REDUNDANCY DETECTION IN LOGIC PROGRAMS IS UNDECIDABLE

Zheng Zhou

Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
zhou@umvlsi.ecs.umass.edu

Benjamin W. Wah

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, IL 61801
wah%aquinas@uxc.cso.uiuc.edu

ABSTRACT

Redundancy detection is one of the major techniques in reducing the complexity of a logic program. For recursive and non-recursive function-free inference rules, redundancy detection has been proved to be decidable and NP-complete. However, redundancies in inference rules with functions, which are more common in logic programs, have not been explored completely. This paper formally defines and represents these redundancies based on domain theory, then proves that redundancy detection in recursive or non-recursive inference rules with functions is undecidable. The proof is done by mapping the problem to Hilbert's Tenth Problem. Our work, therefore, completes the exploration of complexities of redundancy detection in logic programs.

INDEX TERMS: Decidability, Hilbert's Tenth Problem, logic program, redundancy detection.

I. INTRODUCTION

In recent years, logic languages like Prolog have been widely used in artificial intelligence applications, such as natural languages understanding, abstract problems solving, and database retrievals. Researchers have found inefficient execution of logic programs due to unification and automatic backtracking. These handicaps are caused by the separation of logic and control in logic programs: the knowledge about problems and assumptions are explicitly described and stated as logical axioms, and the control strategy is simple and inefficient. Consequently, great emphasis has been placed on providing more efficient control features for logic programming systems in order to improve its performance [9]. Two general methods can be applied.

(1) *Reorder clauses and predicates* in a logic program to find the successes or failures as early as possible, hence reducing the cost of evaluation.
(2) *Prune redundant executions* of identical predicates.

We have studied the first method, which can be considered as the restructuring of an AND/OR graph so that successes or failures can be asserted as quickly as possible [6, 7]. Consider the search of a conjunction of subgoals. Owing to the simple control strategy, which traverses a search tree from left to right and depth first, it may traverse a large subgraph, only to fail on a later goal. By reordering clauses and subgoals using statistics collected in the logic program so that inexpensive goals with high failure probabilities are searched first, the average total search cost can be reduced. For logic programs that search large databases, reordering predicates and clauses results in significant improvement in performance. However, for programs that are algorithmic in nature, dependencies and side effects within the program significantly restrict the amount of reordering that can be performed.

The second method for improving performance can also be explained in terms of searching an AND/OR graph. If a subgoal appears more than once as nodes in an AND/OR graph, then all occurrences

Research supported by National Aeronautics and Space Administration Contract NCC 2-481.

IEEE Computer Software and Applications Conference, 1990

except the first are redundant and should be detected and pruned. This implies that we need an efficient method for identifying redundancies and recalling previous results obtained for the redundant subgoals. A simple method for dynamically storing previously generated results is the extension table [5,11]. Note that an extension table stores *instances* of previously generated predicates, but not rules for eliminating redundant predicates. This approach is similar to a combination of forward and backward chaining. A similar idea has been used in dominance tests of dynamic programming algorithms for removing redundant computations [1].

There are two solutions to redundancy detection and elimination. First, redundancies can be detected and eliminated at *run time*. We have implemented a run-time detection system, which detects all potentially redundant subgoals by inserting a specially defined built-in predicate into users' programs so that the inference engine can automatically check whether or not a subgoal has a solution obtained before and stored in the extension table. Whenever a redundant predicate is found, it is looked up in the extension table to determine if the predicate has been evaluated before with the same bindings. Experimental results with a recursive implementation of the generation of Fibonacci numbers indicate that exponential speedup in execution time and exponential reduction in memory space can be obtained. However, results with other problems, such as the Eight-Queens and the Tower-of-Hanoi problems, are not satisfactory. The reason lies in the large overhead in searching the extension table, and this overhead outweighs the benefit. More efficient implementations of the extension table, such as using hashing techniques, can improve the performance. However, performance is still limited by the size of the extension tables to be searched.

Another solution for redundancy detection and elimination is to analyze the program at *compile time* and detect and eliminate the redundancies in the logic program, based on static information such as syntactical structures and universal relations. Obviously, if redundant execution can be detected at compile-time, then expensive one-by-one matching at run-time can be avoided.

Whatever detection methods we are using, it is necessary to first decide whether all potential redundant executions can be detected. Aho, Sagiv and Ullmans' pioneering work [2] on equivalence detection among relational expressions tells us that redundancies in function-free nonrecursive inference rules (with no function symbols in arguments) are decidable and that its detection is NP-complete. Naughton proved that redundancies in function-free recursive inference rules are undecidable [10]. These results can be summarized in Table 1.

Table 1. Summary of Current Results

Decidability of Redundancy Detection		
RECURSION	FUNCTION	FUNCTION-FREE
Recursive	?	Undecidable
Nonrecursive	?	Decidable (NP-Complete)

In this paper we explore the decidability of redundant execution in a logic program composed of recursive and nonrecursive inference rules with functions. We assume that the reader is familiar with the basic concepts of logic programming languages [9]. In Section 2, redundancy is defined and classified. Section 3 introduces the concepts of domains and information systems and discusses the properties of interval representations. In Section 4 we prove that redundancy detection for inference rules with functions, at least for arithmetic functions, is undecidable. Conclusions are drawn in Section 5.

II. REDUNDANT EXECUTION AND MONOTONIC OPERATIONS

A query (or a subgoal) in a logic program may have more than one successful binding found by backtracking. These bindings make up a solution set for this subgoal. The same subgoal occurring in two different places in a logic program does not imply that the two solutions sets are identical because they may have different environments defined by given or known bindings to variables stated in the subgoal. In addition, the order of subgoals appearing in a program can also affect their solution sets. This dynamic behavior renders the detection of redundancies very difficult. To understand the problem, we first formally define redundant execution in a logic program.

Definition 2.1. In evaluating a logic program, if a predicate G appears more than once, say G^1 and G^2, and every solution found for G^2 is always a solution of G^1, *i.e.*, solution set $S(G^2)$ is a subset of $S(G^1)$, then G^2 is *completely redundant* with G^1; if only some of the solutions found for G^2 are included in the solution set of G^1, then G^2 is *partially redundant* with G^1.

Assuming that G^1 is evaluated before G^2, then partial redundancy can be further classified into the following two cases.

(1) $S(G^1)$ is a proper subset of $S(G^2)$, *i.e.*, $S(G^1) \subset S(G^2)$.

(2) $S(G^1)$ and $S(G^2)$ each have some solutions that are not shared by the other, *i.e.*, $S(G^1) \cap S(G^2) \neq \varnothing$ and $S(G^1) - S(G^2) \neq \varnothing$ and $S(G^2) - S(G^1) \neq \varnothing$.

All the relationships between $S(G^1)$ and $S(G^2)$, namely, *complete redundancy* and *partial redundancy*, can be represented in the Venn diagrams shown in Figure 1.

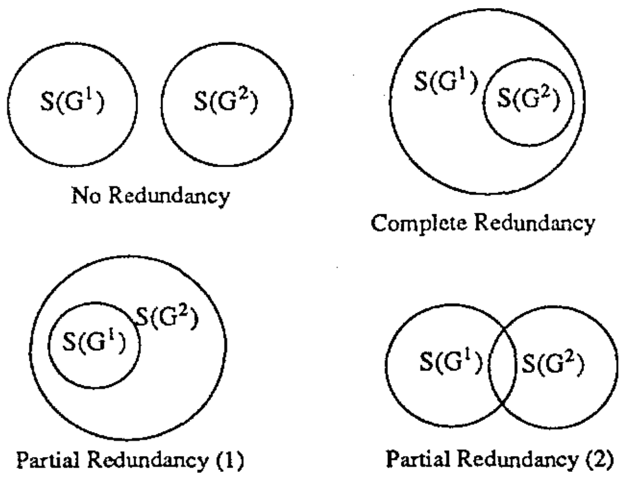

Figure 1. Redundancy representations in solution sets.

The definition above explicitly indicates that the decidability of redundancy detection is equivalent to the decidability of the relations among solution sets in a logic program. Generally, S, the solution set of a given query with n arguments, is composed of an n-tuple, $(a_1, a_2, ..., a_n)$, where a_i is a ground term bounded to the i'th argument and is a member of domain D_i. Hence, S is a subset of $D_1 \times D_2 \times \cdots \times D_n$, and the relation between two solution sets, S^1 and S^2, can be mapped to the relation between two corresponding argument domains, D^1_i and D^2_i. For example, $S^2 \subseteq S^1$ must have $D^2_i \subseteq D^1_i$. This property implies that our redundancy detection will be under the monotone assumption; that is, the output ranges of all functions represented in the form of queries should be monotonic in their input domains. Formally, we have the following definition (based on Ullman's definition).

Definition 2.2 [13]. Let $f(D_1, ..., D_m)$ be a function whose arguments and results are each relations, and $S^1 = D^1_1 \times \cdots \times D^1_m$ and $S^2 = D^2_1 \times \cdots \times D^2_m$ be two assignments of relations to the relation variables of f. If $S^1 \subseteq S^2$, that is, each relation D^1_i is a subset of the corresponding relation D^2_i, then f is *monotone* for S^1 and S^2, and $f(S^1) \subseteq f(S^2)$.

As we have known, monotone functions are quite common, since almost all basic relational operators like union, selection and projection are monotone, so are all basic arithmetic operations like addition, subtraction, and multiplication.

Our definition of redundancy based on the inclusion of solution sets is applicable to the more general cases. It also suggests an ideal redundancy pruning system to be an integration of compile-time analysis and run-time checking. Compile-time analysis can detect some decidable redundancies by detecting the inclusion of solution sets, while run-time checking can prune individual redundant execution using the extension table. Such an integration provides a potentially effective methodology for efficient execution of logic programs.

III. DOMAINS AND THEIR REPRESENTATIONS

In the previous section, we have defined a domain to be the set of all bindings such that the defined predicate is true. Obviously, a domain is closely related to the computation in a given program. A formal definition should be derived from this inherent connection. Scott's information system is a suitable representation and is used here to define a domain based on its "possible elements" [12].

An information system consists of three components: a set of tokens, a consistent predicate, and an entailment relation. Tokens are units of information which may be valid for a computation. In a logic program, tokens are terms (ground or non-ground) that appear in the corresponding arguments of predicates. The result of a computation is to find a Herbrand Model for the given logic program P [9]; that is, it needs to find not only its Herbrand Universe U_p, but also the true value subset of its Herbrand Base U_p. Thus, a solution should, at least, satisfy the syntactic requirements of the elements of U_p, which have been given in the form of ground terms and expressions composed of functions on the arguments. This is why we can use them as basic information units in our information system. For a set of tokens, all its elements should be kept consistent. This can be implemented by some particular predicates, but in our discussion, we assume that these consistencies are found by enumerating all ground terms and some given expressions. Thus, the entailment relation between a set of tokens and a token is obvious.

Larsen and Winskel's definition of information system [8] is selected as the formal definition in this paper. Their definition is different from Scott's original one in that it does not assume a distinguished element Δ in the token sets. However, all their theorems are isomorphic.

Definition 3.1 [8]. An *Information System* is defined as a structure (A,Con,=>), where A is a token set, Con is a set of finite subsets of A, *i.e.*, the set of the consistent sets, and => is a subset of Con × A, *i.e.*, the entailment relation. It satisfies the following conditions.

<1>. $(X \subseteq Y \in Con) \rightarrow (X \in Con)$
<2>. $(a \in A) \rightarrow (\{a\} \in Con)$
<3>. $(X \Rightarrow a) \rightarrow (X \cup \{a\} \in Con)$
<4>. $(X \in Con \;\&\; a \in X) \rightarrow (X \Rightarrow a)$
<5>. $(X, Y \in Con \;\&\; b \in Y.\; X \Rightarrow c) \rightarrow (X \Rightarrow c)$.

An information system determines a family of subsets of tokens, called its elements. Intuitively, an element of an information system is the set of tokens that can be asserted about a possible computation. This set of tokens can be viewed as the information content of the computation. The tokens should not contradict each other, and should be closed under entailment.

Definition 3.2 [8]. The *elements*, */A/*, of an information system (A,Con,=>) are those subsets *X* of A which are
<1>. *consistent:* $(X \subseteq X) \rightarrow (X \in Con)$
<2>. *closed:* $(X \subseteq X \;\&\; X \Rightarrow a) \rightarrow (a \in X)$.

Larsen and Winskel have proved that a closed family composed of all elements will form a consistently complete, algebraic complete partial order (CPO), *i.e.*, a domain. Their theorem is as follows.

Theorem 3.1 [8]. Let *A* = (A, Con, =>) be an information system. Its elements, */A/*, ordered by inclusion, form a *domain*, i.e. *a consistently complete, algebraic complete partial order.* Its finite elements are of the form: $X = \{a \in A | X \Rightarrow a\}$, where $X \in Con$, and the least element of */A/* is $\emptyset$.

As a direct application of an information system, we can define an information system for logic program P.

Theorem 3.2. For an argument of a given predicate in a logic program, let (a) the token set, A, be composed of all possible terms appearing in that argument, (b) the consistent sets, Con, be subset of tokens appearing in that position, and (c) an entailment relation, =>, be defined by enumerating all members of set Con×A. The resulting information system (A, Con, =>) satisfies all five properties given in Definition 3.1.
Proof. The theorem can be proved easily by showing that our defined (A, Con, =>) satisfies all properties of an information system. □

According to this theorem, it is easy to define by observation the domain of an argument of a logic program. This is shown in the following example.

Example 3.1. Given the following logic program,
<1>. Fib(1,1).
<2>. Fib(2,1).
<3>. Fib(N,A) :- Fib(N−1, A_1), Fib(N−2, A_2), A is $A_1 + A_2$.
the first argument of predicate Fib has an information system (A_1, Con_1, $\Rightarrow_1$), where $A_1 = \{1, 2, N, (N-1), (N-2)\}$; Con_1 contains all possible subset of A_1, since they are all consistent; $\Rightarrow_1$ is an entailment relation such as $\{1, 2, (N-1)\} \Rightarrow_1 1$. The domain generated from *A* is $(D_1, \subseteq)$, where D_1 is $\{\{1\}, \{2\}, \{(N-1)\}, \{(N-2)\}, \{1,2\}, \{1,(N-1)\}, \ldots, \{1,2,(N-1),(N-2)\}\}$, *i.e.*, all possible subsets of A.

From the example above, we find that the form of representation of a domain is not elegant in that it is ineffective and inconvenient for determining the inclusion between two domains, *i.e.*, determining the order between domains. Since an argument may be either a value in the numerical space, such as integers and reals, or a symbol string in the nonnumerical space, such as names of people and objects, these terms all have valid representations in logic programming languages. Note that the difference between numerical and nonnumerical space is based on the operations that can be defined on them.

For a domain in the numerical space (assume that it is real), a possible form is an interval. A simple transformation is stated below.

A set of numbers can form a chain-CPO, i.e. a partially ordered set U such that every directed subset (or equivalently, every chain) of U has a least upper bound and a least element. Thus a subset can be represented as an interval by using its largest element, x_u *and the least element,* x_b*:* $[x_b, x_u]$.

After this transformation, the original order of elements within the domain can be preserved. We can prove that this transformation is an approximable mapping between two information systems, (A, Con_A, =>) and (A', $Con_{A'}$, =>'), where A' is a set of intervals corresponding to the numerical sets in A. An approximable mapping satisfies the following properties.

Definition 3.3 [8]. Let $A = (A, Con_A, \Rightarrow_A)$ and $B = (B, Con_B, \Rightarrow_B)$ be information systems. An *approximable mapping* r: $A \rightarrow B$ is a relation $r \subseteq Con_A \times Con_B$ such that
<1>. $\emptyset\, r\, \emptyset$,
<2>. $(X\, r\, Y \;\&\; X\, r\, Y') \rightarrow X\, r\, (Y \cup Y')$ for all $X \in Con_A$ and $Y, Y' \in Con_B$,
<3>. $(X \Rightarrow_A X' \;\&\; X\, r\, Y \;\&\; Y \Rightarrow_B Y') \rightarrow X'\, r\, Y'$ for all $X, X' \in Con_A$ and $Y, Y' \in Con_B$.

The intuitive interpretation of an approximable mapping is that information X in *A* entails information Y in *B*. An approximable mapping among information systems will have a corresponding mapping among these domains. The following result has been given by Larsen and Winskel.

Theorem 3.3 [8]. Let r: $A \rightarrow B$ be an approximable mapping. Then $|r|: |A| \rightarrow |B|$ given by
$$|r|(x) = \cup\{Y | X \subseteq x, X\, r\, Y\}$$
is a continuous function between the domains |*A*| and |*B*| ordered by inclusion.

Based on this theorem, we can prove the following theorem

Theorem 3.4. For a given subset X of domain D in numerical space, we can construct a corresponding subset J of intervals which corresponds to a set in X. This subset of intervals preserves the original order, *i.e.*, the largest set in X corresponds to the largest interval in J.
Proof. First, we can prove that our defined transformation from sets to intervals implies that the mapping from a domain of a set of sets to a domain of a set of intervals satisfies the definition of |r| in Theorem 3.3. This is true since a given set x can be mapped to |r|(x), an interval which contains all points appearing between its smallest and largest elements. Thus, this mapping is a continuous function.

Second, since this mapping is order preserving, it keeps the original order in the mapped domain, *i.e.*, the largest sets in X correspond to the largest intervals in J. □

We have formally defined the concept of domains in a logic program, and have proved that an interval representation can replace the ineffective form using sets without losing any information on the order of sets in the original domain. We should note that $I_1 \leq I_2$ in the domain composed of intervals must have $X_1 \leq X_2$ in the domain composed of the original number sets, but $I_1 = I_2$ does not imply $X_1 = X_2$.

IV. REDUNDANCY DETECTION AND HILBERT'S TENTH PROBLEM

From the discussion above, a domain, *D*, in a real field generated by a set of arguments appearing in an argument of a predicate can be represented by an interval [a, b], where a and b are the minimum and maximum values in *D*. In general, the terms in an argument of a predicate are polynomials with some given real values, *i.e.*, bases. Thus, we should first find out the maximum and minimum values for each polynomial in this set. Consequently, the maximum and minimum values for the whole domain can be decided, and the order between two polynomials is defined based on their extreme values. Determining the extreme value in a polynomial is, therefore, a fundamental problem.

For a given polynomial defined on a real set, can we always determine its extreme value? We classify this problem into two cases based on whether the size of the set is finite or infinite. For a finite discrete set, we can always find out the extreme value of the given polynomial by a finite number of comparisons, so the problem is decidable. For an infinite continuous set in a closed interval, the polynomial also has extreme values, because a polynomial in a continuous function always has the extreme values in a closed interval. Thus, it is also a decidable problem. How about the extreme value in an infinite discrete set? This is a common case, but no direct answer has been given in the literature.

The relevant problem is Hilbert's Tenth Problem, which asks whether a given (multivariate) polynomial with integral coefficients has integral roots. The problem has been proved to be undecidable [4]. Therefore, if we can map our extrema-finding problem to Hilbert's Tenth Problem, then redundancy detection in inference rules with functions is also undecidable. To simplify our discussion, we consider a simple case in which a domain is generated by two polynomials based on a constant. If we cannot find an algorithm to decide its extreme values, then the general case must be obviously undecidable.

Suppose we are given an arbitrary polynomial $P(x_1, ..., x_n)$ with integral coefficients. Consider the basis element {1}, and the two polynomials $Q(x) = x+1$, and $R(x_1, ..., x_n) = P^2(x_1, ..., x_n)$. An easy argument shows

<1>. that all positive integers are in the closure,
<2>. that no negative integers are in the closure,
<3>. and that only integers can be in the closure.

Thus the minimum element in the closure is either 0 or 1. The decision can be transformed to deciding whether P has integral solutions (or roots), which is a new lemma to Hilbert's Tenth Problem [3].

Theorem 4.1. The minimum element in closure {1,Q,R} is 0 if and only if P has integral roots.
Proof. To prove the if part, suppose P is a polynomial of n variables, with integral roots $x_1, ..., x_n$. That is, $P(x_1, ..., x_n) = 0$, where x_i is a positive integer. Since x_i is in the closure of [1,Q,R], so is $R(x_1, ..., x_n) = P^2(x_1, ..., x_n) = 0 * 0 = 0$.

To prove the only-if part, let 0 be in the closure. Since 0 cannot be obtained from Q(x) for any x, it must be obtained from $R(x_1, ..., x_n)$ for some positive integers $x_1, ..., x_n$ in closure [1,Q,R]. Hence, for some $x_1, ..., x_n$, we have

$$0 = R(x_1, ..., x_n) = P^2(x_1, ..., x_n).$$

But then $P(x_1,...,x_n) = 0$. (All x_i are integers by a simple argument showing that all elements of a closure are integers.) □

Thus if we can decide which of {0,1} is the smallest element in closure {1,Q,R}, we can decide Hilbert's (undecidable) Tenth Problem. We can prove similarly about finding the maximum. Hence, there is no algorithm for finding the extreme values in the domain generated by finite polynomials given some bases. This is stated formally in the following theorem.

Theorem 4.2. Redundancy detection in inference rules with functions is undecidable.

V. CONCLUSIONS

In this paper we study the decidability of redundancy detection in logic programs, which are composed of inference rules with functions in their arguments. We formally define redundancies based on the solution sets in logic programs, and represent solution sets in well-defined domains. We prove that redundancy detection in inference rules with functions is undecidable. Our theoretical results complete the previously unknown properties of redundancy detection of logic programs. Except for non-recursive inference rules, redundancy detection is undecidable in general. Although our results reveal the inherent difficulty of efficient execution of logic programs through redundancy detection, special redundant cases can still be detected and pruned, as Naughton has found [10]. Tradeoffs between compile-time analysis of decidable cases and run-time checking of undecidable cases can still be made.

ACKNOWLEDGMENTS

We would like to thank Dr. L. Pitt, Dr. J. Benaloh and Dr. S. Cook for their help and suggestions in this paper.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] A. V. Aho, Y. Sagiv, and J. D. Ullman, "Equivalence Among Relational Expressions," *SIAM J. Computing*, vol. 8, no. 2, pp. 218-246, 1979.

[3] J. Benaloh and S. Cook, *Private Communcation*, 1989.

[4] M. Davis, "Hilbert's Tenth Problem is Unsolvable," *American Mathematical Monthly* , vol. 80, no. 3, pp. 233-269, 1973.

[5] S. W. Dietrich, "Extension Tables: Memo Relations in Logic Programming," *Proc. Symp. on Logic Programming*, pp. 264-272, IEEE, 1987.

[6] M. M. Gooley and B. W. Wah, "Efficient Reordering of Prolog Programs," *Proc. 4th Int'l Conf. on Data Engineering*, pp. 71-75, IEEE, Los Angeles, CA, Feb. 1988.

[7] M. M. Gooley and B. W. Wah, "Efficient Reordering of Prolog Programs," *Trans. on Knowledge and Data Engineering*, vol. 1, no. 4, pp. 470-482, IEEE, Dec. 1989.

[8] K. G. Larsen and G. Winskel, "Using Information Systems to Solve Recursive Domain Equations Effecitively," *Lecture Notes in Computer Science*, vol. 173, pp. 109-129, 1984.

[9] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1984.

[10] J. F. Naughton, "Optimizing Function-Free Recursive Inference Rules," Report No. STAN-CS-86-1114, Department of Computer Science, Stanford University, Stanford, CA, 1986.

[11] E. Sciore and D. S. Warren, "Integrating Data Bases," *Expert*, pp. 38-44, IEEE, 1988.

[12] D. Scott, "Domains for Denotational Semantics," *Lecture Notes in Computer Science*, vol. 140, pp. 577-613, 1982.

[13] J. D. Ullman, *Principals of Database Systems, 2nd Edition*, Computer Science Press, 1988.