Functional Computations in Logic Programs†

Saumya K. Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

David S. Warren
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract: While the ability to simulate nondeterminism and compute multiple solutions for a single query is a powerful and attractive feature of logic programming languages, it is expensive in both time and space. Since programs in such languages are very often *functional*, i.e. do not produce more than one distinct solution for a single input, this overhead is especially undesirable. This paper describes how programs may be analyzed statically to determine which literals and predicates are functional, and how the program may then be optimized using this information. Our notion of ‘‘functionality’’ subsumes the notion of ‘‘determinacy’’ that has been considered by various researchers. Our algorithm is less reliant on language features such as the *cut*, and thus extends more easily to parallel execution strategies, than others that have been proposed.

Categories and Subject Descriptors: D.3 [**Software**]: Programming Languages; D.3.2 [**Programming Languages**]: Language Classifications – *nonprocedural languages*; D.3.4 [**Programming Languages**]: Processors – *compilers*, *optimization*; I.2 [**Computing Methodologies**]: Artificial Intelligence; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving – *logic programming*.

---

† A preliminary version of this paper was presented at the Third International Conference on Logic Programming, London, July 1986.

This work was supported in part by the National Science Foundation under grant number DCR-8407688.

1. Introduction

In recent years, there has been a great deal of interest in logic programming languages, the best known of these being Prolog. The ability to simulate nondeterminism is a powerful feature of such languages. It permits the succinct and readily understandable expression of logical alternatives that require complex constructs in many programming languages. However, the additional runtime support needed for this, e.g. the ability to remember previous states and backtrack to them on failure, can incur a significant overhead. This is especially undesirable since predicates in logic programs are very often *functional*, and do not need this generalized backtracking ability. Knowledge about the functionality of predicates can be used to make significant improvements in the space and time requirements of a program. Knowing that a predicate is functional may make it possible, for example, to avoid having to record a system state to backtrack to, effect early reclamation of space on the runtime stack, and avoid unnecessary search.

Traditionally, the means of controlling Prolog's search has been through *cuts* inserted by the programmer. This, however, makes programs harder to understand and reason about declaratively [20, 22]. An alternative is to treat the cut as a low-level primitive that should be used infrequently by the programmer, if at all, but which may be introduced by compilers in the course of generating optimized code for execution in a sequential environment. In this view, the cut is seen, not as a language feature intrinsic to logic programming, but as an implementation feature of sequential Prolog. (It is not obvious whether cuts are very useful in parallel execution schemes.) To emphasize the distinction between user-supplied cuts and those generated by the compiler, we will refer to the latter as ''*savecp/cutto pairs*'' (the reason for these names is discussed in Section 5.1). It then becomes the responsibility of the compiler to determine which parts of the program involve redundant search that can be eliminated by inserting *savecp/cutto* pairs. This paper explores ways of doing this by inferring functionality of predicates and literals.

A special case of functionality, that of determinacy, has been investigated by Mellish [16], Naish [19] and Sawamura and Takeshima [23]. Deransart and Maluszynski, taking a different approach, have characterized such behavior of logic programs in terms of attribute grammars, but in the restricted setting of definite clause programs [Deransart Maluszynski 1985]. These authors have not considered the relationship between functional computations and negation by failure, or investigated connections with dependency theory in databases. A notion similar to that of functionality has been considered by Mendelzon in the restricted setting of databases, i.e. assuming that function symbols are absent, and that some predicates are defined entirely by ground facts [17]. Our approach is both more general and less operational. It does not rely exclusively on user-supplied cuts to infer functionality, thereby promoting what we believe is a better programming style. It also enables us to optimize certain cases where a particular call of a predicate may be functional even though the predicate itself is not.

The reader is assumed to be acquainted with the basic concepts of logic programming, an introduction to which may be found in [14]. The remainder of this paper is organized as follows: Section 2 introduces various concepts that are used later in the paper. Section 3 defines and discusses the notions of functionality and mutual exclusion. Section 4 describes an algorithm for the static inference of functionality. Section 5 discusses some compile-time optimizations that are possible with knowledge of functionality. Section 6 concludes with a summary.

2. Preliminaries

2.1. The Language

The language considered here is essentially that of first order predicate logic. It has countable sets of variables, function symbols and predicate symbols, these sets being mutually disjoint. Each function and predicate symbol is associated with a unique natural number called its *arity*. A function symbol of arity 0 is called a *constant*. A *term* is either a variable, or a constant, or a compound term $f(t_1, ..., t_n)$ where f is a function symbol of arity n, and the t_i are terms, $1 \le i \le n$. The *principal functor* of a term t is defined as follows: if t is a constant c, then the principal functor is c, while if t is a compound term $f(t_1, ..., t_n)$, then its principal fiunctor is f; the principal functor of a variable is undefined. An *atomic goal*, or *atom*, is of the form $p(t_1, ..., t_n)$, where p is an n-ary predicate symbol and the t_i are terms, $1 \le i \le n$.

A logic program consists of a finite set of predicate definitions. A predicate definition consists of a finite set of clauses. Each clause is a finite set of literals, which are either atomic goals or negations of atomic goals. The clauses are generally constrained to be definite Horn, i.e. have exactly one positive literal: the positive literal is called the *head* of the clause, and the remaining literals, if any, constitute the *body* of the clause; a clause with only negative literals is referred to as a *goal*. It is possible to have negative literals in the body of a clause. The semantics of negated goals is given in terms of unprovability by finite failure; it coincides with logical negation (with respect to the ‘‘completed’’ program) under certain conditions: the set of program clauses must have a minimal model, and each negated goal G to be proved must be of the form $\forall \bar{X}\, not(G_1)$ where $\bar{X}$ is a list of the variables occurring in G_1 at the time G is selected for resolution [14]. This is usually enforced by requiring that only ground negated goals be considered [4]. However, some systems permit negated goals to contain variables, provided such variables are explicitly quantified to comply with the requirement above [18].

The meaning of a clause is the disjunction of its literals; that of the program is the conjunction of its clauses. We adhere to the syntax of Edinburgh Prolog and write clauses in the form

$$\mathrm{p} :- \mathrm{q}_1, \ldots, \mathrm{q}_n.$$

which can be read as ‘‘p *if* q_1 *and* . . . *and* q_n’’. Variables are written starting with upper case letters, while predicate and function symbols are written starting with lower case letters. In addition, the following syntax is used for lists: the empty list is written [], while a list with head *Hd* and tail *Tl* is written [*Hd*|*Tl*]. The symbol ‘!’ is used to denote the Prolog atom *cut*, which provides a mechanism for controlling backtracking. Operationally, the effect of executing a cut is to discard all backtrack points up to that of the parent goal.

A literal is a static component of a clause. In an execution of the program, the corresponding dynamic entity is a call, which is a substitution instance of an alphabetic variant of the literal. A call to an n-ary predicate p can therefore be considered to be a pair $\$langle\ p/n, \bar{T}\$rangle$ where $\bar{T}$ is an n-tuple of terms which are arguments to the call. When the predicate being referred to in a call is clear from the context, we omit the predicate name and refer to the call by its tuple of arguments. Calls corresponding to a literal are said to *arise from* that literal. If the predicate symbol of a literal L is p, then L is said to *refer to* the predicate p.

The declarative semantics of logic programs is usually given in terms of the model theory of first order logic: the meaning of a logic program is its least Herbrand model [14, 24]. Computationally, however, predicates, clauses and literals in a program can be thought of as denoting binary relations over tuples of terms. Thus, if D is the set of terms (possibly containing variables) in a program, then an n-ary predicate (clause, literal, call) denotes a subset of $D^n \times D^n$. The first element of the pair represents the ''input'', or calling, values of its arguments, and the second, the ''output'', or returned values.† We refer to such relations as *input-output relations*. If a pair $\$langle\ t_1, t_2\$rangle$ is in the input-output relation of a predicate p, and θ is the most general unifier of t_1 and t_2, then a call t_1 to p can *succeed* with the substitution θ. Notice that for any pair $\$langle\ t_1, t_2\$rangle$ in an input-output relation, t_2 must be a substitution instance of t_1.

We sometimes wish to ignore values returned for ''void'' or ''anonymous'' variables, i.e. variables that occur only once in a clause. Given an n-tuple T and a set of argument positions $A = \{m_1, \ldots, m_k\}$, $1 \le m_1 < \ldots < m_k \le n$, let the *projection* of T on A denote the k-tuple obtained by considering only the values of the argument positions $m_1 \ldots m_k$ of T. Then, the projection of an input-output relation on a set of argument positions is defined as follows:

Definition: Given an input-output relation R and a set of argument positions A, the *projection of R on A*, written $\pi_A(R)$, is the set of pairs $\$langle\ S_I, S_O\$rangle$ such that for some pair $\$langle\ T_I, T_O\$rangle$ in R, S_I is the projection of T_I on A, and S_O is the projection of T_O on A. ●

This is analogous to the projection operation of relational databases. Notice that in ''pure'' logic programs, i.e. programs not containing nonlogical features such as *cut*, *var*, *nonvar*, etc., the projection of an input-output relation of a predicate p on a set of argument positions A is precisely the input-output relation obtained for a new predicate defined by p, but with the arguments in the head restricted to those in A. Thus, for $A = \{1, \ldots, k\}$ and R the input-output relation for an n-ary predicate p in a pure program, $\pi_A(R)$ coincides with the input-output relation of a predicate p' defined by the single clause

$$p'(X_1, \ldots, X_k) :- p(X_1, \ldots, X_n).$$

Additionally, we sometimes wish to restrict our attention to a ''horizontal slice'' of the input-output relation of a predicate. This is given by the notion of input restriction:

† This is a straightforward abstraction of denotational semantics that have been proposed for Prolog [9, 12]

Definition: Let R be the input-output relation for an n-ary predicate (clause, literal) in a program, and let T be a set of n-tuples of terms. The *input restriction* of R to T, written $\sigma_T(R)$, is the set of pairs *$langle* S_I, S_O*$rangle* in R such that S_I is in T. •

Implementations of logic programming languages may impose an ordering on the clauses and literals comprising a program. For example, Prolog implementations execute clauses according to their textual top-to-bottom order in the search for a proof, and resolve literals within a clause according to their textual left-to-right order. These orderings induce data and control dependencies that are crucial to the analysis of program properties such as functionality. To simplify the presentation, Prolog's evaluation ordering on clauses and literals is assumed throughout this paper. However, our algorithm is not dependent on this ordering in any fundamental way, and its adaptation to other execution orders is straightforward.

2.2. Modes

In general, Prolog programs are undirected, i.e. can be run either ''forwards'' or ''backwards'', and do not distinguish between ''input'' and ''output'' arguments for a predicate. However, in most programs, individual predicates tend to be used with some arguments as input arguments and others as output arguments. Knowledge of such directionality, expressed using *modes*, enables various compile-time optimizations to be made. Mode information can either be supplied by the user, in the form of mode declarations, or be inferred from a global analysis of the program [8, 16].

It is convenient to think of the mode of an n-ary predicate as representing a set of n-tuples of terms, or equivalently, a set of calls. The modes considered here are quite simple: **c** represents the set of ground terms, **f** the set of variables and **d** the set of all terms. Thus, if a predicate *p/3* has mode *$langle* **c**, **f**, **d***$rangle* in a program, then it will always be called with its first argument ground and its second argument uninstantiated in that program; however, nothing definite can be said about the instantiation of its third argument. In general, a mode for an n-ary predicate is an n-tuple over {**c**, **d**, **f**}. A call to a predicate with arguments $\bar{T}$ is *consistent* with the mode of that predicate if $\bar{T}$ is in the set of tuples of terms represented by that mode.

Given an ''input'' mode for a predicate or clause, it is possible to propagate it (from left to right, if we assume Prolog's evaluation order) to literals in the body and obtain modes for these literals [8]. The modes so inferred are said to be *induced* by the input mode. Thus, given the clause

$p(X, Y) :- q(X, Z), r(Z, Y).$

and the mode *$langle* **c**, **f***$rangle* for p, the induced mode for the literal $q(X, Z)$ is *$langle* **c**, **f***$rangle*. If we also know that q always binds its arguments to ground terms on success, then it can be inferred that if execution succeeds through $q(X, Z)$, Z will be bound to a ground term, so that the induced mode for $r(Z, Y)$ is *$langle* **c**, **f***$rangle*.

2.3. Functional Dependencies

Functional dependencies are well known in relational database theory. Given a predicate $p(\bar{X})$, where $\bar{X}$ is a sequence of distinct variables, if there exist subsets of its arguments $\bar{U}$ and $\bar{V}$ such that a ground instantiation of the arguments $\bar{U}$ uniquely determines the instantiation of the arguments $\bar{V}$, then $\bar{U}$ is said to *functionally determine* $\bar{V}$ in p, and $\bar{V}$ is said to *depend functionally* on $\bar{U}$. More formally, if $\bar{U}$ functionally determines $\bar{V}$ in $p(\bar{U}, \bar{V})$, and $\bar{u}$ is a ground instantiation of $\bar{U}$, then for all $\bar{v}_1$ and $\bar{v}_2$, whenever $p(\bar{u}, \bar{v}_1)$ and $p(\bar{u}, \bar{v}_2)$ are true, it must be the case that $\bar{v}_1 = \bar{v}_2$. We use the notation ''$L : S_1 \rightarrow S_2$'', where L is a literal and S_1 and S_2 are sets of variables occurring in L, to indicate that S_1 functionally determines S_2 in L, i.e. that if L is executed with ground instantiations for the variables in S_1, then the instantiations of the variables in S_2 are uniquely determined if the call succeeds.

The following axioms, known as Armstrong's axioms, are sound and complete for functional dependencies:

Reflexivity: If $S_2 \subseteq S_1$, then $L : S_1 \rightarrow S_2$ for any L.

Transitivity: If $L : S_1 \rightarrow S_2$ and $L : S_2 \rightarrow S_3$ then $L : S_1 \rightarrow S_3$.

Augmentativity: If $L : S_1 \rightarrow S_2$, and $S = S_0 \cup S_1$ for some S_0, then $L : S \rightarrow S_2$.

This extends in a natural way to conjunctions of literals, and to clauses: if L is a member of a conjunct C and $L : S_1 \rightarrow S_2$, then $C : S_1 \rightarrow S_2$; if Cl is a clause ''H :– B'' and $B : S_1 \rightarrow S_2$, then $Cl : S_1 \rightarrow S_2$.

Let S be a set of variables in a clause C, and F a set of functional dependencies that hold in the clause. The set of all variables in that clause that can be inferred to be functionally determined by S under F, using the axioms above, is called the *closure* of S under F. If S_2 is the closure of S_1 under a set of functional dependencies F, we write $C : S_1 \overset{*}{\underset{F}{\longrightarrow}} S_2$. Given a set of functional dependencies F and a set of variables S, the closure of S under F can be determined in time linear in the size of F [15].

3. Functionality

The notion of ''determinacy'' has usually been identified with ''having no alternatives''. For example, Sawamura et al. define determinacy as, essentially, ''succeeding at most once'' [23]; Mellish defines a goal as determinate if it ''will never be able to backtrack to find alternative solutions'' [16]. Unfortunately, such definitions are inherently operational in nature, and procedures to infer determinacy tend to rely heavily on the presence of cuts in the user's program. This has two drawbacks: it encourages bad programming style, and does not extend gracefully to parallel execution schemes even though such schemes would benefit from knowledge of determinacy.

We consider a more general property of predicates, *functionality*, where all alternatives produce the same result, which therefore need not be computed repeatedly. The difference between determinacy and functionality is illustrated by the following example:

Example 1: The predicate

$p(a)$.
$p(X) :- p(X)$.

is functional, since the set of solutions it produces is the singleton $\{p(a)\}$. However, since it can produce this solution infinitely many times, it is not determinate in the traditional sense. ●

Functionality subsumes determinacy: clearly, determinacy implies functionality; however, as the example above shows, the converse is not true.

Functionality can be considered at the level of literals, clauses and predicates. We define these notions as follows:

Definition: Let R be the input-output relation of a literal L, in a program and A the set of its non-void argument positions. Then, L is *functional relative to a call C* in the program iff $\pi_A(\sigma_{\{C\}}(R))$ is a function. A literal is *functional relative to a mode M* iff it is functional relative to every call consistent with M. ●

In other words, a literal is functional if any call that can arise from it is functional on its non-void arguments.

Definition: Let C be a call to a predicate (clause) with input-output relation R. The predicate (clause) is functional relative to C iff $\sigma_{\{C\}}(R)$ is a function. A predicate (clause) is functional relative to a mode M iff it is functional relative to every call consistent with M. ●

A somewhat different view of functionality may be had by considering SLD-resolution as a goal rewriting process. Then, a literal L is functional if SLD-resolution of L is Church-Rosser with respect to the substitutions returned for the variables of L [Downey communication]. The idea extends without difficulty to clauses and predicates.

If a literal (clause, predicate) is not functional, it is said to be *relational*. Notice that whether or not a predicate, clause or literal is functional is determined by its input-output relation, and need not have anything to do with whether arguments in calls to it are instantiated to ground terms on success. For example, the predicate defined by the single clause

$p(X, Y)$.

is functional − its input-output relation is the identity function over pairs of terms − even though it succeeds with the nonground arguments when called with nonground arguments. Not surprisingly, the functionality of a predicate is in general undecidable ([23] proves the recursive unsolvability of deciding the special case where a call can succeed at most once). However, sufficient conditions can be given for functionality:

Proposition 3.1: A literal $p(\overline{X})$ is functional relative to mode M if the predicate p is functional relative to mode M. *$box*

Proposition 3.2: A literal $p(\overline{X})$ is functional relative to mode M if there are subsets $\overline{U}, \overline{V} \subseteq \overline{X}$ such that (i) $\overline{U} \cup \overline{V} = \overline{X}$; ($ii$) $\overline{U}$ functionally determines $\overline{V}$ in $p(\overline{X})$; and (iii) in any call consistent with mode M, each argument in $\overline{U}$ is ground.

Proof: From the definition of functional dependency. *$box*

Functional dependencies are especially relevant here because, as many researchers have pointed out, logic programming languages are very well suited for querying relational databases, and functional dependencies are among the most frequently encountered integrity constraints for such databases. The detection of functionality is therefore important if futile searches through large relations are to be avoided. For the purposes of this paper, it is assumed that the relevant functional dependencies for base predicates have been supplied to the functionality analyzer.

Proposition 3.2 can be generalized to take void variables into account. Consider a literal $p(\overline{X})$, with $\overline{U}, \overline{V}, \overline{W} \subseteq \overline{X}$ such that $\overline{X} = \overline{U} \cup \overline{V} \cup \overline{W}$ where $p(\overline{X}) : \overline{U} \rightarrow \overline{V}$, and $\overline{W}$ consists only of void variables. Then, $p(\overline{X})$ can be rewritten as a literal $q(\overline{Y})$, where $\overline{Y} = \overline{U} \cup \overline{V}$ and q is a new predicate, defined by the clause

$q(\overline{Y})$:– $p(\overline{X})$.

From Proposition 3.2, the literal $q(\overline{Y})$ is functional relative to a mode M if for any call consistent with M, each argument in $\overline{U}$ is ground. This argument extends in a straightforward way to the original literal, $p(\overline{X})$, leading to the following proposition:

Proposition 3.3: A literal $p(\overline{X})$ is functional relative to a mode M if there are subsets $\overline{U}, \overline{V}, \overline{W} \subseteq \overline{X}$ such that (i) $\overline{X} = \overline{U} \cup \overline{V} \cup \overline{W}$; ($ii$) $\overline{U}$ functionally determines $\overline{V}$ in $p(\overline{X})$; (iii) in any call consistent with M, each term in $\overline{U}$ is ground; and (d) $\overline{W}$ consists only of void variables. *$box*

Example 3: Consider a predicate *emp*(*Id, Name, Dept, Sal, PhoneNo*), which is an employee relation whose arguments are the employee's identification number, name, the department he works for, his salary and phone number. Assume that the predicate has the functional dependencies $Id \rightarrow Name$ (''an employee can have only one name'') and $Id \rightarrow Sal$ (''an employee can have only one salary''). Then, the literal

emp(12345, *EmpName*, _ , *Sal*, _)

is functional. Here, the arguments {*Id*, *Name*, *Dept*, *Sal*, *PhoneNo*} can be partitioned into the sets {*Id*}, {*Name, Sal*} and {*Dept, PhoneNo*} where $Id \rightarrow Name$ and $Id \rightarrow Sal$, *Id* is a ground term in the literal and {*Dept, PhoneNo*} correspond to anonymous variables. However, the literal

emp(12345, *EmpName*, _ , *Sal*, *PhoneNum*) may not be functional, since an employee can have more than one phone number. ●

In general, a clause is functional if each literal in its body is functional. It is possible to permit literals that are not functional in the body, as long as they occur in contexts that are functional. The notion of a functional context can be defined more formally as follows:

Definition: A literal L occurs in a *functional context* in a clause C if the body of C is of the form ''G, !, G_1'' or ''G_1, *not*(G), G_2'', and L occurs in G. ●

This definition applies to simple Horn clauses extended with negation. It can be extended in a straightforward way to take other kinds of connectives, such as Prolog's *if-then-else* construct, into account.

Proposition 3.4: A clause C is functional if any literal L in its body that is not functional relative to its mode induced by M occurs in a functional context. *$box*

Notice that this proposition does not require the presence of cuts in the clause: if each literal in the body of the clause can be shown to be functional, then the clause can be inferred to be functional even if there are no cuts in the body. As stated, however, the proposition suffers from the problem that it may be sensitive to the order of literals in the body of the clause. Consider, for example, the program defined by

$q(a, b)$.
$q(c, d)$.
$r(b, c)$.
$r(d, e)$.
$p1(X, Y) :- q(X, Z), r(Z, Y)$.
$p2(X, Y) :- r(Z, Y), q(X, Z)$.

Assume that *p1* and *p2* have mode *$langle* **c**, **d***$rangle*, and that in both q and r, the first argument functionally determines the second. Each literal in the body of the clause for *p1* is functional from Proposition 3.2, and hence the clause for *p1* is also functional relative to the mode *$langle* **c**, **d***$rangle*. Now consider the clause for *p2*: the induced mode for the literal $r(Z, Y)$ is *$langle* **d**, **d***$rangle*, so the literal is not functional. Further, this literal does not occur in a functional context. Thus, the clause cannot be inferred to be functional from Proposition 3.4.

The problem here is that the functional dependencies are being encountered in the wrong order. A simple solution that suggests itself is to try reordering the literals in the body of the clause. However, this does not always work: consider the clause

$p(X, Z) :- q(X, Y, U, Z), r(Y, U)$.

Assume that p has mode *$langle* **c**, **d***$rangle*; that the functional dependencies $X \rightarrow Y$ and $U \rightarrow Z$ hold in $q(X, Y, U, Z)$, and the functional dependency $Y \rightarrow U$ holds in $r(Y, U)$. Further, assume that both q and r

succeed binding their arguments to ground terms. Then, the clause is functional. However, it cannot be shown to be functional from Proposition 3.4 by reordering the literals in its body.

The key to the problem lies in determining what other functional dependencies are implied by those that are already known to hold in the clause:

Proposition 3.5: Let F be the functional dependencies that hold in a clause C, and S the set of variables appearing in the head of C. Given a mode M for C, let $gd(M) \subseteq S$ be the set of variables appearing in arguments in the head of the clause that are guaranteed to be ground in any call consistent with M, and let $C : gd(M) \underset{F}{\overset{*}{\longrightarrow}} S_0$. Then, C is functional relative to mode M if $S \subseteq S_0$.

Proof: By definition, S_0 is the closure of $gd(M)$ under F. Further, M guarantees that each variable in $gd(M)$ will be instantiated to a ground term in any call to C. It follows that in any call to C consistent with M, each variable in S_0 will be uniquely determined on success. Since $S \subseteq S_0$, it follows that any call to C consistent with M will, if it succeeds, succeed with its variables uniquely determined. Thus, C is functional. *$box*

Returning to the example above, given the mode M = *$langle* **c**, **d***$rangle* for the clause

$p(X, Z) :- q(X, Y, U, Z), r(Y, U).$

the set $gd(M)$ is $\{X\}$, and the closure of $\{X\}$ under the functional dependencies $\{X \rightarrow Y, Y \rightarrow U, U \rightarrow Z\}$ is $\{X, Y, U, Z\}$. Since the set of variables appearing in the head of the clause is contained in this closure, the clause is functional relative to mode M. Notice that using this proposition, it may be possible to infer a clause to be functional even though it contains literals that neither have functional dependencies nor are functional or occur in a functional context. For example, the clause

$p(X, Z) :- q(X, Y, U, Z), r(Y, U), s(U, W).$

with functional dependencies $\{X \rightarrow Y, Y \rightarrow U, U \rightarrow Z\}$ can be inferred to be functional relative to mode *$langle* **c**, **d***$rangle* irrespective of whether the literal $s(U, W)$ is functional.

The rules given above enable us to reason about the functionality of literals and clauses. The next step is to extend them to allow reasoning about the functionality of predicates. A sufficient condition for the functionality of a predicate in a program is that each clause of the predicate be functional, and further that at most one clause succeed for any call to that predicate in that program. The latter is expressed using the notion of mutual exclusion of clauses:

Definition: Two clauses $C1$ and $C2$ for a predicate, with input-output relations R_{C1} and R_{C2} respectively, are *mutually exclusive relative to a call G* iff either $\sigma_{\{G\}}(R_{C1}) = \emptyset$ or $\sigma_{\{G\}}(R_{C2}) = \emptyset$. Two clauses for a predicate are *mutually exclusive relative to a mode M* iff they are mutually exclusive relative to every call consistent with M. ●

In other words, if two clauses of a predicate are mutually exclusive relative to a call G, then it is not possible for G to succeed through both clauses. Clauses that are mutually exclusive relative to all calls to the corresponding predicate in a program are referred to simply as *mutually exclusive*. The following propositions establish sufficient conditions for the static determination of mutual exclusion among clauses.

Proposition 3.6: Two clauses are mutually exclusive relative to any mode if there is a cut in the body of the textually antecedent clause. *$box*

Proposition 3.7: Two clauses are mutually exclusive relative to a mode M if there is a subset $\bar{U}$ of the argument positions in their heads whose values are not unifiable, and each term in $\bar{U}$ is ground in any call to the corresponding predicate that is consistent with M. *$box*

Example 4: The clauses

$p(a, f(X), Y) :- q1(X, Y).$
$p(b, f(g(Y)), Z) :- q2(Y, Z).$

are mutually exclusive relative to the mode *$langle* **c**, **d**, **d***$rangle*; however, they may not be mutually exclusive relative to the modes *$langle* **d**, **c**, **d***$rangle* or *$langle* **d**, **d**, **c***$rangle*. ●

It is possible to weaken this condition somewhat, so that the relevant terms in the calls are not required to be ground, as long as they are ''sufficiently instantiated'' to discriminate between the clauses. While the extension is conceptually straightforward, it needs a more expressive language for the specification of variable instantiations than the simple mode set {**c**, **d**, **f**} considered in this paper; this is not pursued further here.

Proposition 3.8: Two clauses for a predicate p of the form

$p(\bar{X}) :- G_{11}, r(\bar{Y}_0), G_{12}.$
$p(\bar{X}) :- G_{21}, \forall not(r(\bar{Y}_1)), G_{22}.$

where each of the G_{ij} consist of zero or more literals, are mutually exclusive relative to a mode M if (*i*) each literal in G_{11} and G_{21} is functional relative to its mode induced by M, and (*ii*) for any call to p that is consistent with M, the call arising from the literal $r(\bar{Y}_0)$ in the first clause is subsumed by the call arising from the literal $\forall not(r(\bar{Y}_1))$ in the second.

Proof: If each literal in G_{11} and G_{21} is functional relative to its mode induced by M, then for any call consistent with M, there can be at most one call arising from each of the literals $r(\bar{Y}_0)$ and $\forall not(r(\bar{Y}_1))$. Assume that the subsumption condition of the proposition is satisfied for these. Then, at runtime, if the goal $r(\bar{Y}_0)$ is called with substitution σ and succeeds, then the goal $r(\sigma(\bar{Y}_1))$ will also succeed. Therefore, the goal $\forall not(r(\sigma(\bar{Y}_1)))$ will fail. Conversely, the goal $\forall not(r(\bar{Y}_1))$ can succeed only if no instance of $r(\bar{Y}_1)$ succeeds, which means that $r(\bar{Y}_0)$ must fail. Thus, the two clauses can never both succeed for any call consistent with M, i.e. they are mutually exclusive relative to M. *$box*

For this proposition to hold, it is necessary that $r(\bar{Y}_1)$ subsume $r(\bar{Y}_0)$ in the program. This does not guarantee subsumption at runtime, of course, but sufficient conditions for runtime subsumption can be given. One such sufficient condition is that G_{11} and G_{21} be identical and not share any variables with $r(\bar{Y}_1)$; another is that none of the literals in G_{21} instantiate any variables, e.g. when they are all negated literals.

It is easy to see how to extend this proposition to situations where two literals can be inferred to be complementary even though they do not have the same predicate symbol, e.g. from knowledge about builtin predicates.

Example 5: The clauses

$length([], N) :- N =:= 0.$
$length([H \mid L], N) :- N > 0, plus(M, 1, N), length(L, M).$

are mutually exclusive relative to the mode $\langle \mathbf{d}, \mathbf{c} \rangle$, because '$N$ =:= 0' implies '*not*(N > 0)', and 'N > 0' implies '*not*(N =:= 0)'. ●

Proposition 3.8 assumes a finite failure semantics for negation. Analogous conditions can be found for other treatments of negation as well. It gives a weaker condition for the mutual exclusion of clauses than requiring cuts in their bodies, since the clauses

$p(\bar{X}) :- q(\bar{Y}_0), r(\bar{Y}).$
$p(\bar{X}) :- not(\ q(\bar{Y}_1)\ ), s(\bar{Z}).$

where $q(\bar{Y}_1)$ subsumes $q(\bar{Y}_0)$, is not equivalent to the clauses

$p(\bar{X}) :- q(\bar{Y}_0), !, r(\bar{Y}).$
$p(\bar{X}) :- s(\bar{Z}).$

if q($\bar{Y}_0$) is not functional. Proposition 3.8 is applicable even in parallel evaluation contexts, while conditions involving cuts do not extend naturally to execution strategies that are not sequential.

From the point of view of inference, we distinguish between two kinds of mutual exclusion: that which can be inferred without any knowledge of the functionality of any user-defined predicate or literal, and that which requires knowledge of the functionality of user-defined predicates. The former is referred to as *simple mutual exclusion*, the latter as *derived mutual exclusion*. Note that in Proposition 3.8, in the case where G_{11} and G_{21} consist of built-in predicates whose functionality is known, the proposition can be used to infer simple mutual exclusion.

Proposition 3.9: A predicate is functional relative to a mode M if its clauses are pairwise mutually exclusive relative to mode M, and each clause is functional relative to mode M.

Proof: Since the clauses are pairwise mutually exclusive relative to mode M, at most one clause can succeed for any call to the predicate consistent with M. Since each clause is functional relative to M, any invocation of it can succeed in at most one way. Hence any call to the predicate consistent with M can succeed in at most one way, i.e. the input restriction of its input-output relation to these calls is a function. *$always*

4. Functionality Inference

The basic idea in the inference of functionality is to solve a set of simultaneous, possibly recursive, equations over a set of propositional variables. This is similar to the technique for the inference of determinacy used by Mellish [16]. As an example, consider a predicate p whose definition is of the form

$$
\begin{array}{ll}
(cl_1) & p :- p_{11}, p_{12}, \ldots, p_{1n_1}. \\
(cl_2) & p :- p_{21}, p_{22}, \ldots, p_{2n_2}. \\
 & \quad\ldots \\
(cl_m) & p :- p_{m1}, p_{m2}, \ldots, p_{mn_m}.
\end{array}
$$

By Proposition 3.9, p is functional if each of its clauses $cl_1, \ldots, cl_m$ is functional, and they are pairwise mutually exclusive. A strengthening of this condition to *if and only if*† induces a set of equations of the form

$$func\_p = MutEx_p \; \$and \; func\_cl_1 \; \$and \; \ldots \$and \; func\_cl_m$$

where each of the variables is propositional: *func_p* is true only if p is functional, $func\_cl_1$ if clause cl_1 is functional, and so on; $MutEx_p$ is true only if the clauses for p are pairwise mutually exclusive. The functionality of each clause depends, from Proposition 3.4, on the functionality of the literals in its body. This allows us to add the equations

$$
\begin{array}{l}
func\_cl_1 = func\_p_{11} \; \$and \; func\_p_{12} \; \$and \; \ldots \$and \; func\_p_{1n_1}. \\
\qquad\qquad \ldots \\
func\_cl_m = func\_p_{m1} \; \$and \; func\_p_{m2} \; \$and \; \ldots \$and \; func\_p_{mn_m}.
\end{array}
$$

Each of the variables *func_p*, $func\_p_{11}$, $func\_cl_1$ etc., is referred to as a *functionality status flag*. Equations are also set up for the propositional variable $MutEx_p$ if necessary.

We first present an algorithm that takes only simple mutual exclusion of clauses into account. The algorithm is proved sound. The fact that only simple mutual exclusion is considered does not affect soundness, but results in some predicates, which are actually functional, being inferred to be relational. A later section considers derived mutual exclusion and shows how it can be reduced to simple mutual exclusion, so that the algorithm and its soundness proof go through as before.

† Note that this strengthening is conservative, i.e. it may lead to a loss of precision but not of soundness.

4.1. Functionality Inference with Simple Mutual Exclusion of Clauses

4.1.1. An Algorithm

The algorithm for functionality analysis takes as input a set of clauses comprising the program, a set (possibly empty) of modes for predicates in the program, and a set (possibly empty) of functional dependencies. The output is an annotated program, where each predicate, clause and literal is annotated with a flag that indicates whether or not it is functional. Associated with each literal, clause and predicate A is a flag $A.fstat$, its *functionality status*, that ranges over {⊥, **true**, **false**} and initially has the value ⊥. The idea behind the algorithm is to first detect predicates whose clauses are not pairwise simple mutually exclusive and set their functionality status flags to **false**, and then to propagate these values in the program call graph in a depth first manner.

Nodes in the call graph represent predicates in the program. The node corresponding to a predicate p contains the functionality status flag $p.fstat$ for that predicate, initialized to ⊥, together with a bit *p.visited* that initially has the value **false**. The set of edges in the call graph of the program is denoted by *CG_EDGES*: if *$langle p, q$rangle* ∈ *CG_EDGES* then there is a directed edge from p to q in the call graph. The graph is represented using adjacency lists. The algorithm also maintains, as an auxiliary data structure, a stack of predicates *RELPREDS* that is initially empty. Pseudocode for the algorithm is given in Figure 1.†

The algorithm proceeds in three stages. In the first stage, various flags are initialized and the call graph is constructed. Functionality status flags are initialized to ⊥, unless the value of the flag can be determined *a priori* without any information about the functionality of any other predicate. Thus, literals for built-in predicates that are known to be functional, negated literals, and literals that can be determined to be functional from Propositions 3.1, 3.2 or 3.3, have their functionality status set to **true** at this stage. Then, if a clause can be inferred to be functional based on Proposition 3.4 or Proposition 3.5, its functionality status is set to **true**. If the clauses for a predicate cannot be determined to be pairwise simple mutually exclusive, then its functionality status is set to **false**, and it is marked as visited; otherwise, if each of its clauses has been inferred to be functional, then its functionality status is set to **true** and it is marked as visited (recall that it is possible for a clause to be functional, based on functional contexts and functional dependencies, even if its body contains literals that are not functional: in such cases, a predicate can be functional even if it calls relational predicates). If a literal $L \equiv q(\ldots)$ is inferred to be functional at this stage, or occurs in a functional context in a clause for a predicate p, then whether or not the predicate q is functional has no influence on whether p is functional, so L is not considered when the call graph is constructed.

The order in which predicates are processed may affect which predicates have their functionality status flags set to **true** in this stage. This is because of the possibility of mutual recursion: consider

† The algorithm is presented in this form, rather than more declaratively in a language such as Prolog, in order to simplify subsequent reasoning about computational aspects of the algorithm, such as termination and asymptotic complexity.

predicates p, and q that are mutually recursive:

p :– . . ., q, . . .

. . .

q :– . . ., r, . . .

Suppose p is processed first: then, since p is relational if q is relational, it is necessary to add an edge $\langle p, q\rangle$ in the call graph. If, during the subsequent processing of q, it is discovered from Proposition 3.5 that the clauses for q are functional independently of whether or not p is functional, then *q.fstat* is set to **true**. However, this is information is not propagated back to p. If q were to be processed before p, however, the fact that q is functional would be detected when p was being processed. It is easy to see from this example that no particular order of processing the predicates, e.g. a postorder traversal of a depth-first spanning tree of the program call graph, will always detect every predicate that can be detected as functional in the first stage of the algorithm. As a result, some predicates may have their functionality status set to $\perp$ at the end of the first stage even though they are actually functional. Such predicates have their functionality status flags set in the third stage.

Once functionality status flags have been initialized, they are propagated iteratively in the second stage of the algorithm. The rule for propagation is given by Proposition 3.9. Whenever the functionality status of a predicate q is set to **false**, this value is propagated to any predicate p that has a clause in which a literal referring to q occurs in a non-functional context. As shown above, it is possible to have an edge $\langle r, s\rangle$ in *CG_EDGES* where it is not known, in Stage I, whether s will be functional or not. Now if *s.fstat* is set to **false**, this information has to be propagated back to r; but if *s.fstat* is **true**, then it does not necessarily follow that r is also functional. Thus, when propagating functionality status values along the call graph, it is necessary to ensure that the only value being propagated is **false**.

At the end of the iteration, every predicate that can be inferred to be relational has its functionality status set to **false**. It is possible that some predicates still have a functionality status of $\perp$: these are set to **true** in the third stage of the algorithm. It is also possible that some clauses and literals may still have functionality status flags with the value $\perp$. These are also set appropriately at this stage.

4.1.2. Correctness

To establish the soundness of the algorithm, we show that any predicate inferred to be functional is in fact functional. To this end, it suffices to show that any predicate that is relational has its functionality status set to **false** when the algorithm terminates. Define the notion of *lowest common ancestor* for nodes in a tree as follows: a node n_0 is a *common ancestor* of a set of nodes N in a tree T if n_0 is an ancestor of n in T for each $n \in N$; n_0 is the *lowest common ancestor* of N in T if n_0 is a common ancestor of N, and there is no other common ancestor $n_1 \neq n_0$ of N such that n_0 is an ancestor of n_1. Each literal in a program can give rise to a number of calls during execution, each call defining a tree called a *search tree* or an *AND/OR tree*. The algorithm can be thought of as computing over an abstraction of these trees, which can be characterized by defining the notion of *relational depth*:

Definition: Let T_C be the search tree corresponding to a call C in a program. For each pair of succesful leaf nodes *$langle i, j$rangle* in T_C giving distinct sets of substitutions for the non-void variables in C, let $\delta(i, j)$ be the depth of the lowest common ancestor of the pair in T_C. The relational depth $\rho(C)$ of the call C is defined to be the least depth $\delta(i, j)$ of all such pairs *$langle i, j$rangle*, if the call is relational, and ∞ otherwise. ●

Definition: Let C be the set of calls to a relational predicate p in a program. The relational depth of the predicate p is defined to be $\min\limits_{c \in C} \rho(c)$. ●

Intuitively, the relational depth of a call is the least depth in the search tree for that call at which a predicate is encountered whose clauses are not pairwise mutually exclusive. The relational depth of a predicate is simply the least relational depth of all calls to it that can arise in the program.

Lemma 4.1: If a literal $L \equiv p(\bar{T})$ is relational in a program, then the predicate p is also relational in that program.

Proof: Assume that p is functional in the program. It follows, from Proposition 3.1, that L must also be functional in the program, which is a contradiction. *$box*

Lemma 4.2: If a predicate p is relational in a program, then either

(1) the clauses for p are not pairwise simple mutually exclusive, and the functionality status flag of p is set to **false** in Stage I of the algorithm; or

(2) there is a relational predicate q in the program, such that there is an edge $\langle p, q \rangle$ in the call graph of the program.

Proof: If p is relational, then its clauses are either pairwise simple mutually exclusive, or they are not. If they are not pairwise simple mutually exclusive, then from the soundness of Propositions 3.6, 3.7 and 3.8, it follows that p's functionality status flag will be set to **false** in Stage I of the algorithm.

If p's clauses are pairwise simple mutually exclusive, then since p is relational, it must be the case that there is a literal $L \equiv q(\ \ldots\ )$ in the body of a clause C for p, such that L is relational and does not occur in a functional context in C. From Lemma 4.1, it follows that q is relational. In this case, it can be seen from the algorithm that an edge $\langle p, q \rangle$ is added to the call graph of the program in Stage I. *$box*

Lemma 4.3: The relational depth of a relational call (predicate) is finite, and fixed for a given program.

Proof: If the call is relational, then there is a pair of successful leaf nodes in its search tree that give distinct substitutions for at least one non-void variable in the call. These leaf nodes are at finite depths in the search tree, hence their lowest common ancestor is also at a finite depth. This establishes that the relational depth is finite for a relational call. That it is fixed for a given program follows from the fact that the search tree defined by the call is fixed for a given program. The statement for relational predicates follows from the fact that the relational depth of a relational predicate is the least relational depth of all calls to it in the program, which must be finite, and fixed with respect to the program. *$box*

Theorem 4.4 (Soundness): If a predicate is relational in a program, then its functionality status is inferred to be **false** by the algorithm.

Proof: By induction on the relational depth N of the predicate.

If $N = 0$, then the clauses of p cannot be mutually exclusive, and hence cannot be pairwise simple mutual exclusive. From Lemma 4.2, it follows that the functionality status of p is set to **false** in Stage I of the algorithm. Since the value of a functionality status flag does not change once it has been set to **false**, the functionality status of p will be **false** if the algorithm terminates.

Assume that the theorem holds for predicates with relational depth less than k, $k > 0$. Consider a predicate p with relational depth $N = k$. Since $N > 0$, the clauses of p are mutually exclusive. Consider the search tree corresponding to a call $p(\bar{T})$ for which the relational depth is k (there must be such a call, since the relational depth of the predicate p is k). Let the root of this tree be α. There is a node χ, at depth k in this tree, that has two paths leading to successful leaf nodes that yield distinct sets of substitutions for the variables in the call (see Figure 2).

Consider the subtree S, rooted at a node β that is a child of α, that contains the node χ. Let β be labelled by the goal $L_1, \ldots, L_m$. Then, there is a clause C, with head H and body B, in the program, such that $p(\bar{T})$ and H are unifiable with most general unifier θ, and $\theta(B) \equiv L_1, \ldots, L_m$. Consider the goal $L_1, \ldots, L_m$: for some literal $L_l \equiv q_l(\ldots)$, $1 \le l \le m$, the predicate q_l must have relational depth less than k. From the soundness of Propositions 3.1, 3.2 and 3.3, L_l is not inferred to be functional in Stage I. It follows from Lemma 4.2 that the edge *$langle p*, q_l*$rangle* is in *CG_EDGES*. From the induction hypothesis, q_l.*fstat* is set to **false** by the algorithm. Assume that this is set in the $i^{th.}$ iteration of the algorithm. Then, it is evident from the algorithm that q_l is inserted into *RELPREDS* in the $i^{th.}$ iteration. At the end of this iteration, therefore, *RELPREDS* is nonempty, and the iteration does not stop at this point. Since *$langle p*, q_l*$rangle* is in *CG_EDGES*, it follows that *p.fstat* is set to **false** in a later iteration, if it has not already been set earlier. Since functionality status flags do not change once they have been set to **false**, the functionality status of p remains set to **false**. From Lemma 4.3, every relational predicate has finite relational depth. This implies that every relational predicate has its functionality status set to **false** when Stage II of the algorithm terminates.

That the functionality status of each relational literal and clause is set to **false** at the end of Stage III follows from the fact the functionality status of each relational predicate is set to **false** at the end of Stage II of the algorithm, as proved above.

A program contains only finitely many predicates, clauses and literals, and hence only finitely many functionality status flags. This implies that the call graph of the program is finite. The value of a functionality status flag can only go from $\perp$ to **true** or **false**, never in the other direction, so the value of a functionality status flag can change at most once. A predicate p is added to *RELPREDS* only if *p.visited* is **false**, and once its node in the call graph has been visited, *p.visited* is set to **true**, so the node for p is not visited again. This means that a predicate is not added to *RELPREDS* more than once. Since *RELPREDS* is always finite and has length bounded by the number of predicates defined in the program, each predicate added to it is eventually be processed and removed from it. This implies that the algorithm eventually terminates. *$always*

4.1.3. An Example

The following example illustrates how the algorithm works:

Example 6: Consider the program defined by the clauses

p([ ], [ ]).
p([*X*|*L1*], [*Y*|*L2*]) :– *q*(*X*, *Z*, _), *r*(*Z*, *Y*), *p*(*L1*, *L2*).

q(0, 0, 0).
q(0, 0, 1).
q(1, 1, 0).
q(1, 1, 1).

r(*X*, *Y*) :– *even*(*X*), *Y* is 2 * *X* + 1.
r(*X*, *Y*) :– *not*(*even*(*X*)), *Y* is 2 * *X*.

s(*X*, *Y*, *Z*) :– *q*(*X*, *Y*, *Z*), *r*(*X*, *Z*).

even(*X*) :– *X* mod 2 =:= 0.

Assume that it is known that *p/2* has the mode *$langle* **c**, **f** *$rangle*, *q/3* has the mode *$langle* **c**, **f**, **f** *$rangle* and that in the predicate *q*(*X, Y, Z*), the functional dependency $X \rightarrow Y$ holds. At the end of Stage I of the algorithm, the only predicate whose clauses cannot be inferred to be pairwise mutually exclusive is q. All functionality status flags except *q.fstat* therefore have value **true**, and *RELPREDS* = [q]. The call graph for the program (with nodes for built-in predicates omitted) is given in Figure 3.

In the first iteration in Stage II of the algorithm, *s.fstat* is set to **false**, and s is added to *relpreds*. Thus, after the first iteration of the while-loop, *RELPREDS* = [s]. However, since there is no node with an edge going to s, there is no change to any functionality status flag after this, and the while loop terminates after the next iteration.

The third stage of the algorithm then results in the functionality status of the clause for s getting the value **false**.

The output of the algorithm, therefore, is that the predicates p, r and *even* are functional, the literal referring to q in the recursive clause for p is functional, but that the predicates q and s are relational. ●

4.1.4. Complexity

Consider a program with p predicates of arity a, each predicate with at most m clauses, each clause containing at most n literals; assume that the number of functional dependencies that hold in the program is F. There are at most mp clauses and mnp literals in the program, so that the number of functionality status flags is $mnp + mp + p$.

Inspecting the literals in Stage I of the algorithm to determine whether they are functional takes time $O(mnpa)$. Determining the closure of the functional dependencies takes time $O(F)$; each clause has $O(a)$ variables in the head, so checking whether each clause is functional according to Proposition 3.5 takes time $O(a)$. The complexity of checking for functionality of clauses in Stage I is therefore $O(F + ma)$. The checking of pairwise mutual exclusion of m clauses takes time $O(m^2a)$, and this has to be done for p predicates. Typically, m can be expected to be $O(n)$, whence m^2ap is $O(mnap)$.† Letting $mnap = N$ be the size of the program, the time complexity of Stage I is therefore $O(mnap + F + ma + m^2ap) = O(N + F)$. If there are E edges in the call graph of the predicate, then the worst case complexity of Stage II, which is a depth first search of a graph with p nodes and E edges, is $O(max(p, E))$. In Stage III, it may be necessary to test and set $O(mnp)$ functionality status flags, which takes time $O(mnp)$. The overall time complexity of the algorithm is therefore $O(N + F + max(p, E))$.

The space used for the various functionality status flags is $O(N)$. The space required to store F functional dependencies is $O(F)$. Since there are p vertices and E edges in the call graph for the program, its adjacency list representation requires $O(E + p)$ space. Since the iterations in the algorithm can be accomplished in $O(1)$ space, the space complexity is $O(N + F + E + p) = O(N + F + E)$.

4.2. Functionality Inference for Derived Mutual Exclusion

The algorithm for functionality inference given earlier considers only simple mutual exclusion of clauses. It is possible to expand the set of predicates inferred to be functional if derived mutual exclusion is also taken into account. The problem of functionality inference in the presence of derived mutual exclusion of clauses turns out to be easily reducible to the case of simple mutual exclusion, so that results from the previous sections remain directly applicable.

From Proposition 3.8, two clauses for a predicate p of the form

$$p(\bar{X}) :- G_{11}, r(\bar{Y}_0), G_{12}.$$
$$p(\bar{X}) :- G_{21}, not(\ r(\bar{Y}_1)\ ), G_{22}.$$

are mutually exclusive relative to a mode M if the literals in G_{11} and G_{21} are functional relative to their modes induced by M; if $r(\bar{Y}_1)$ subsumes $r(\bar{Y}_0)$; and if the clauses meet some other conditions to satisfy Proposition 3.8, e.g. if G_{11} and G_{21} are identical and functional, and variable-disjoint with $r(\bar{Y}_0)$; or if the

† This may not be true for database predicates, which are defined by a large number of unit clauses. As mentioned, however, it is assumed that the relevant information for such predicates is supplied separately in the form of functional dependencies. Thus, database predicates need not be considered here.

literals in G_{21} are all negated. These clauses are derived mutually exclusive if any of the literals in G_{11} or G_{21} refers to a user-defined predicate. In this case, mutual exclusion cannot be detected in Stage I of the algorithm. Notice, however, that there are two components to inferring derived mutual exclusion: (*i*) the checking of conditions, such as subsumption of $\bar{Y}_0$ by $\bar{Y}_1$, and variable disjointness between G_{11} and $r(\bar{Y}_0)$, that can be performed in Stage I; and (*ii*) the verification of the functionality of user-defined predicates in G_{11} and G_{21}, which has to be deferred to Stage II of the algorithm. Once the subsumption of literals and the satisfaction of the other constraints has been verified in Stage I, derived mutual exclusion depends only on the functionality of user-defined predicates. This can now be handled simply by adding an equation describing this condition to the system of equations considered earlier. For example, if derived mutual exclusion for the clauses of a predicate p depends on a set of user-defined predicates Q, then the equation that is added is

$$MutEx_p = \bigwedge_{q \in Q} func\_q$$

where $MutEx_p$ is the mutual exclusion bit for p, and $func\_q$ is the functionality status of a predicate q. This gives a set of equations where the functionality of each predicate depends only on those of other predicates. These equations can be solved as described earlier. The reader may verify that the space and time complexity in this case remains linear in the size of the program.

5. Functional Optimizations

This section considers some of the optimizations that can be made with knowledge about functionality and mutual exclusion.

5.1. Controlling Backtracking: savecp and cutto

One of the functional optimizations discussed is the insertion of cuts. For this, we briefly describe the primitives used in our system to implement *cut*. In any implementation of cut, it is necessary to know how far to cut back to in the stack of choice points. One way of doing this is to note the current choice point at an appropriate point in execution, and cut back to this point when a cut is encountered. In our system, this is done via two primitives, *savecp*/1 and *cutto*/1. These are internal primitives that are introduced by the compiler, and are unavailable to the user. The call *savecp*(X) saves the current choice point in X, while the call *cutto*(X) sets the current choice point to be that saved in X. Thus, a predicate with a cut in it,

$p(\bar{X}) :- q(\bar{Y}), !, r(\bar{Z}).$
$p(\bar{X}) :- s(\bar{U}).$

is transformed by the compiler to

$p(\bar{X}) :- savecp(W), p1(\bar{X}, W).$

$p1(\bar{X}, W) :- q(\bar{Y}), cutto(W), r(\bar{Z}).$
$p1(\bar{X}, \_) :- s(\bar{U}).$

where W is a new variable not occurring in the original definition of p, and $p1$ is a new predicate not appearing in the original program. In general, *savecp* and *cutto* can be used to bracket calls whose choice points are to be cut.

5.2. Functionality and the Insertion of Cuts

If a call is functional, it can succeed with at most one distinct answer. Once this answer has been obtained, further search for other solutions for that call cannot produce any new solutions. A *savecp/cutto* pair may therefore be inserted by the compiler around the corresponding literal without (in most cases) affecting the semantics of the program. (There are certain non-logical contexts in which cuts so introduced can affect program semantics: this is discussed later.)

It is usually profitable to insert *savecp/cutto* pairs around functional literals referring to nonfunctional predicates, as in Example 3 above. If a predicate is itself functional, then it is generally preferable to insert *savecp/cutto* pairs in the clauses defining it, rather than around literals referring to it. A point to note is that when inserting cuts in clauses, care should be exercized to ensure that opportunities for tail recursion optimization are not being lost as a result.

Example 7: Consider the predicate

big_shot(*Id*, *EmpName*) :– *emp*(*Id*, *EmpName*, _, *Salary*, _), *Salary* > 100000.

Given the mode *$langle* **c**, **d***$rangle* for *big_shot/2* and the functional dependencies $Id \rightarrow Name$, $Id \rightarrow Salary$ for the predicate *emp*(*Id, Name, Dept, Salary, PhoneNo*), it can be inferred that the literal referring to *emp* in the clause above is functional. Since the predicate *emp* is not itself functional, this should be transformed to

big_shot(*Id*, *EmpName*) :–
  savecp(*X*), *emp*(*Id*, *EmpName*, _, *Salary*, _), *cutto*(*X*), *Salary* > 100000.

•

Further improvement is possible if we consider sequences of functional literals. Define a *fail_back_to* relation over pairs of literals, such that *fail_back_to*(*p1, p2*) is true if execution should backtrack to the (most recent) goal corresponding to literal *p2* if a goal corresponding to literal *p1* fails (this relation is fairly trivial given Prolog's naive backtracking strategy, but nontrivial *fail_back_to* relations can be given for more sophisticated backtracking strategies [1, 3]). For functional calls, the *fail_back_to* relation is transitive. In other words, given a sequence of literals

$$\ldots p1, \ldots, p2, \ldots, p3, \ldots$$

where *p1*, *p2* and *p3* are functional literals and the *fail_back_to* relation has the tuples *$langle p3, p2$rangle* and *$langle p2, p1$rangle*, execution can fail back directly to *p1* on failure of the goal *p3*, i.e. *$langle p3, p1$rangle* is in the *fail_back_to* relation. This property can be used to produce more efficient code for contiguous functional calls, as the example shows:

Example 8: Consider the clause

$p(X, Y) :- q(X, Z, \_), r(X, Y, \_), s(Y, Z).$

Assume that *p* has the mode $\langle \mathbf{c}, \mathbf{f} \rangle$, so that both *q/3* and *r/3* have the induced modes *$langle* **c**, **f**, **f***$rangle* , and that both *q* and *r* are functional relative to this mode. The calls to *q/3* and *r/3* in the clause above are therefore functional, and the clause can therefore be transformed to the following in a straightforward way:

$p(X, Y) :- savecp(U), q(X, Z, \_), cutto(U), savecp(V), r(X, Y, \_), cutto(V), s(Y, Z).$

However, since the two functional calls were contiguous in the original clause, the transitivity of the *fail_back_to* relation can be used to obtain the following clause, which is more efficient in both space and time:

$p(X, Y) :- savecp(U), q(X, Z, \_), cutto(U), r(X, Y, \_), cutto(U), s(Y, Z).$

•

5.3. Avoiding Cuts in Functional Predicates

The obvious way to improve functional predicates and literals is to cut away useless choice points, as illustrated in the preceding examples. However, this still involves the creation of choice points, which is not inexpensive. Under certain circumstances, more efficient code can be generated for functional predicates if cuts are not generated. This section considers two such situations.

5.3.1. Clause Indexing

The creation of a choice point for a call to a functional predicate can often be avoided by proper clause indexing. In such cases, the addition of cuts to the predicate can actually result in redundant work. A better strategy is to have the compiler either build more sophisticated indices, or transform the program, based on mode information and analysis of mutual exclusion of clauses, so that Prolog's usual indexing scheme will suffice to avoid the creation of choice points (typically, Prolog systems, e.g. [25, 27], index on the principal functor of the first argument of each clause). The additional effort involved at compile time can very well be offset by the space and time savings accruing from not having to put down a choice point at each call. In such cases, where no choice points are being created, cuts in the bodies of clauses are no-ops if they serve only to cut the clause selection alternatives, and constitute unproductive overhead.

Example 9: Consider the predicate

process([ ], [ ]).
process([','(*X*, *Y*) | *Rest*], [*T* | *TRest*]) :– *process_comma*(*X*, *Y*, *T*), *process*(*Rest*, *TRest*).
process([';'(*X*, *Y*) | *Rest*], [*T* | *TRest*]) :– *process_semicolon*(*X*, *Y*, *T*), *process*(*Rest*, *TRest*).
process([*not*(*X*) | *Rest*], [*T* | *TRest*]) :– *process_not*(*X*, *T*), *process*(*Rest*, *TRest*).

The clauses for the predicate *process* are mutually exclusive if it is always called with its first argument ground. However, in order to avoid creating a choice point for it, it is necessary to look beyond the principal functor of its first argument, which is the same for three of its four clauses. This can be done either by building a more sophisticated index for this predicate, or by transforming it to the following at compile time:

process([ ],[ ]).
process([*H* | *L*], [*TH* | *TL*]) :– *process1*(*H*, *TH*), *process*(*L*, *TL*).

process1(’,’(*X*, *Y*), *T*) :– *process_comma*(*X*, *Y*, *T*).
process1(’;’(*X*, *Y*), *T*) :– *process_semicolon*(*X*, *Y*, *T*).
process1(*not*(*X*), *T*) :– *process_not*(*X*, *T*).

Here, *process1* is a new predicate not appearing elsewhere in the program. It is evident that in the transformed program, the usual indexing mechanism of Prolog suffices to avoid the creation of choice points for calls to *process/2* or *process1/2*. ●

5.3.2. Transformations for Mutually Exclusive Clauses

Mutual exclusion between clauses is often based upon complementary tests. Such clauses can often be transformed in a manner that avoids the creation of choice points. In such cases, it is possible to avoid cuts in the clauses without sacrificing efficiency.

Mutual exclusion due to complementary tests is addressed in Proposition 3.8, which considers clauses of the form

$$p(\bar{X}) :- G_{11}, q(\bar{T}), G_{12}.$$
$$p(\bar{X}) :- G_{21}, not(q(\bar{U})), G_{22}.$$

where the G_{ij} consist of zero of more literals. This proposition is applicable only if the heads of the clauses being considered are identical (modulo variable renaming). In practice, it is rarely the case that clauses have identical heads, but mode and functionality information can often be used to effect transformations that permit the application of Proposition 3.8. The essence of the transformation is to use mode information to move certain unifications from the heads of clauses into their bodies. Consider a clause

$$p(t_1, \ldots, t_{m-1}, t_m, \ldots, t_n) :- Body.$$

and assume, without loss of generality, that the mode of the predicate indicates that all arguments between and including the $m^{\text{th.}}$ and $n^{\text{th.}}$ are guaranteed to be free variables in any call to p. The transformation replaces these arguments with new variables $X_m, \ldots, X_n$ that do not appear elsewhere in the clause, and introduces explicit unifications ‘$X_i = t_i$’, $m \le i \le n$, immediately before the body of the clause.† The transformed clause is therefore

$$p(t_1, \ldots, t_{m-1}, X_m, \ldots, X_n) :- X_m = t_m, \ldots, X_n = t_n, Body.$$

† Strictly speaking, it is necessary to do this only for those arguments where the term t_i is not a variable.

While this transformation may, in many cases, suffice to ensure that the two clauses have identical heads (modulo variable renaming), the clauses may still not be in a form where Proposition 3.8 can be applied. In such cases, it is often useful to try to move the complementary goals $q(\overline{T})$ and $not(q(\overline{U}))$ towards the head of the clause by reordering literals. In general, changing the order of literals in a clause can affect its operational behavior, by changing the order in which solutions are found or affecting its termination characteristics. However, simple special cases can be considered where these problems do not arise: for example, two literals

$$. \ . \ ., r(\overline{V}), s(\overline{W}), . \ . \ .$$

in the body of a clause can be transposed without affecting the behavior of the clause, provided that

(*i*) $r(\overline{V})$ and $s(\overline{W})$ are guaranteed to be independent, so that the order in which variables are bound does not pose problems;

(*ii*) both the literals $r(\overline{V})$ and $s(\overline{W})$ are functional, so that order of solutions is not an issue;

(*iii*) the termination of $r(\overline{V})$ and $s(\overline{W})$ is guaranteed, e.g. when they consist only of simple tests or do not involve any recursion; and

(*iv*) $r(\overline{V})$ and $s(\overline{W})$ are free of side effects.

An example application of this transformation is given by the following:

Example 10: Consider the following predicate, used to partition lists in the quicksort algorithm:

> *part*(_, [], [], []).
> *part*(*M*, [*E* | *L*], [*E* | *U1*], *U2*) :– *E* =< *M*, *part*(*M*, *L*, *U1*, *U2*).
> *part*(*M*, [*E* | *L*], *U1*, [*E* | *U2*]) :– *E* > *M*, *part*(*M*, *L*, *U1*, *U2*).

While the second and third clause intuitively seem to be mutually exclusive because of the complementary tests, they cannot be so inferred from Proposition 3.8. If it is assumed, however, that the first two arguments to the predicate are input arguments while the third and fourth are output arguments, i.e. that it has the mode ⟨**c**, **c**, **f**, **f**⟩, then it can be transformed, as described above, to

> *part*(_, [], [], []).
> *part*(*M*, [*E* | *L*], *X*, *U2*) :– *X* = [*E* | *U1*], *E* =< *M*, *part*(*M*, *L*, *U1*, *U2*).
> *part*(*M*, [*E* | *L*], *U1*, *Y*) :– *Y* = [*E* | *U2*], *E* > *M*, *part*(*M*, *L*, *U1*, *U2*).

At this point, the literals *X* = [*E*|*U1*] and *E* =< *M* in the second clause satisfy the four conditions listed above for literal reordering (independence follows from the fact that both *E* and *M* are ground given the mode under consideration). A similar comment applies to the literals *Y* = [*E*|*U2*] and *E* > *M* in the third clause. Literal reordering then yields

> *part*(_, [], [], []).
> *part*(*M*, [*E* | *L*], *X*, *U2*) :– *E* =< *M*, *X* = [*E* | *U1*], *part*(*M*, *L*, *U1*, *U2*).
> *part*(*M*, [*E* | *L*], *U1*, *Y*) :– *E* > *M*, *Y* = [*E* | *U2*], *part*(*M*, *L*, *U1*, *U2*).

At this point, the third and fourth clauses can be inferred to be mutually exclusive based on Proposition

3.8. The clauses can in fact be fused, with the recognition of complementary tests yielding

$part(\_, [], [], [])$.
$part(M, [E \mid L], X, Y)$:–
$E =\!< M \rightarrow (X = [E \mid U1], part(M, L, U1, Y))\ ;\ (Y = [E \mid U2], part(M, L, X, U2))$.

From Proposition 3.7, these clauses can be seen to be mutually exclusive relative to the mode $\langle \mathbf{c}, \mathbf{c}, \mathbf{f}, \mathbf{f} \rangle$. In an implementation, it is possible to execute this predicate without creating a choice point, since a type test on the second argument suffices to discrimate between the two clauses, while an arithmetic test suffices to determine which alternative in the body of the second clause should be taken. A variant of this transformation is used in the SB-Prolog compiler [7]: the optimization resulting from this leads to a speed increase of over 30% for this example. ●

5.4. Functional Optimizations in Parallel Execution Strategies

Functionality and mutual exclusion can also be exploited in parallel evaluation strategies. Parallel execution of logic programs can be broadly divided into two classes: OR-parallel execution, where alternative search paths are explored concurrently, and AND-parallel execution, where subtasks of a computation are solved concurrently. Different flavors of OR-parallelism have been proposed by a number of researchers [2, 10, 21, 28, 29]. Other proposals incorporate both AND- and OR-parallelism [5, 13, 30, 31].

OR-parallel execution may be controlled via *commit* operators, which are symmetric generalizations of the *cut*. There are two kinds of commit: *strict commit*, which prevents any solutions or side effects in other execution branches from becoming visible; and *cavalier commit*, which makes no guarantees about side effects [29]. During OR-parallel execution, if two clauses are known to be mutually exclusive, then *commit*s may be introduced by the compiler at appropriate places. This allows processes for one clause to be killed off, and the machine resources used by it reclaimed, as soon as the other one succeeds. In situations where the absence of side effects can be guaranteed, cavalier commits can be inserted instead of strict commits, reducing the need for process suspension and synchronization. Indeed, in cases involving simple mutual exclusion, e.g. based on simple arithmetic tests, the compiler can move fork points lower in the execution tree, delaying the creation of OR-parallel processes until the outcome of the test becomes known, in a manner analogous to the transformation illustrated in Example 10. This can, in many cases, avoid the cost of creating useless processes and thereby reduce the overall cost of the computation. The early elimination of execution branches in this manner also allows early ‘‘promotion’’ of variable bindings from *conditional*, where bindings are maintained in binding lists local to processes, to *unconditional*, where the binding is actually written out to the value cell for that variable. Since operations involved in unification are typically faster for unconditional bindings than for conditional ones [29], this also reduces the cost of subsequent unifications involving such promoted variables. It should be noted here that since the cut is a control mechanism designed for sequential execution strategies, it forces sequentialization of execution under parallel evaluation strategies in order to give the expected behaviour. It may be therefore be preferable not to have cuts in the original program, but instead let the compiler infer mutual exclusion and functionality and generate code appropriate to the execution environment.

Functionality information is also useful in AND-parallel systems [5, 6, 11]. For example, in Conery's AND/OR process model [5], OR-processes for literals that are known to be functional can be killed, and their resources reclaimed, as soon as they have delivered a solution. This, in turn, can improve backtracking behavior within AND-processes, since the number of goals that have to be considered for backtracking can be reduced. Functionality information can also be used to influence scheduling decisions in AND-parallel systems: consider a situation where there are two independent goals G_1 and G_2 that have to both be solved, where G_1 is functional and G_2 is not. If there is only one processor available at this point, then the scheduler can schedule eiher G_1 or G_2 on it. In such a case, unless G_1 is certain to succeed, it may be better to schedule G_1 for execution in preference to G_2. This is because the fact that G_1 is functional suggests that scheduling it early can reduce backtracking costs and enable earlier reclamation of resources allocated to it.

5.5. Functional Optimizations in Other Execution Strategies

Information about functionality and functional dependencies can also be used in sequential execution strategies that depart from Prolog's left-to-right control regime for literals within a clause. For example, both CHAT-80 [26] and MU-Prolog [19] use heuristic estimates of the number of solutions that can be returned by a database predicate to determine the order of execution of literals within a goal. In both systems, *priority declarations* are supplied by the user for database predicates, to provide information about the number of clauses and the probability of match for each argument position. An important assumption that is made is that the probabilities of arguments in a clause matching the corresponding arguments in a call are independent for different argument positions. However, the specification of functional dependencies may require reasoning about more than one argument position at one time. For example, consider a predicate p defined as

$p(0, 0, 0)$.
$p(0, 1, 1)$.
$p(1, 0, 1)$.
$p(1, 1, 0)$.

In this predicate, the first two arguments together functionally determine the third: there is no functional dependency between just the first and third arguments, or the second and third arguments. Thus, priority declarations cannot express functional dependencies in general. Since functional dependencies specify semantic properties of the database, information about functionality and functional dependencies is likely to be more accurate than that obtained from priority declarations, and can be used to obtain better estimates for the number of solutions for a literal, improving the performance of the query optimizer.

Since the notion of functionality is applicable even for predicates that are not necessarily database predicates, it can also be taken into consideration when planning the execution order for predicates that do not involve database relations. As an example, Naish considers computation rules that select *locally deterministic* calls, where a locally deterministic call is defined as one that ''has at most one matching clause for any (non-delaying) call to it'' [19]. Clearly, the notion of local determinism can be generalized to that of functionality. Alternatively, the definition of local determinism can be generalized to incorporate the notion of mutual exclusion of clauses, as illustrated in Example 10.

5.6. Functionality and Cut Insertion : Some Caveats

Section 5.2 discussed the bracketing of deterministic calls with *savecp/cutto* pairs to avoid useless backtracking. There are situations, however, where such transformations can alter the semantics of the program. The example below illustrates this:

Example 11: Consider a predicate to count the number of occurrences of an element in a list:

$$numocc(Elt, L, N) :- bagof(Elt, member(Elt, L), EltList), length(EltList, N).$$

where *member*/2 and *length*/2 are defined in the usual way. If *numocc*/3 is always called with the first two arguments ground, then the call to *member*/2 is functional. However, bracketing this call to *member*/2 with a *savecp/cutto* pair would give incorrect answers. ●

The problem arises because in this case the number of successes is what is important, not just the answer. One could argue that *numocc*/3 is better written as a recursive predicate free of non-logical constructs such as *bagof*: the point of the example is to illustrate the fact that cuts should not be inserted blithely without taking the context into account. Other such examples can be constructed, involving side effects such as *read* or *write* operations, where altering the number of successes can affect the semantics of the program. For this reason, caution should be exercised in inserting cuts. For example, cuts should be inserted at a point only if it can be guaranteed that the search tree below that point is free of side effects.

6. Implementation

A prototype functionality inference system based on the ideas described here has been implemented for the SB-Prolog system. The functionality inference system, which is written in Prolog, uses a simple mode inference system to infer predicate modes [8]. The system was tested on some simple programs (quicksort, four-queens, a simple rewriting theorem prover) as well as significant modules from the SB-Prolog compiler (the scanner and parser, preprocessor, peephole optimizer and assembler). The results, given in Table 1, indicate that functionality analysis takes about 2% to 4% of the total compilation time, and that about 65% to 80% of the predicates can typically be inferred to be functional. A closer examination indicates that where the analysis is conservative, it is so principally because of a conservative mode inference system, suggesting that the precision of the functionality inference system could be improved even further given a more sophisticated mode inference system, or via user-declared modes. Our experiments indicate that functionality inference can be a practical and useful tool in the analysis and optimization of logic programs.

7. Summary

The paper considers the question of inferring the functionality of predicates in logic programs. The notion of functionality subsumes that of determinacy. Not being an inherently operational notion, it tends

Program	*No. of predicates*	*No. inferred functional*
quicksort	4	4
theorem prover	9	6
fourqueens	11	8
peephole	13	10
assembler	33	21
preprocessor	38	14
scanner/parser	42	31
func-inf	47	38

Table 1

to rely on features such as the ‘‘cut’’ to a much lesser extent. This encourages a better style of programming and extends gracefully to parallel evaluation strategies. Sufficient conditions for functionality are given, and an algorithm described for the automatic inference of functionality of predicates. Some program optimizations based on information about functionality and mutual exclusion of clauses are described, both for sequential and parallel execution strategies.

Acknowledgements

Yves Villiers pointed out a flaw in an earlier version of Proposition 3.8. Peter J. Downey pointed out the relationship between functionality and the Church-Rosser property of goal rewriting. Comments by the referees helped improve the paper considerably.

References

1. M. Bruynooghe and L. M. Pereira, Deduction revision by intelligent backtracking, in *Implementations of Prolog*, J. A. Campbell (ed.), Ellis Horwood Ltd., Chichester, 1984, pp. 194-215.

2. R. Butler, E. L. Lusk, R. Olson and R. A. Overbeek, ANLWAM: A Parallel Implementation of the Warren Abstract Machine, Internal Report, Argonne National Laboratory, USA, 1986.

3. J. Chang and A. M. Despain, Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis, in *Proc. 1985 Symposium on Logic Programming*, Boston, July 1985, pp. 10-21.

4. K. L. Clark, Negation as Failure, in *Logic and Data Bases*, H. Gallaire and J. Minker (ed.), Plenum Press, New York, 1978.

5. J. S. Conery, *Parallel Execution of Logic Programs*, Kluwer Academic Publishers, 1987.

6. D. DeGroot, Restricted AND-Parallelism, in *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1984.

7. S. K. Debray, The SB-Prolog System, Version 2.3.2: A User Manual, Tech. Rep. 87-15, Department of Computer Science, University of Arizona, Tucson, AZ, Dec. 1987. (Revised March 1988).

8. S. K. Debray and D. S. Warren, Automatic Mode Inference for Logic Programs, *J. Logic Programming 5*, 3 (Sep. 1988), pp. 207-229.

9. S. K. Debray and P. Mishra, Denotational and Operational Semantics for Prolog, *J. Logic Programming 5*, 1 (Mar. 1988), pp. 61-91.

10. B. Hausman, A. Ciepielewski and S. Haridi, Or-parallel Prolog made efficient on shared memory multiprocessors, in *Proc. 1987 IEEE Symposium on Logic Programming*, San Francisco, CA, Aug. 1987, pp. 69-79.

11. M. V. Hermenegildo, An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs, in *Proc. 3rd. International Conference on Logic Programming*, London, July 1986, pp. 25-39. Springer-Verlag LNCS vol. 225.

12. N. D. Jones and A. Mycroft, Stepwise Development of Operational and Denotational Semantics for PROLOG, in *Proc. 1984 Int. Symposium on Logic Programming*, IEEE Computer Society, Atlantic City, New Jersey, Feb. 1984, 289-298.

13. L. V. Kale, The REDUCE-OR Process Model for Parallel Evaluation of Logic Programs, in *Proc. Fourth International Conference on Logic Programming*, Melbourne, May 1987, pp. 616-632.

14. J. W. Lloyd, *Foundations of Logic Programming*, Springer Verlag, 1984.

15. D. Maier, *The Theory of Relational Databases*, Computer Science Press, 1983.

16. C. S. Mellish, Some Global Optimizations for a Prolog Compiler, *J. Logic Programming 2*, 1 (Apr. 1985), 43-66.

17. A. O. Mendelzon, Functional Dependencies in Logic Programs, in *Proc. 11th. ACM Int. Conf. on Very Large Data Bases*, Stockholm, Sweden, Aug. 1985.

18. L. Naish, Negation and Quantifiers in NU-Prolog, in *Proc. Third International Conference on Logic Programming*, London, July 1986, pp. 624-634. Springer-Verlag LNCS vol. 225.

19. L. Naish, *Negation and Control in Prolog*, Springer-Verlag, 1986. LNCS vol. 238.

20. R. A. O'Keefe, On the Treatment of Cuts in Prolog Source-Level Tools, in *Proc. 1985 Symposium on Logic Programming*, Boston, July 1985, 73-77.

21. R. A. Overbeek, J. Gabriel, T. Lindholm and E. L. Lusk, Prolog on Multiprocessors, Internal Report, Argonne National Laboratory, USA, 1985.

22. J. A. Robinson, Logic Programming – Past, Present and Future, *New Generation Computing 1*, 2 (1983), pp. 107-124, Springer-Verlag.

23. H. Sawamura and T. Takeshima, Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and Their Applications to Prolog Optimization, in *Proc. 1985 Symposium on Logic Programming*, Boston, July 1985, 200-207.

24. M. H. van Emden and R. A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *J. ACM 23*, 4 (Oct. 1976), pp. 733-742.

25. D. H. D. Warren, Implementing Prolog – Compiling Predicate Logic Programs, Research Reports 39 and 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

26. D. H. D. Warren, Efficient Processing of Interactive Relational Database Queries Expressed in Logic, in *Proc. 7th. Conf. on Very Large Data Bases*, Cannes, 1981, 272-281.

27. D. H. D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, CA, Oct. 1983.

28. D. S. Warren, M. Ahamad, S. K. Debray and L. V. Kale, Executing Distributed Prolog Programs on a Broadcast Network, *Proc. 1984 Int. Symposium on Logic Programming*, Atlantic City, New Jersey, Feb. 1984.

29. D. H. D. Warren, The SRI Model for Or-Parallel Execution of Prolog – Abstract Design and Implementation Issues, in *Proc. 1987 IEEE Symposium on Logic Programming*, San Francisco, CA, Aug. 1987, pp. 92-102.

30. H. Westphal, P. Robert, J. Chassin and J. Syre, The PEPSys Model: Combining Backtracking, AND- and OR-parallelism, in *Proc. 1987 IEEE Symposium on Logic Programming*, San Francisco, CA, Aug. 1987, pp. 436-448.

31. M. J. Wise, Epilog: Reinterpreting and Extending Prolog for a Multiprocessor Environment, in *Implementations of Prolog*, J. A. Campbell (ed.), Ellis Horwood Ltd., Chichester, 1984, pp. 341-351.

```
begin
  RELPREDS := nil;                                   /* Stage I : initialization of flags */
  CG_EDGES := ∅;
  for each literal, clause and predicate A in the program do A.fstat := ⊥;
  for each predicate p in the program do
      if p's clauses cannot be inferred to be pairwise simple mutually exclusive then begin
          p.fstat := false; p.visited := true;
          RELPREDS := push(p, RELPREDS);
      end
      else begin
          for each clause C of p do begin
              for each literal L ≡ q( ... ) in the body of C do
                  if L can be inferred functional from Propositions 3.1, 3.2 or 3.3 then L.fstat := true
                  else if L is not in a functional context then CG_EDGES := CG_EDGES ∪ $langle p, q$rangle ;
              if C can be inferred to be functional from Propositions 3.4 or 3.5 then C.fstat := true;
          end;
          if C.fstat = true for each clause C of p then begin
              p.fstat := true; p.visited := true;
          end
          else p.visited := false;
      end;      /* if */
  while RELPREDS ≠ nil do begin                      /* Stage II : iterative propagation */
      p := head(RELPREDS);
      if ($exists q)[$langle q, p$rangle ∈ CG_EDGES and p.fstat = false and ¬q.visited] then begin
           q.fstat := false;
           q.visited := true;
           RELPREDS := push(q, RELPREDS);
      end
      else RELPREDS := pop(RELPREDS);
  end;      /* while */
  for each predicate p in the program do if p.fstat = ⊥ then p.fstat = true;      /* Stage III : cleanup */
  for each clause C in the program do begin
      for each literal L ≡ q( ... ) in the body of C do if L.fstat = ⊥ then L.fstat := q.fstat;
      if C.fstat = ⊥ then
          C.fstat := ($exists a literal L in the body of C)[¬L.fstat and L is not in a functional context];
  end
end.
```

Figure 1: Algorithm for Functionality Inference

move 3i; Root: box ht 0.3i invis "α: p(. . .) "; B0: box wid 1.5i with .ne at Root.s invis B1: box wid 1.5i with .nw at Root.s invis Lev1: box ht 0.3i wid 2.5i with .n at B0.se invis " β: $L_1, \ldots, L_l, \ldots, L_m$ "; B2: box wid 1.5i ht 1.5i with .ne at Lev1.s invis B3: box wid 1.5i ht 1.5i with .nw at Lev1.s invis Alpha: box ht 0.2i with .ne at B2.se invis " • χ"; B4: box wid 1i ht 1i with .ne at Alpha.s invis B5: box wid 1i ht 1i with .nw at Alpha.s invis # box wid 0.1i ht 0.1i with .n at B4.sw box wid 0.1i ht 0.1i with .n at B5.se # " (*depth* = 1)" at Lev1.e ljust " (*depth* = *k*)" at B3.se ljust # line from Root.s to Lev1.n line from B0.ne to 2/3<B0.ne, B0.sw> line dashed from 2/3<B0.ne, B0.sw> to B0.sw line from B1.nw to 2/3<B1.nw, B1.se> line dashed from 2/3<B1.nw, B1.se> to B1.se line from B2.ne to 2/3<B2.ne, B2.sw> line dashed from 2/3<B2.ne, B2.sw> to B2.sw line from B3.nw to 2/3<B3.nw, B3.se> line dashed from 2/3<B3.nw, B3.se> to B3.se line from B4.ne to 2/3<B4.ne, B4.sw> line dashed from 2/3<B4.ne, B4.sw> to 9/10<B4.ne, B4.sw> line from B5.nw to 2/3<B5.nw, B5.se> line dashed from 2/3<B5.nw, B5.se> to 9/10<B5.nw, B5.se> spline from Lev1.s down 0.5i then left 0.1i then down 0.5i left 0.1i then left 0.1i then to Alpha.n

Figure 2

---

move 2i; B: box wid 2i ht 2i invis; "•" at B.n; "•" at B.w; "•" at B.e "•" at B.s C: box wid 0.7i with .w at B.e invis box ht 0.3i wid 0.2i with .n at B.n invis "*p*" box ht 0.3i wid 0.2i with .n at B.s invis "*s*" "• *even*" at C.e ljust box wid 0.2i ht 0.2i with .ne at B.w invis "*q*" box wid 0.2i ht 0.2i with .nw at B.e invis "*r*" arrow from 1/20<B.n, B.w> to 19/20<B.n, B.w> arrow from 1/20<B.n, B.e> to 19/20<B.n, B.e> arrow from 1/20<B.s, B.w> to 19/20<B.s, B.w> arrow from 1/20<B.s, B.e> to 19/20<B.s, B.e> arrow from 1/10<C.w, C.e> to 19/20<C.w, C.e> spline -> from B.n then left 0.2i up 0.2i then right 0.2i up 0.2i then right 0.2i down 0.2i then to B.n

Figure 3: Call Graph for Example 6