INTELLIGENT CONTROL USING INTEGRITY CONSTRAINTS

Madhur Kohli Jack Minker

Department of Computer Science, University of Maryland, College Park, MD 20742

ABSTRACT

This paper describes how integrity constraints, whether user supplied or automatically generated during the search, and analysis of failures can be used to improve the execution of function free logic programs. Integrity constraints are used to guide both the forward and backward execution of the programs. This work applies to arbitrary node and literal selection functions and is thus transparent to the fact whether the logic program is executed sequentially or in parallel.

1. Introduction

1.1. The Problem

Interpreters for logic programs have employed, in the main, a simple search strategy for the execution of logic programs. PROLOG (Roussel [1975], Warren [1979], Roberts [1977]) the best known and most widely used interpreter for logic programs employs a straightforward depth first search strategy augmented by chronological backtracking to execute logic programs. This control strategy is 'blind' in the sense that when a failure occurs, no analysis is done to determine the cause of the failure and to determine the alternatives which may avoid the same cause of failure. Instead the most recent node where an alternative exists, is selected. This strategy has the advantage that it is efficient in that no decisions need to be made as to what to select next and where to backtrack to. However, the strategy is extremely inefficient when it backtracks blindly and thus repeats failures without analyzing their causes.

Pereira [1982], Bruynooghe [1978] and others have attempted to improve this situation by incorporating the idea of intelligent backtracking within the framework of the PROLOG search strategy. In their work the forward execution component remains unchanged, however, upon failure their systems analyze the failure and determine the most recent node which generated a binding which caused the failure. This then becomes the backtrack node. This is an improvement over the PROLOG strategy but still suffers from several drawbacks. Their scheme works only for a depth first search strategy and always backtracks to the most recent failure causing node. Also, once the backtrack node has been selected, all information about the cause of the failure is discarded. This can lead to the same failure in another branch of the search tree.

A node in the search space is said to be closed when it has provided all the results possible from it. In most PROLOG based systems a node cannot be closed until every alternative for that node is considered. However, by using integrity constraints as will be shown later, a node can be closed once it is determined that exploring further alternatives for that node will not provide any more results.

When executing a logic program it is often desirable to permit an arbitrary selection function and to have several active nodes at any given time. It is also useful to be able to remember the causes of failures and use this information to guide the search process.

1.2. Function Free Logic Programs and Integrity Constraints

The theory we treat is that of function-free Horn clauses as described in Kohli and Minker [1983]. It is assumed that the reader is familiar with Horn clause logic programs as described in Kowalski [1979]. We assume throughout the paper that the inability to prove an atom implies its negation (Clark[1978] and Reiter[1978]).

An integrity constraint is an invariant that must be satisfied by the clauses in the knowledge base. That is, if T represents a theory of function free logic programs and IC represents a set of integrity constraints applicable to T, then $T \cup IC$ must be consistent.

Integrity constraints are closed function free Horn formulae of the form:

(a) $\leftarrow P_1,\ldots,P_m$, or

(b) $Q \leftarrow P_1,\ldots,P_m$, or

(c) $E_1,E_2,\ldots,E_n \leftarrow P_1,\ldots,P_m$
where the E_i, $i=1,\ldots,n$, are equality predicates i.e. each E_i is of the form $x_i = y_i$ where at least one of the x_i, y_i are variables.

Thus, an integrity constraint of the form (a), represents negated data, in the sense that

* This work was supported in part by AFOSR grant 82-0303 and NSF grant MCS-79-19418.

$P_1 \wedge P_2 \wedge \ldots \wedge P_m$ can never hold if T U IC is consistent.

An integrity constraint of the form (b), states that if $P_1 \wedge P_2 \wedge \ldots \wedge P_m$ holds then Q must be provable from the knowledge base.

Integrity constraints of the form (c), represent dependencies between the arguments of $P_1, P_2, \ldots, P_m$.

2. Goals and Integrity Constraints

2.1. Integrity Constraints to Limit Forward Execution

Though integrity constraints are not necessary for finding the solution of a given set of goals with respect to a given logic program (Reiter [1978]), they can greatly enhance the efficiency of the search process and thus improve the performance of the problem solver (McSkimin and Minker [1977], King [1981]).

Integrity constraints enable the semantics of the given domain to direct the search strategy by enabling the problem solver to prune those alternatives which violate integrity constraints and thus focus the search. Thus integrity constraints influence the forward execution of the problem solver by enabling it to detect which sets of goals are unsolvable because they violate integrity constraints. This avoids exploring alternatives which must fail after a, possibly lengthy, full search.

Thus whenever a new set of subgoals is generated, this set can be tested to determine if it violates any integrity constraints. If so, the node in question can be discarded and another path considered.

2.2. Implementation and Search Strategy

There are several forms of integrity constraints as described in Section 1.2.

Whenever a new set of goals is generated it must be tested to determine if it violates an integrity constraint. Though each of the forms (a), (b), and (c) above require slightly different treatments to determine if they are violated, the underlying mechanism for each is the same.

Form (c) constraints can be transformed into form (a) by moving the disjunction of equalities on the left into a conjunction of inequalities on the right, i.e.,

$$E_1, E_2, \ldots, E_n \leftarrow P_1, \ldots, P_m$$

is equivalent to

$$\leftarrow P_1, \ldots, P_m, \overline{E_1}, \overline{E_2}, \ldots, \overline{E_n}.$$

Form (b) can be interpreted to mean that solving Q is equivalent to solving $P_1, \ldots, P_m$ and thus $P_1, \ldots, P_m$ can be replaced by Q in the set of goals.

Since it is only necessary to determine if the right hand side of some integrity constraint can subsume the goal clause, an extremely straightforward algorithm can be used. A clause C subsumes a clause D iff there exists a substitution σ such that $C\sigma \subseteq D$. The subsumption algorithm executes in linear time and does not increase the complexity of the search (Chang and Lee [1973]).

Consider now, how the various forms of integrity constraints can be used to limit the forward execution.

If a form (a) constraint subsumes a newly generated goal clause, the goal violates the constraint and can be deleted from the search space.

Whenever a literal is solved, it must be determined whether it unifies with a literal in the right hand side of a form (c) constraint. If so, the resulting substitution is applied to the constraint, the solved literal is deleted, and the clause is added to the set of integrity constraints.
For example, if

$$x_1 = x_2 \leftarrow P(x, x_1), P(x, x_2)$$

is a constraint and P(a,b) is solved then P(a,b) unifies with a literal in the above constraint with the substitution set $\{a/x,\ b/x_1\}$. The revised constraint

$$x_2 = b \leftarrow P(a, x_2)$$

is then obtained and added to the set of integrity constraints. Now any node containing P(a,x) can be considered a deterministic node, since only one possible solution for P(a,x) exists.

Finally, if the right hand side of a form (b) constraint subsumes the goal, then the resulting substitution is applied to the left hand side of the constraint and a new alternative goal with the left hand side substituted for the right hand side of the constraint, is generated. For example, if

$$Q(x,z) \leftarrow P_1(x,y), P_2(y,z)$$

is a constraint, and the goal clause under consideration is

$$\leftarrow R(a,u), P_1(a,y), P_2(y,z), S(b,z)$$

then

$$\leftarrow P_1(x,y), P_2(y,z)$$

subsumes the goal with substitution $\{a/x\}$. Applying this substitution to <- Q(x,z) results in <- Q(a,z). Generating an alternative node with Q replacing P_1, P_2 then results in the above goal node being replaced by the following OR-node

$$\begin{tikzcd}[column sep=tiny] & \text{Parent Node} \arrow[dl, no head] \arrow[dr, no head] & \\ R(a,u),Q(a,z),S(b,z) & & R(a,u),P_1(a,x),P_2(y,z),S(b,z) \end{tikzcd}$$

Whenever a violation of an integrity constraint occurs, it is treated as a failure. This results in failure analysis and backtracking which are detailed in the next section.

3. Local and Global Conditions

Global conditions are integrity constraints which are applicable to every possible node in the search space. Local conditions are integrity constraints which are generated during the proof process and which are applicable only to the descendant nodes of some given node in the search space.

3.1. Failure

The failure of a literal can provide information for directing the search. A literal 'fails' when it cannot be unified with the head of any clause in the knowledge base. Since this failure means that the literal cannot be proven in the current knowledge base, because of the assumption of failure by negation, the literal's negation can be assumed to hold. Thus, this literal can be viewed as an implicit integrity constraint, and the failure can be viewed as a violation of the integrity constraint. Thus, every failure can be viewed as a violation of some integrity constraint, implicit or explicit. This allows us to extract useful information from every failure, and to use this information in directing the search.

The possible causes of unification conflicts are:
(a) The literal is a pure literal. That is, there is no clause in the knowledge base, which has as its head the same predicate letter as the literal selected. This implies that any literal having the same predicate letter as the selected literal, will fail anywhere in the search space. This information can be useful in terminating other branches of the search tree in which a literal containing this predicate letter occurs. Thus if P(a,x) is a pure literal, then all of its argument positions can be replaced by distinct variables and the resulting literal can be added to the set of integrity constraints as a form (a) constraint, i.e.,

<- P(x_1,x_2)

is added to the set of constraints.

(b) There are clauses in the knowledge base which could unify with the selected literal, but which do not unify because of a mismatch between at least two constant names. In this case the selected literal can never succeed with that particular set of arguments. This information can be used as an integrity constraint. For example, if the selected literal is

P(x,a,x)

and the only P clauses in the knowledge base are

P(nil,nil,nil) <-
P(z,z,b) <- P_1(z,b),P_2(z)

then the unification fails and <-P(x,a,x) can be added to the set of integrity constraints.

3.2. Explicit and Implicit Integrity Constraints

Integrity constraints may be either explicit or implicit. Explicit constraints are those provided initially in the domain specification. These constraints affect the forward execution of the problem solver as detailed in Section 2 and can be used in the derivation of implicit constraints.

Implicit constraints are generated during the proof process, i.e., during the solution of a specific set of goals. These constraints arise out of the information gleaned from failure as shown in section 3.1, and from successes in certain contexts as will be shown in later sections. These constraints may be considered to be implicit in the sense that they are not explicitly supplied but are derived during the proof process.

3.3. Applicability of Integrity Constraints

An integrity constraint may be globally or locally applicable. It is globally applicable if it can be applied to any node in the search space. Explicit constraints are always globally applicable since they are defined for the domain and are independent of any particular proof tree. Implicit constraints may be either locally or globally applicable.

A locally applicable constraint is one which must be satisfied by a given node and all its children. Any node which is not part of the subtree rooted at the node to which the constraint is locally applicable, need not satisfy the constraint. Locally applicable constraints are derived from the failure of some path in the search space. The analysis of the cause of the failure results in the generation of a locally applicable constraint which is transmitted to the parent node of the failure node. This local constraint must then be satisfied by any alternative expansions of the node to which it applies. This effectively prunes those alternatives which cannot satisfy the constraint. The following example illustrates these techniques.

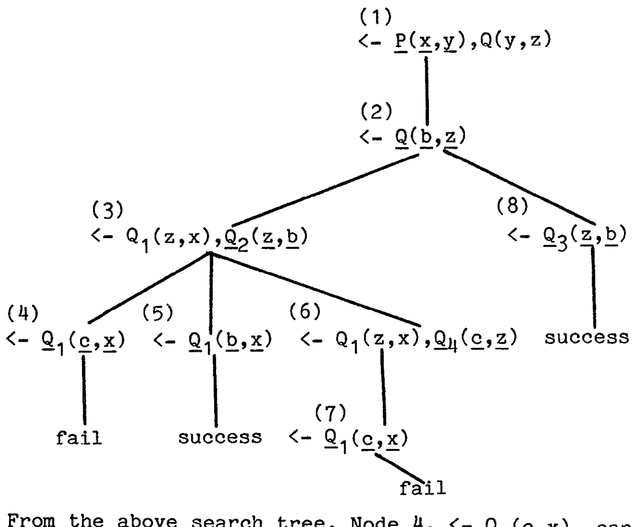
Logic Program:

P(a,b) <-
Q(y,z) <- Q_1(z,x), Q_2(z,y)
Q(y,z) <- Q_3(z,y)
Q_1(b,d) <-
Q_2(b,b) <-
Q_2(c,c) <-
Q_2(c,b) <-
Q_2(x,y) <- Q_4(c,x)
Q_3(c,b) <-
Q_4(x,x) <-

Query:

<- P(x,y),Q(y,z)

Search Tree:



From the above search tree, Node 4, <- Q_1(c,x) can be propagated as a global implicit constraint since <- Q_1(c,x) can never be solved. Also, z = c can be propagated as a local implicit constraint to node 3 and thus later prevent the generation of node 7. This constraint is local to node 3 and its children since that is the node that bound z to c. As can be seen from the example an alternative expansion of node 2 giving node 8 succeeds with z bound to c.

3.4. Generation and Propagation of Conditions

Implicit constraints are generated at leaf nodes of the search space and are propagated either globally or as locally applicable constraints to some parent of the leaf node. Rules for generating and propagating implicit constraints are detailed below.

When a goal fails along all paths, then that goal along with its current bindings is propagated as a global integrity constraint. Thus, if $P(\alpha_1, \alpha_2, \ldots, \alpha_n)$, where the α_i, $i = 1,\ldots,n$ are constants or variables, fails for every expansion of P, then $\leftarrow P(\alpha_1, \alpha_2, \ldots, \alpha_n)$ is a global constraint because it can never succeed in the current knowledge base.

Since that goal can never succeed with its current bindings, alternatives which give rise to different bindings for its arguments must be tried. Thus those nodes which created these bindings receive as local constraints the information that these bindings must not be repeated along alternative expansions of those nodes. That is, if $P(\alpha_1, \alpha_2, \ldots, \alpha_n)$ fails and there is some ancestor P' of P such that some α_i of P is bound to some x_i of P' and x_i is contained in some literal (other than P') in the clause containing P', then $\leftarrow x_i = \alpha_i$ is a local integrity constraint for the clause containing P'. If there are several α_i which have been bound in different ancestor clauses of P, the conjunction of these bindings must be propagated to the binding clauses.

Local constraints which are propagated to a node by a descendant of the node must then be propagated to all other descendants of that node. This is because, as was noted above, the binding to x_i in the node containing P_i was due to the selection of some literal other than P_i in that node. Thus, P_i will be present in every expansion of that node and the binding of x_i will cause P_i to eventually fail.

Consider a node P which has several children $P_1, P_2, \ldots, P_n$. Associated with each P_i is a set of local integrity constraints generated by its descendent nodes. Then if there is some local integrity constraint associated with every P_i, that integrity constraint is propagated to P.

4. Summary

A control strategy has been developed for function-free logic programs to permit intelligent search based both on domain specific information in the form of integrity constraints and on an analysis of failures. Integrity constraints limit search in the forward direction, while failure analysis results in the creation of integrity constraints. Failure analysis is also used to determine backtrack points more likely to succeed. The concepts of local and global constraints are used to inhibit exploring fruitless alternatives. Subsumption is employed to take advantage of the constraints. In Kohli and Minker [1983], a logic program is specified for an interpreter which will perform the above.

We intend to incorporate these concepts into PRISM, a parallel logic programming system [Kasif, Kohli and Minker 1983], under development at the University of Maryland.

REFERENCES

[1] Bruynooghe, M., Intelligent Backtracking for an Interpreter of Horn Clause Logic Programs, Report CW 16, Applied Math and Programming Division, Katholieke Universiteit, Leuven, Belguim, 1978.

[2] Chang, C.L., and Lee, R.C.T., Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

[3] Clark, K.L., "Negation as Failure", in Logic and Databases, H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp 293-322.

[4] Kasif, S., Kohli, M., and Minker, J., PRISM: A Parallel Inference System for Problem Solving, Technical Report, TR-1243, Dept. of Computer Science, University of Maryland, College Park, 1983.

[5] King, J.J., Query Optimization by Semantic Reasoning, Ph.D Thesis, Dept of Computer Science, Stanford University, May 1981.

[6] Kohli, M., and Minker, J., Control in Logic Programs using Integrity Constraints, Technical Report, Department of Computer Science, University of Maryland, College Park, 1983.

[7] Kowalski, R.A., Logic for Problem Solving, North-Holland, New York, 1979.

[8] McSkimin, J.R., and Minker, J., The Use of a Semantic Network in a Deductive Question Answering System, Proceedings IJCAI-77, Cambridge, MA, 1977, pp 50-58.

[9] Pereira, L.M., and Porto, A., Selective Backtracking, in Logic Programming, K.L. Clark and S-A. Tarnlund, Eds., Academic Press, New York, 1982, pp 107-114.

[10] Reiter, R., On Closed World Data Bases, in Logic and Databases, H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp 55-76.

[11] Roberts, G.M., An Implementation of PROLOG, M.S. Thesis, University of Waterloo, 1977.

[12] Roussell, P., PROLOG: Manuel de Reference et d'Utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, 1975.

[13] Warren, D.H.D., Implementing PROLOG: Compiling Predicate Logic Program, Department of Artificial Intelligence, University of Edinburgh. Research Reports 39 and 40, 1979.