact real arithmetic, real analysis, graphics, or networks of communicating processes [Kahn & MacQueen 1977]. The precise interpretation of lazy evaluation depends, of course, on the formalism in which we are programming. This thesis provides such an interpretation in the context of the rewrite rule system F* which it develops.

4.1 Lazy evaluation in functional languages

Many implementations of lazy evaluation have been proposed for purely functional languages. Only four of the more well known ones are outlined. Friedman & Wise [1976] modify Lisp by making cons non-strict in both its arguments. A function is non-strict in its ith argument if it can return a value without evaluating its ith argument. Thus, the expression (car (intfrom 1)), in the presence of the definition (intfrom n)=(cons n (intfrom (plus 1 n))), would evaluate to 1 instead of leading to a non-terminating recursion.

However, efficient implementation of non-strict cons is substantially more difficult than that of strict cons. Moreover, often multiple copies of unevaluated expressions are created. To avoid redundant computation, one has to ensure that whenever one copy is evaluated, all copies of it are simultaneously evaluated. This further complicates the implementation.

A similar idea is proposed in Henderson [1980]. He wraps *delay*, and *force* operators around certain Lisp expressions, for example, before arguments to cons. These postpone or activate evaluation of expressions. This idea is also difficult to efficiently implement for much the same reasons as non-strict cons is.

O'Donnell [1985] describes a non-terminating rewrite rule system, intended for doing functional programming. The system is reduction-complete, so it exhibits a weak form of laziness. It is claimed in a later paper by O'Donnell that an implementation of this system is as efficient as Franz Lisp.

A quite interesting approach to lazy evaluation is based upon a theorem of Curry and Feys [1958] which states that normal-order reduction of lambda terms yields their normal forms whenever they exist. This approach has been realized by Turner in his SASL language [Turner 1979]. He compiles SASL programs into expressions in his combinatory logic, and reduces them using normal-order graph reduction. In graph reduction it is easy to ensure that when an expression is reduced all copies of it are simultaneously reduced. Thus, this approach appears both elegant as well as practical.

Vuillemin [1974], Berry & Levy [1979] show that provided whenever a term is reduced, all copies of it are simultaneously reduced, a call-by-name reduction strategy is optimal. In other words, it computes normal forms of terms in a minimum number of steps. Thus, this scheme exhibits a strong form of laziness. However, this result is derived only for rewrite rules whose left hand sides are of the form f(X1,..,Xm), each Xi a variable. Thus, the existence of a finite number of primitive functions, such as if-then-else, must be assumed. These cannot be defined using such rules alone.

4.2 Lazy evaluation in logic programming

One of the first proposals for achieving lazy evaluation in logic languages was made in IC-Prolog [Clark 1980]. The behavior of the IC-Prolog interpreter could be controlled by annotating variables in the logic program. For example, it could be made to suspend proving a predication if its arguments were not sufficiently instantiated. A similar idea is found in Concurrent Prolog [Shapiro 1983] and Parlog [Clark & Gregory 1986]. As with non-strict cons, these extended Prologs are difficult to efficiently implement on sequential machines. However, some efficient implementations on parallel machines seem to be under way.

In keeping with the philosophy of logic programming--to write clauses in such a way that SLD-resolution behaves as intended--an efficient technique for doing lazy evaluation in Prolog was presented in [Narain 1986]. In contrast to previous approaches for realizing lazy evaluation, this technique does *not* require any change to the Prolog interpreter. Instead, *pure* logic programs are written in such a way that their interpretation, using the Prolog interpreter directly yields lazy evaluation.

However, the analysis of this technique relies upon certain questionable semantical ideas such as the equality of infinite lists. Also, the conditions that programs must satisfy in order to behave lazily can be sometimes difficult to verify. This thesis is partly motivated by a desire to obtain a purely syntactic understanding of this technique.

5.0 COMBINING LOGIC PROGRAMMING AND REWRITING

5.1 Logic programming in rewriting

Several attempts have been made to implement the SLD-resolution procedure in a rewrite rule system such as Lisp, e.g. LOGLISP [Robinson & Sibert 1982] or QLOG [Komorowski 1982]. In these, terms which occur as arguments to predications can be Lisp expressions. Clauses can contain calls to Lisp predicates. The result of a deduction is a Lisp data object subject to arbitrary manipulation by Lisp procedures.

It appears difficult for such an approach to lead to an efficient system. SLD-resolution can compute with unbound variables in expressions, but Lisp cannot. So, logic programs are different enough from Lisp programs that they cannot be interpreted using the very efficient Lisp interpreters available today. Neither can they be compiled using Lisp compilers. Thus, a separate interpreter in Lisp is developed, but since it is itself interpreted by Lisp, deductions are not very efficient.

5.2 Equality in logic programming

A rewrite rule system can be interpreted as an equality theory by interpreting => as =. Equality theories and axioms can be expressed as Horn clauses. To reason with them, one can simply add them to a logic programming system such as Prolog, and so obtain more than a combination of rewriting and logic programming. However, as discussed in section 3.1, this approach leads to a computationally infeasible search space.

To alleviate this large search space, Robinson and Wos [1969] proposed *paramodulation*, a rule of inference which simulates all the equality axioms, except reflexivity. *These axioms then need not be included with the equality theory*, so a reduced search space results. For example, from Q(a) and a=b one can infer in a single step, via paramodulation, Q(b). With the full equality theory, the inference would be longer.

Paramodulation is sound and complete for the clausal form of first order logic. However, it only seems to hide details of applications of equality axioms, not eliminate their use, so the search space is still quite large. For example, there is still an infinite branch due to symmetricity.

An important, recent approach to implementing equality in logic programming has been suggested by van Emden and Yukawa [1986]. They present alternative equality axioms, which logically imply the original axioms, but which, for purpose of solving identities, have much better computational properties. These axioms also form an executable Prolog program. However, this approach is restricted to *terminating* theories, which cannot be used to represent infinite structures, or all computable functions.

Tamaki [1984] has also shown how to compile equality theories into Horn clauses with a smaller search space. However, as he himself points out, these clauses can still be seriously inefficient, particularly, when manipulating representations of infinite structures.

5.3 Extended unification

Like paramodulation, this is another approach for excluding equality axioms, yet obtaining their logical consequences. Let T be an equality theory interpreted as a rewrite rule system. Given terms t1 and t2 one tries to algorithmically determine a substitution θ such that $t1\theta=t2\theta$, as an identity, is a logical consequence of T and the equality axioms. Such a substitution is called a *T-unifier* of t1 and t2. Conventional unification is T-unification with T={ }.

T-unification is the dual of the word problem. Given a theory T and terms t1,t2, T-unification is determining whether t1=t2, treated as an equation, is a logical consequence of T and the equality axioms. In particular, all variables in t1=t2 are existentially quantified.

A substitution σ is said to be an *instance* of substitution τ if there is a substitution δ such that for every variable x, $x\sigma=x\tau\delta$ as an identity, is a logical consequence of T and the equality axioms [Fay 1979]. A set S of T-unifiers of t1 and t2 is said to be *complete*, if for any T-unifier σ of t1 and t2, there exists a unifier τ in S such that σ is an instance of τ. A set of T-unifiers of t1 and t2 is said to be *independent* if no two members of S are instances of each other. A set of T-unifiers of t1 and t2 is said to be *maximally general* if it is complete, and independent.

One can generalize resolution to use T-unifiers. For example, given the query P(t1) and the clause P(t2)<-Q one first T-unifies t1 and t2 to obtain σ, and then infers the query $Q\sigma$. *Thus T-unification steps do not appear as part of the deduction,* and so its length is considerably shorter, than it would be had the equality axioms been included.

Many T-unification algorithms have been proposed. Those of Fay [1979] and Hullot [1980] are based upon narrowing, and are shown to yield complete sets of T-unifiers when the equality theories, interpreted as rewrite rules, are confluent and terminating. Kornfeld [1983], and Subrahmanyam & You [1984] provide many interesting examples, but no rigorous analysis of their algorithms. Goguen and Meseguer [1986] use Fay's or Hullot's algorithm for solving equations in their Eqlog language. The algorithm of Miller & Nadathur [1986] is for unifying terms in the typed lambda calculus and is based upon one by Huet [1975].

In contrast to conventional unification, the maximally general set of unifiers of two terms need not be a singleton. In fact it can be infinite. For example, with the theory defining multiplication of natural numbers, the maximally general set of unifiers of X*X=Z*W is infinite [Goguen & Meseguer 1986]. Moreover, T-unification can require searching through an infinite space even with very simple theories. For example, when T={0+x=x,s(x)+y=s(x+y)}, and u is a variable, T-unification of u+u and s(s(0)) via narrowing generates an infinite branch. *The usefulness of a computational model whose innermost loop involves such search is highly questionable.*

5.4 Completion procedure as interpreter

Dershowitz & Josephson [1984] show how to interpret rewrite rules using a linear version of the completion procedure [Knuth & Bendix 1970], to obtain the effect of logic programming. For example, they define the append function as follows:

append(X.U,V)=X.W -> append(U,V)=W

append(nil,V)=V -> true

append(V,nil)=V -> true

An example of a query is append(A,B)=1.2.nil->ans(t(A,B)), which is added to the above set. The goal is to complete this set in the sense of [Knuth & Bendix 1970] till the rule ans(t(A,B)))->true is derived. A and B are then read off as answers. They also show how to transform logic programs into rewrite rules. This approach is interesting to the extent of showing that the completion procedure can be regarded as an interpreter. However, the completion procedure does not specify any reduction strategy, so computationally, this approach does not seem to have much advantage over narrowing.

5.5 Logic programming with sets

Robinson [1987] and Darlington et al. [1986] propose unifying functional and logic programming by means of sets. They observe that a logic program computes a relation, which can be thought of as a set, namely the set of those tuples which are in the relation. Hence if a rewrite rule system could have a facility for defining and computing sets, one could obtain the power of logic programming. This idea is embodied in the SUPER language of Robinson's group. Darlington's group has extended the HOPE language with it. This approach appears to be quite promising, and it remains to be seen how practical SUPER and extended HOPE will be.

5.6 Narrowing

Reddy [1985], proposes interpreting rewrite rules using narrowing. In narrowing input variables can be bound. This feature, which mainly distinguishes logic programming from reduction can thus be achieved with rewrite rules. So, narrowing can be used as basis for subsuming both rewriting and logic programming. For example, with:

append([],Y)=Y.

append([A|X],Y)=[A|append(X,Y)].

the expression append(A,[3])=[1,2,3], A a variable, cannot be reduced at all. But it can be narrowed to append([1,2],[3])= [1,2,3], to yield the substitution A=[1,2]. However, this scheme seems to pose computational problems as serious as with semantic unification, or paramodulation.

5.7 Residuation

Ait-Kaci & Nasr [1987] allow the possibility that arguments of predicate symbols be evaluable. In this, their scheme resembles LOGLISP [Robinson & Sibert 1982]. It differs from LOGLISP in that unification of terms is suspended till all their evaluable subterms become ground. These subterms are then evaluated and unification resumed. A residuation is simply a suspended unification. However, they do not discuss lazy evaluation.

5.8 Miscellaneous

Fribourg [1984] proposes oriented equational clauses, i.e. Horn clauses with a rewrite rule as the head part, and a list of equations as the body part. He also proposes an interpreter based upon the rule of clausal superposition, or substitution of equals for equals, and the derivation of resolvents. He shows how this language can be used to do both rewriting, and conventional logic programming. However, he does not discuss a reduction strategy, or lazy evaluation.

Yamamoto [1987] extends SLD-resolution with narrowing for treating equational theories. However, the completeness theorem is only proved for confluent, Noetherian theories. Moreover, no reduction strategy is discussed.

Barbuti, et al. [1986] extend SLD-resolution with their atom elimination rule to handle rewriting in a demand-driven fashion. However, they do not provide much rigorous discussion of the computational properties of this rule.

CHAPTER III

A REWRITE RULE SYSTEM F*

1.0 INTRODUCTION

A first order, non-deterministic, non-terminating rewrite rule system F*, and a lazy reduction strategy for it, select, are defined. The emphasis in F* is on computing simplified forms, instead of normal forms. Thus, certain termination problems faced by previous approaches are avoided.

The main result proved is that F* is reduction-complete, in that select reduces ground terms to their simplified, or normal forms, whenever possible. Reduction-completeness yields a weak form of laziness. A term may denote an infinite object, and so fail to have a finite normal form. However, if it has a finite simplified form, it is obtained in finite time. By repeatedly simplifying subterms of this simplified form, the structure of the infinite object can be revealed to any arbitrary depth.

2.0 DEFINITION OF F*

Variables. There is a countably infinite list of variables.

Function symbols. There is a countably infinite list of 0-ary function symbols. In particular, [], 0, true, false, are 0-ary function symbols. There is a countably infinite list of 1-ary function symbols. In particular, s is a 1-ary function symbol. There is a countably infinite list of 2-ary function symbols. In particular, | is a 2-ary function symbol. And so on, for all other arities.

Connectives. The connectives are =>, (,), ','.

Constructor Symbols. There is an infinite subset of the function symbols called Constructors. Each element of Constructors is called a constructor symbol. For each n, n>=0, Constructors contains an infinite number of n-ary function symbols. In particular, 0, true, false, [] and | are constructor symbols. It is intended that data be represented by combinations of only constructor symbols.

Terms. A term is either a variable, or an expression of the form f(t1,..tn) where f is an n-ary function symbol, n>=0, and each ti is a term. A term is called ground if it contains no variables. _It is the intention in F* to reduce only ground terms,_ and most of the propositions below are about these. Non-ground terms such as left hand sides of reduction rules do arise, but in very few propositions. **Hence, unless explicitly stated otherwise, by a term is meant a ground term.**

Subterms. Let E be a term. Then E is said to be a subterm of itself. Also, if E=f(t1,..,tn), n>0, then X is said to be a subterm of E, if X is a subterm of some ti. Let X be a subterm of E. Then X is said to occur in E. Also, if X≠E, then X is said to be a proper subterm of E, or be properly contained in E. Two subterms A and B of E are said to overlap, if A is properly contained within B.

Substitutions. A substitution is a, possibly empty, set {<X1,t1>,..,<Xn,tn>} where the X1,..,Xn are distinct variables, and each ti is a term, possibly containing variables. A variable X is defined in a substitution σ iff for some possibly non-ground term s, <X,s> occurs in σ. In this thesis, we will be concerned almost exclusively with substitutions in which for each pair <X,s>, s is a ground term.

Applying substitutions to terms. Let $\sigma=\{<X1,t1>,..,<Xn,tn>\}$ be a substitution and E be a term, possibly containing variables. The result of applying σ to E, $E\sigma$, is the result of replacing, for each i, every occurrence of Xi in E by ti.

Matching. A ground term E is said to **match** a possibly non-ground term F, with substitution α, if $E=F\alpha$.

Unification. Two terms, E and F, possibly containing variables, are said to unify with substitution σ if $E\sigma=F\sigma$. Note that matching is a special case of unification.

Reduction Rules. A reduction rule is of the form:

LHS=>RHS

where LHS and RHS are terms, possibly containing variables. LHS is called the head of the rule. The following restrictions are placed on LHS and RHS:

(a) LHS is not a variable.

(b) LHS is not of the form c(t1,..,tn) where c is a constructor symbol.

(c) If LHS=f(t1,t2,..,tn), then each ti is a variable, or a term of the form c(X1,..,Xm) where c is an m-ary constructor symbol, and each Xi a variable.

(d) There is at most one occurrence of any variable in LHS.

(e) All variables of RHS appear in LHS.

These restrictions are very reasonable, and as examples throughout the thesis show, very expressive programs can be written adhering to these. *Note that F* is more expressive than first order Lisp, as the latter does not admit patterns in left hand sides of function definitions.*

Restriction (a) is to enable functional programs to be written in F*.

Restriction (b) ensures that a term of the form c(t1,..,tn), c a constructor symbol, cannot be reduced as a whole. This yields a simple halting condition for the basic simplification process. If further simplification is required, the process may be called recursively.

Restriction (c) limits heads of rules to be of depth at most two, and so greatly simplifies analysis. However, no generality is lost, since rules with heads of arbitrary depth can easily be expressed in terms of rules with heads of depth at most two. For example, the rule:

```
fib(s(s(X)))=>plus(fib(X),fib(s(X)))
```

can be expressed as:

```
fib(s(A))=>g(A)
g(s(X))=>plus(fib(X),fib(s(X)).
```

Restriction (d) is the linearity assumption. It ensures that to match a ground term f(t1,..,tn) with the left hand side of a rule f(L1,..,Ln), it is sufficient to match, for each i, ti with Li.

Restriction (e) ensures that a ground term is never reduced to a non-ground term. Again, this is necessary if F* is to be used for functional programming.

F* programs. An F* program is a finite set of reduction rules. Where t is a binary constructor symbol, some examples of F* programs are:

```
quicksort([])=>[].
quicksort([A|B])=>quicksort1(A,partition(A,B,[],[])).
quicksort1(A,t(L,R))=>append(quicksort(L),[A|quicksort(R)]).

partition(U,[],L,R)=>t(L,R).
partition(U,[A|B],L,R)=>
        if(lesseq(A,U),partition(U,B,[A|L],R),partition(U,B,L,[A|R])).

append([],X)=>X
append([U|V],W)=>[U|append(V,W)]

if(true,X,Y)=>X.
if(false,X,Y)=>Y.

lesseq(0,X)=>true.
lesseq(s(X),s(Y))=>lesseq(X,Y).
```

```
lesseq(s(X),0)=>false.


zero(X)=>0.
prim_rec_f(0,Y1,Y2,Y3)=>g(Y1,Y2,Y3).
prim_rec_f(s(X),Y1,Y2,Y3)=>h(prim_rec_f(X,Y1,Y2,Y3),X,Y1,Y2,Y3).
minim_p(X,K)=>if(equal(p(X),K),X,minim_p(s(X),K)).


equal(0,0)=>true.
equal(0,s(X))=>false.
equal(s(X),0)=>false.
equal(s(X),s(Y))=>equal(X,Y).


merge([A|B],[C|D])=>if(lesseq(A,C),[A|merge(B,[C|D])],[C|merge([A|B],D)]).


int(N)=>[N|int(s(N))].


greater(X,Y)=>not(lesseq(X,Y)).


not(true)=>false.
not(false)=>true.
```

We now consider the reduction of **terms**. Again, unless explicitly stated, by a term we mean a ground term.

E=>$_{\mathbf{P}}$E1. Let P be an F* program and E and E1 be terms. We say E=>$_{P}$E1 if there is a rule LHS=>RHS in P, and a substitution σ such that E=LHSσ, and E1=RHSσ. We also say that E reduces to E1 by the rule LHS=>RHS, or that the rule applies to the whole of E. The subscript on => is dropped, if clear from context.

F=E[G/H]. Where E,F,G,H, are terms, let F be the result of replacing an occurrence of G in E by H. Then we say F=E[G/H].

E->$_P$E1,E-*>$_P$E1. Let P be an F* program and E be a term. Let G be a subterm of E such that G=>$_P$H. Let E1 be the result of substituting H for G in E. Then we say that E->$_P$E1. Note that if E=>$_P$E1 then E matches the left hand side of some rule in P. If E->$_P$E1 then some subterm of E, including possibly E, matches the left hand side of some rule in P. We define -*>$_P$ to be the reflexive transitive closure of ->$_P$. Again, the subscript on -> or -*> is dropped, if clear from context.

Reductions. Let P be an F* program. A reduction in P is a, possibly infinite, sequence E1,E2,... such that for each i, when Ei and Ei+1 both exist, Ei->$_P$Ei+1.

Lengths of reductions. The length of a finite reduction E0,E1,..,En is n.

Simplified forms. A term is said to be in simplified form or simplified if it is of the form c(t1,..,tn) where c is an n-ary constructor symbol, n>=0, and each ti is a term. F is called a simplified form of E, if E-*>F and F is in simplified form.

Normal forms. A term is said to be in normal form if each function symbol in it is a constructor symbol. F is called a normal form of E if E-*>F and F is in normal form.

Successful reductions. Let P be an F* program. A successful reduction in P is a finite reduction E0,..,En, n>=0, in P, such that En is simplified.

R_P(G,H,A,B). Let P be an F* program. Where G,H,A,B are terms, R_P(G,H,A,B) if (a) G=>H, and (b) B is identical with A except that zero or more occurrences of G in A are *simultaneously* replaced by H. Note that A and G can be identical. Again, if P is clear from context we omit the subscript on R.

Reduction strategy. Let P be an F* program. A reduction strategy for P takes as input a term E and selects a subterm G of E such that there exists a term H such that G=>$_P$H.

A special reduction strategy. Let P be an F* program. We now define a reduction strategy, select$_P$ for P. Informally, given a term E it will select that subterm of E whose reduction is necessary in order that some => rule in P apply to the whole of E. Where f(T1,..,Tn) is a term, the relation select$_P$ is defined by the following pseudo-Horn clauses:

> select$_P$(f(T1,..,Tn),f(T1,..,Tn)) if f(T1,..,Tn)=>$_P$X.
> select$_P$(f(T1,..,Ti,..,Tn),X) if
>> there is a rule f(L1,..,Li,..,Ln)=>RHS in P, and
>> there is no substitution σ such that Ti=Liσ, and
>> select$_P$(Ti,X).

The second rule is a schema, so that an instance of it is assumed written for each each i, 1=<i=<n. Again, the subscript on select is dropped, if clear from context. Note the following:

(1) When $select_P$ takes as input E and returns G, it also, implicitly, computes a position, or occurrence of G in E. This occurrence can be obtained from the proof of $select_P$(E,G).

(2) If $select_P$(E,G), there is a term H such that $G{=}{>}_P H$.

(3) Select is non-deterministic, in that given term E, it is possible for it to select more than one subterm A1,..,Ak, k>0, within E. Also, it is possible that for some i,j, i≠j, Ai is a proper subterm of Aj.

(4) Since, by restriction (b) there is no rule in P of the form c(t1,..,tn)=>RHS, where c is a constructor symbol, if E is simplified, $select_P$ is undefined for E.

For example, where P is the set of reduction rules which appear above, and 1,2,... are abbreviations, respectively, for s(0),s(s(0)),.., we have the following:

select(merge(int(1),int(2)),int(1)).
select(merge(int(1),int(2)),int(2)).
select(merge([1,3],int(2)),int(2)).
select(merge([1,2],[3,4]),merge([1,2],[3,4])).
If E=[1|merge(int(1),int(2))] then select is undefined for E.
select(lesseq(0,zero(1)),lesseq(0,zero(1))).
select(lesseq(0,zero(1)),zero(1)).

Let E,G,H be terms. In the following, when we say that select(E,G), and G is to be replaced by H in E, we mean that the occurrence of G derived from the proof of select(E,G), is to be replaced by H.

N-step. Let P be an F* program and E,G,H be terms. Let $select_P(E,G)$, and $G =>_P H$. Let E1 be the result of replacing G by H in E. Then we say that E reduces to E1 in an N-step in P. The qualification "in P" is omitted when P is clear from context. The prefix N in N-step is intended to connote normal order.

N-reduction. Let P be an F* program. An N-reduction in P is a reduction E1,E2,.... in P such that for each i, when Ei and Ei+1 both exist, Ei reduces to Ei+1 in an N-step in P. In particular, the sequence E where E is a term, is an N-reduction in P. The qualification "in P" is omitted when P is clear from the context.

select-r. This reduction strategy repeatedly uses select to reduce terms. The suffix r stands for recursive, or repeated. Where P is an F* program:

$select\text{-}r_P(E,F)$ if $select_P(E,F)$.
$select\text{-}r_P(c(T1,..,Ti,..,Tm),F)$ if
c is a constructor symbol, and
$select\text{-}r_P(Ti,F)$.

Again, the second rule is a schema, so that an instance of it is assumed written for each i, 1=<i=<n. Thus, select-r is like select except that if a term is in simplified form, it recursively uses select on one of the arguments of the outermost constructor symbol. So, its repeated use can yield normal-forms of terms.

For example, with the usual rules for append, the query select([1|append([],[])],X) fails, whereas the query select-r([1|append([],[])],X) succeeds with X=append([],[]). The subscript on select-r is dropped, if clear from context.

NR-step. Let P be an F* program and E,G,H be terms. Suppose select-r_P(E,G) and G$=>_P$H. Let E1 be the result of replacing G by H in E. Then we say that E reduces to E1 in an NR-step in P. The qualification "in P" is omitted when clear from context.

NR-reduction. Let P be an F* program. An NR-reduction in P is a reduction E1,E2,.... in P such that for each i, when Ei and Ei+1 both exist, Ei reduces to Ei+1 in an NR-step in P. In particular, the sequence E where E is a term, is an NR-reduction in P.

NR-reductions are needed to compute normal-forms of terms. For example, the term append([1],[2]) has the only N-reduction append([1],[2]), [1|append([],[2])]. However, it has the NR-reduction append([1],[2]), [1|append([],[2])], [1,2]. The qualification "in P" is omitted when clear from context.

3.0 REDUCTION-COMPLETENESS OF F*

Lemma 1. Let P be an F* program. If A->B and B is simplified but A is not, then A=>B.

Proof. Since A is not simplified, A=f(t1,..,tn) where f is not a constructor symbol and each ti is a term. Since the reduction of A to B eliminates this symbol, it follows that A must reduce as a whole to B. Thus A=>B. **QED.**

Lemma 2. Let P be an F* program. Let X1,..,Xn be variables, G,H,t1,..,tn,t1*,..,tn* be terms such that for each i, R(G,H,ti,ti*). Let $\sigma=\{\langle X1,t1\rangle,..,\langle Xn,tn\rangle\}$ and $\tau=\{\langle X1,t1^*\rangle,..,\langle Xn,tn^*\rangle\}$ be substitutions. Let M be a term, possibly containing variables, but only from {X1,..,Xn}. Then $R(G,H,M\sigma,M\tau)$.

Proof. By induction on length of M. Since M is a term, possibly containing variables, it is either a variable, a 0-ary function symbol or of the form f(N1,..,Nk) where f is an n-ary function symbol and each Ni is a term, possibly containing variables.

If M is a variable Xi, then $M\sigma$=ti and $M\tau$=ti* and so clearly $R(G,H,M\sigma,M\tau)$. If M is a 0-ary function symbol then $M\sigma$=M and $M\tau$=M and obviously R(G,H,M,M). Let M=f(N1,..,Nk). Assume the lemma holds for N1,..,Nk, i.e., for all i, $R(G,H,Ni\sigma,Ni\tau)$. $f(N1,..,Nk)\sigma=f(N1\sigma,..,Nk\sigma)$. Similarly, $f(N1,..,Nk)\tau=f(N1\tau,..,Nk\tau)$, and hence $R(G,H,M\sigma,M\tau)$. **QED**.

Lemma 3. Let P be an F* program. If:

(1) G, H, E1=f(t1,..,tn) and F1=f(t1*,..,tn*) are terms, and

(2) R(G,H,ti,ti*) for every i in 1,..,n, and

(3) B=f(L1,..,Ln) is the head of some rule in P, and

(4) E1=$B\sigma$ for some substitution σ, which defines only the variables in B.

Then there exists a substitution τ such that:

(1) F1=$B\tau$, and

(2) σ and τ define exactly the same variables, and

(3) If pair <X,s> occurs in σ and <X,s*> occurs in τ then R(G,H,s,s*).

Proof. Since by restriction (d) a variable occurs at most once in B=f(L1,..,Ln), a term f(d1,..,dn) matches B iff for each i, di matches Li with substitution σi. So, f(d1,..,dn) matches B with the union of σ1,..,σn. Consider some Li in L1,..,Ln. By restriction (c) there are the following cases.

Case 1. Li is a variable. Then Li matches ti* with substitution τi={<Li,ti*>}. Also, the pair <Li,ti> appears in σ. By assumption, R(G,H,ti,ti*).

Case 2. Li=c(X1,..,Xm), m>=0, c a constructor symbol and each Xj a variable. Then since ti matches Li, ti=c(s1,..,sm) where each si is a term. Thus the pairs {<X1,s1>,..,<Xm,sm>} appear in σ.

If ti is identical with ti*, ti* also matches Li with substitution τi={<X1,s1>,..,<Xm,sm>}. Of course, for every i, R(G,H,si,si).

If ti is not identical with ti* then since R(G,H,ti,ti*), ti contains at least one occurrence of G and G=>H. Since ti=c(s1,..,sm), c a constructor symbol, by restriction (b) ti$\neq$G. Hence ti*=c(s1*,..,sm*) each si* a term and for every i R(G,H,si,si*). Hence ti* matches Li with substitution τi={<X1,s1*>,..,<Xm,sm*>}.

The same argument can be repeated for every other Li. Let τ be the union of the τi. Then B and F1 match with τ. Thus (1).

By definition of τ, τ defines only those variables which occur in B. Thus σ and τ

define exactly the same variables. Thus (2).

If some pair <X,d*> appears in τ, then, by the above discussion <X,d> appears in σ and R(G,H,d,d*). Thus (3). **QED.**

Lemma 4. Let P be an F* program. If:

(1) f(t1,..,ti,..,tn) is a term, and

(2) f(L1,..,Li-1,c(X1,..,Xm),Li+1,..,Ln)=>RHS is a rule in P, and

(3) ti=d1,d2,d3,..,dr, r>0, is an N-reduction.

Then, f(t1,..,ti-1,d1,ti+1,..,tn), f(t1,..,ti-1,d2,ti+1,..,tn), .., f(t1,..,ti-1,dr,ti+1,..,tn) is also an N-reduction.

Proof. Let Li=c(X1,..,Xm). Since f(L1,..,Li,..,Ln)=>RHS is a rule, by restriction (b), f is not a constructor symbol. If r=1 then, by definition of N-reduction, the lemma is obvious. So, assume r>1.

By definition of N-reduction, at most the last member of the sequence d1,d2,d3,..,dr can be in simplified form. Hence, since Li=c(X1,..,Xm), none of the di, 1=<i<r matches Li.

We now show that for all j, 1=<j<r, f(t1,..,ti-1,dj,ti+1,..,tn) reduces to f(t1,..,ti-1,dj+1,ti+1,..,tn) in an N-step. Since dj is not simplified, it does not match Li. Hence, by definition of select, for every X, select(f(t1,..,ti-1,dj,ti+1,..,tn),X) if select(dj,X).

Since dj reduces to dj+1 in an N-step there are terms pj and qj such that $select_P(dj,pj)$, pj=>qj and dj+1 is the result of replacing pj by qj in dj. Then f(t1,..,ti-1,dj,ti+1,..,tn) reduces to f(t1,..,ti-1,dj+1,ti+1,..,tn) in an N-step. Hence, f(t1,..,ti-1,d1,ti+1,..,tn), f(t1,..,ti-1,d2,ti+1,..,tn), .., f(t1,..,ti-1,dr,ti+1,..,tn) is an N-reduction. **QED.**

Theorem 1. Let P be an F* program. Let E1,F1,F2,G,H be terms such that

(1) R(G,H,E1,F1), and

(2) F1 reduces to F2 in an N-step

Then there is an N-reduction E1,..,E2 in P such that R(G,H,E2,F2).

Proof. It is helpful to draw the following diagram:

```
    N-step
F1------------>F2
|
| R(G,H,E1,F1)
|
E1-----------*>E2
    N-reduction
```

We have to show that R(G,H,E2,F2). We proceed by induction on length of E1. Suppose E1 is a 0-ary function symbol. If E1=F1 then E1,F2 is an N-reduction and R(G,H,F2,F2). If E1≠F1 then since R(G,H,E1,F1), E1=G and E1=>F1. Thus, there is an N-reduction E1,F1,F2 and R(G,H,F2,F2). In both cases, take E2=F2.

Otherwise, E1=f(t1,..,tn), n>0. Assume the theorem for every term whose length is less than that of f(t1,..,tn). If E1=F1 then E1,F2 is an N-reduction and R(G,H,F2,F2).

Otherwise E1≠F1. If E1=G then since R(G,H,E1,F1), E1=>F1. Thus, there is an N-reduction E1,F1,F2, and R(G,H,F2,F2). Again, in both cases, take E2=F2.

We now arrive at the interesting cases, with E1≠F1, but G≠E1. Hence F1=f(t1*,..,tn*) where for every i, R(G,H,ti,ti*). We now consider the following cases:

Case 1. F1=>F2. Then there is a rule f(L1,..,Ln)=>RHS in P, such that F1 matches f(L1,..,Ln) with substitition τ, and F2=RHSτ.

Case 1-1. E1 matches f(L1,..,Ln) with substitution σ. By Lemma 3, there exists substitution β such that F1=f(L1,..,Ln)β. Since F1=f(L1,..,Ln)τ, $\tau=\beta$.

E1=>RHSσ, so let E2=RHSσ. The N-reduction is E1,E2. Of course F2=RHSτ. By Lemma 3, σ and τ define exactly the same variables, and if <X,s> occurs in σ and <X,s*> appears in τ then R(G,H,s,s*). Hence, by Lemma 2, R(G,H,E2,F2).

Case 1-2. E1 does not match f(L1,..,Ln). Then, since E1 is ground and each variable occurs at most once in f(L1,..,Ln), there is some Li in L1,..,Ln, and some ti in t1,..,tn, such that ti does not match Li. Hence Li is not a variable, so Li=c(X1,..,Xm), c a constuctor symbol and each Xi a variable.

Moreover, since R(G,H,ti,ti*), and ti does not match Li, by restriction (c), ti is not simplified. Since F1 matches f(L1,..,Ln), ti* matches Li, and so ti* is simplified. Since R(G,H,ti,ti*), ti=>ti*. Thus select(E1,ti). Hence f(t1,..,ti,..,tn) reduces to f(t1,..,ti*,..,tn) in an N-step.

Hence there exists an N-reduction E1=P1,P2,P3,... such that for each i, Pi=f(s1,..,sn), and for each sk in s1,..,sn, sk=tk or sk=tk*. Moreover, Pi+1 is derived from Pi by selecting some sk in s1,..,sn such that sk does not match Lk in L1,..,Ln, and replacing sk, in Pi, by tk*. We also have for each i, R(G,H,Pi,F1). Since n is finite, this reduction cannot be infinite and must end in Pm such that Pm matches f(L1,..,Ln) with substitution σ. Then Pm=>RHSσ. Hence we have the N-reduction E1,P2,P3,..,Pm,RHSσ. Take E2=RHSσ. By Lemma 3, F1 and f(L1,..,Ln) match with some substitution, and clearly this is τ. Already, F2=RHSτ. By Lemma 2, R(G,H,E2,F2).

Case 2. Not F1=>F2. We are given that F1 reduces to F2 by an N-step. We now have to show that there is an N-reduction E1,..,E2 such that R(G,H,E2,F2).

Suppose select(F1,u). Then u occurs in some ti*. That is, there is some ti* in t1*,..,tn*, such that select(ti*,u). Let u=>v and let ti** be the result of replacing u in ti* by v. Hence ti* reduces to ti** in an N-step, and also F2=f(t1*,..,ti**,..,tn*). By definition of select, there is a rule f(L1,..,Li,..,Ln)=>RHS in P such that ti* does not match Li. Hence Li=c(X1,..,Xm), m>=0, where c is a constructor symbol and each Xi is a variable.

Clearly, ti* is not simplified. So, by restriction (b) ti is also not simplified. ti* reduces to ti** in an N-step. We already have R(G,H,ti,ti*). Since the length of ti is less than that of f(t1,..,ti,..,tn), by induction hypothesis there is an N-reduction ti=d1,d2,..,dr, r>=1, such that R(G,H,dr,ti**). By Lemma 4, the sequence f(t1,..,ti-1,ti,ti+1,..,tn), f(t1,..,ti-1,d2,ti+1,..,tn),.., f(t1,..,ti-1,dr,ti+1..,tn) is an N-reduction. Take E2=f(t1,..,ti-1,dr,ti+1..,tn). We already have F2=f(t1*,..,ti**,..,tn*) and for each k, R(G,H,tk,tk*). Hence R(G,H,E2,F2). **QED.**

Lemma 5. Let P be an F* program. Let R(G,H,E0,F0) and F0,F1,..,Fn be an N-reduction. Then there is an N-reduction E0,..,E1,..,En such that R(G,H,En,Fn).

Proof. By induction on length n of F0,F1,..,Fn. If n=0 then clear. Otherwise assume lemma for the N-reduction F1,..,Fn. Since F0 reduces to F1 in an N-step and R(G,H,E0,F0), by Theorem 1, there exists an N-reduction E0,..,E1 such that R(G,H,E1,F1). By induction hypothesis, there exists an N-reduction E1,..,En, such that R(G,H,En,Fn). Hence there exists the N-reduction E0,..,E1,..,En such that R(G,H,En,Fn). **QED**.

Theorem 2. Reduction-completeness of F* for simplified forms. Let P be an F* program and D0 a term. Let D0,D1,..,Dn, n>=0, be a successful reduction in P. Then there is a successful N-reduction D0,E1,..,Em in P, such that Em-*>Dn.

Proof. By induction on length n of D0,D1,..,Dn. If n=0, D0 is already simplified, so D0 is a successful N-reduction, and D0-*>D0.

Let n>0 and assume Theorem for D1,..,Dn. Then there is a successful N-reduction D1,F2,..,Fp such that Fp-*>Dn. The situation can be laid out as follows:

```
Dn
:
:
D2
|
D1->F2-*>Fp
|
|        R(G,H,Em,Fp),  Fp-*>Dn
|
D0->E1-*>Em
```

Since D0->D1 there are terms G,H, such that G=>H and D1=D0[G/H]. Hence R(G,H,D0,D1). Since D1,F2,..,Fp is a successful N-reduction, by Lemma 5, there is an N-reduction D0,E1,..,Eq such that R(G,H,Eq,Fp). If Eq is simplified, take Em=Eq. Now D0,E1,..,Em is a successful N-reduction. Since R(G,H,Em,Fp), and Fp-*>Dn, Em-*>Dn, as required.

If Eq is not simplified, then since R(G,H,Eq,Fp), and Fp is simplified, Eq=>Fp. Now take Em=Fp, so D0,E1,..,Eq,Em is a successful N-reduction, and Em-*>Dn, as required. **QED.**

Theorem 3. Let P be an F* program. Let E1,F1,F2,G,H be terms such that

(1) R(G,H,E1,F1), and
(2) F1 reduces to F2 in an NR-step

Then there is an NR-reduction E1,..,E2 in P such that R(G,H,E2,F2).

Proof. By induction on length of E1. Let E1 be a 0-ary function symbol. If E1=F1 then clear. If E1≠F1, then E1=G, and so, clear. Otherwise, let E1=f(t1,..,tn), n>0.

Assume theorem for t1,..,tn.

Case 1. E1 is unsimplified. If F1 is simplified, then since R(G,H,E1,F1), E1=G, so the theorem is clear. If F1 is unsimplified, then by definition of NR-reduction, F1 reduces to F2 in an N-step. By Theorem 1, there exists an N-reduction E1,..,E2 such that R(G,H,E2,F2). But this is also an NR-reduction.

Case 2. E1 is simplified. Then, since R(G,H,E1,F1), F1 is also simplified. Let F1=f(s1,..,sn) where f is a constructor symbol. Hence for each i, 1=<i=<n, R(G,H,ti,si). Since F1 reduces to F2 in an NR-step, there is some si in s1,..,sn, such that si reduces to some si* in an NR-step and F2=f(s1,..,si*,..,sn). By induction hypothesis, there exists an NR-reduction ti=ti1,ti2,..,tik such that R(G,H,tik,si*). It can easily be shown that the reduction f(t1,..,ti1,..,tn), f(t1,..,ti2,..,tn), .., f(t1,..,tik,..,tn) is also an NR-reduction. Clearly R(G,H,f(t1,..,tik,..,tn),F2). **QED.**

Lemma 6. Let P be an F* program. Let R(G,H,E0,F0) and F0,..,Fn, n>=0, be an NR-reduction. Then there is an NR-reduction E0,..,Ek such that R(G,H,Ek,Fn).

Proof. Similar to that of Lemma 5. **QED.**

Theorem 4. Reduction-completeness of F* for normal forms. Let P be an F* program and D0 a term. Let D0,D1,..,Dn, n>=0, be a reduction in P, where Dn is in normal form. Then there is an NR-reduction D0,E1,..,Em=Dn, m>=0, in P.

Proof. By induction on length n of D0,D1,..,Dn. If n=0, D0 is already in normal form, so D0 is the required NR-reduction.

Let n>0 and assume theorem for D1,..,Dn. Then there is an NR-reduction D1,F2,..,Fp=Dn. The situation can be laid out as follows:

```
Dn (in normal form)
:
|
D2
|
D1->F2-*>Fp=Dn
|
D0->E1-*>Em
```

Since D0->D1 there are terms G,H, such that G=>H and D1=D0[G/H]. Hence R(G,H,D0,D1). Since D1,F2,..,Fp is an NR-reduction, by Lemma 6, there is an NR-reduction D0,E1,..,Eq such that R(G,H,Eq,Fp). It can easily be shown, by induction on length of terms, that there is an NR-reduction Eq,..,Fp. In each step in it, an occurrence of G is replaced by H. The required NR-reduction is then D0,E1,..,Eq,..,Fp=Dn=Em. **QED.**

CHAPTER IV

DETERMINISTIC F*

1.0 INTRODUCTION

A class of F* programs called **Deterministic F* (DF*)** is now defined and shown to possess several useful computational properties. In particular, every DF* program satisfies confluence and directedness. Confluence is shown to hold for any F* program which satisfies just restriction (f) below.

Confluence, means that if for terms M,N,P, M-*>N, and M-*>P, then there exists term Q such that N-*>Q, and P-*>Q. It has the immediate consequence that every term has at most one normal form. Hence DF* can be used as a functional programming system. Also, if a DF* program is interpreted as an equality theory, equality of two terms can be determined by checking whether their normal forms are syntactically identical.

Directedness, for simplified forms, means that if a term has a simplified form then *any* N-reduction starting at that term, if extended far enough, computes it. Moreover, all successful N-reductions are of equal length. Directedness, for normal forms, means that if a term has a normal form, then *any* NR-reduction starting at that term, if extended far enough, computes it. Moreover, all NR-reductions ending in normal forms are of equal length. Due to directedness, no searching among alternative N- or NR-reductions is necessary.

2.0 DEFINITION OF DF*

A DF* program is an F* program P satisfying two restrictions:

(f) Let LHS1 and LHS2 be variants of heads of two rules in P, such that LHS1 and LHS2 have no variables in common. Then LHS1 and LHS2 do not unify.

(g) Let f(L1,..,Li,..,Lm)=>RHS be a rule in P, where Li is not a variable. Then in every other rule f(K1,..,Ki,..,Km)=>RHS1 in P, Ki is not a variable.

Again, as can be seen from examples in this thesis, and especially in Chapter VII & VIII, DF* is also quite expressive. Note that restrictions (a)-(e) are upon rules while (f) and (g) are upon the entire program. In the following, the first three rules do not constitute a DF* program because they violate (f) and the next four do not because they violate (g):

```
insert(A,[])=>[A].
insert(A,[U|V])=>[A,U|V].
insert(A,[U|V])=>[U|insert(A,V)].
```

```
f(X,[])=>[].
f([],[U|V])=>[].
a=>a.
b=>[].
```

Restriction (f), supported by (a)-(e), ensures confluence. Restriction (g), supported by (a)-(f), ensures directedness. In particular, given a term f(t1,..,ti,..,tn), it is possible to determine, at compile time, whether ti needs to be simplified. It needs to be, only if the ith argument in the head of any F* rule defining f is a non-variable. Moreover, as shown below, the arguments of f which do need to be simplified can be simplified in any order.

The importance of select may be emphasized from the observation that even with restrictions (a)-(g), every outermost reduction strategy is not reduction complete. For example, given the DF* program:

g([],X)=>[].
g([U|V],W)=>h(U,V,W).

and the term E=g(b,a), a rightmost-outermost reduction strategy would compute the infinite reduction g(b,a),g(b,a),.... However, there exists a successful leftmost-outermost reduction g(b,a),g([],a),[]. Select, of course, would compute this second reduction.

Thus, select is more than an outermost reduction strategy. For DF*, it may perhaps be called *outermost-call-by-need.* Note that it is still non-deterministic. However due to directedness, the non-determinism is benign.

S_P(A,B). Let P be an F* program, and A,B be terms. Let G1,..,Gm, m>=0, be mutually non-overlapping subterms in A, and H1,..,Hm be terms such that for each i=<m, Gi=>Hi, and B is the result of *simultaneously* replacing G1,..,Gm, in A, by respectively, H1,..,Hm. Then we say S_P(A,B). Note that G1,..,Gm need not include all, or even one, of the mutually non-overlapping subterms of A which reduce as a whole. The subscript on S is dropped if clear from context.

R@i. Let R be an N-reduction E0,E1,..,Em, m>=0, where for no i, Ei=>Ei+1. Then there is some function symbol f, such that each Ei is of the form f(p1,..,pk). Let E0=f(t1,..,tk), and for any p, let Rp be E0,E1,..,Ep.

For any 1=<i=<k, R0@i is defined to be the singleton sequence ti. For any j≠m, let Ej=f(a1,..,an-1,an,an+1,..,ak), and Ej+1=f(a1,..,an-1,bn,an+1,..,ak) such that an reduces to bn in an N-step. If n=i, then Rj+1@i=Rj@i:bn, otherwise, Rj+1@i=Rj@i. Here : is concatenation of a term at the end of a sequence of terms.

For example, with the rules a=>a, b=>b1, b1=>b2, and the N-reduction R=f(a,b),f(a,b1), f(a,b1),f(a,b2), R@1=a,a, and R@2=b,b1,b2. Thus, roughly, R@i is the sequence, without duplicates, of ith arguments of the outermost function symbol of the members of the N-reduction R.

R@u. R@u is a generalization of R@i to positions in terms. Let R be an NR-reduction E0,E1,..,Em, m>=0, where E0 is simplified. Let A0,A1,..,Ap be unsimplified terms in E0 such that no Ai is properly contained in any other unsimplified term. Let the positions of A0,A1,..,Ap, in E0 be u1,..,up respectively. Then, for each j≠m, Ej+1 can be thought of as being derived from Ej by replacing a term A at one of the ui, by another term B. Moreover, A reduces to B in an NR-step.

For any ui in u1,..,up, R0@ui is defined to be the singleton sequence Ai. For any j, j≠m, let Ej+1 be derived from Ej by replacing a term P at position u in Ej by Q, where u is in u1,..,up. If u=ui, Rj+1@ui=Rj@ui:Q, otherwise, Rj+1@ui=Rj@ui.

3.0 CONFLUENCE AND DIRECTEDNESS OF DF*

Confluence and directedness are shown by deriving the following results, for any DF* program P:

(a) Let F1,E1,F2 be terms such that S(F1,E1), and F1->F2. Then there exists a term E2 such that E1-*>E2, and S(F2,E2).

(b) If there are two N-reductions starting at the same term and ending in terms in simplified form, then these terms are identical, and the N-reductions are of equal length.

(c) Let E0,E1,..,En be a successful N-reduction. Let E0=F0,F1,..,Fp, be an *unsuccessful* N-reduction, i.e. Fp is not simplified. Then p<n, and there exists Fp+1 such that Fp reduces to Fp+1 in an N-step.

Now, (a) is iterated to obtain confluence. (c) requires (b). From (c) we infer that if a term E0 has a successful N-reduction then no N-reduction starting at E0 is infinite, or terminates in failure. Hence, every N-reduction must terminate in a term in simplified form. Hence directedness for simplified forms. Similarly, directedness for normal forms.

Lemma 1. Select never chooses overlapping terms. Let P be a DF* program. Let E and F be terms such that select(E,F). Then, for all G, select(E,G) implies that G is not properly contained in F.

Proof. By induction on length of E. As before, the definition of select, for any term f(T1,..,Tn) is:

> select_P(f(T1,..,Tn),f(T1,..,Tn)) if f(T1,..,Tn)=>$_P$X.
>
> select_P(f(T1,..,Ti,..,Tn),X) if
>
> > there is a rule f(L1,..,Li,..,Ln)=>RHS in P, and
> >
> > there is no substitution σ such that Ti=Liσ, and
> >
> > select_P(Ti,X).

If E is a 0-ary function symbol, the lemma holds. Otherwise let E=f(t1,..,ti,..,tm) and let the lemma hold for each of t1,..,tm. Suppose select(E,F) but F≠E. Then for some ti in t1,..,tm, select(ti,F). Suppose select(E,G). If G=E then G is clearly not contained in F. Otherwise, for some tj in t1,..,tm, select(tj,G). If j=i, by induction hypothesis, G is not properly contained in F. If j≠i, of course, G is not properly contained in F.

Suppose select(E,F) and F=E. Then there exists a rule f(M1,..,Mi,..,Mm)=>RHS such that E matches its head. Now suppose that there also exists G such that select(E,G), and G is properly contained in F. Hence, for some ti in t1,..,tm, select(ti,G). Hence there is a rule f(L1,..,Li,..,Lm)=>RHS1 such that ti does not match Li. Hence Li is not a variable. By restriction (g) Mi is also not a variable. Hence ti is in simplified form. But then select(ti,G) fails. Contradiction. **QED.**

Lemma 2. Let P be a DF* program. Let G be a term. Then there is at most one term H such that G=>H.

Proof. Suppose G=>H1, G=>H2 and H1≠H2. Then there are two rules LHS1=>RHS1 and LHS2=>RHS2, such that G matches LHS1 with substitution σ1, and LHS2 with substitution σ2. Assume without loss of generality that LHS1 and LHS2 do not have any variables in common. Then LHS1 and LHS2 have a unifier σ1σ2, violating restriction (f). Contradiction. **QED.**

Lemma 3. Let σ and τ be two substitutions each defining only the variables X1,..,Xm, m>=0, such that for any i=<m, where <Xi,si> appears in σ, and <Xi,ti> in τ, S(si,ti). Let M be a term, possibly containing variables, but only from X1,..,Xm. Then S(Mσ,Mτ).

Proof. By induction on length of M. If M is a variable, clear. If M is a 0-ary function symbol, then clear. Otherwise, M=f(p1,..,pk), k>0. Assume Lemma for p1,..,pk. Mσ=f(p1σ,..,pkσ), and Mτ=f(p1τ,..,pkτ). Clearly, S(Mσ,Mτ). **QED.**

Lemma 4. Let P be a DF* program. Let A=f(t1,..,tm), m>0, and B=f(t1*,..,tm*), such that for each i=<m, S(ti,ti*). Let A=>C. Then there exists D, such that B=>D and S(C,D).

Proof. Let A reduce to C by the rule LHS=>RHS. Then, there exists a substitution σ such that A=LHSσ and C=RHSσ. It is easily verified that there exists substitution τ such that B=LHSτ, σ and τ define the same variables, and for each i, where <Xi,pi> appears in σ and <Xi,qi> in τ, S(pi,qi). Hence B=>RHSτ=D. By Lemma 3, S(C,D).

QED.

Lemma 5. Let P be a DF* program. Let F1,E1,F2 be terms such that S(F1,E1), and F1->F2. Then there exists term E2 such that E1-*>E2, and S(F2,E2).

Proof. By induction on length of F1. The situation can be visualized in the following diagram:

```
        F1
       / \
      *    \
     /      \
    E1       F2      S(F1,E1), F1->F2
     \      /
      *    *
       \ /
        E2
```

Case 1. F1 is a 0-ary function symbol. Since F1->F2, F1=>F2. If F1=E1, take E2=F2. Then E1=>E2, and S(F2,E2), as required. Otherwise, F1=>E1. By restriction (f), E1=F2. Take E2=F2. Again, E1-*>E2, and S(F2,E2), as required.

Case 2. F1=f(t1,..,tm), m>0. Assume Lemma for t1,..,tm.

Case 2-1. F1 reduces to E1 as a whole. If F1 reduces to F2 as a whole, by restriction (f), F2=E1. Take E2=E1. Then E1-*>E2, and also S(F2,E2), as required. Otherwise, there is some j=<m, such that tj->tj*, and F2=f(t1,..,tj*,..,tm). Hence S(F1,F2). By Lemma 4, there exists E2 such that F2=>E2 and S(E1,E2). But then E1-*>E2. Since F2=>E2, S(F2,E2), as required.

Case 2-2. F1 does not reduce as a whole to E1. Then E1=f(s1,..,sm), and for each i=<m, S(ti,si).

If F1=>F2, then, by Lemma 4, there exists E2 such that E1=>E2, and S(F2,E2), as required. Otherwise, there is some j=<m, such that tj->tj*, and F2=f(t1,..,tj-1,tj*,tj+1,..,tm). By induction hypothesis, there exists p, such that sj-*>p and S(tj*,p). Take E2=f(s1,..,sj-1,p,sj+1,..,sm). Clearly, E1-*>E2, and also, S(F2,E2), as required. **QED.**

Lemma 6. Let P be a DF* program, and M,N,P be terms such that S(M,N) and M-*>P. Then there exists term Q such that N-*>Q and S(P,Q).

Proof. By iterating Lemma 5. **QED.**

Lemma 7. Let P be a DF* program, and M,N,P be terms such that M->N, and M-*>P. Then there exists term Q such that N-*>Q, and P-*>Q.

Proof. Since M->N, S(M,N). By Lemma 6, there exists term Q such that N-*>Q and S(P,Q). Hence P-*>Q, as required. **QED.**

Theorem 1. Confluence of DF*. Let P be a DF* program, and M,N,P be terms such that M-*>N, and M-*>P. Then there exists term Q such that N-*>Q, and P-*>Q.

Proof. By iterating Lemma 7. **QED.**

Corollary. Uniqueness of normal forms. Let P be a DF* program. Then every term has at most one normal form.

Lemma 8. Let R be an N-reduction f(p1,..,pk)=E0,E1,..,Em such that for no i, Ei=>Ei+1. Then the length m of E0,E1,..,Em is equal to the sum of the lengths of R@1, R@2,..,R@k.

Proof. By induction on m. If m=0, then clear. Assume lemma for E0,..,Em-1. There exists exactly one n, such that Em-1=f(t1,..,tn-1,tn,tn+1,..,tk), Em=f(t1,..,tn-1,un,tn+1,..,tk), and tn reduces to un in an N-step. So, for every i, i≠n, (E0,..,Em-1)@i=(E0,..,Em-1,Em)@i. Only for n, (E0,..,Em-1,Em)@n=(E0,..,Em-1)@n:un. By induction hypothesis, the lemma is clear. **QED.**

Lemma 9. Let P be a DF* program. Let E0,E1,..,En be a successful N-reduction. Then for every successful N-reduction E0,F1,..,Fp, p=n and Fp=En.

Proof: By induction on n. If n=0 then E0 is simplified and the lemma holds trivially. Let n>1. Assume hypothesis for all successful N-reductions of length less than n.

Since E0,E1,..,En is a successful N-reduction, there exists Ek, 0=<k<n, such that for no i, 0=<i<k, Ei=>Ei+1, but Ek=>Ek+1, and Ek+1,..,En is a successful N-reduction. Let E0=f(t1,..,tm), m>=0. Since n>0, f is not a constructor symbol. Then Ek=f(s1,..,sm) for terms s1,..,sm. Similarly, for the successful N-reduction F0,F1,..,Fp, there exists Fj with properties similar to those of Ek. The situation can be laid out in the following diagram:

```
E0=f(t1,..,tm)----*>Ek=f(s1,..,sm)=>Ek+1--*>En

F0=f(t1,..,tm)----*>Fj=f(q1,..,qm)=>Fj+1--*>Fp
```

Consider any ti in t1,..,tm.

Case 1. ti is simplified. Then, by definition of select, ti=si. Similarly, ti=qi. Hence qi=si.

Case 2. ti is unsimplified. If si is simplified, then there exists a successful N-reduction (E0,..,Ek)@i, of length less than n.

By restriction (g), the ith argument in the head of any rule in P, defining f, is a non-variable. Hence, qi is also simplified. Hence there exists a successful N-reduction (F0,..,Fj)@i. By induction hypothesis, its length is equal to that of (E0,..,Ek)@i, and qi=si.

If si is unsimplified, then, by restriction (g), and definition of select, ti=si=qi.

Hence Ek=Fj. By Lemma 8, the length of E0,..,Ek is the sum of lengths of (E0,..,Ek)@i such that ti is unsimplified but si is simplified. Again, by Lemma 8, and **Case 2**, this is also the length of F0,..,Fj. Hence, k=j. By restriction (f), Ek+1=Fj+1. Now, Ek+1,..,En, is a successful N-reduction of length less than n. By induction hypothesis, its length is equal to that of the successful N-reduction Fj+1,..,Fp, and En=Fp. Hence, also, the length of E0,..,En is equal to that of F0,..,Fp. **QED**.

Lemma 10. Let P be a DF* program. Let E0,E1,..,En and E0=F0,F1,..,Fm be two NR-reductions such that En and Fm are in normal form. Then En=Fm and n=m.

Proof: Note that En=Fm follows directly from the confluence of DF*. We focus on showing that n=m, and proceed by induction on length of E0,E1,..,En. If n=0 then clear. Otherwise, let n>0 and assume the lemma for NR-reductions of length less than n.

Case 1. E0 is unsimplified. Then, there exists Ek, 0<k=<n such that E0,..,Ek is a successful N-reduction. Also, there exists Fj, 0<j=<m such that E0,..,Fj is a successful N-reduction. By Lemma 9, Fj=Ek and j=k. Now Ek,..,En, and Fj,..,Fm are also NR-reductions. The length of Ek,..,En is less than n, so by induction hypothesis, n=m.

Case 2. E0 is simplified. Then E0=c(t1,..,tq) for terms t1,..,tq and constructor symbol c. If E0 is in normal form then the theorem holds. Otherwise, let A0,A1,..,Ap be unsimplified terms in E0 such that no Ai is properly contained in any unsimplified term. Consider any Ai in A0,A1,..,Ap.

Let the position at which Ai occurs in E0 be ui. Then, since En is in normal form, (E0,..,En)@ui is an NR-reduction ending in a normal form. Similarly obtain (F0,..,Fm)@ui.

By reasoning as in **Case 1**, the length of (E0,..,En)@ui is equal to that of (F0,..,Fm)@ui. It can be shown, analogously to Lemma 8, that the length of E0,..,En is equal to the sum of the lengths of each (E0,..,En)@ui. Similarly, for length of

F0,..,Fm. Hence m=n. **QED.**

Lemma 11. Let P be a DF* program and E0 a term. Let E0,E1,..,En be a successful N-reduction. Let E0=F0,F1,..,Fp, be an *unsuccessful* N-reduction, i.e. Fp is not simplified. Then p<n, and there exists Fp+1 such that Fp reduces to Fp+1 in an N-step.

Proof: By induction on the length of E0,..,En. If n=0 then E0 is in simplified form and the only N-reduction starting at E0 is E0 itself, so the lemma is clear. Let n>0. Then E0 is not simplified. Assume lemma for all successful N-reductions of length less than n. Since E0,E1,..,En is a successful N-reduction, there exists Ek, 0=<k<n such that for no i, 0=<i<k, Ei=>Ei+1, but Ek=>Ek+1, and Ek+1,..,En is a successful N-reduction. Let E0=f(t1,..,tm), m>=0. Let Ek=f(s1,..,sm). We have two cases:

Case 1. There exists Fj, 0=<j<p such that for no i, 0=<i<j, Fi=>Fi+1, but Fj=>Fj+1, and Fj+1,..,Fp is an N-reduction. Let Fj=f(q1,..,qm). The situation can be visualized in the following diagram:

```
E0=f(t1,..,tm)--*>Ek=f(s1,..,sm)=>Ek+1--*>En
F0=f(t1,..,tm)--*>Fj=f(q1,..,qm)=>Fj+1--*>Fp
```

By reasoning as in Lemma 9 above, Ek=Fj, and k=j. By restriction (f), Ek+1=Fj+1. By induction hypothesis, p<n. Furthermore, there exists Fp+1 such that Fp reduces to Fp+1 in an N-step.

Case 2. There does not exist Fj in F0,..,Fp such that Fj=>Fj+1. Let Fp=f(q1,..,qm). The situation can be visualized as:

```
E0=f(t1,..,tm)--*>Ek=f(s1,..,sm)=>Ek+1--*>En

F0=f(t1,..,tm)--*>Fp=f(q1,..,qm)  (unsimplified)
```

It is easily seen that either Fp=>Fp+1, or there exists i in 1,..,m such that qi is not simplified, but si is. By induction hypothesis, there exists ri such that qi reduces to ri in an N-step. Hence Fp reduces to f(q1,..,ri,..,qm) in an N-step. Also, by Lemma 8, and induction hypothesis, p<n. **QED.**

Theorem 2. Directedness for simplified forms. Let P be a DF* program. Let E0 be a term and let E0,..,En be a successful reduction. Then any N-reduction starting at E0, if extended far enough, would terminate in a term in simplified form.

Proof. By reduction-completeness for simplified forms, (Theorem 2, Chapter III), and iterating Lemma 11. **QED.**

Lemma 12. Let P be a DF* program and E0 a term. Let E0,E1,..,En be an NR-reduction such that En is in normal form. Let E0=F0,F1,..,Fp, be an NR-reduction such that Fp is not in normal form. Then, p<n, and there exists Fp+1 such that Fp reduces to Fp+1 in an NR-step.

Proof: By induction on length of E0,E1,..,En. If n=0, then clear. Otherwise, assume Lemma for all NR-reductions of length less than n, and ending in normal forms.

Case 1. E0 is unsimplified. By a reasoning very similar to that in proof of Lemma 11.

Case 2. E0 is simplified. If E0 is in normal form, the lemma trivially holds. Otherwise, as in Case 2 of Lemma 10. **QED.**

Theorem 3. Directedness for normal forms. Let P be a DF* program. Let E0 be a term and let E0,..,En be a reduction where En is in normal form. Then any NR-reduction starting at E0, if extended far enough, would terminate in a normal form.

Proof By reduction-completeness for normal forms, (Theorem 4, Chapter III), and iterating Lemma 12. **QED.**

CHAPTER V

LABELED DETERMINISTIC F*

1.0 INTRODUCTION

Intuitively, it can be seen that in an N-reduction a term is reduced only when it is necessary for simplifying the first term in the reduction. In this sense, an N-reduction conserves computation. For example, with the rules:

```
f(X)=>[X].
a=>[].
```

there exists the N-reduction f(a),[a]. There is no N-reduction starting at f(a) in which a is reduced. However, it can also happen that in an N-reduction, several copies of the same term are reduced. This can happen when use is made of a rule in which a variable occurs more than once on the right hand side. In this sense, an N-reduction wastes computation. For example, with the rules:

```
f(X)=>g(X,X).
g([],[])=>[].
a=>[].
```

there exists the N-reduction f(a),g(a,a),g([],a),g([],[]),[]. The two occurrences of a in the second term are copies of each other, yet they are reduced separately. This is the same problem which arises with a call-by-name procedure call mechanism in programming languages.

If we can arrange that when a term is reduced, all copies of it are also reduced, then N-reductions could become considerably shorter. In fact, it is shown that they become *minimal*. An N-step in which all copies of a term are replaced is called an NA-step. Thus it is distinguished from an N-step in which only a single copy of a term is replaced. A sequence of NA-steps is called an NA-derivation. The prefix NA stands for "normal-all". Minimality yields a strong form of laziness, since terms are simplified with minimum computational effort.

The notion of a copy of a term, however, has two legitimate interpretations. The first is simply that any two occurrences of a subterm in a term are copies of each other.

The second is obtained from examining representations of terms as directed acyclic graphs. A 0-ary function symbol f is represented as a graph consisting of just a single node with f stored in it. A term f(t1,..,tn) is represented as a graph whose root is a node with n+1 fields. The first field stores f and for each 1=<i=<n, the ith field stores a pointer to the graph representation of ti. A term can have many graph representations. For example, the two occurrences of a in f(a,a) can be represented by a single graph, or by distinct graphs. Now, two occurrences of a subterm in term E are said to be copies of each other *only* if, in the graph representation of E, they have the same graph representation.

We adopt the second interpretation since it enables us to develop a simple proof of minimality and also to implement replacement of all copies of a subterm with a small overhead. Graph representations of terms are, in turn, represented using labeled terms. The address of each node in a graph is represented by a label. Let there be a graph G with root node N. Let N contain m+1 fields, where the first field contains the symbol f and the rest of the m fields contain pointers to, respectively, graphs G1,..,Gm. Let the address of N be represented by label α. Then the representation of G is the labeled term f(α,t1,..,tm) where for each i, the representation of Gi is the labeled term ti.

A subset of DF* called, Labeled Deterministic F* (LDF*), is defined. Notions of labels [Vuillemin 1974], labeled terms, ordinary terms, and ordinary programs are introduced. Reductions in LDF* are intended to mimic graph reduction. In particular, it is ensured that when a new node is allocated in graph reduction, a label not previously used in the LDF* reduction is generated.

LDF* is shown to be minimal in the following sense: where P* is an LDF* program, and E a proper term, let there be a shortest successful reduction of E. Then there is a successful NA-derivation of E of lesser or equal length.

It is also shown that with each ordinary DF* program P, one can associate an LDF* program P*, such that if there is a successful reduction in P, there is a successful reduction in P* of exactly equal length. Hence, to simplify terms in P in a minimum number of steps, it is sufficient to transform P to P* and use NA-derivations.

Some main ideas in our proof are (a) in each reduction step, a label is eliminated, so (b) the size of the elimination-set (E-set) of a reduction, i.e. the set of labels eliminated in the reduction, is a lower-bound on its length, (c) the size of the E-set of an NA-derivation of a proper term, is exactly equal to its length.

2.0 DEFINITION OF LDF*

Labels. Let α, β, ξ, α1, β1, ξ1,.. be an enumerably infinite subset of the set of 0-ary function symbols in F*. Each member of this list is called a primitive label.

Let * be a binary function symbol in F*. A label is defined as follows. A primitive label is a label. If x and y are labels then x*y is also a label. A label α is said to be a proper initial segment of label β if either β=α*δ, or β=ξ*δ and α is a proper initial segment of ξ.

Labeled terms. Where f is an n+1-ary function symbol, n>=0, f≠*, α a label and t1,..,tn labeled terms, f(α,t1,..,tn) is a labeled term. α is called the outermost label of f(α,t1,..,tn).

For example, where f is a 4-ary function symbol and a,b are 1-ary function symbols, f(δ,a(α),b(β),a(ξ)) is a labeled term, and δ is the outermost label of this term. Note that a label standing alone is *not* a labeled term. Neither is a labeled term of the form A*B. *Also, note that a labeled term never contains any variables.*

A labeled term is said to be in normal form if it contains only constructor symbols and labels.

Maximal labels. A label is maximal in a labeled term if it is not a proper initial segment of any other label in that term.

Proper terms. A labeled term E is called proper if (a) all its labels are maximal, and

(b) for every two subterms A and B of E, if A and B have the same outermost label then A=B. For example, f(α,b(ξ),c(δ)) is a proper term. However, g(α,a(α*δ)) is not a proper term since it violates (a), and f(α,b(α)) is not a proper term since it violates (b).

Ordinary terms, ordinary rules, and ordinary programs. A term in F*, possibly containing variables, a rule in F*, or an F* program, is said to be ordinary if it does not contain any labels, nor any occurrence of the symbol *.

A mapping Σ. Let F be the set of all function symbols in F*, except *, and the primitive labels. Let there be an injection Σ between F and F which maps each n-ary function symbol in F to an n+1-ary function symbol in F. Moreover, Σ always maps a constructor symbol to a constructor symbol, and a non-constructor symbol to a non-constructor symbol.

Labeled versions of ordinary terms, possibly containing variables. Let E be an ordinary term, possibly containing variables. If E is a variable then its labeled version is E itself. Otherwise, let E=f(t1,..,tn), n>=0. Its labeled version is f_(α,t1_,..,tn_) where α is a label, Σ(f)=f_, and for each i, ti_ is a labeled version of ti. For example, where Σ maps f to f_ and a to a_, a labeled version of f(a,a) is f_(α,a_(β),a_(δ)).

Labeled versions of ordinary rules. Let LHS=>RHS be an ordinary F* rule. A labeled version of this rule, LHS*=>RHS*, is defined as follows:

Let LHS=f(L1,..,Lm). Then LHS*=f_(L,L1_,..,Lm_) satisfying the following conditions: (a) f_=Σ(f), (b) L is a variable, (c) If Li is a variable, Li_=Li, otherwise

Li=c(X1,..,Xm), and Li_=c_(Ki,X1,..,Xm), Ki a variable, Σ(c)=c_, and (d) a variable occurs at most once in LHS*.

Let RHS1 be a labeled version of RHS in which all labels are distinct, and for no two of these is one a proper initial segment of the other. Let these labels be β1,..,βk. Then, where LHS*=f(L,L1_,..,Lm_), RHS* is obtained by replacing, in RHS1, each βi by L*βi.

Labeled Deterministic F* Programs. Let P be an ordinary DF* program, and let P* consist of labeled versions of rules in P. Then P* is called a labeled deterministic F* (LDF*) program. For example, where P consists of:

```
append(nil,X)=>X
append(cons(U,V),W)=>cons(U,append(V,W))
```

and Σ maps append, nil, and cons to append_, nil_ and cons_ respectively, P* consists of:

```
append_(L,nil_(L1),X)=>X.
append_(L,cons_(K1,U,V),W)=>cons_(L*β1,U,append_(L*β2,V,W))
```

where β1, and β2 are distinct labels, and neither is a proper initial segment of each other. Note that each LDF* program is a DF* program as well as an F* program. Also, each labeled term is an F* term. *Hence, results of all previous chapters also hold for LDF* programs and labeled terms.*

NA-steps and NA-derivations. Let P be an LDF* program and E,G,H be labeled terms. Suppose $select_P$(E,G) and G=>$_P$H. Let E1 be the result of replacing *all* occurrences of G by H in E. Then we say that E reduces to E1 in an NA-step in P. The prefix NA in N-step stands for "normal-all". A sequence of labeled terms E0,E1,.. is an **NA-derivation** if for each i, when Ei, and Ei+1 both exist, Ei reduces to Ei+1 in an NA-step. For example, given the rule a(L)=>b(L*α), the term f(β,a(δ),a(δ)) reduces in an NA-step to f(β,b(δ*α), b(δ*α)).

Leftmost steps and reductions. Let P be an F* program, not necessarily ordinary. Let E be a term, and G and G1 two of its subterms. G is said to be to the left of G1 in E, if either (a) they both occur at the same position in E, or (b) in the depth-first or preorder traversal of the tree representation of E, the function symbol which is the root of G occurs before the function symbol which is the root of G1.

Let select(E,G), G=>H. Then E reduces to F in a leftmost N-step, if for every G1, select(E,G1) implies G is to the left of G1 in E, and F=E[G/H].

Let select(E,G), G=>H. Then E reduces to F in a leftmost NA-step, if for every G1, select(E,G1) implies G is to the left of G1 in E, and F is the result of replacing all occurrences of G in E by H.

Definitions of leftmost N-reductions and leftmost NA-derivations are the obvious ones.

Elimination sets or E-sets. Let A0,A1,..,An be a reduction where each Aj is a labeled term. For any i, let Ai+1=Ai[G/H] where G=f(α,t1,..,tm). Then we say that the function-label pair (FL-pair) <f,α> has been eliminated in the reduction Ai,Ai+1. The elimination set or E-set of a reduction is defined as the set of all FL-pairs eliminated in the reduction. Since elimination of an FL-pair requires one reduction step, the size of this set (number of elements in it) is a lower bound on the length of the reduction (the number of steps in it).

Since an NA-step can be thought of as a sequence of reductions steps, an NA-derivation can be thought of as a reduction. Hence, the E-set of an NA-derivation is the E-set of the corresponding reduction.

3.0 MINIMALITY OF LDF*

Let P be an LDF* program and E a labeled term. We already know from completeness of DF* that if E has a successful reduction, it has a successful N-reduction. By directedness of DF*, it has a successful leftmost N-reduction. We now show the following:

(a) If E has a successful reduction R0, E has a successful N-reduction R1 whose E-set is a subset of the E-set of R0.

(b) If R1 and R2 are two successful N-reductions of E, their E-sets are identical.

(c) If E has a successful leftmost N-reduction R2, it has a successful leftmost NA-derivation R3. Furthermore, the E-set of R3 is a subset of the E-set of R2.

(d) If E is a proper term, the size of the E-set of any NA-derivation starting at E is equal to the number of NA-steps in that derivation.

(e) Let P be an ordinary DF* program, and P* its labeled version. Let E0 be an ordinary term, and E0* a labeled version of E0. Let E0,E1,E2,... be a reduction in P. Then there exists a reduction E0*,E1*,E2*,... in P*, such that for each i, Ei* is a labeled version of Ei.

Let E be a proper term. Let R0 be a shortest successful reduction of E. Then its length is greater than or equal to the size of its E-set. By (a), there exists R1, an N-reduction of E whose E-set is a subset of that of R0. By directedness of DF*, there exists R2, a successful leftmost N-reduction of E. By (b), the E-set of R1 is identical to that of R2. By (c), there exists R3, a successful leftmost NA-derivation of E whose E-set is a subset of that of R2. By (d), the length of this NA-derivation is at most the length of R0. Hence, leftmost NA-derivations are minimal for simplifying proper terms.

Now, let P be an ordinary DF* program and P* its labeled version. Let there be a successful reduction R of a term E in P. By (e) there exists a successful reduction R* of a labeled version E* of E. This version can always be chosen to be proper. By minimality of LDF*, there is a successful NA-derivation starting at E* of length less than or equal to that of R* or R. Hence, to simplify terms in a minimum number of steps, it is sufficient to transform P to P* and use leftmost NA-derivations. This reasoning is now carried out formally and in detail.

3.1 Existence of successful leftmost NA-derivations

Lemma 1. Let P be an LDF* program. Let E0,F0 be labeled terms such that E0->F0. If E0 reduces to E1 in a leftmost N-step then there exists F1 such that E1-*>F1, and (a) either F1=F0, or (b) F0 reduces to F1 in a leftmost N-step, and the FL-pair eliminated in F0,F1 is the same as that eliminated in E0,E1.

Proof. By induction on length of E0. We can draw the following diagram:

```
E0------> F0
|         |
leftmost  | F0=F1 or
N-step    | F0 reduces to F1 in a leftmost N-step
|         |
E1-----*> F1
```

Case 1. E0=f(α) for some 1-ary function symbol f and label α. Since E0->F0, E0=>F0. So, select(E0,E0) and due to restriction (f), E1=F0. Take F1=F0. Clearly, E1-*>F1.

Case 2. E0=f(α,t1,..,tm), for some m+1-ary function symbol f, m>0, label α, and labeled terms t1,..,tm.

Case 2-1. E0=>F0. Similar to Case 1. E1=F1=F0 and E1-*>F1.

Case 2-2. Not E0=>F0. Then F0=f(α,t1*,..,tm*), and there exists j such that tj->tj* and for each i, i$\neq$j implies ti=ti*.

Suppose E0=>E1. Then the FL-pair eliminated in E0,E1 is $<f,\alpha>$. It is easily verified by induction on length of E0, that F0=>F1 and E1-*>F1. Also, F0 reduces to F1 in a leftmost N-step. Finally, the FL-pair eliminated in F0,F1 is also $<f,\alpha>$.

Suppose not E0=>E1. Then, there is some k, such that E1=f(t1,..,tk-1,sk,tk+1,..,tm), and tk reduces to sk in a leftmost N-step. By induction hypothesis, there exists sk* such that sk-*>sk*, and either tk*=sk*, or tk* reduces to sk* in a leftmost N-step, and the FL-pair eliminated in tk,sk is the same as that eliminated in tk*,sk*.

If tk*=sk*, let F1=F0. Clearly, E1-*>F1. Otherwise, let F1=f(α,t1*,..,tk-1*,sk*,tk+1*,..,tm*). Since tk* reduces to sk* in an N-step, tk* is not simplified. Hence, using restriction (g), it is easily verified that F0 reduces to F1 in a leftmost N-step. In particular, in this step, tk* reduces to sk* in a leftmost N-step. Hence, E1-*>F1, and the FL-pair eliminated in E0,E1 is the same as that eliminated in F0,F1. **QED.**

Lemma 2. Let P be an LDF* program. Let E0,F0 be labeled terms such that E0-*>F0. If E0 reduces to E1 in a leftmost N-step then there exists F1 such that E1-*>F1, and (a) either F1=F0, or (b) F0 reduces to F1 in a leftmost N-step, and the FL-pair eliminated in F0,F1 is the same as that eliminated in E0,E1.

Proof. By induction on length of the reduction E0,..,F0. We can draw the following diagram:

```
E0-----*>F0
|         |
leftmost  | F0=F1 or
N-step    | F0 reduces to F1 in a leftmost N-step
|         |
E1-----*>F1
```

If the length is 0 then clear. Otherwise let E0,..,F0 be E0,G1,..,Gk=F0, k>0. Assume lemma for G1,..,Gk. Let E0 reduce to E1 in a leftmost N-step. Then, by Lemma 1, there exists H1 such that E1-*>H1, and either G1=H1, or G1 reduces to H1 in a leftmost N-step, and the FL-pair eliminated in G1,H1 is the same as that eliminated in E0,E1. This situation can be laid out in the following diagram:

```
E0------>G1 -> G2..->F0
|         |
leftmost  | G1=H1 or,
N-step    | G1 reduces to H1 in a leftmost N-step.
|         |
E1-----*>H1
```

If G1=H1 then take F1=F0. Since E1-*>H1=G1, and G1-*>F0, E1-*>F0=F1.

If G1 reduces to H1, then by induction hypothesis, there exists F1 such that H1-*>F1, and either F0=F1, or F0 reduces to F1 in a leftmost N-step, and the FL-pair eliminated in F0,F1 is the same as that eliminated in G1,H1. Since E1-*>H1, E1-*>F1. Also, if F0 reduces to F1, the FL-pair eliminated in F0,F1 is the same as that eliminated in E0,E1. **QED**.

Lemma 3. Let P be an LDF* program. Let E0,F0 be labeled terms such that E0-*>F0 and let there be a successful leftmost N-reduction E0,E1,..,En. Then there is a successful leftmost NA-derivation F0,F1,F2,..,Fk whose E-set is a subset of that of

E0,E1,..,En.

Proof. It is helpful to draw the following diagram.

```
E0------*>F0
|          |
leftmost   | F0=F1 or,
N-step     | F0 reduces to F1 in a leftmost N-step.
|          |
E1------*>F1
|          |
leftmost   | F1=F2 or,
N-step     | F1 reduces to F2 in a leftmost N-step.
:          :
:          :
En------*>Fn
```

By induction on length n of E0,..,En. If n=0, then clear. Otherwise, let n>0 and assume lemma for E1,..,En. Since E0 reduces to E1 in a leftmost N-step and E0-*>F0, by Lemma 2, there exists A1 such that E1-*>A1, and either F0=A1, or F0 reduces to A1 in a leftmost N-step, and the FL-pair eliminated in E0,E1 is the same as that eliminated in F0,A1.

Case 1. F0=A1. Let F1=F0. Then E1-*>F1. By induction hypothesis, there exists a successful leftmost NA-derivation F1,..,Fk whose E-set is a subset of that of E1,..,En. Hence the E-set of F0,F1,..,Fk is a subset of that of E0,E1,..,En.

Case 2. F0 reduces to A1. Let there exist G,H such that A1=F0[G/H]. Let F1 be obtained from A1 by replacing the remaining occurrences of G in A1 by H. Then F0 reduces to F1 in a leftmost NA-step. Moreover, the FL-pair eliminated in the NA-step F0,F1 is the same as that eliminated in E0,E1.

Since E1-*>A1, E1-*>F1. By induction hypothesis, there exists a successful leftmost NA-derivation F1,..,Fk whose E-set is a subset of that of E1,..,En. Hence there exists a successful leftmost NA-derivation F0,F1,..,Fk whose E-set is a subset of that of E0,E1,..,En. **QED.**

Theorem 1. Let P be an LDF* program. Let E0 be a labeled term and E0,E1,..,En a successful leftmost N-reduction. Then there is a successful leftmost NA-derivation E0,F1,F2,..,Fk whose E-set is a subset of that of E0,E1,..,En.

Proof. Since E0-*>E0, apply Lemma 3. **QED.**

3.2 E-sets of N-reductions

Lemma 4. Let P be an LDF* program and E1,F1,G,H be labeled terms such that:

(a) R(G,H,E1,F1), and

(b) F1 reduces to F2 in an N-step, and

(c) The outermost function symbol and label of G are, respectively, g and α, and

(d) $<r,\beta>$ is the FL-pair eliminated in the reduction F1,F2.

Then there exists an N-reduction E1,..,E2 such that its E-set is included in $\{<g,\alpha>,<r,\beta>\}$ and R(G,H,E2,F2).

Proof. The proof proceeds exactly as that of Theorem 1, Chapter III. Thus, we already know that there exists an N-reduction E1,..,E2 such that R(G,H,E2,F2). We now show that the E-set of E1,..,E2 is included in $\{<g,\alpha>,<r,\beta>\}$. This is easy to see in the extreme cases, i.e. when E1=h(δ) for some 1-ary function symbol h, and label δ, E1=G or E1=>F1. Otherwise, let E1=f(ξ,t1,..,tm), m>0. Then F1=f(ξ,t1*,..,tm*), and for each i, R(G,H,ti,ti*). We have the following cases:

Case 1. F1=>F2. Then $<f,\xi>$ is eliminated, r=f and $\beta=\xi$. In particular, there is some rule f(L,L1,..,Lm)=>RHS in P such that F1 matches f(L,L1,..,Lm).

Case 1-1. E1 matches f(L,L1,..,Lm). Then $<f,\xi>$ is eliminated again, which is in $\{<g,\alpha>,<r,\beta>\}$ as required.

Case 1-2. E1 does not match f(L,L1,..,Lm). Then there is some Li in L1,..,Lm such that ti* matches Li but ti does not. Hence Li is not a variable, so ti* is simplified, but ti is not. Since R(G,H,ti,ti*), ti=G. As in Case 1-2, Theorem 1, Chapter III, $<g,\alpha>$ is eliminated several times and then $<f,\xi>=<r,\beta>$ is eliminated. So, the E-set of E1,..,E2 is $\{<g,\alpha>,<r,\beta>\}$, as required.

Case 2. There is no F2 such that F1=>F2. Then there is some ti* in t1*,..,tm* such that ti* reduces to ti** in an N-step and the E-set of ti*,ti** is $\{<r,\beta>\}$. By induction hypothesis, there is an N-reduction ti,..,dr such that R(G,H,dr,ti**). Further, the E-set of this reduction is contained in $\{<g,\alpha>,<r,\beta>\}$. From the argument in Case 2 of Theorem 1, Chapter III, the E-set of E1,..,E2 is contained in $\{<g,\alpha>,<r,\beta>\}$, as required. **QED.**

Lemma 5. Let P be an LDF* program. Let E1,F1,G,H, be labeled terms such that R(G,H,E1,F1). Let the outermost function symbol and label of G be g and α

respectively. Let F1,F2,..,Fm be a successful N-reduction. Then there exists a successful N-reduction E1,..,En whose E-set is contained in the union of {<g,α>} and the E-set of F1,F2,..,Fm.

Proof. By induction on length of F1,..,Fm. If m=1 then Fm=F1 is simplified and its E-set is empty. If E1 is simplified, then clear. If not, then since R(G,H,E1,F1), E1=>F1. Thus, there exists the successful N-reduction E1,F1. Hence the E-set of E1,F1 is {<g,α>}, as required.

Let m>1. Assume lemma for F2,..,Fm. Since R(G,H,E1,F1), by Lemma 4, there exists an N-reduction E1,..,E2 such that R(G,H,E2,F2) and whose E-set is contained in the union of {<g,α>} and the E-set of F1,F2.

By induction hypothesis, there exists an N-reduction E2,..,En whose E-set is contained in the union of {<g,α>} and the E-set of F2,..,Fm. Hence, the E-set of E1,..,En is contained in the union of {<g,α>}, and the E-set of F1,F2,..,Fm. **QED**.

Lemma 6. Let E1,F1,G2,..,Gm be a successful reduction. Then there is a successful N-reduction E1,..,En such that the E-set of E1,..,En is contained in that of E1,F1,G2,..,Gm.

Proof. By induction on length of E1,F1,G2,..,Gm, and Lemma 5. **QED**.

Lemma 7. Let P be an LDF* program. Let E0 be a labeled term and E0,E1,..,En and E0,F1,..,Fp two successful N-reductions. Then, the E-set of one is identical to that of the other.

Proof. Exactly analogous to the proof of Lemma 9, Chapter IV, that any two successful N-reductions of a term are of equal length, and end in the same simplified form. **QED.**

3.3 Reductions of proper terms

Lemma 8. Let P be an LDF* program. Let E be a proper term and let E reduce to F in an NA-step. Then all labels of F are maximal.

Proof. Let select(E,G). Then G=>H and F is obtained by replacing all occurrences of G in E by H. Let the rule by which G=>H be LHS=>RHS, and let G=g(α,t1,..,tm), m>=0, and each ti a labeled term. Take any two labels β and ξ in F. There are four cases.

Case 1. β and ξ are both in E. Since E is proper, these labels are not proper initial segments of each other.

Case 2. Only β is in E. Then, by the nature of LDF* rules, ξ=α*δ for some label δ in RHS. Since E is proper, β and α are not proper initial segments of each other. If β≠α then β and α*δ are also not proper initial segments of each other.

Suppose β=α. Since β occurs in E, E has a subterm f(β,s1,..,sn), n>=0. Since E is proper, f(β,s1,..,sn)=G. But since all occurrences of G are replaced by H, β cannot occur in F, as assumed. Hence this subsubcase cannot arise.

Case 3. Only ξ is in E. Same as case 2.

Case 4. None of β and ξ is in E. Then, by definition of LDF* rules, $\beta=\alpha*\delta$ and $\xi=\alpha*\varepsilon$, for some labels δ and ε in RHS. Since δ and ε are not proper initial segments of each other, neither are β and ξ. **QED.**

Lemma 9. Let P be an LDF* program. Let E be a proper term and let E reduce to F in an NA-step. Let A and B be two subterms of F such that the outermost label of A and of B is β. Then, A=B.

Proof. Let select(E,G). Then G=>H and F is obtained by replacing all occurrences of G in E by H. Let the rule by which G=>H be LHS=>RHS, and let the outermost label of G be α. There are four cases.

Case 1. A, but not B, is a subterm of H. Let the label of A, and of B be β. Since B is not a subterm of H, there occurs B1 in E, with label β, such that B is the result of replacing all occurrences of G in B1 by H.

Since A is a subterm of H, β occurs in H. However, $\beta\neq\alpha*\delta$ since α, and β both occur in E, and E is proper. Hence, β occurs in G. But then, since E is proper, B cannot properly contain G. Hence B1=B, so B occurs in E.

Now β is also the label of A, $\beta\neq\alpha*\delta$, and A occurs in H. Hence A occurs in G, and so in E. Since E is proper, A=B, as required.

Case 2. B, but not A, is a subterm of H. Then, as in the previous case, B occurs in E. Hence A=B.

Case 3. Both A and B are subterms of H. Then A and B are also contained in G. Suppose A is not, but B is. Then, $\beta=\alpha*\delta$. Since B occurs in G, β also occurs in E. Contradiction with E is proper. Similarly, for A in G, but not B. Suppose none of A and B are in G. Without loss of generality assume A and B occur at distinct positions. Then, there must be distinct labels ε and ϕ in RHS such that $\beta=\alpha*\varepsilon$ and $\beta=\alpha*\phi$. But this implies $\varepsilon=\phi$ which, by the nature of labeled rules, is impossible. Hence both A and B are contained in G, and hence in E. Since E is proper, A=B.

Case 4. None of A and B is a subterm of H. Hence, there exist terms A1 and B1 in E such that A is obtained by replacing all occurrences of G in A1 by H and B is obtained from B1 similarly. Since $A\neq H$, $B\neq H$, the outermost label of A1 and B1 is also β. Since E is proper A1=B1. Hence A=B. **QED.**

Lemma 10. Let P be an LDF* program. Let a proper term E reduce to F in an NA-step. Then F is a proper term.

Proof. By Lemmas 8 and 9, F is a proper term. **QED.**

Lemma 11. Let P be an LDF* program. Let E0 be a proper term. Let E0,E1,..,Ek be an NA-derivation. Then, in this reduction, an FL-pair is eliminated at most once.

Proof. By induction on length k of E0,E1,..,Ek. If k=0, then clear. Otherwise, assume the theorem for E1,..,Ek. Let select(E0,G), G=>H and let E1 be obtained by replacing all occurrences of G in E0 by H. Let the outermost function symbol of G be f and its outermost label be α. Hence, $\langle f,\alpha\rangle$ is the pair eliminated in the reduction E0,E1.

If we can show that there is no term $f(\alpha,t1,..,tm)$ in E1,..,Ek, then, by induction, hypothesis, we can conclude that no FL-pair is eliminated more than once in E0,E1,..,Ek. To show this, it is sufficient to show that the label α never occurs in E1,..,Ek. Since E0 is proper, and $<f,\alpha>$ is eliminated, α does not occur in E1. Let ξ be a label in E2,..,Ek. Then, either $\xi=\beta$ for some label β in E1 in which case $\xi\neq\alpha$. Otherwise, $\xi=\beta*\varepsilon$ for some label β in E1 and label ε. We show that it is not possible that $\beta*\varepsilon=\alpha$.

Case 1. β occurs in E0. Since E0 is proper, β is not a proper initial segment of α. Hence, it is not possible that $\beta*\varepsilon=\alpha$.

Case 2. β does not occur in E0. Then, by the nature of LDF* rules, $\beta=\alpha*\delta$, for some label δ. Hence, $\beta*\varepsilon$ is longer than α, and so $\beta*\varepsilon\neq\alpha$. **QED.**

Theorem 2. Minimality of LDF*. Let P be an LDF* program. Let E0 be a proper term. Let E0,E1,..,Ek be a successful reduction. Then there exists a successful leftmost NA-derivation E0,F1,..,Fm such that $m=<k$.

Proof. Since E0,E1,..,Ek is a successful reduction, by reduction-completeness for F*, Theorem 2, Chapter III, there exists a successful N-reduction E0,G1,..,Gn. Let the E-set of E0,E1,..,Ek be S1 and that of E0,G1,..,Gn be S2. Then, by Lemma 6, S2 is a subset of S1. By directionality of DF*, there exists a successful leftmost N-reduction E0,H1,..,Hn. By Lemma 7, its E-set is also S2.

By Theorem 1 above, there exists a successful leftmost NA-derivation E0,F1,..,Fm whose E-set, S3, is a subset of S2. Hence S3 is a subset of S1. By Lemma 11, the size of S3 is m. The size of S1 is a lower bound on the number of steps in E0,E1,..,Ek. Hence m=<k. **QED.**

4.0 EXTENSION OF MINIMALITY RESULT TO NORMAL FORMS

We have shown that leftmost NA-derivations reduce proper terms to simplified forms in a minimum number of steps. It appears to be straightforward to extend this result to normal forms.

E reduces to F in an NAR-step if select-r(E,p), p=>q and F is the result of replacing each occurrence of p in E by q. Definitions of NAR-derivations and leftmost NAR-derivations are the obvious ones. The proof that leftmost NAR-reductions reduce proper terms to normal forms in a minimum number of steps appears to be very similar to the above proof.

5.0 DERIVED MINIMALITY OF DF*

Lemma 12. Let P be a DF* program and E0 a term, where both P and E0 are ordinary. Let P* and E0* be, respectively, their labeled versions. Let $E0\text{-}_{P}{>}E1$. Then there exists E1* such that $E0^*\text{-}_{P^*}{>}E1^*$ and E1* is a labeled version of E1.

Proof. There exist G,H such that E1=E0[G/H]. Proceed by induction on length of E0. If E0 is a 0-ary function symbol g, then clear.

Otherwise, E0=f(t1,..,tm), m>0. Then E0*=f*(α,t1*,..,tm*) where Σ(f)=f* and for each i, 0=<i=<m, ti* is a labeled version of ti. Assume lemma for each of t1,..,tm.

Suppose G occurs in some ti in t1,..,tm, di=ti[G/H] and E1=f(t1,..,ti-1,di,ti+1,..,tm). Then, by induction hypothesis, there exists di* such that ti*->di* and di* is a labeled version of di. Let E1*=f*(α,t1*,..,ti-1*,di*,ti+1*,..,tm*). Clearly, E1* is a labeled version of E1.

Suppose G=E0. Then there is a rule f(L1,..,Lm)=>RHS such that E0 matches f(L1,..,Lm) with some substitution Φ and H=RHSΦ. Let a labeled version of this rule be f*(L,L1*,..,Lm*)=>RHS*. It is easily verified that E0* matches the head of this rule with substitution Φ* such that <L,α> is in Φ* and for each pair <X,t> in Φ, the pair <X,t*> is in Φ* where t* is a labeled version of t. It is also easily verified that RHS*Φ* is a labeled version of RHSΦ. **QED.**

Theorem 3. Derived minimality for DF*. Let P be a DF* program and E0 a term, where both P and E0 are ordinary. Let P* and E0* be, respectively, their labeled versions such that E0* is proper. Let E0,E1,..,Ek be a successful reduction in P. Then there exists a successful NA-derivation E0*,F1*,..,Fp* in P* such that p=<k.

Proof. We can ensure that E0* is a labeled version of E0 which is proper, simply by choosing distinct maximal labels for function symbols of E0. By Lemma 12, there exists a successful reduction E0*,E1*,..,Ek* such that for each i, Ei* is a labeled version of Ei. Since P* is an LDF* program, and E0* is proper, by Theorem 2, there exists a successful NA-derivation E0*,F1*,..,Fp* such that p=<k. **QED.**

CHAPTER VI

COMPILATION OF F* INTO HORN CLAUSES

1.0 INTRODUCTION

A very simple algorithm is described, which compiles F* programs into Horn clauses in such a way that when SLD-resolution interprets them, it directly simulates the behavior of select. This is accomplished by compiling each F* rule into a distinct Horn clause, and combining in that clause, information about the logic of the rule, and information about the control of select when interpreting that rule. Thus, a specialized interpreter is produced for each rule.

If the F* program satisfies restriction (g) in Chapter IV, Section 2.0, the clauses resulting from its translation can be transformed to eliminate all redundant backtracking. If the program also satisfies restriction (f), i.e. is in DF*, SLD-search trees automatically contain exactly one branch. All the time however, only *pure* clauses are produced.

The nature of logical variables is utilized to implement the assumption necessary for minimality. This is that when a term is reduced, all copies of it are simultaneously reduced. A logical variable has the property that when one occurrence of it in a term is bound to some term, all occurrences of it are simultaneously bound to the same term. Unfortunately, use must now be made of a metalogical feature (var), and an extra logical feature (cut). This is the *only* impure aspect in the entire LOG(F) system. Consequently, SLD-resolution, augmented with these features, computes NA-reductions.

LOG(F) is defined to be a logic programming system augmented with an F* compiler, and the equality axiom X=X. A ready-made implementation of LOG(F) is obtained by implementing the F* compiler in Prolog and using Prolog in place of SLD-resolution. Due to its depth-first search strategy, Prolog may sometimes not be able to simplify terms, even though select would. *However, if P is in DF*, Prolog always simplifies terms whenever select does.*

In all of the following, except in Section 6.0, Prolog clauses and Prolog are synonymous with Horn clauses and SLD-resolution. Only in Section 6.0 do they refer to the *programming language*. An implementation of the compiler in Prolog is listed in APPENDICES 1-2.

2.0 COMPILATION ALGORITHM

Let P be an F* program. The compilation of P into Prolog proceeds in two stages.

Stage 1. For each n-ary, n>=0, constructor symbol c in P, and where X1,..,Xn are distinct variables, generate the clause:

reduce(c(X1,..,Xn),c(X1,..,Xn))

Stage 2. Let f(L1,..,Lm)=>RHS be a rule in P where f is an m-ary, m>=0, non-constructor function symbol and each of RHS and L1,..,Lm is a term, possibly containing variables. For each such rule perform the following steps:

(a) Let A1,..,Am be distinct Prolog variables none of which occur in the rule.

If Li is a variable let Qi be Ai=Li. If Li is c(X1,..,Xn) where c is a constructor symbol, and each Xi a variable, let Qi be reduce(Ai,c(X1,..,Xn)).

(b) Let Out be a Prolog variable not occurring in the rule, and different from A1,..,Am. Generate the predication reduce(RHS,Out).

(c) Generate the clause:

```
reduce(f(A1,..,Am),Out):-Q1,..,Qm,reduce(RHS,Out).
```

For example the F* rules:

```
append([],X)=>X
append([U|V],W)=>[U|append(V,W)]
intfrom(N)=>[N|intfrom(s(N))].
if(true,X,Y)=>X.
if(false,X,Y)=>Y.
```

are compiled into:

```
reduce([],[]).
reduce([U|V],[U|V]).
reduce(true,true).
reduce(false,false).

reduce(append(A1,A2),Out):-reduce(A1,[]),A2=X,reduce(X,Out).
```

```
reduce(append(A1,A2),Out):-
        reduce(A1,[U|V]),A2=W,reduce([U|append(V,W)],Out).
reduce(intfrom(N),Out):-reduce([N|intfrom(s(N))],Out).
reduce(if(T,X,Y),Out):-reduce(T,true),reduce(X,Out).
reduce(if(T,X,Y),Out):-reduce(T,false),reduce(Y,Out).
```

It can be seen that where reduce(f(A1,..,Am),Out):-Q1,..,Qm,reduce(RHS,Out) is the translation of f(L1,..,Lm)=>RHS, Q1,..,Qm represent the attempt to match some term f(t1,..,tm) with f(L1,..,Lm). If these succeed, the match succeeds with some substitution α. Now, reduce(RHS,Out) represents simultaneously, application of α to RHS, and recursive simplification of RHSα. The correctness of compilation algorithm is formally proved in Section 7.0

In practice, in stage 2(a) if Li is a variable, then Ai in f(A1,..,Am) is replaced by Li, and Ai=Li is not generated. This eliminates a procedure call, and so yields substantially faster code. However, proofs of propositions below are easier to derive without this optimization.

3.0 COMPUTING AND PRINTING NORMAL FORMS

If there is a method to compute simplified forms of terms, it can be applied repeatedly to compute normal forms of terms. *This is guranteed by reduction-completeness for normal forms, Theorem 4, Chapter III.* In particular, for each m-ary constructor symbol we can add the following rule:

```
nf(E,c(X1,..,Xm)):-reduce(E,c(T1,..,Tm)),nf(T1,X1),..,nf(Tm,Xm).
```

Now, to compute the normal form of a term E, we can execute nf(E,X), where X is a variable. The correctness of this rule for computing normal forms can easily be proved from the arguments of Section 7.0.

Clearly, computing normal forms is only sensible when they are finite. If they are not, we can at least print finite portions of them as they are generated. For example, we can print members of an infinite list as follows:

```
print_list(X):-reduce(X,[U|V]),write(U),write(' '),print_list(V).
```

4.0 OPTIMIZING RULES SATISFYING RESTRICTION (g)

Let P be an F* program and PC its compiled version. Let f(t11,..,t1i,..,t1m)=>RHS1, .., f(tn1,..,tni,..,tnm)=>RHSn be the n rules defining f in P, and C1,..,Cn be, respectively, their compiled versions. Let the rules satisfy restriction (g), Chapter IV, Section 2.0. Then, if t1i is a variable, the ith literal in bodies of C1,..,Cn will be, respectively, Ai=t1i,..,Ai=tni, for some variable Ai. Otherwise, the ith literals in C1,..,Cn would be, respectively, reduce(Ai,t1i),..,reduce(Ai,tni).

If t1i is not a variable, the query reduce(f(a1,..,ai,..,am),Z) may, due to backtracking, cause evaluation of each of reduce(ai,t1i),..,reduce(ai,tni). We can ensure that reduce is called just once for ai by taking advantage of the fact that all reduce clauses have the same form. That is, we can collapse them all into the single clause:

```
reduce(f(A1,..,Am),Z):-R1,..,Rm,f(X1,..,Xm)=>RHS,reduce(RHS,Z).
```

where X1,..,Xm are distinct variables not occurring in any of the clauses, and if t1i is a variable, Ri is Ai=Xi, otherwise Ri is reduce(Ai,Xi). Now reduce would be called just once for ai. Of course, the => rules now need to be included with the reduce clauses. Thus Prolog execution can be considerably speeded up.

Furthermore, if P is a DF* program then f(X1,..,Xm)=>RHS will succeed at most once. Hence, for any ground terms t1,..,tm, and variable Z, the search tree rooted at reduce(f(t1,..,tm),Z) will contain exactly one branch. *Thus, the reduce clauses would form a deterministic logic program.* For example, consider the DF* program:

```
append([],X)=>X.
append([U|V],W)=>[U|append(V,W)].
```

Its compiled version, excluding rules for constructor symbols, is:

```
reduce(append(A1,A2),Z):-reduce(A1,[]),A2=X,reduce(X,Z).
reduce(append(A1,A2),Z):-
        reduce(A1,[U|V]),A2=W,reduce([U|append(V,W)],Out).
```

These two rules can be collapsed into a single one:

```
reduce(append(A1,A2),Z):-
        reduce(A1,X1),A2=X2,append(X1,X2)=>RHS,reduce(RHS,Z).
```

Now, given the query reduce(append([1],[2]),Z), an attempt would be made to simplify [1] just once, and not twice, as with the original pair of reduce clauses. Also, since the append rules are in DF*, the SLD-search tree rooted at reduce(append([1],[2]),Z) contains exactly one branch.

5.0 COMPUTING FUNCTIONS EAGERLY IN F*

If a function is defined in F*, it is computed lazily. Often it is very desirable that some functions, such as arithmetic functions, be computed eagerly. We show one way to accomplish this.

A lazy function symbol is one which is defined in F*. An eager function symbol is one which is defined in Prolog. Only right hand sides of F* rules can contain calls to eager functions. Let E be a subterm, possibly containing variables, of the right hand side of an F* rule. Let the outermost function symbol of E be eager. Then E must not contain any lazy function symbol. For example, where length is eager, and append is lazy, the term length(append([],[1])) must not appear in any F* rule.

Now, let LHS=>RHS be an F* rule, f an eager function, and f(t1,..,tn) a subterm, possibly containing variables, of RHS. Let f be defined by an n+1 ary predicate symbol p(A1,..,An,A), such that A1,..,An are input positions and A the output position. Let RHS1 be the result of replacing f(t1,..,tn) in RHS by X, where X is a variable not occurring in LHS=>RHS. Generate the condition p(t1,..,tn,X), and add it to the conditions generated in Stage 2 (a) of Section 2.0. Of course, if t1,..,tn themselves involve calls to eager functions, they must be treated similarly. For example, let multiple be an eager function defined in Prolog as follows:

```
multiple(A,B,true):-0 is A mod B.
```

```
multiple(A,B,false):-not(0 is A mod B).
```

Now the rule:

```
filter(A,[U|V])=>if(multiple(U,A),filter(A,V),[U|filter(A,V)]).
```

is compiled into:

```
reduce(filter(A,X),Z):-
        reduce(X,[U|V]),
        multiple(U,A,T),
        reduce(if(T,filter(A,V),[U|filter(A,V)]),Z).
```

However, some care still needs to be exercised. For example, where zerop and / are eager functions, defined in Prolog by, respectively, zerop and div, the rule:

```
f(X)=>if(zerop(X),[X],[1/X]).
```

will be compiled into:

```
reduce(f(X),Z):-zerop(X,T),div(1,X,A),reduce(if(T,[X],[A]),Z).
```

Now, if X is 0, the call to div will cause an unintended division by 0. At present, the F* compiler, listed in APPENDICES 1-2, does not guard against such a possibility, so one has to rewrite the above rule as:

f(X)=>if(zerop(X),[X],h(X)).

h(X)=>[1/X].

6.0 COMPILING LDF* PROGRAMS

We now show how to represent labeled terms in Prolog, and compile LDF* programs into Prolog in such a way that NA-steps can be performed efficiently. The main idea is that labels can be represented by logical variables. These have the property that if one occurrence of a variable in term E is bound to term F, all occurrences of the variable in E are simultaneously bound to F.

Let E be a proper term and let E reduce to F in an NA-step. Then there is a subterm G of E such that G=>H, and F is obtained by replacing all occurrences of G in E by H. Note that each of G,H,F is proper. Let E contain the labels α1,..,αn. Let V1,..,Vn be distinct variables and E* the result of replacing for each i, all occurrences of αi in E by Vi. Then E* is a Prolog representation of E. Similarly, let G*,H*,F* be Prolog representations of G,H,F respectively such that H* and E* do not have any variables in common. Then G*=f(V,t1,..,tm) where V is a variable. If we now bind V to H*, all occurrences of V in E* are bound to H*. Let the result be F1*.

Now, before attempting to match a term with a non-variable term, we take the precaution of checking whether its label is already bound to some term. If so, we attempt to match this term with the non-variable term. Otherwise, we proceed as usual. Thus, after V has been bound to H*, if another occurrence of G* is to be matched with some term, we attempt to match H* with it.

At a later stage it is possible that the label of H* itself be bound to a term. Thus, before matching a term, it may be necessary to "dereference" its label a number of times. It is not unreasonable to assume that the cost of dereferencing is small compared to that of reduction. *In the next section we show how the length of the dereferencing chain can be made exactly one.* Thus, we can work with F1* instead of F*, so replacement of all occurrences of a term is implemented efficiently. Moreover, F* can be obtained from F1* by dereferencing.

The algorithm for compiling LDF* programs can now be given. It proceeds in three stages.

Stage 1. For each n+1-ary constructor symbol c in P, and where L,X1,..,Xn are distinct variables, generate the clause:

reduce(c(L,X1,..,Xn),c(L,X1,..,Xn))

Stage 2. Let f(L,L1,..,Lm)=>RHS be a rule in P where f is an m+1-ary, m>=0, non-constructor function symbol and each of L1,..,Lm, and RHS is a term, possibly containing variables. For each such rule perform the following steps:

(a) Let A,A1,..,Am be distinct Prolog variables none of which occur in the rule. If Li is a variable let Qi be Ai=Li. If Li is c(Ki,X1,..,Xn) where c is a constructor symbol and each of Ki,X1,..,Xn a variable, let Qi be reduce(Ai,c(Ki,X1,..,Xn)).

(b) Let P1,P2,..,Pk be all terms in RHS of the form L*β where β is a label.

Let V1,...,Vk be distinct variables, distinct from any variables in the rule, and from A,A1,..,Am. Let RHS1 be obtained from RHS by replacing each Pi by Vi in RHS.

(c) Let Out be a Prolog variable not occurring in the rule, distinct from A,A1,..,Am and from V1,..,Vk. Generate the predication reduce(RHS1,Out).

(d) Generate the clause:

reduce(f(A,A1,...,Am),Out):-Q1,..,Qm,A=RHS1,reduce(RHS1,Out).

Stage 3. Before any reduce rules for f, add the clause:

reduce(f(A,A1,..,Am),Z):-not var(A),reduce(A,Z),!.

The literal A=RHS1 ensures that all occurrences of A in the term of which f(A,A1,..,Am) is a subterm, are replaced by RHS1. The dereferencing is performed by the above clause.

Since Prolog evaluates literals from left to right, it automatically computes leftmost N-reductions. Since variables are indivisible, all labels in Prolog representations of labeled terms are maximal. Also, upon each procedure entry, Prolog instantiates V1,..,Vk to variables not occurring previously in the deduction. These facts help to ensure that proper terms are reduced to proper terms and that a label is eliminated at most once. Hence Prolog also simplifies terms in a minimum number of NA-steps.

It will be recognized that our scheme for implementing NA-derivations is exactly the graph-reduction scheme with indirection nodes described in [Turner 1979] and [O'Donnell 1982]. In practice, DF* programs can be compiled directly into reduce clauses with labels, without first transforming them into LDF* programs. The appropriate algorithm can easily be worked out. For example, where nil is a zero-ary constructor symbol, let P be the following DF* program:

```
merge(nil,nil)=>nil.
double(X)=>merge(X,X).
h=>d.
```

This is compiled into:

```
(1) reduce(merge(V,A1,A2),Z):-not var(V) ,reduce(V,Z),!.
(2) reduce(double(V,A1),Z):-not var(V),reduce(V,Z),!.
(3) reduce(h(V),Z):-not var(V),reduce(V,Z),!.

(4) reduce(nil(N),nil(N)).
(5) reduce(merge(V,A1,A2),Z):-
        reduce(A1,nil(N1)),
        reduce(A2,nil(N2)),
        V=nil(N3),
        reduce(nil(N3),Z).
(6) reduce(double(V,A1),Z):-
        V=merge(N,A1,A1),reduce(merge(N,A1,A1),Z).
```

(7) reduce(h(V),Z):-V=d(D),reduce(d(D),Z).

Now, consider the evaluation of the query reduce(double(A,h(B)),Z). This yields the query reduce(merge(N,h(B),h(B)),Z). Suppose the first call, reduce(h(B),nil(N1)), in (5) succeeds, but only after a long and complicated deduction. Then B is bound to d(D), and the result of dereferencing D is nil(N1). Now, due to (3), the second call, reduce(h(B),nil(N2)), in (5) will perform a sequence of dereferencing steps starting at B, and infer that h(B) is reducible to nil(N1). The cut (!) will prevent (7) from being tried all over again.

6.1 Keeping length of dereferencing chain constant.

Consider the clause added in Stage 2 above:

```
reduce(f(V,A1,..,Am),Z):-Q1*,..,Qm*,V=RHS1,reduce(RHS1,Z).
```

Instead of it, if we add:

```
reduce(f(Z,A1,..,Am),Z):-Q1*,..,Qm*,reduce(RHS1,Z).
```

then, whenever Z is bound, it is always to a simplified form. Thus, dereferencing can terminate in just one step, instead of in several steps, as before. For example, the program:

```
merge(nil,nil)=>nil.
double(X)=>merge(X,X).
```

```
h=>d.
```

is compiled into:

```
(1) reduce(merge(V,A1,A2),Z):-not var(V),reduce(V,Z),!.
(2) reduce(double(V,A1),Z):-not var(V),reduce(V,Z),!.
(3) reduce(h(V),Z):-not var(V),reduce(V,Z),!.

(4) reduce(nil(N),nil(N)).
(5) reduce(merge(Z,A1,A2),Z):-
        reduce(A1,nil(N1)),reduce(A2,nil(N2)),reduce(nil(N3),Z).
(6) reduce(double(Z,A1),Z):-reduce(merge(N,A1,A1),Z).
(7) reduce(h(Z),Z):-reduce(d(D),Z).
```

Consider again the query reduce(double(A,h(B)),Z), which yields reduce(merge(N,h(B),h(B)),Z). After the call reduce(h(B),nil(N1)) in (5) succeeds, B is bound directly to nil(N1), not to d(D). Now, reduce(h(B),nil(N2)) terminates in just three inference steps, of which just one is a dereferencing step.

7.0 CORRECTNESS OF F* COMPILATION ALGORITHM

Lemma 1. Let P be an F* program. If:

(1) E0=f(t1,..,ti,..,tm), and

(2) Ek=f(s1,..,si,..,sm), and

(3) si is simplified, and

(4) E0,..,Ek, k>=0, is an N-reduction such that for no i, Ei=>Ei+1.

Then there is a successful N-reduction ti,..,si of length less than or equal to the length k of E0,E1,..,Ek.

Proof: By Lemma 8, Chapter IV. **QED.**

Lemma 2. Let P be an F* program, and PC its compiled version. Let A be a ground term and B a term, possibly containing variables, such that reduce(A,B) succeeds, in the sense of SLD-resolution, with answer substitution σ. Then Bσ is ground.

Proof: By induction on length n of successful SLD-derivation reduce(A,B),G1,..,Gn=□. If n=1 then A=c(t1,..,tm), c a constructor symbol each ti a term, m>=0. The query reduce(A,B) will succeed by unifying with the head of the clause reduce(c(X1,..,Xm),c(X1,..,Xm)). The answer substitution σ will be such that Bσ=A. Clearly Bσ is ground.

Assume lemma for successful SLD-derivations of length less than n. Let the successsful derivation starting at reduce(A,B) be of length n, n>1. Then A=f(t1,..,tm), m>=0, where f is a function symbol, but not a constructor symbol, and each ti is a ground term. Then there is a clause:

reduce(f(X1,..,Xm),Z):-Q$\cup${reduce(RHS,Z)}.

such that it is the compilation of a rule f(L1,..,Lm)=>RHS. Now, reduce(f(t1,..,tm),B) unifies with the head of this clause with some m.g.u. τ and its immediate descendant (Q$\cup${reduce(RHS,Z)})τ has a successful SLD-derivation of length n-1. Clearly, τ={<X1,t1>,..,<Xm,tm>,<Z,B>} and so Zτ=B. Also, since RHS does not contain any of the Xi, RHSτ=RHS.

Let Q1,..,Qm, m>=0, be the members of Q. If Qi is Xi=Li then Qiτ=(ti=Li) and succeeds with answer substitution σi={<Li,ti>}. If Qi is reduce(Xi,Li) then Qiτ=reduce(ti,Li) and has a successful SLD-derivation of length less than or equal to n-1. Hence, by induction hypothesis, Qiτ succeeds with answer substitution σi such that Liσi is ground.

By restriction (e) all variables of RHS occur in L1,..,Lm. Hence, since each Liσi is ground, RHS$\tau\sigma$1,..,σm is ground. Already Zτ=B. Since B does not contain any variables in L1,..,Lm, Bσ1,..,σm=B. Hence reduce(RHS,Z)$\tau\sigma$1,..,σm=reduce(RHSσ1,..,σm,B). By induction hypothesis, this succeeds with answer substitution σ such that Bσ is ground. So, reduce(A,B) succeeds with answer substitution σ such that Bσ is ground. **QED.**

Lemma 3. Let P be an F* program and PC its compiled version. Let A and B be ground terms such that reduce(A,B) succeeds. Let D be a term, possibly containing variables, such that for some substitution α, $D\alpha=B$. Then reduce(A,D) succeeds with answer substitution α.

Proof: By induction on length n of successful SLD-derivation starting at reduce(A,B). If n=1 then A=B=c(t1,..,tm), c a constructor symbol each ti a term, m>=0. The query reduce(A,D) will succeed with answer substitution which is the m.g.u. of B and D. Since B is ground this m.g.u. is α.

Let the successful derivation starting at reduce(A,B) be of length n, n>1. Assume lemma for successful derivations of length less than n. Then A=f(t1,..,tm) where f is a function symbol, but not a constructor symbol, and each ti is a term. Then there is a clause:

$$\text{reduce(f(X1,..,Xm),Z)}:\!-Q\cup\{\text{reduce(RHS,Z)}\}$$

which is the translation of some rule in P. Also, reduce(f(t1,..,tm),B) unifies with the head of this clause with some m.g.u. $\tau=\{<X1,t1>,..,<Xn,tn>,<Z,B>\}$ and its immediate descendant is $(Q\cup\{\text{reduce(RHS,Z)}\})\tau$. Since RHS does not contain any of the Xi, this is $Q\tau\cup\{\text{reduce(RHS,B)}\}$. It has a successful derivation of length n-1.

If Q is empty, by restriction (e) $RHS\tau$ is ground. Otherwise let Q1,..,Qm be the members of Q. Consider some Qi. If Qi is Xi=Li, then $Qi\tau$=(ti=Li) which succeeds with answer substitution $\sigma i=\{<Li,ti>\}$. Otherwise Qi=reduce(Xi,Li), so $Qi\tau$=reduce(ti,Li). By Lemma 2, reduce(ti,Li) succeeds with answer substitution σi such that $Li\sigma i$ is ground. Since all the variables of RHS are in L1,..,Lm, $RHS\sigma 1,..,\sigma m$ is again ground.

Since reduce(RHSσ1,..,σm,B) succeeds, by induction hypothesis, reduce(RHSσ1,..,σm,D) succeeds with answer substitution α. Now consider the query reduce(A,D). Again, by reasoning as above, reduce(RHSσ1..σm,D) appears in an SLD-derivation of reduce(A,D). Hence reduce(A,D) also succeeds with answer substitution α. **QED**.

Lemma 4. Let P be an F* program. Let PC be the compiled version of P. Let E0,..,En be a successful N-reduction. Then reduce(E0,En) succeeds in the presence of PC.

Plan of Proof: By induction on length of successful N-reduction E0,..,En. We show that there is some Ej, j>0, in E0,..,En such that an SLD-derivation of reduce(E0,En) contains the goal reduce(Ej,En). Since Ej,..,En is also a successful N-reduction, by induction hypothesis, reduce(Ej,En) succeeds. Hence reduce(E0,En) succeeds.

Proof: By induction on length n of successful reduction E0,..,En. If n=0 then E0 is already simplified. In particular, E0=c(t1,..,tm) where c is an m-ary constructor symbol, m>=0, and t1,..,tm are terms. There is a clause in PC reduce(c(X1,..,Xm),c(X1,..,Xm)) where each Xi is a variable. Clearly reduce(E0,E0) succeeds.

Let n>0 and E0=f(t1,..,tm), f not a constructor symbol, each ti a term and m>=0. Assume theorem holds for all successful reductions of length less than n.

Since E0 is not simplified, the N-reduction is of the form E0,..,Ek-1,Ek,..,En, 0<k=<n, such that Ek-1=>Ek, but for no i, 0=<i<k-1, Ei=>Ei+1. Hence, Ek-1=f(s1,..,sm) for some terms s1,..,sm. Since Ek-1=>Ek, there is some rule f(L1,..,Lm)=>RHS such that

Ek-1 matches f(L1,..,Lm) with some substitution σ and Ek=RHSσ. Since L1,..,Lm do not share any variables, σ is the union of $\sigma1,..,\sigma m$ such that for each Li in L1,..,Lm, si matches Li with substitution σi.

For each i, if Li is not a variable, then since si matches Li, si is in simplified form. For such i, there is, by Lemma 1, a successful N-reduction ti,..,si of length less than or equal to k-1.

The rule f(L1,..,Lm)=>RHS is compiled into the Horn clause

reduce(f(X1,..,Xm),Z):- $Q\cup\{$reduce(RHS,Z)$\}$

in accordance with the compilation rules stated above. This clause is contained in PC.

Consider the query reduce(E0,En), i.e. reduce(f(t1,..,tm),En). It unifies with reduce(f(X1,..,Xm),En) with m.g.u. $\tau=\{\langle X1,t1\rangle,..,\langle Xm,tm\rangle,\langle Z,En\rangle\}$ and its immediate descendant is $(Q\cup\{\text{reduce(RHS,Z)}\})\tau$. Since RHS does not contain any of the Xi, this is $Q\tau\cup\{\text{reduce(RHS,En)}\}$.

Let Q1,..,Qm be the members of Q. Consider some Qi. If Qi is Xi=Li, then $Qi\tau$=(ti=Li) which succeeds with answer substitution $\{\langle Li,ti\rangle\}$. Of course, ti matches Li, so $\{\langle Li,ti\rangle\}=\sigma i$.

Otherwise, Qi=reduce(Xi,Li), so $Qi\tau$=reduce(ti,Li). Since there is a successful N-reduction ti,..,si of length less than or equal to k-1, by induction hypothesis, reduce(ti,si) succeeds. Since $Li\sigma i$=si, by Lemma 3, reduce(ti,Li) also succeeds with answer substitution σi.

By repeating the same argument for each Qi, we see that an SLD-derivation starting at reduce(E0,En) contains reduce(RHSσ1,..,σm,En) as a member. Since σ is the union of σ1,..,σm and no variable is defined in more than one σi in σ1,..,σm, RHSσ1,..,σm=RHSσ. But RHSσ=Ek. Hence the SLD-derivation starting at reduce(E0,En) contains reduce(Ek,En). Since the length of the successful reduction Ek,..,En is less than n, by induction hypothesis, reduce(Ek,En) succeeds. Thus, the query reduce(E0,En) succeeds. **QED**.

Lemma 5. Let P be an F* program. Let PC be the compiled version of P. Let E0 and En be terms such that reduce(E0,En) succeeds in the presence of PC. Then there is a successful N-reduction E0,..,En.

Plan of Proof: By induction on length of successful SLD-derivation reduce(E0,En),..,□. We show that there is some goal reduce(Ej,En), j>0, in this derivation such that there is an N-reduction E0,..,Ej. Since reduce(Ej,En) succeeds, by induction hypothesis, there is a successful N-reduction Ej,..,En. So there is a successful N-reduction E0,..,Ej,..,En.

Proof: By induction on length n of successful SLD-derivation starting at reduce(E0,En). If n=1 then there is a clause reduce(c(X1,..,Xm),c(X1,..,Xm)) in PC such that reduce(E0,En) unifies with the head of this clause. Clearly, then, E0=En, En is simplified and the required N-reduction is simply E0.

Let n>0. Assume lemma for all successful derivations of length less than n. Assume

E0=f(t1,..,tm) for some non-constructor function symbol f and terms t1,..,tm. Since reduce(E0,En) succeeds there is a clause in PC:

reduce(f(X1,..,Xm),Z):-Q∪{reduce(RHS,Z)}

such that it is the compilation of a rule f(L1,..,Lm)=>RHS in P. Moreover, reduce(f(t1,..,tm),En) unifies with the head of the above clause with m.g.u. τ={<X1,t1>,..,<Xm,tm>,<Z,En>} and Qτ U {reduce(RHS,Z)}τ has a successful derivation of length n-1. Also, RHSτ=RHS and Zτ=En.

If Q is empty, m=0. So, by restriction (e) RHS is ground. By induction hypothesis there is a successful N-reduction RHS,..,En. E0 matches f(L1,..,Lm) and so E0=>RHS. Hence E0,RHS,..,En is a successful N-reduction.

Suppose Q is non-empty. Let Q1,..,Qm be the members of Q. Consider Qi. If Qi=(Xi=Li) then ti unifies with Li with substitution σi={<Li,ti>}. Construct the singleton sequence f(t1,..,ti,..,tm). This sequence is an N-reduction.

If Qi=reduce(Xi,Li) then Li=c(U1,..,Uk) for some constructor symbol c and variables U1,..,Uk. Also Qiτ=reduce(ti,Li). Clearly, reduce(ti,Li) succeeds. Let the answer substitution be σi. By Lemma 2, Liσi is ground. Then reduce(ti,Liσi) also succeeds. The successful derivation of reduce(ti,Liσi) is the same as that of reduce(ti,Li) with Li replaced by Liσi. So, the length of this derivation is also less than n. By induction hypothesis, there is a successful N-reduction ti,..,Liσi. By Lemma 4 of Chapter III, the sequence f(t1,..,ti,..,tm),..,f(t1,..,Liσi,..,tm) is an N-reduction.

Hence we obtain the N-reductions f(t1,..,tm),..,f(L1σ1,..,tm) and f(L1σ1,t2,..,tm),..,f(L1σ1,L2σ2,..,sm) and .. f(L1σ1,L2σ2,..,tm),..,f(L1σ1,L2σ2,..,Lmσm). The concatenation of these reductions is itself an N-reduction. Since L1,..,Lm do not share variables, f(L1σ1,..,Lmσm) matches f(L1,..,Lm) with a substitution which is the union of σ1,..,σm. Let σ be this union. Hence f(L1σ1,..,Lmσm)=>RHSσ. Since all the variables of RHS are in L1,..,Lm and for each σi, Liσi is ground, RHSσ is ground.

The predication reduce(RHSσ,En) succeeds and the length of the associated successful derivation is less than n. By induction hypothesis, there is a successful N-reduction RHSσ,..,En. Hence there is a successful N-reduction f(t1,..,tn),..,f(L1σ1,..,Lmσm),RHSσ,..,En. **QED.**

Theorem 1. The correctness of the compilation of F*. Let P be an F* program and PC be its compilation. Let E0 and En be ground terms. Then there is a successful N-reduction beginning with E0 and ending with En iff PC|-reduce(E0,En).

Proof: Lemmas 4 and 5 state, respectively, the if and only if parts of the theorem. By their proofs, we obtain the proof of the theorem. **QED.**

CHAPTER VII

PROGRAMMING IN LOG(F)

1.0 INTRODUCTION

This chapter describes six examples of programming in LOG(F). The first illustrates non-determinism of LOG(F), and usefulness of lazy evaluation even when manipulating finite data structures. The second shows how useful cases of the rule of substitution of equals for equals can be implemented. The third obtains a new proof of confluence of combinatory logic. The fourth shows how a pair of communicating processes can be simulated. The fifth illustrates the power of NA-derivations, and manipulation of infinite numerical structures. The sixth illustrates manipulation of infinite graphical structures. In each case, clauses listed are those obtained after performing optimizations discussed in previous chapters.

2.0 NON-DETERMINISM IN F*

As discussed in Chapter I, permutations of lists can be computed by the following F* program:

```
perm([])=>[].
perm([U|V])=>insert(U,perm(V)).
insert(U,X)=>[U|X].
insert(U,[A|B])=>[A|insert(U,B)].
```

This is compiled, and optimized into:

```
reduce([],[]).
reduce([A|B],[A|B]).

reduce(insert(A,B),[A|B]).
reduce(insert(A,B),[C|insert(A,D)]):-reduce(B,[C|D]).
reduce(perm(A),B):-reduce(A,C),perm(C)=>D,reduce(D,B).

perm([])=>[].
perm([A|B])=>insert(A,perm(B)).
```

Note that some => rules survive in the compiled version. This is due to the method, discussed in Section 5.0, Chapter VI, of compiling F* programs satisfying restriction (g). If we now type reduce(perm([1,2,3]),Z), we obtain Z=[1|perm([2,3])], Z=[2|insert(1,perm([3]))], Z=[3|insert(1,insert(2,perm([])))]. However, if we define:

```
make_list(X,[]):-reduce(X,[]).
make_list(X,[U|V]):-reduce(X,[U|B]),make_list(B,V).
```

and then type make_list(perm([1,2,3]),Z), we obtain Z=[1,2,3],..,Z=[3,2,1].

The above program can be used to implement a very efficient solution to the N-queens problem which is to place N queens on an NxN chess board so that no two queens attack each other. It is easily seen that each queen must be in a distinct row and column, so that candidates for solutions can be represented by permutations of the list [1,2,..,N]. The position of the ith queen in a permutation p is [i,q] where q is is the ith element of q. The problem now reduces to generating all permutations of [1,2,..,N] and testing whether they are safe, or represent a solution.

Lazy evaluation guarantees that permutations are tested as soon as they are generated. If it is determined that [A1,..,Am], m=<N is unsafe then no permutation with [A1,..,Am] as initial segment is generated. This yields a drastic pruning of the search space. For example, for a 15x15 chess board, a solution is found in about 32 cpu seconds in Quintus Prolog on a SUN-3/60. The number of permutations of [1,2,..,15] is over 1.3 trillion. The program is:

```
if(true,X,Y)=>X.
if(false,X,Y)=>Y.
queens(X)=>safe(perm(X)).
safe([])=>[].
safe([U|V])=>[U|safe(nodiagonal(U,V,1))].
nodiagonal(U,[],N)=>[].
nodiagonal(U,[A|B],N)=>if(noattack(U,A,N),[A|nodiagonal(U,B,N+1)],none).
noattack(U,A,N)=>neg(equal(abs(U-A),N)).
```

This is compiled into:

```
reduce([],[]).
reduce([U|V],[U|V]).
reduce(true,true).
reduce(false,false).

reduce(queens(A),B):-queens(A)=>C,reduce(C,B).
```

```
reduce(safe(A),B):-reduce(A,C),safe(C)=>D,reduce(D,B).

reduce(if(A,B,C),D):-reduce(A,E),if(E,B,C)=>F,reduce(F,D).

reduce(noattack(A,B,C),D):-noattack(A,B,C)=>E,reduce(E,D).

reduce(nodiagonal(A,B,C),D):-reduce(B,E),nodiagonal(A,E,C)=>F,reduce(F,D).


queens(A)=>safe(perm(A)).

safe([])=>[].

safe([A|B])=>[A|safe(nodiagonal(A,B,1))].

if(true,A,B)=>A.

if(false,A,B)=>B.

nodiagonal(A,[],B)=>[].

nodiagonal(A,[B|C],D)=>
        if(noattack(A,B,D),[B|nodiagonal(A,C,E)],none):-E is D+1.

noattack(A,B,C)=>D:-E is A-B,abs(E,F),equal(F,C,G),neg(G,D).
```

The eager functions are defined in Prolog:

```
abs(X,X):-X>=0.

abs(X,Y):-X<0,Y is -X.

neg(true,false).

neg(false,true).

equal(A,A,true).

equal(A,B,false):-not A=B.

less_than(U,A,true):-U<A.

less_than(U,A,false):-U>=A.
```

If we now type make_list(queens([1,2,3,4]),Z), we obtain Z=[2,4,1,3] and Z=[3,1,4,2].

3.0 IMPLEMENTING SUBSTITUTION OF EQUALS FOR EQUALS

If a DF* program is interpreted as an equality theory, reduce clauses can be thought of as implementing an equality theory in Prolog with the restriction that it be used only for simplification of terms. Now, given a clause of the form p(c(X1,..,Xm)):-Body, where c is a constructor symbol, we can add another clause stating a rule of substitution of equals:

```
p(X):-reduce(X,c(X1,..,Xm)),p(c(X1,..,Xm)).
```

Now, even when a term E is not of the form c(X1,..,Xm), p can still be inferred for E, provided E is reducible to a term of the form c(X1,..,Xm). For example, from:

```
married(X):-spouse(X,Y).
spouse(scott,a).
```

one can infer married(scott). One can now add the clause:

```
married(X):-reduce(X,Y),married(Y).
```

An equality theory is:

```
author(waverley)=>author(ivanhoe).
```

```
author(ivanhoe)=>scott.
```

The reduce clauses for the last two => rules are:

```
reduce(scott,scott).
reduce(ivanhoe,ivanhoe).
reduce(waverley,waverley).

reduce(author(X),Z):-reduce(X,waverley),reduce(author(ivanhoe),Z).
reduce(author(X),Z):-reduce(X,ivanhoe),reduce(scott,Z).
```

Here scott, waverley, and ivanhoe are constructor symbols. Now one can infer, in Prolog, married(author(waverley)), i.e. the result of substituting author(waverley) for scott in married(scott).

4.0 COMBINATORY LOGIC

A new proof is obtained of the theorem that the SKI calculus is confluent. Following the ideas of Ait-Kaci & Nasr [1986], SKI reduction rules can be expressed as a DF* program:

```
apply(k,X)=>k1(X).
apply(k1(X),Y)=>X.
apply(s,F)=>s1(F).
apply(s1(F),G)=>s2(F,G).
apply(s2(F,G),X)=>apply(apply(F,X),apply(G,X)).
```

Here k,s,k1,s1,s2 are constructor symbols, and apply a non-constructor symbol. From confluence of DF*, it follows that the SKI calculus is also confluent. These rules are translated into the following reduce clauses:

```
reduce(s,s).
reduce(k,k).
reduce(k1(X),k1(X)).
reduce(s1(X),s1(X)).
reduce(s2(X,Y),s2(X,Y)).

reduce(apply(A,B),Z):-reduce(A,k),reduce(k1(B),Z).
reduce(apply(A,B),Z):-reduce(A,k1(D)),reduce(D,Z).
reduce(apply(A,B),Z):-reduce(A,s),reduce(s1(B),Z).
reduce(apply(A,B),Z):-reduce(A,s1(C)),reduce(s2(C,B),Z).
reduce(apply(A,B),Z):-
        reduce(A,s2(D,E)),reduce(apply(apply(D,B),apply(E,B)),Z).
```

These clauses can be used to contemplate higher-order programming in LOG(F).

5.0 TWO WAY COMMUNICATION

This example models communcation between two users, each of who types a stream of tokens on his screen. Each token is of the form [A] or [send,M] in which case M appears on both screens. The communication is modeled by:

```
extract_messages([[A]|X])=>extract_messages(X).
```

```
extract_messages([[send,M]|X])=>[M|extract_messages(X)].

screen1=>fair_merge(key1,extract_messages(key2)).
screen2=>fair_merge(key2,extract_messages(key1)).
```

Here send, [] and | are constructor symbols. We assume there exists a function fair_merge which takes as input two streams and interleaves their tokens into an output stream. If two tokens appear in some order in an input, then they appear in the same order in the output. Finally, fair_merge consumes each input at the rate at which it is produced.

Note that the second extract_messages rule has a left hand side of depth greater than two, so, strictly speaking, it is not an F* rule. However, it can be expressed in F* as follows:

```
extract_messages([A|X])=>g(A,X).
g([U|V],X)=>h(U,V,X).
h(send,V,X)=>[V|extract_messages(X)].
```

Here g and h are auxiliary function symbols. For convenience, the F* compiler in APPENDICES 1-2 allows => rules with left hand sides of arbitrary depth (but not containing any non-constructor function symbols). It compiles these into reduce clauses equivalent to those produced when reexpressed as above.

Assuming that key1 and key2 are streams of tokens typed by, respectively, the first and second user, the term screen1 will reduce to the stream of tokens appearing on the first user's screen. Similarly for screen2. The reduce clauses are:

```
reduce([],[]).
reduce([U|V],[U|V]).
reduce(send,send).

reduce(extract_messages(A),B):-
        reduce(A,[C|D]),reduce(C,[E]),reduce(extract_messages(D),B).
reduce(extract_messages(A),[B|extract_messages(C)]):-
        reduce(A,[D|C]), reduce(D,[E|F]), reduce(E,send), reduce(F,[B]).
reduce(screen1,A):-reduce(fair_merge(key1,extract_messages(key2)),A).
reduce(screen2,A):-reduce(fair_merge(key2,extract_messages(key1)),A).
```

6.0 HAMMING'S PROBLEM

The problem, described in [Dijkstra 1976], is to generate, in increasing order, all those numbers which are divisible by no primes other than 2,3 or 5. Dijkstra states that an equivalent problem is to generate the sequence of numbers, in ascending order, defined by the following axioms:

(a) 1 is in the sequence

(b) If x is in the sequence, then so are 2*x, 3*x and 5*x.

(c) The sequence contains no values except those on account of (a) and (b).

These axioms can be expressed by the following DF* program:

```
hamming=>hamming_aux([1|hamming]).
hamming_aux(X)=>
        merge(times_list(2,X),merge(times_list(3,X),times_list(5,X))).

merge([U|V],[A|B])=>if(U<A,[U|merge(V,[A|B])],merge_aux(U,V,A,B)).
merge_aux(U,V,A,B)=>if(equal(U,A),[U|merge(V,B)],[A|merge([U|V],B)]).

times_list(N,[])=>[].
times_list(N,[U|V])=>[U*N|times_list(N,V)].
```

Function times_list multiplies each element of its input list by a fixed number. Function merge takes two lists in ascending order and merges their elements in increasing order. Functions hamming and hamming_aux are implementations of axioms (a),(b),(c).

This program illustrates the power of NA-derivations. The definition of hamming_aux contains three occurrences of X on the right hand side. If care is taken that whenever the term at one occurrence of X is reduced, terms at the other two occurrences of X are also reduced, the list hamming is produced with little overhead. If not, then the overhead increases exponentially. This can be felt by comparing the speed with which elements of hamming are printed on the screen in the two cases. The labeled version of this program is compiled into the following reduce clauses:

```
reduce(hamming(A),B):-not var(A),B=A,!.
reduce(hamming_aux(A,B),C):-not var(A),C=A,!.
reduce(merge(A,B,C),D):-not var(A),D=A,!.
```

```
reduce(merge_aux(A,B,C,D,E),F):-not var(A),F=A,!.
reduce(times_list(A,B,C),D):-not var(A),D=A,!.
reduce(if(A,B,C,D),E):-not var(A),E=A,!.

reduce([],[]).
reduce([A|B],[A|B]).
reduce(true,true).
reduce(false,false).

reduce(hamming(A),A):-hamming(B)=>C,reduce(C,A).
reduce(hamming_aux(A,B),A):-hamming_aux(C,B)=>D,reduce(D,A).
reduce(merge(A,B,C),A):-
        reduce(B,D),reduce(C,E),merge(F,D,E)=>G,reduce(G,A).
reduce(merge_aux(A,B,C,D,E),A):-merge_aux(F,B,C,D,E)=>G,reduce(G,A).
reduce(times_list(A,B,C),A):-reduce(C,D),times_list(E,B,D)=>F,reduce(F,A).
reduce(if(A,B,C,D),A):-reduce(B,E),if(F,E,C,D)=>G,reduce(G,A).

hamming(A)=>[1|hamming_aux(B,hamming(C))].
hamming_aux(A,B)=>
        merge(C,times_list(D,2,B),
                merge(E,times_list(F,3,B),times_list(G,5,B))).
merge(A,[B|C],[D|E])=>
        if(F,G,[B|merge(H,C,[D|E])],merge_aux(I,B,C,D,E)):-less_than(B,D,G).
merge_aux(A,B,C,D,E)=>
        if(F,G,[B|merge(H,C,E)],[D|merge(I,[B|C],E)]):-equal(B,D,G).
times_list(A,B,[])=>[].
```

```
times_list(A,B,[C|D])=>[E|times_list(F,B,D)]:-E is C*B.
if(A,true,B,C)=>B.
if(A,false,B,C)=>C.
```

Definitions of the eager functions less_than and equal are as in Section 2.0. If we now type print_list(hamming(_)), we obtain 1,2,3,4,5,6,8,9,10,12,...

7.0 INFINITE GRAPHICAL STRUCTURES.

Henderson [1982] has shown how to use functional programming for defining and manipulating graphical structures. In particular, he shows how to construct Square Limit, an Escher woodcut. We use Henderson's building blocks to tile the x-y plane in an interesting way. A picture is represented by a list of vectors, each of the form v(A,B)--v(X,Y), where A,B,X,Y are real numbers. Transformations on pictures, such as composition, translation, scaling, or rotation (about the origin) are defined as follows:

```
union([],X)=>X.
union([FX|RX],Y)=>[FX|union(Y,RX)].

rotate([],_)=>[].
rotate([v(X,Y)--v(A,B)|L],Theta)=>
 [v(X*cos(Theta)-Y*sin(Theta),X*sin(Theta)+Y*cos(Theta))--
 v(A*cos(Theta)-B*sin(Theta),A*sin(Theta)+B*cos(Theta))|rotate(L,Theta)].

translate([],_,_)=>[].
```

```
translate([v(X,Y)--v(A,B)|L],Dx,Dy)=>
        [v(X+Dx,Y+Dy)--v(A+Dx,B+Dy)|translate(L,Dx,Dy)].
scale([],_,_)=>[].
scale([v(X,Y)--v(A,B)|L],Kx,Ky)=>
        [v(X*Kx,Y*Ky)--v(A*Kx,B*Ky)|scale(L,Kx,Ky)].
```

The basic pictures are p,q,r,s, drawn in a 36x36 grid, and shown in order in the top row in Figure 1. (The vectors can be found, not unfortunately, in Henderson's paper, but in [Robinson & Green 1987]). These are combined by quartet into t, shown in the second row. The third and fourth rows show, respectively, block1(t) and block2(t), the two basic 144x72 rectangles. row(Block,0) repeats Block, infinitely often, at intervals of 144 units, in the x and -x directions. alt_rows(Row,0) repeats a row infinitely often, at intervals of 144 units, in the y and -y directions. mosaic(Block1,Block2) computes rows of Block1 and Block2, alternates these, and then composes these to tile the x-y plane, Figure 2. The program is:

```
rot_pos(X)=>translate(rotate(translate(X,-72,-72),-1.57),72,0).
rot_neg(X)=>translate(rotate(translate(X,-72,-72),1.57),0,72).

block1(X)=>union(X,translate(rot_neg(X),72,0)).
block2(X)=>
 union(rot_pos(X),
     translate(rotate(translate(rot_pos(X),-72,0),-1.57),72,0)).
```

```
row(Block,N)=>union(translate(Block,144*N,0),
                union(translate(Block,-144*N,0),row(Block,N+1))).

alt_rows(Row,N)=>union(translate(Row,0,144*N),
                union(translate(Row,0,-144*N),alt_rows(Row,N+1))).

mosaic(Block1,Block2)=>union(alt_rows(row(Block1,0),0),
                        translate(alt_rows(row(Block2,0),0),0,72)).
beside(A,B)=>union(A,translate(B,36,0)).
above(A,B)=>union(A,translate(B,0,36)).
quartet(P1,P2,P3,P4)=>above(beside(P3,P4),beside(P1,P2)).

t=>quartet(p,q,r,s).

p=>[v(0,7)--v(6,9),v(6,9)--v(0,18),v(0,18)--v(0,7),v(13,0)--v(9,9),
v(9,12)--v(9,23),v(9,23)--v(16,14),v(16,14)--v(9,12),v(24,0)--v(22,9),
v(22,9)--v(18,18),v(18,18)--v(9,30),v(9,30)--v(0,36),v(0,36)--v(13,34),
v(13,34)--v(18,36),v(18,36)--v(26,27),v(26,27)--v(36,27),
v(18,27)--v(36,23),v(18,18)--v(27,21),v(27,21)--v(36,18),
v(32,36)--v(36,34),v(27,36)--v(29,34),v(29,34)--v(36,32),
v(22,36)--v(26,32),v(26,32)--v(36,29),v(20,14)--v(27,16),
v(27,16)--v(36,14),v(22,9)--v(29,11),v(29,11)--v(36,9),
v(24,0)--v(31,5),v(31,5)--v(36,5)].

q=>[v(0,27)--v(7,29),v(7,29)--v(11,31),v(11,31)--v(16,34),
v(16,34)--v(18,36),v(0,23)--v(16,25),v(0,27)--v(0,36),v(0,0)--v(0,18),
```

v(0,18)--v(9,16),v(9,16)--v(13,16),v(13,16)--v(27,22),
v(27,22)--v(36,36),v(4,36)--v(7,29),v(9,36)--v(11,31),
v(14,36)--v(16,34),v(18,34)--v(25,34),v(25,34)--v(20,30),
v(20,30)--v(18,34),v(20,27)--v(27,27),v(27,27)--v(22,23),
v(22,23)--v(20,27),v(36,36)--v(34,22),v(34,22)--v(36,18),
v(36,18)--v(29,9),v(29,9)--v(27,0),v(29,0)--v(36,14),v(32,0)--v(36,9),
v(34,0)--v(36,4),v(32,25)--v(23,0),v(5,0)--v(9,11),v(9,11)--v(9,16),
v(9,0)--v(13,11),v(13,11)--v(13,16),v(14,0)--v(18,13),
v(18,13)--v(18,18),v(18,0)--v(22,14),v(22,14)--v(22,20)].

r=>[v(24,36)--v(27,28),v(27,28)--v(36,18),v(0,36)--v(4,27),
v(4,27)--v(10,22),v(10,22)--v(17,18),v(17,18)--v(31,14),
v(31,14)--v(36,9),v(13,36)--v(25,23),v(25,23)--v(36,14),
v(27,28)--v(36,36),v(29,30)--v(36,23),v(31,32)--v(36,28),
v(33,34)--v(36,32),v(2,2)--v(8,0),v(4,4)--v(18,0),v(7,7)--v(18,4),
v(18,4)--v(27,0),v(10,11)--v(27,7),v(27,7)--v(36,0),v(0,0)--v(17,18),
v(0,8)--v(10,22),v(0,18)--v(4,27),v(0,27)--v(2,32)].

s=>[v(18,36)--v(16,30),v(16,30)--v(16,23),v(16,23)--v(16,18),
v(16,18)--v(18,14),v(18,14)--v(23,9),v(23,9)--v(36,0),
v(23,36)--v(25,23),v(27,36)--v(30,30),v(30,30)--v(32,25),
v(32,25)--v(34,21),v(34,21)--v(36,18),v(29,16)--v(34,18),
v(34,18)--v(34,11),v(34,11)--v(29,16),v(22,14)--v(27,16),
v(27,16)--v(27,9),v(27,9)--v(22,14),v(30,30)--v(36,32),
v(32,25)--v(36,27),v(34,21)--v(36,22),v(0,0)--v(9,5),v(9,5)--v(17,5),
v(17,5)--v(36,0),v(0,9)--v(4,2),v(0,14)--v(16,9),v(0,18)--v(18,14),

```
v(0,23)--v(16,18),v(0,28)--v(16,23),v(0,32)--v(16,30),
v(0,36)--v(18,36),v(27,36)--v(36,36)].
```

Note that mosaic computes an infinite row, an infinite number of times. *However, reduction-completeness of DF* precludes infinite runaway.* Vectors are displayed as they are generated. The above program is compiled into:

```
reduce(rotate(A,B),[]) :- reduce(A,[]).
reduce(rotate(A,B),[v(C,D)--v(E,F)|rotate(G,B)]) :-
        reduce(A,[H|G]), reduce(H,I--J), reduce(I,v(K,L)),
        reduce(J,v(M,N)), cos(B,O), P is K*O, sin(B,Q),
        R is L*Q, C is P-R, sin(B,S), T is K*S,
        cos(B,U), V is L*U, D is T+V, cos(B,W),
        X is M*W, sin(B,Y), Z is N*Y, E is X-Z,
        sin(B,A1), B1 is M*A1, cos(B,C1), D1 is N*C1, F is B1+D1.
reduce(translate(A,B,C),[]) :-
 reduce(A,[]).
reduce(translate(A,B,C),[v(D,E)--v(F,G)|translate(H,B,C)]) :-
        reduce(A,[I|H]), reduce(I,J--K), reduce(J,v(L,M)),
        reduce(K,v(N,O)), D is L+B, E is M+C,
        F is N+B, G is O+C.
reduce(scale(A,B,C),[]) :- reduce(A,[]).
reduce(scale(A,B,C),[v(D,E)--v(F,G)|scale(H,B,C)]) :-
        reduce(A,[I|H]), reduce(I,J--K), reduce(J,v(L,M)),
        reduce(K,v(N,O)), D is L*B, E is M*C, F is N*B, G is O*C.
```

```
reduce(true,true).
reduce(false,false).
reduce([],[]).
reduce([A|B],[A|B]).
reduce(A--B,A--B).
reduce(v(A,B),v(A,B)).

reduce(p,A) :- p=>B, reduce(B,A).
reduce(q,A) :- q=>B, reduce(B,A).
reduce(r,A) :- r=>B, reduce(B,A).
reduce(s,A) :- s=>B, reduce(B,A).
reduce(t,A) :- t=>B, reduce(B,A).
reduce(block1(A),B) :- block1(A)=>C, reduce(C,B).
reduce(block2(A),B) :- block2(A)=>C, reduce(C,B).
reduce(cycle(A),B) :- cycle(A)=>C, reduce(C,B).
reduce(rot(A),B) :- rot(A)=>C, reduce(C,B).
reduce(rot_neg(A),B) :- rot_neg(A)=>C, reduce(C,B).
reduce(rot_pos(A),B) :- rot_pos(A)=>C, reduce(C,B).
reduce(above1(A,B),C) :- above1(A,B)=>D, reduce(D,C).
reduce(alt_rows(A,B),C) :- alt_rows(A,B)=>D, reduce(D,C).
reduce(beside1(A,B),C) :- beside1(A,B)=>D, reduce(D,C).
reduce(mosaic(A,B),C) :- mosaic(A,B)=>D, reduce(D,C).
reduce(row(A,B),C) :- row(A,B)=>D, reduce(D,C).
reduce(union(A,B),C) :- reduce(A,D), union(D,B)=>E, reduce(E,C).
reduce(quartet(A,B,C,D),E) :- quartet(A,B,C,D)=>F, reduce(F,E).
```

```
union([],A)=>A.
union([A|B],C)=>[A|union(C,B)].
rot_pos(A)=>translate(rotate(translate(A,-72,-72),-1.57),72,0).
rot_neg(A)=>translate(rotate(translate(A,-72,-72),1.57),0,72).
block1(A)=>union(A,translate(rot_neg(A),72,0)).
block2(A)=>union(rot_pos(A),
    translate(rotate(translate(rot_pos(A),-72,0),-1.57),72,0)).
row(A,B)=>union(translate(A,C,0),
            union(translate(A,D,0),row(A,E))) :-
                C is 144*B, D is-144*B, E is B+1.
alt_rows(A,B)=>
        union(translate(A,0,C),union(translate(A,0,D),alt_rows(A,E))):-
        C is 144*B, D is-144*B, E is B+1.
mosaic(A,B)=>union(alt_rows(row(A,0),0),
              translate(alt_rows(row(B,0),0),0,72)).
beside(A,B)=>union(A,translate(B,36,0)).
above(A,B)=>union(A,translate(B,0,36)).
quartet(A,B,C,D)=>above(beside(C,D),beside(A,B)).
rot(A)=>rotate(A,1.57).
cycle(A)=>union(A,union(rot(A),union(rot(rot(A)),rot(rot(rot(A)))))).
t=>quartet(p,q,r,s).
p=>[..].
q=>[..].
r=>[..].
s=>[..].
```

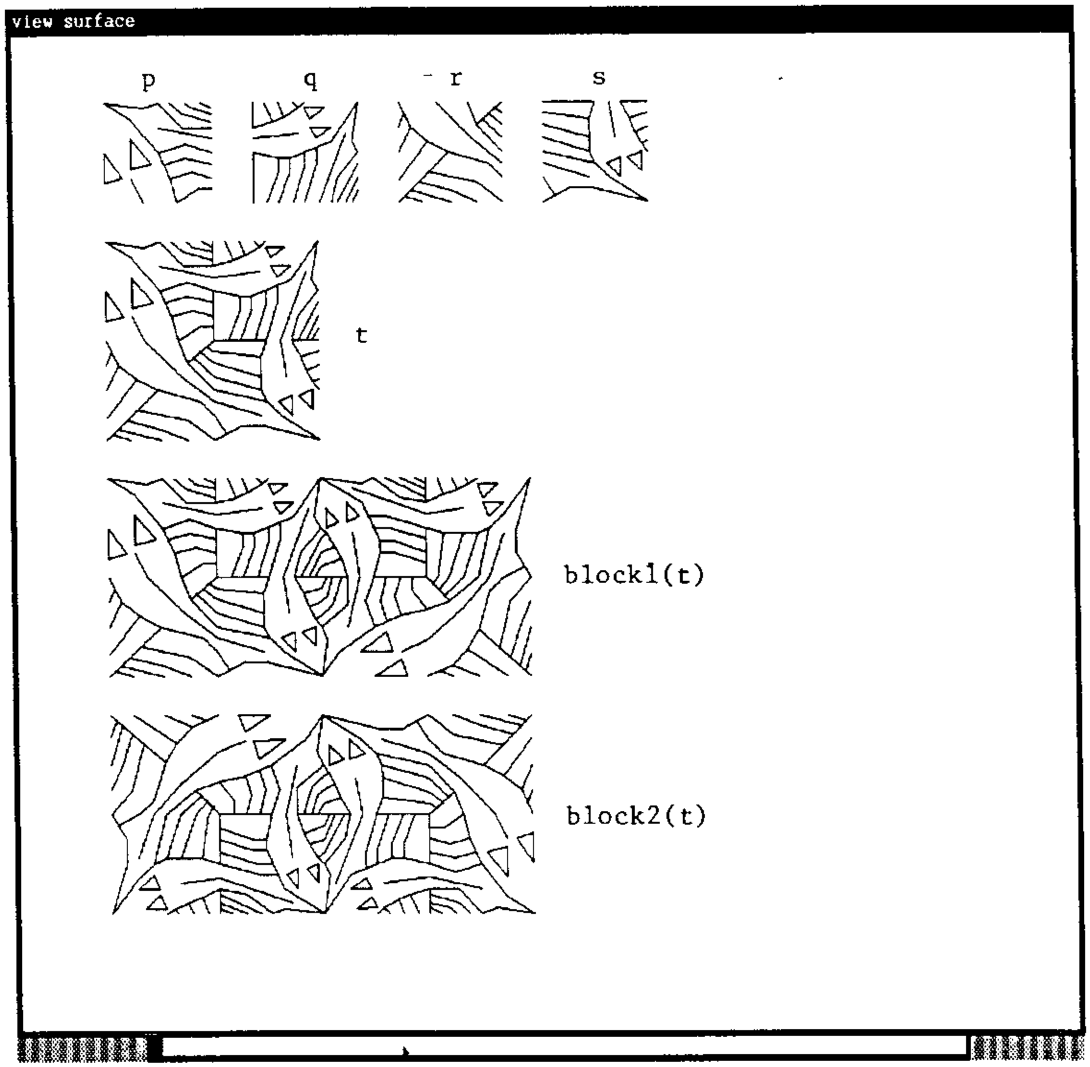

Figure 1. Some Graphical Pimitives

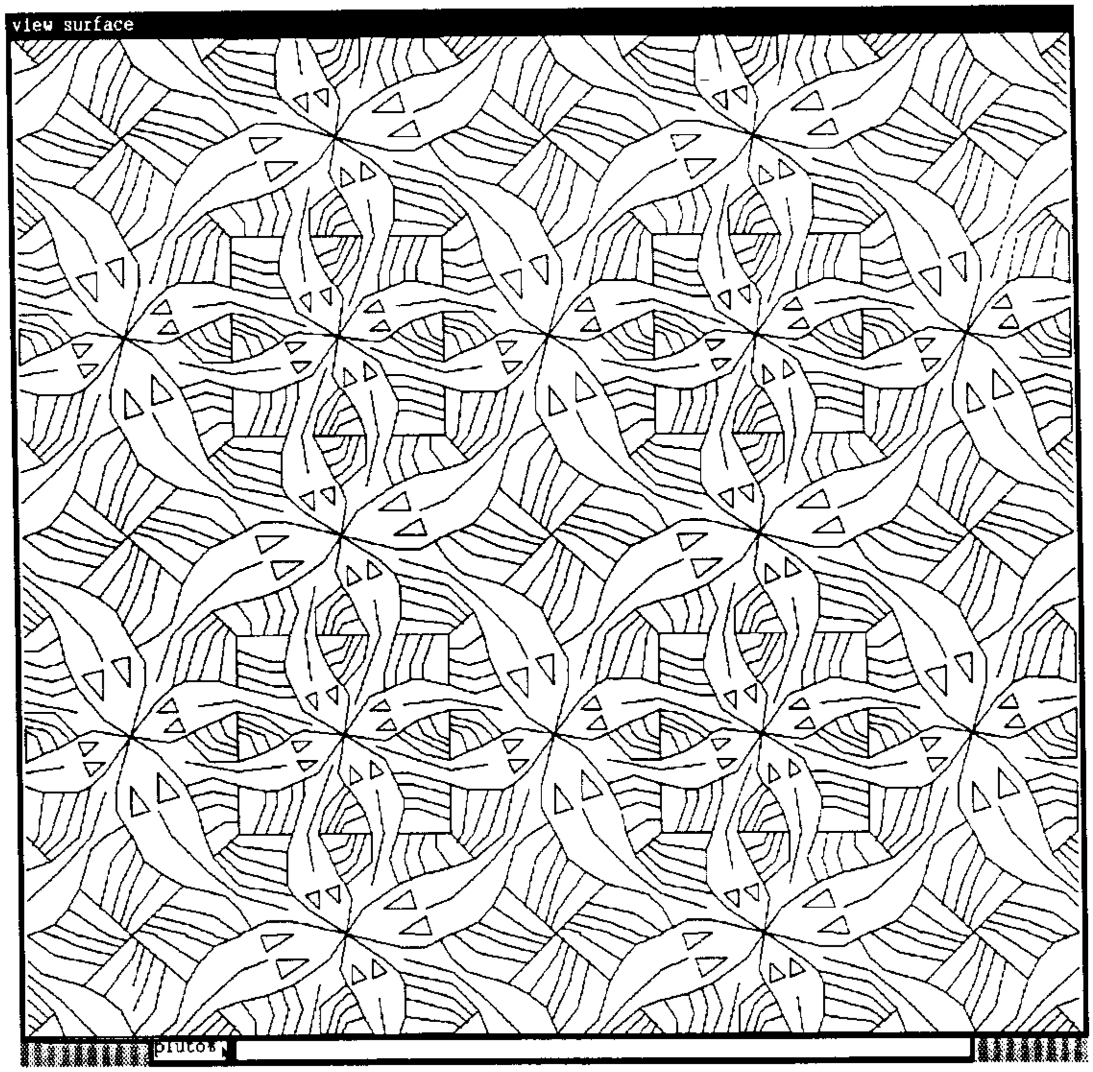

Figure 2. Square Unlimit

CHAPTER VIII

COMPARING LOG(F) PERFORMANCE WITH THAT OF PROLOG

1.0 INTRODUCTION

This chapter compares performance of LOG(F) with that of Prolog. Programs of similar length, and intellectual complexity are written in both F* and in Prolog. The former are compiled into Prolog, and optimized, before being tested. Sections 2.0-7.0 list the F* and Prolog programs. Section 8.0 contains the performance figures, and provides some empirical verification of the assertion that LOG(F) can be used to do efficient, lazy rewriting.

For problems in which data structures are always completely evaluated, lazy evaluation cannot reduce lengths of computation. Such problems include list reversal, or sorting. For these, LOG(F) is, on an average, five times slower than Prolog. However, the slowdown for a given problem appears to stay the same, regardless of the size of the input.

For problems in which data structures need only be partially evaluated, e.g. the N-queens problem, or tiling an infinite plane, lazy evaluation can reduce lengths of computation. For these, LOG(F) can be faster than Prolog by factors which are unbounded, i.e. grow with input size, and by factors which are infinite.

2.0 LINEAR LIST REVERSAL

The F* version is:

```
reverse([],A)=>A.
reverse([U|V],W)=>reverse(V,[U|W]).
```

This is compiled into:

```
reduce([],[]).
reduce([U|V],[U|V]).
reduce(reverse(A,B),Z):-reduce(A,X),reverse(X,B)=>RHS,reduce(RHS,Z).

reverse([],A)=>A.
reverse([U|V],W)=>reverse(V,[U|W]).
```

The Prolog version is:

```
reverse([],A,A).
reverse([U|V],A,Z):-reverse(V,[U|A],Z).
```

3.0 QUICKSORT

The F* version is:

```
quicksort([])=>[].
```

```
quicksort([A|B])=>quicksort1(A,partition(A,B,[],[])).
quicksort1(A,t(L,R))=>append(quicksort(L),[A|quicksort(R)]).
append([],X)=>X.
append([U|V],W)=>[U|append(V,W)].
if(true,X,Y)=>X.
if(false,X,Y)=>Y.
partition(U,[],L,R)=>t(L,R).
partition(U,[A|B],L,R)=>
        if(A=<U,partition(U,B,[A|L],R),partition(U,B,L,[A|R])).
```

This is compiled into:

```
reduce([],[]).
reduce([U|V],[U|V]).
reduce(true,true).
reduce(false,false).
reduce(t(X,Y),t(X,Y)).

reduce(quicksort(A),B):-reduce(A,C),quicksort(C)=>D,reduce(D,B).
reduce(quicksort1(A,B),C):-reduce(B,D),quicksort1(A,D)=>E,reduce(E,C).
reduce(append(A,B),C):-reduce(A,D),append(D,B)=>E,reduce(E,C).
reduce(if(A,B,C),D):-reduce(A,E),if(E,B,C)=>F,reduce(F,D).
reduce(partition(A,B,C,D),E):-
        reduce(B,F),partition(A,F,C,D)=>G,reduce(G,E).

quicksort([])=>[].
```

```
quicksort([A|B])=>quicksort1(A,separate(A,B,[],[])).
quicksort1(A,t(L,R))=>append(quicksort(L),[A|quicksort(R)]).
append([],X)=>X.
append([U|V],W)=>[U|append(V,W)].
if(true,X,Y)=>X.
if(false,X,Y)=>Y.
partition(U,[],L,R)=>t(L,R).
partition(U,[A|B],L,R)=>if(T,partition(U,B,[A|L],R),partition(U,B,L,{A|R]))
        :-less_than_equal(A,U,T).
less_than_equal(U,A,true):-U=<A.
less_than_equal(U,A,false):-U>A.
```

The Prolog version is:

```
quicksort([],[]).
quicksort([U|V],W):-partition(V,U,L,R),
            quicksort(L,L1),
            quicksort(R,R1),
            append(L1,[U|R1],W).
append([],X,X).
append([U|V],W,[U|Z]):-append(V,W,Z).
partition([],_,[],[]).
partition([U|V],A,[U|L],R):-U=<A,partition(V,A,L,R).
partition([U|V],A,L,[U|R]):-U>A,partition(V,A,L,R).
```

4.0 PERMUTATIONS

The F* version and its compilation are as given in Chapter VII, Section 2.0. The Prolog version is:

```
perm([],[]).
perm([U|V],Z):-perm(V,W),insert(U,W,Z).
insert(U,X,[U|X]).
insert(U,[A|B],[A|Z]):-insert(U,B,Z).
```

5.0 SIEVE OF ERATOSTHENES

The F* version is:

```
primes=>sieve(intfrom(2)).
sieve([U|V])=>[U|sieve(filter(U,V))].
intfrom(X)=>[X|intfrom(X+1)].
filter(A,[])=>[].
filter(A,[U|V])=>if(multiple(U,A),filter(A,V),[U|filter(A,V)]).
```

This is compiled into:

```
reduce([],[]).
reduce([U|V],[U|V]).
reduce(true,true).
reduce(false,false).
```

```
reduce(primes,A):-primes=>B,reduce(B,A).
reduce(sieve(A),B):-reduce(A,C),sieve(C)=>D,reduce(D,B).
reduce(intfrom(A),B):-intfrom(A)=>C,reduce(C,B).
reduce(filter(A,B),C):-reduce(B,D),filter(A,D)=>E,reduce(E,C).

primes=>sieve(intfrom(2)).
sieve([U|V])=>[U|sieve(filter(U,V))].
intfrom(X)=>[X|intfrom(X1)]:-X1 is X+1.
filter(A,[])=>[].
filter(A,[U|V])=>if(T,filter(A,V),[U|filter(A,V)]):-multiple(U,A,T).

multiple(U,A,true):- 0 is U mod A.
multiple(U,A,false):- not(0 is U mod A).
```

The Prolog version is:

```
sieve([],[]).
sieve([U|X],[U|Z]):-filter(U,X,V),sieve(V,Z).
filter(A,[],[]).
filter(A,[U|V],[U|Z]):- not divisible(U,A),filter(A,V,Z).
filter(A,[U|V],Z):-divisible(U,A),filter(A,V,Z).
divisible(U,A):-0 is U mod A.
intbetween(M,M,[M]).
intbetween(M,N,[M|Z]):- not M==N,M1 is M+1,intbetween(M1,N,Z).
```

6.0 N-QUEENS

The F* version and its compiled version are as in Chapter VII, Section 2.0. The Prolog version is:

```
queens(X,Y):-perm(X,Y),safe(Y).
safe([]).
safe([U|V]):-nodiagonal(U,V,1),safe(V).
nodiagonal(U,[],N).
nodiagonal(U,[A|B],N):-noattack(U,A,N),N1 is N+1,nodiagonal(U,B,N1).
noattack(U,A,N):- Z is U-A,abs(Z,Z1),not Z1==N.
```

7.0 INFINITE GRAPHICAL STRUCTURES

Again, the F* version and its compilation can be found in Chapter VII, Section 7.0. The Prolog definition of an infinite row of Block is:

```
row(Block,N,Z):-LeftN is -144*N,
                RightN is 144*N,
                N1 is N+1,
                translate(Block,LeftN,LeftBlock),
                translate(Block,RightN,RightBlock),
                row(Block,N1,Z1),
                union(LeftBlock,RightBlock,A),
                union(A,Z1,Z).
```

Note that row contains a call to itself, so is non-terminating.

8.0 TABLE OF RUNNING TIMES

Time in milliseconds			
	Prolog	LOG(F)	Prolog/LOG(F)
Reverse: 3200 elements	83	883	0.09
Reverse: 6400 elements	150	1755	0.08
Quicksort: 60 elements	83	539	0.15
Quicksort: 120 elements	250	1261	0.19
Sieve: First 50 primes	422	1261	0.33
Sieve: First 100 primes	1816	4511	0.40
All permutations of [1,2,3,4,5]	427	516	0.82
All permutations of [1,2,3,4,5,6]	3000	3172	0.94
8-Queens: All solutions	62783	17511	3.58
9-Queens: All solutions	635144	86539	7.33
15-Queens: First solution	>30 minutes	30300	>60
Infinite plane: First vector	∞	~ 0	∞

Table 1. Comparing LOG(F) performance with that of Prolog

Note that in the 15-Queens problem, Prolog did not yield any solution even after 30 minutes of elapsed time. Also, note that for problems in which lazy evaluation does not reduce lengths of computation, e.g. reverse, quicksort, sieve, or all permutations, LOG(F) is slower than Prolog. However, the slowdown varies little with problem size. For problems in which lazy evaluation does reduce lengths of computation, such as N-queens, or tiling the infinite plane, LOG(F) is faster than Prolog. However, the speedup varies considerably with problem size, and can even be infinite.

CHAPTER IX

SUMMARY AND CONCLUSIONS

A new approach for combining logic programming, rewriting, and lazy evaluation is described. It rests upon *subsuming* within logic programming, instead of upon *extending* it with, rewriting, and lazy evaluation.

F* is a non-terminating, non-deterministic rewrite rule system. The reduction strategy for it, select, is reduction-complete. DF* is a subset of F*, and is also non-terminating. DF* satisfies confluence, directedness, and minimality. Reduction-completeness, and minimality enable select to exhibit, respectively, weak and strong forms of laziness.

F* can be compiled into Horn clauses in such a way that when SLD-resolution interprets them, it directly simulates the behavior of select. In particular, it is made to exhibit laziness. LOG(F) is defined to be a logic programming system augmented with an F* compiler, and the equality axiom X=X. Since clauses obtained by compiling F* programs can be called from other logic programs, LOG(F) is proposed as a combination of logic programming, rewriting, and lazy evaluation.

LOG(F) offers, perhaps for the first time, an efficient implementation of lazy evaluation within a widely used language, namely, Prolog. For problems in which lazy evaluation cannot reduce lengths of computation, LOG(F) is somewhat slower than Prolog. For problems in which lazy evaluation does reduce lengths of computation, LOG(F) can be faster than Prolog by factors which are unbounded, i.e. grow with input size, and factors which are infinite.

LOG(F) can also be used to implement useful cases of the rule of substitution of equals for equals. Confluence of DF* yields a new proof of the confluence of combinatory logic. Finally, DF* seems to be a good candidate for implementation on parallel machines. It seems to offer a reasonable compromise between sequential execution and unbounded parallelism. Due to directedness of DF*, arguments of f in f(t1,..,tm) can be simplified in parallel, however, they would be simplified lazily.

REFERENCES

Abramson, H. [1986]. A prolog definition of HASL, a purely functional language with unification-based conditional binding expressions. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.

Ait-Kaci, H., Nasr, R. [1986]. Residuation: A paradigm for integrating logic and functional programming. MCC Technical Report AI-359-86, Austin, TX.

Ait-Kaci, H., Lincoln, P., Nasr, R. [1987]. Le Fun: Logic, Equations and Functions. *Proceedings of symposium on logic programming,* San Francisco.

Apt, K.R., van Emden, M.H. [1982]. Contributions to the theory of logic programming. *Journal of the ACM* vol. 29, no. 3, July 1982.

Barendregt, H.P. [1977]. The type free lambda-calculus, in: *Handbook of Mathematical Logic*, ed. John Barwise, North Holland Publishing Company.

Barbuti, R., Bellia, M., Levi, G. [1986]. LEAF: A language which integrates logic, equations, and functions. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.

Barrow, H. [1983]. Proving the Correctness of Digital Hardware Designs. *Proceedings of the National Conference on Artificial Intelligence,* Washington, D.C.

Bellia, M., Levi, G. [1986]. The relation between logic and functional languages: a survey. *Journal of Logic Programming*, October 1986.

Berry, G., Levy, J.-J. [1979]. Minimal and optimal computations of recursive programs. *Journal of the ACM*, vol. 26, no. 1, pp. 148-175.

Bundy, A., Welham, W. [1981]. Using meta-level inference for selective application of multiple rewrite rule sets in algebraic manipulation. *Artificial Intelligence*, vol 16, no. 2, May.

Burstall, R.M., MacQueen, D.B., Sannella, D.T. [1980]. HOPE: An experimental applicative language. *Proceedings of 1980 Lisp Conference*. Stanford, California.

Burstall, R., Darlington, J. [1977]. A transformation system for developing recursive programs. *Journal of the ACM*, vol. 24, No. 1.

Church, A. [1941]. *The calculi of lambda-conversion*. Annals of mathematical studies number 6. Princeton University Press, Princeton.

Clark, K.L., McCabe F. [1979]. Programmer's guide to IC-Prolog. *CCD Report 79/7*, London: Imperial College, University of London.

Clark, K.L. [1980]. Predicate Logic as a computational formalism. Research monograph, Imperial college, University of London.

Clark, K.L., McCabe, F. [1982]. PROLOG: A Language for implementing expert systems. *Machine Intelligence 10*, (eds.) J. Hayes and D.J. Michie.

Clark, K.L., Gregory, S. [1986]. PARLOG: Parallel programming in logic. *ACM transactions on programming languages and systems*, 8,1.

Curry, H.B., Feys, R. [1958]. *Combinatory Logic, vol I*, North Holland, Amsterdam.

Darlington, J., Field, A.J., Pull, H. [1986]. The unification of functional and logic languages. *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, New Jersey.

Davis, M., Putnam, H. [1960]. A computing procedure for quantification theory. *Journal of the ACM*, 7, pp. 201-215.

DeGroot, D., Lindstrom, G. (editors) [1986]. *Logic programming. Functions, relations and equations.* Prentice Hall, New Jersey.

Dershowitz, N., Josephson, N.A. [1984]. Logic Programming by completion. *Proceedings of second international logic programming conference,* Uppsala University, Sweden.

Digricoli, V.J., Harrison, M.C. [1986]. Equality-based binary resolution. *Journal of the ACM*, vol. 33, no. 2, April.

Dijkstra, E. [1976]. *A discipline of programming*. Prentice-Hall, Englewoods Cliffs, N.J.

Dincbas, M., van Hentenryck, P. [1987]. Extended unification algorithms for the integration of functional and logic languages. *Journal of logic programming*, vol. 4, No. 3.

Fay, M. [1979]. First order unification in an equational theory. *Proceedings of the 4th conference on automated deduction.*

Frege, G. [1879]. Begriffsschrift. A formula language, modelled upon that of arithmetic, for pure thought. in *From Frege to Goedel: A source book in mathematical logic, 1879-1931*. Harvard University Press, Cambridge, MA.

Fribourg, L. [1984]. Oriented equational clauses as a programming language. *Journal of Logic Programming* vol 1, pp. 165-177.

Friedman, D.P., Wise, D.S. [1976]. CONS should not evaluate its arguments, in: *Automata, Languages and Programming,* eds. S. Michaelson, R. Milner, Edinburgh University Press, Edinburgh.

Gallagher J. [1982]. Simulating Coroutining for the 8 Queens Problem. *Logic Programming Newsletter 3*, Summer 1982.

Gallaire, H., Lasserre, C. [1982]. Metalevel Control for Logic Programs, *Logic Programming* eds. K. Clark, S.-A. Tarnlund, Academic Press, New York.

Gallaire, H., Minker, J. (editors) [1978]. *Logic and Databases*, Plenum Press, New York.

Gallier, J.H., Raatz, S. [1986]. SLD-resolution methods for Horn clauses with equality based on E-unification. *Proceedings of 1986 symposium on logic programming*, Salt Lake City, Utah.

Goguen, J.A., Meseguer, J. [1986]. Equality, types and generic modules for logic programming. *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, New Jersey.

Green, C.C. [1969]. Theorem proving by resolution as a basis for question-answering systems. *Machine Intelligence*, 4, Edinburgh University Press, pp 183-205.

Greene, K.J. [1985]. A fully lazy higher order purely functional programming language with reduction semantics. CASE Center technical report no. 8503, Syracuse University, N.Y.

Hansson, A., Haridi, S., Tarnlund, S.-A. [1982]. Properties of a logic programming language, in: *Logic Programming* eds. K. Clark, S.A. Tarnlund, Academic Press, New York.

Henderson, P. [1980]. *Functional Programming: Application and Implementation.* Prentice Hall International, New Jersey.

Henderson, P. [1982]. Purely functional operating systems. In *Functional programming and its applications. An advanced course.* (eds.) J. Darlington, P. Henderson, D.A. Turner.

Henderson, P. [1982]. Functional Geometry. *Proceedings of the ACM Symposium on Lisp and Functional Programming*. Pittsburgh, PA.

Hill, R. [1974]. LUSH Resolution and its completeness. DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh.

Hoffman, C.M., O'Donnell, M.J. [1982]. Programming with equations. *ACM Transactions on programming languages and systems*. January.

Hoelldobler, S. [1987]. Equational logic programming. *Proceedings of Fourth Symposium on logic programming*, San Francisco, CA.

Hopcroft, J., Ullman, J. [1979]. *Introduction to automata theory, languages and computation*. Addison Wesley, Menlo Park, CA.

Huet, G. [1980]. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27:797-821.

Huet, G., Levy, J.-J. [1979]. Call by need computations in non-ambiguous linear term rewriting systems. IRIA technical report 359.

Huet, G. [1975]. A unification algorithm for typed λ-calculus. *Theoretical computer science* 1 (1975) 27-57.

Hullot, J.-M. [1980]. Canonical forms and unification. *Proceedings of 5th conference on automated deduction, LNCS 87*, Springer Verlag.

Jaffar, J., Lassez, J.-L., Maher, M.J. [1984]. A theory of complete logic programs with equality. *Journal of logic programming*, vol. 1, no. 3.

Kahn, K. [1986]. Uniform -- A language based upon unification which unifies (much of) Lisp, Prolog, and Act 1. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, New Jersey.

Kahn, G., MacQueen, D. [1977]. Coroutines and Networks of Parallel Processes. *Information Processing-77*, North-Holland, Amsterdam.

Knuth, D.E., Bendix, P.B. [1970]. Simple word problems in universal algebras. *Computational problems in abstract algebra*, ed. J. Leech, Pergamon.

Kohavi, Z. [1978]. *Switching and finite automata theory*. McGraw Hill, New York.

Komorowski, H.J. [1982]. QLOG - The programming environment for Prolog in Lisp. *Logic Programming* eds. K. Clark, S.-A. Tarnlund, Academic Press, New York.

Kornfeld, W. [1983]. Equality for Prolog. *Proceedings of IJCAI-83*. Karlsruhe, West Germany.

Kowalski, R. [1979]. *Logic for Problem Solving*, Elsevier North Holland, New York.

Lloyd, J. [1984]. *Foundations of logic programming*. Springer Verlag, New York.

Malachi, Y., Manna, Z. [1986]. Tablog: A new approach to logic programming. In

Logic programming: functions, relations and equations (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.

McCarthy, J. [1960]. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3, pp. 184-195.

Miller, D.A., Nadathur, G. [1986]. Higher-order logic programming. *Proceedings of third international conference on logic programming*. Lecture notes in computer science 225, (ed.) E. Shapiro, Springer Verlag, New York.

Narain, S. [1985]. A technique for doing lazy evaluation in logic. *Proceedings of IEEE symposium on logic programming*, Boston, MA.

Narain, S. [1986]. MYCIN: The expert system and its implementation in LOGLISP. In *Logic programming and its applications*, eds. D.H.D. Warren, M. van Caneghem, Ablex publishing company, N.J.

Narain, S. [1986]. A Technique for Doing Lazy Evaluation in Logic. *Journal of Logic Programming*, vol. 3, no. 3, October.

O'Donnell, M.J. [1985]. Equational logic as a programming language. MIT Press, Cambridge, MA.

Pereira, F.C.N., Warren, D.H.D. [1980]. Definite clause grammars for natural language analysis. A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence Journal*, 13, pp. 231-278.

Pingali, K., Arvind. [1985]. Efficient demand-driven evaluation. Part 1. *ACM transactions on programming languages and systems,* April 1982.

Rabin, M.O. Theoretical impediments to artificial intelligence.

Reddy, U.S. [1985]. Narrowing as the operational semantics of functional languages. *Proceedings of the 1985 symposium on logic programming*, Boston.

Robinson, G., Wos, L. [1969]. Paramodulation and theorem proving in first order theories with equality. *Machine Intelligence 4*, (eds.) B. Meltzer, D. Michie, M. Swann.

Robinson, J.A. [1965]. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12, pp 23-41.

Robinson, J.A. [1979]. *Logic: Form and Function. The Mechanization of Deductive Reasoning*. Elsevier North Holland, New York.

Robinson, J.A., Sibert, E.E. [1982]. LOGLISP: Motivation, Design and Implementation. *Logic Programming* eds. K. Clark, S.-A. Tarnlund, Academic Press, New York.

Robinson, J.A. [1984]. Editor's introduction. *Journal of logic programming*, vol. 1, no. 1, June.

Robinson, J.A. [1987]. Beyond LOGLISP: Combining functional and relational programming in a reduction setting. *Machine Intelligence 11*.

Robinson, J.A., Greene, K.J. [1987]. New Generation Knolwedge Processing, vol. III. RADC-TR-87-165. Rome Air Development Center, Griffis Air Force Base, NY.

Rosser, J.B. [1982]. Highlights of the history of the lambda-calculus. *Proceedings of the ACM Symposium on Lisp and Functional Programming*. Pittsburgh, PA.

Sato, M., Sakurai, T. [1986]. QUTE: A functional language based on unification. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.

Shapiro, E. [1983]. A subset of Concurrent Prolog and its interpreter. *ICOT technical report TR-003*, February 1983.

Shapiro, E., Takeuchi, A. [1983]. Object-oriented Programming in Concurrent Prolog. *New Generation Computing I* (1983), OHMSA LTD and Springer Verlag.

Subrahmanyam, P.A. and You J.-H. [1984]. Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming. *Proceedings of 1984 IEEE Logic Programming Symposium,* Atlantic City, N.J.

Tamaki, H. [1984]. Semantics of a logic programming language with a reducibility predicate. *Proceedings of 1984 international symposium on logic programming*, Atlantic City, N.J.

Turner, D. [1979]. A New Implementation Technique for Applicative Languages, *Software Practice and Experience*, 9, pp. 31-49.

Ueda, K. [1986]. Guarded Horn Clauses. Ph.D. Thesis. University of Tokyo. Tokyo, Japan.

van Emden, M.H., Yukawa, K. [1987]. Logic programming with equations. *Journal of Logic Programming*, vol. 4, no. 4.

Vuillemin, J. [1974]. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9, pp. 332-354.

Wadsworth, C.P. [1976]. The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus. *SIAM Journal of Computing*, vol. 5, no. 3, September.

Warren, D.H.D., van Caneghem, M. (editors) [1986]. *Logic Programming and its Applications*, Ablex Publishing, N.J.

Warren, D.H.D., Pereira, L.M., Pereira, F.C.N. [1977]. Prolog - the language and its implementation comparted with Lisp. *Proc. Symp. on AI and Programming Languages*, SIGPLAN Notices 12, No. 8, and SIGART Newsletter 64, pp 109-115.

Yamamoto, A. [1987]. A theoretical combination of SLD-resolution and narrowing. *Proceedings of fourth international conference on logic programming*, Melbourne, Australia.

APPENDIX 1
F* COMPILER IN PROLOG

```
/*
==============================================================
A compiler which accepts => rules and produces reduce clauses.
==============================================================
*/

:-op(650,xfx,=>).

translate_fdf:-compute_consistent_arg_heads,translate_f,translate_df.

translate_all:-compute_consistent_arg_heads,
        translate_f,
        write('translated non-consistent-argument rules'),nl,
        translate_df,
        write('translated consistent-argument rules'),nl,
        translate_ldf,
        write('attached output variables'),nl,nl,write('done').
/*
==============================================================
Generating consistent argument clauses.
==============================================================
*/

/*
Determining whether a list of lists of arguments is consistent. If the
list contains just A, then we still have to check whether members of A are
F* arguments. This can be done, rather indirectly, by
consistent_pair(A,A) which checks for this.

All this is necessary because we are allowing => rules with lhs of
arbitrary depth since there is an easy way to compile them into reduce
clauses.
*/

consistent_list([]).
consistent_list([A]):-consistent_pair(A,A).
```

```
consistent_list([A,B|C]):-consistent_list(C),
                          consistent_pair(A,B).

consistent_pair([],[]).
consistent_pair([X|Y],[U|V]):-var(X),var(U),consistent_pair(Y,V).
consistent_pair([X|Y],[U|V]):- nonvar(X),nonvar(U),
                              nonvar_fstar_arg(X),
                              nonvar_fstar_arg(U),
                              consistent_pair(Y,V).
/*
Input should not be a variable.
*/

nonvar_fstar_arg(X):-atomic(X),!.
nonvar_fstar_arg(X):-X=..[F|Args],variable_list(Args).

variable_list([]).
variable_list([U|V]):-var(U),variable_list(V).

/*
Input is a sorted list of all heads of => rules. Output is a list of
lists of heads. Each member list contains all heads for a fixed function.
*/

extract_same_heads([],[]).
extract_same_heads([U|V],[Z|L]):-extract_same_heads1(U,V,Z,Rem_Heads),
                              extract_same_heads(Rem_Heads,L).

extract_same_heads1(A,[],[A],[]).
extract_same_heads1(A,[U|V],[U|Z],L):-same_principal_functor(A,U),
                                   extract_same_heads1(A,V,Z,L).
extract_same_heads1(A,[U|V],[A],[U|V]):- \+same_principal_functor(A,U).

same_principal_functor(A,B):-functor(A,F,N),functor(B,F,N).

/*
This is the toplevel procedure. It creates consistent_arg clauses.
*/
```

```
compute_consistent_arg_heads:-
        setof(LHS,B^RHS^clause((LHS=>RHS),B),S),
        extract_same_heads(S,L), /* L is a list of list of lhsides */
        consistent_heads(L,M),  /* M is a list of f(X1,..,Xm) where
                                   heads of rules for f are consistent */
        assert_consistent_clauses(M).

assert_consistent_clauses([]).
assert_consistent_clauses([U|V]):-
        U=..[F|Args],
        length(Args,N),
        variables(N,Vlist),
        U1=..[F|Vlist],
        assertz(consistent_args(U1)),
        assert_consistent_clauses(V).

consistent_heads([],[]).
consistent_heads([U|V],[A|Z]):-extract_arguments(U,U1),
                        consistent_list(U1),
                        U=[A|_],
                        consistent_heads(V,Z).
consistent_heads([U|V],Z):-extract_arguments(U,U1),
                        \+consistent_list(U1),
                        consistent_heads(V,Z).


extract_arguments([],[]).
extract_arguments([U|V],[Args|V1]):-U=..[F|Args],
                                extract_arguments(V,V1).


/*
=================================================================
Produces reduce rules for => rules satisfying (g).
=================================================================
*/

translate_df:-
        create_det_reduce_rules,
```

```
        create_det_arrow_rules.

create_det_reduce_rules:-
                consistent_args(Head),
                template(Head,Head1),
                det_reduce_rule(Head1,Rule),
                assertz(Rule),
                fail.
create_det_reduce_rules.

template(Head,Head):-clause((Head=>RHS),true),!.

create_det_arrow_rules:-
                    clause((LHS=>RHS),true),
                    consistent_args(LHS),
                    det_arrow_rule((LHS=>RHS),Rule),
                    retract(((LHS=>RHS):-true)),
                    assertz(Rule),
                    fail.
create_det_arrow_rules.

det_reduce_rule(Head,(reduce(A,Z):-Body)):-
        Head=..[F|Args],
        length(Args,N),
        variables(N,As),
        A=..[F|As],
      variables(N,Xs),
        reduce_conds(Args,As,Xs,ArgConds),
        B=..[F|Xs],
        insert_at_end(((B=>RHS),reduce(RHS,Z)),ArgConds,Body1),
        flatten(Body1,Body2),
        eliminate_trues(Body2,Body).

reduce_conds([],[],[],true).
reduce_conds([Arg|Args],[A|As],[A|Xs],Z):-
        var(Arg),!,
        reduce_conds(Args,As,Xs,Z).
reduce_conds([Arg|Args],[A|As],[X|Xs],(reduce(A,X),Z)):-
        reduce_conds(Args,As,Xs,Z).
```

```
det_arrow_rule((LHS=>RHS),((LHS=>RHS1):-Body)):-
        find_evaluable_calls(RHS,RHS1,B),
        flatten(B,B1),
        eliminate_trues(B1,Body).

/*
================================================================
Produces reduce rules for => rules not satisfying (g).
================================================================
*/

translate_f:-
        clause(LHS=>RHS,true),
        \+ consistent_args(LHS),
        find_evaluable_calls(RHS,RHS1,Conds),
        translate_rule(LHS,RHS1,Conds,(H:-B)),
        flatten(B,B1),
        eliminate_trues(B1,B2),
        assert((H:-B2)),
        fail.
translate_f:-simplified(X),assert(reduce(X,X)),fail.
translate_f.

flatten((A,B),Z):-!,flatten(A,A1),flatten(B,B1),append_conds(A1,B1,Z).
flatten(A,A).

append_conds((A,B),Z,(A,Z1)):-!,append_conds(B,Z,Z1).
append_conds(A,Z,(A,Z)).

eliminate_trues(A,B):-eliminate_trues1(A,Z),
                      eliminate_last_true(Z,B).

eliminate_trues1((true,X),X1):-!,eliminate_trues1(X,X1).
eliminate_trues1((X,Y),(X,Y1)):-!,eliminate_trues1(Y,Y1).
eliminate_trues1(X,X).

eliminate_last_true((A,true),A):-!.
eliminate_last_true((A,B),(A,Z)):-!,eliminate_last_true(B,Z).
eliminate_last_true(A,A).
```

```
find_evaluable_calls(E,E,true):-var(E),!.
find_evaluable_calls(E,P,Conds):-
        E=..[F|Args],
        find_evaluable_calls_each(Args,Args1,Conds1),
        G=..[F|Args1],
        makep(G,Conds1,P,Conds).

makep(G,Conds,P,(Conds,C)):-replace(G,P,C),!.
makep(G,Conds,G,Conds).

find_evaluable_calls_each([],[],true).
find_evaluable_calls_each([U|V],[U1|V1],(Conds1,Conds)):-
        find_evaluable_calls(U,U1,Conds1),
        find_evaluable_calls_each(V,V1,Conds).

evaluable(E):-not(var(E)),replace(E,_,_).

write_functions(F):-
        tell(F),
        write_clauses_for_head(reduce(M,N)),
        write_clauses_for_head((A=>B)),
        told.

write_clauses_for_head(A):-
   clause(A,Body),
   if_then_else(Body=true,
              (write(A),write('.'),nl,fail),
              (write(A),write(':'),write('-'),nl,
               write_body(Body),
               write('.'),nl,fail)).
write_clauses_for_head(A).

write_body(A):-var(A),!,write('      '),write(A).
write_body((A,B)):-!,write('      '),
               write(A),write(','),nl,
               write_body(B).
write_body(A):-write('      '),write(A).
```

```
/*
Conds correspond to evaluable terms. ArgConds correspond to
arguments of LHS.
*/

translate_rule(LHS,RHS,Conds,(reduce(H,Out):-Body)):-
        LHS=..[F|Args],
        length(Args,N),
        variables(N,Vlist),
        translate_args_each(Args,Vlist,ArgConds),
        H=..[F|Vlist],
        rhs_and_conds(RHS,(ArgConds,Conds),Out,Body).

/*
If RHS is simplified, no condition is generated, otherwise reduce(RHS,Out)
is generated.
*/

rhs_and_conds(RHS,Conds,RHS,Conds):- \+var(RHS),simplified(RHS),!.
rhs_and_conds(RHS,Conds,Out,NewConds):-
        insert_at_end(reduce(RHS,Out),Conds,NewConds).

translate_args_each([],[],true).
translate_args_each([A|L],[X|Vars],(A1,L1)):-
        translate_arg(A,X,A1),
        translate_args_each(L,Vars,L1).

translate_arg(A,X,true):-var(A),A=X,!.
translate_arg(A,X,(reduce(X,A1),Conds)):-
                A=..[F|Args], /* F is a constructor symbol */
                length(Args,N),
                variables(N,Vlist),
                A1=..[F|Vlist],
                translate_args_each(Args,Vlist,Conds).

variables(0,[]).
variables(N,[A|L]):-N>0,N1 is N-1,variables(N1,L).

insert_at_end(A,(X,Y),(X,Z)):-!,insert_at_end(A,Y,Z).
```

```
insert_at_end(A,Y,(Y,A)).

/*
=============================================================
Compiler for LDF*. Accepts as input reduce clauses produced by
translate_f, and translate_ldf.
=============================================================
*/

/*
This attaches output vars to function symbols, except constructor symbols.
*/

attach_output_var(X,X):-(number(X);var(X);(atomic(X),simplified(X))),!.
attach_output_var(X,Y):-atomic(X),Y=..[X,A],!.
attach_output_var(X,Z):-X=..[F|Args],
                    !,attach_output_var_each(Args,Args1),
                    if_then_else(simplified(X),
                     Z=..[F|Args1],
                          Z=..[F,Out|Args1]).

if_then_else(C,A,B):-C,!,A.
if_then_else(C,A,B):-B.

attach_output_var_each([],[]).
attach_output_var_each([U|V],[U1|V1]):-attach_output_var(U,U1),
                                   attach_output_var_each(V,V1).

preprocess_cond(true,true).
preprocess_cond(A,A):-replace(Z,_,A),!.
preprocess_cond(A=X,A=X):-!.
preprocess_cond(A=>B,A1=>B):-A=..[F|Args],A1=..[F,Out|Args],!.
                    /*B above is always a variable */
preprocess_cond(reduce(A,X),reduce(A1,X1)):-
        attach_output_var(A,A1),
        attach_output_var(X,X1).


preprocess_body((A,B),(A1,B1)):-preprocess_cond(A,A1),!,
```

```
                         preprocess_body(B,B1).
preprocess_body(X,Y):-preprocess_cond(X,Y).


preprocess_rule((Head:-Body),NewRule):-
        preprocess_cond(Head,reduce(LHS,RHS)),
        preprocess_body(Body,Body1),
     connect_lhs_rhs(LHS,RHS,LHS1),
        construct_rule(LHS1,RHS,Body1,NewRule).

preprocess_arrow_rule(((LHS=>RHS):-C),((LHS1=>RHS1):-C)):-
        attach_output_var(LHS,LHS1),
        attach_output_var(RHS,RHS1).

/*
The first rule is for rules of the form reduce(E,E), E is simplified.
*/

connect_lhs_rhs(LHS,RHS,LHS):-simplified(LHS).
connect_lhs_rhs(LHS,RHS,LHS1):-
        LHS=..[F,A|Args],
     A=RHS,
        LHS1=LHS,!.

construct_rule(LHS,RHS,true,reduce(LHS,RHS)):-!.
construct_rule(LHS,RHS,Body,(reduce(LHS,RHS):-Body)).


preprocess_arrow_rule_set([]).
preprocess_arrow_rule_set([Rule|X]):-
        preprocess_arrow_rule(Rule,R),
        assertz(R),
        preprocess_arrow_rule_set(X).

preprocess_rule_set([]).
preprocess_rule_set([Rule|X]):-preprocess_rule(Rule,R),
                            assertz(R),
                            preprocess_rule_set(X).
```

```
translate_ldf:-bagof((reduce(A,B):-C),clause(reduce(A,B),C),S),
          retract_all((reduce(X,Y):-Z)),
          preprocess_rule_set(S),
          bagof(((LHS=>RHS):-D),clause((LHS=>RHS),D),Set),
        retract_all(((LHS1=>RHS1):-D1)),
        preprocess_arrow_rule_set(Set),
          generate_first_rules.

retract_all(C):-retract(C),fail.
retract_all(C).

/*
Generating the first rule.
*/

non_constructor([F,N]):-clause(reduce(A,B),C),
                       \+simplified(A),
                       A=..[F|Args],
                       length(Args,N).

generate_first_rules:-setof(H,non_constructor(H),S),
                    generate_rules_each(S).

/*
The first rules are generated after the output variables have been
inserted in reduce clauses.  So, arities of function symbols is 1 more
than usual.

Using setof instead of bagof ensures that there are no duplicates in list
of function symbols.
*/

generate_rules_each([]).
generate_rules_each([[F,N]|V]):-
      length([A|Args],N),
      Head=..[F,A|Args],
      asserta((reduce(Head,Out):-nonvar(A),Out=A,!)),
      generate_rules_each(V).
```

```
/*
Unattaching output variables for readability.
*/

remove_var(X,X):-(number(X);var(X);(atomic(X),simplified(X))),!.
remove_var(X,Z):-simplified(X),!,X=..[F|Args],
                remove_var_each(Args,Args1),
                Z=..[F|Args1].
remove_var(X,Z):-X=..[F,A|Args],
                remove_var_each(Args,Args1),
                Z=..[F|Args1].

remove_var_each([],[]).
remove_var_each([U|V],[U1|V1]):-remove_var(U,U1),
                        remove_var_each(V,V1).

/*
==============================================================
Utilities.
==============================================================
*/

make_list(X,Y):-reduce(X,Z),make_list_1(Z,Y).

make_list_1([],[]).
make_list_1([U|V],[U|Z]):-make_list(V,Z).

print_list(X):-reduce(X,[]),write(' '),write(nil).
print_list(X):-reduce(X,[FX|RX]),nl,write(FX),print_list(RX).

make_list_ldf(X,Z):-attach_output_var(X,Y),
                make_list(Y,A),
                remove_var(A,Z).

print_list_ldf(X):-attach_output_var(X,Y),print_list_ldf1(Y).
```

```
print_list_ldf1(X):-reduce(X,[]),write(' '),write(nil).
print_list_ldf1(X):-reduce(X,[FX|RX]),
                remove_var(FX,FX1),nl,write(FX1),print_list_ldf1(RX).

reduce_ldf(X,Y):-attach_output_var(X,X1),reduce(X1,Y1),remove_var(Y1,Y).
```

APPENDIX 2
F* UTILITIES

```
/*
================================================================
File fstarutils. Makes useful initializations.
================================================================
*/

:-op(650,xfx,=>).

simplified(true).
simplified(false).
simplified([]).
simplified([U|V]).

replace((A+B),Z,(Z is A+B)).
replace((A-B),Z,(Z is A-B)).
replace((A*B),Z,(Z is A*B)).
replace((A/B),Z,(Z is A/B)).
replace((A<B),Z,less_than(A,B,Z)).
replace((A>B),Z,greater_than(A,B,Z)).
replace((A>=B),Z,greater_than_equal(A,B,Z)).
replace((A=<B),Z,less_than_equal(A,B,Z)).
replace(equal(A,B),Z,equal(A,B,Z)).
replace(neg(X),T,neg(X,T)).
replace(cos(X),Z,cos(X,Z)).
replace(sin(X),Z,sin(X,Z)).

neg(true,false).
neg(false,true).

greater_than(U,A,true):-U>A,!.
greater_than(U,A,false).

less_than(U,A,true):-U<A,!.
less_than(U,A,false).

equal(A,A,true):-!.
```

```
equal(A,B,false).

greater_than_equal(A,B,true):-A>=B,!.
greater_than_equal(A,B,false).

less_than_equal(A,B,true):-A=<B,!.
less_than_equal(A,B,false).
```

APPENDIX 3

USING THE F* COMPILER

1.0 INSTRUCTIONS

Assume partitioning of function symbols into constructor symbols, and non-constructor symbols. Then, write an F* program, i.e. a collection of rewrite rules each of the form LHS=>RHS, satisfying the following three restrictions:

(a) LHS is of the form f(t1,..,tn), n>=0, f a non-constructor function symbol, and each ti is either a variable, or of the form c(X1,..,Xm), m>=0, c a constructor symbol, and each Xi a variable.

(b) There is at most one occurrence of any variable in LHS.

(c) All variables of RHS appear in LHS.

If the F* program satisfies two additional restrictions, much more efficient code can be generated:

(d) Let LHS1 and LHS2 be variants of heads of two rules in P having no variables in common. Then LHS1 and LHS2 do not unify.

(e) Let f(L1,..,Li,..,Lm)=>RHS be a rule in P, where Li is not a variable. Then, in every other rule f(K1,..,Ki,..,Km)=>RHS1 in P, Ki is not a variable.

Let <foo> be the file in which an F* program is to be defined. Ensure that the first line in <foo> is the operator declaration :-op(650,xfx,=>), and, for Quintus Prolog, => has been declared dynamic.

For each n-ary constructor symbol c in <foo>, include in <foo> a clause simplified(c(X1,..,Xn)) where X1,..,Xn are distinct variables. For convenience, the clauses for c=[],|,true,false have already been included in APPENDIX-2.

A lazy function symbol is one which is defined by F* rules. An eager function symbol is one which is defined in Prolog. Only right hand sides of F* rules can contain calls to eager functions. Let E be a term, possibly containing variables, in the right hand side of an F* rule. Let the outermost function symbol of E be eager. Then E must not contain any lazy function symbol. For example, where length is eager, and append is lazy, the term length(append([],[1])) must not appear in any F* rule. For each n-ary eager symbol f, do the following:

(a) Include in <foo>, the rule:

replace(f(X1,..,Xn),X,p(X1,..,Xn,X)).

where X1,..,Xn,X are distinct variables and f is computed by p in Prolog.

(b) Define p in Prolog. The first n arguments of p are assumed to be the n arguments of f. For convenience, +,-,*,/,neg, equal, >, <, >=, =< are all assumed to be eager and the appropriate replace clauses have been defined in APPENDIX-2.

NOTE. For Quintus Prolog, contents of APPENDIX-2 must be appended to <foo>. It is NOT sufficient to just load it.

Load this file, and <foo> into Prolog and type

translate_fdf.

To simplify some ground term e, type reduce(e,Z). If e denotes a list, and all its elements are to be obtained, type make_list(e,Z). If e denotes an infinite list, and each of its elements are only to be printed from left to right, type print_list(e).

If there are many => rules with more than one occurrence of a variable on their right hand sides, further optimization may be achieved by typing:

translate_ldf.

Now, in place of reduce, make_list and print_list, use respectively, reduce_ldf, make_list_ldf and print_list_ldf. The former will no longer work.

For Quintus Prolog only, if compilation is desired, after translate_fdf, or translate_ldf, select some filename <bar> and type:

write_functions(<bar>),compile(<bar>).

You may wish to start a fresh session of Quintus Prolog, before compiling, since in the current session, reduce clauses are dynamic.