# Control of Logic Programs Using Integrity Constraints

Madhur Kohli Jack Minker

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

## Abstract

This paper presents a theory for the intelligent execution of function free logic programs.

Generally, interpreters for logic programs have employed a simple search strategy for the execution of logic programs. This control strategy is 'blind' in the sense that when a failure occurs, no analysis is performed to determine the cause of the failure and to determine the alternatives which may avoid the same cause of failure.

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.

In this paper we present a theory for using integrity constraints, whether user supplied or automatically generated during the search, to improve the execution of function free logic programs. Integrity constraints are used to guide both the forward and backward execution of the programs. The theory supports 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

This paper presents a theory for the intelligent execution of function free logic programs.

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 performed 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 as to where to backtrack. 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. Pietrzykowski [1981] has considered intelligent backtracking in the framework of general first-order systems.

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.

In this paper we present a theory for using integrity constraints, whether user supplied or automatically generated during the search, to improve the execution of function free logic programs. Integrity constraints are used to guide both the forward and backward execution of programs. The theory supports arbitrary node and literal selection functions and is thus transparent as to whether the logic program is executed sequentially or in parallel.

In the rest of this section we define the class of logic programs to which this theory is applicable. In Section 2 we show how integrity constraints can be used to guide the forward execution of the system. In Section 3 we show how integrity constraints can be extracted from failure and propagated up the search tree. In the Appendix we present the interpreter for this theory.

## 1.2. Function Free Logic Programs

### 1.2.1. Horn Clauses

Horn clauses are a subset of the first order predicate calculus. The language of function free Horn clauses is defined below.

A term is a constant or variable.

An atomic formula is a predicate letter of arity $n \geq 1$ whose arguments are terms, i.e., if P is an n-ary predicate letter and $t_1,\ldots,t_n$ are terms then $P(t_1,\ldots,t_n)$ is an atomic formula.

An atomic formula or its negation is a literal. The classical logical connectors ~ (not), /\ (and), V (or) and the universal quantifier ∀ are used in constructing clauses.

A clause is a disjunction of literals all of whose variables are universally quantified. That is, $B_1 V \ldots V B_m V \sim A_1 V \ldots V \sim A_n$ is a clause. A clause can be written equivalently as $(\forall x_i)\ (B_1 V \ldots V B_m \leftarrow A_1 /\backslash \ldots /\backslash A_n)$ where the $x_i$, i=1,...,k are all the variables in the atomic formulae $A_i$, $B_j$, i=1,...,n, j=1,...,m. Since all variables in a clause are universally quantified, the universal quantifier will be omitted in the rest of this paper.

A Horn clause is a clause which has at most one positive literal, i.e., the clause above is Horn iff $m \leq 1$.

A function free logic program is composed of function free Horn clauses as follows:

1. Assertions are facts or general statements about the domain and are of the form
$$P(x_1,\ldots,x_m) \leftarrow$$
2. Procedures are of the form:
$$P(x_1,\ldots,x_m) \leftarrow P_1(\ldots),\ldots,P_n(\ldots)$$
which states that to solve P we must solve $P_1,\ldots,P_n$.
3. Goals or the problem to be solved are of the form
$$\leftarrow P(\ldots),Q(\ldots),\ldots,S(\ldots)$$

A function free logic program is defined in terms of the following:

(1) Axioms

(a) Domain Closure Axioms, which states that there is a finite set of constants $c_1,\ldots,c_n$ from which all constants in the knowledge base must be drawn.

(b) Unique Name Axioms, which state that the constants are unique i.e., $(c_1 \neq c_2)$, ..., $(c_1 \neq c_n)$, ..., $(c_{n-1} \neq c_n)$

(c) Equality Axioms:

reflexive $(x = x)$

symmetric $((y = x) \leftarrow (x = y))$

transitive $((x = z) \leftarrow (x = y) \wedge (y = z))$

principle of substitution of equal terms

$$P(y_1,\ldots,y_n) \leftarrow P(x_1,\ldots,x_n) \wedge (x_1=y_1) \wedge \ldots \wedge (x_n = y_n)$$

where P is an n-ary predicate letter.

(2) Assertions of the form

$$P(x_1,\ldots,x_n) \leftarrow$$

(3) Procedures of the form

$$P(x_1,\ldots,x_n) \leftarrow P_1(\ldots),\ldots,P_m(\ldots)$$

(4) A meta-rule: Negation as failure to prove positive literals. ([Clark 1978] and [Reiter 1978]).

### 1.2.2. Integrity Constraints

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 U 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) $E_1 \vee E_2 \vee \ldots \vee E_n \leftarrow P_1,\ldots,P_m$ where the $E_i$, i=1,...,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) above, represents negated data, in the sense that $P_1 \wedge P_2 \wedge \ldots \wedge P_m$ can never hold if T U IC is consistent.

An integrity constraint of the form (b) above, states that if

$$P_1 \wedge P_2 \wedge \ldots \wedge P_m$$

holds then Q must also hold.

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

$$\left.\begin{array}{l} P(x,y)\leftarrow F(x,y) \\ F(a,b)\leftarrow \\ F(b,c)\leftarrow \end{array}\right| T$$

and the associated integrity constraint

$$R(x,z) \leftarrow P(x,y), P(y,z) \mid IC$$

In the above example, $\overline{R(a,c)}$ can be proven from T, by using negation by failure. P(a,b) and P(b,c) can be proven from T. R(a,c) can be proven from P(a,b) and P(b,c) and IC. Thus R(a,c) can be proven from T U IC. Thus T U IC is inconsistent. The above integrity constraint is violated by the logic program since T U IC is inconsistent.

If, however, the clause:

$$R(x,z) \leftarrow F(x,y), F(y,z)$$

were added to T, then T U IC would be consistent and the integrity constraint would not be violated. Similarly, one could leave the axiom as an integrity constraint and add R(a,c) to the knowledge base and have a consistent theory.

## 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 [1979], 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. 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 an integrity constraint may take (Section 1.2.2).

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) 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.,

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

is equivalent to

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

These inequalities can then be handled by using predicate evaluation rather than negation.

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 all that is required, is to determine if the literals in the integrity constraint occur in the goal clause, an extremely straightforward algorithm can be used. It is only necessary to determine if the right hand side of some integrity constraint can subsume the goal clause.

A clause C subsumes a clause D iff there exists a substitution $\sigma$ such that

$$C\sigma \subseteq D$$

By $C\sigma$ we mean the result of applying a substitution set $\sigma$, which is composed of pairs of the form $a_i/x_i$ where the $x_i$ are variables of C and the $a_i$ are variables or constants, to C, i.e. each occurrence of $x_i$ in C is replaced by $a_i$.

The subsumption algorithm executes in linear time and does not increase the complexity of the search.

This algorithm (Chang and Lee [1973]) is presented below:

Let C and D be clauses.
Let $\theta = \{a_1/x_1,\ldots,a_n/x_n\}$ be a substitution set, where $x_1,\ldots,x_n$ are all the variables occurring in D and $a_1,\ldots,a_n$ are new distinct constants not occurring in C or D (Skolem constants) and the constants $a_i$ is to be substituted for the variable $x_i$, wherever it appears in C.
Suppose $D = L_1 \vee L_2 \vee \ldots \vee L_m$,
then $D\theta = L_1\theta \vee L_2\theta \vee \ldots \vee L_m\theta$
$D\theta$ is a ground clause since all variables in D have been replaced by Skolem constants.
Thus, $\sim D\theta = \sim L_1\theta \wedge \sim L_2\theta \wedge \ldots \wedge \sim L_m\theta$

1: Let $W = \{\sim L_1\theta,\ldots,\sim L_m\theta\}$

2: Let $k = 0$ and $U^0 = \{C\}$

3: If $U^k$ contains the null clause then terminate; C subsumes D
else let $U^{k+1}$ = {resolvents of $C_1$ and $C_2$ | $C_1 \in U^k$ and $C_2 \in W$}

4: If $U^{k+1}$ is empty then terminate; C does not subsume D
else set k to k+1; go to Step 3.

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

If a constraint is a form (a) constraint then all that is required is to apply the subsumption algorithm to the newly generated goal clause. If the constraint subsumes the goal clause, the goal violates the constraint and should be deleted from the search space.

Whenever a literal is solved, it must be determined whether it unifies with any 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 resulting 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 the right hand side of the above constraint with the substitution set $\{a/x, b/x_1\}$. Applying this substitution to the above constraint, and noticing that P(a,b) has been solved permits the revised constraint

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

to be obtained. This is then added to the set of integrity constraints. Also, this allows any node containing P(a,x) to be considered as a purely deterministic node, since only one possible solution for P(a,x) exists.

Finally form (b) constraints can be used as follows. If the right hand side of the 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

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

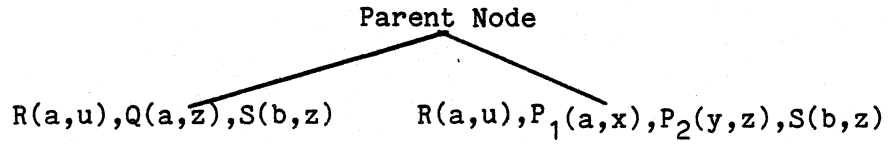
is a constraint, and the goal clause under consideration is

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

then

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

subsumes the goal with substitution $\{a/x\}$. Applying this substitution to $\leftarrow Q(x,z)$ results in $\leftarrow Q(a,z)$. 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




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.

In this section we show that both local and global conditions exist. We also show how implicit global and local conditions may be determined and how they can be used to improve the efficiency of a problem solver. We also show how both failure nodes and fully expanded nodes may be used to derive these conditions.

## 3.1. Failure

The failure of a literal can provide valuable information for directing the search. A literal 'fails' when it cannot be unified with the head of any clause (intensional or extensional) 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, the negation of the literal can be viewed as an implicit integrity constraint, and the failure can be viewed as a violation of this 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.,

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

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. This mismatch can occur in two ways:

(i) one of the constant names occurs in the literal and the other occurs textually in the head of the clause with which it is being matched.

(ii) both the constants, which are distinct, occur in only one of the literal or the clause head being matched, but are to be bound together by the repeated occurrence of a variable in the clause head or literal, respectively.

In both cases (i) and (ii) it is obvious that the selected literal can never succeed with that particular set of arguments. This information can be used as an integrity constraint by placing the literal as a form (a) constraint. For example, if the selected literal is

$$P(x,a,x)$$

and the only P clauses in the knowledge base are

$$P(nil,nil,nil) \leftarrow$$
$$P(z,z,b) \leftarrow P_1(z,b),P_2(z)$$

then the unification fails and $\leftarrow P(x,a,x)$ 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 integrity constraints are those which are provided initially in the domain specification. These constraints affect the forward execution of the problem solver as detailed in Section 2. These constraints can also be used in the derivation of implicit constraints.

Implicit integrity 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 integrity constraints may be considered to be implicit integrity constraints in the sense that they are not explicitly supplied integrity constraints but are derived from the proof process.

## 3.3. Applicability of Integrity Constraints

An integrity constraint may be globally or locally applicable. An integrity constraint is said to be globally applicable if it can be applied to any node in the search space. That is, it must be satisfied by every node on every success path in the proof tree. Explicit integrity constraints are always globally applicable since they are defined for the domain and are independent of any particular proof tree. Implicit integrity constraints may be either locally or globally applicable.

A locally applicable integrity 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 integrity 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 integrity constraint which is transmitted to the parent node of the failure node. This local integrity 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. For example, consider the following logic program fragment,

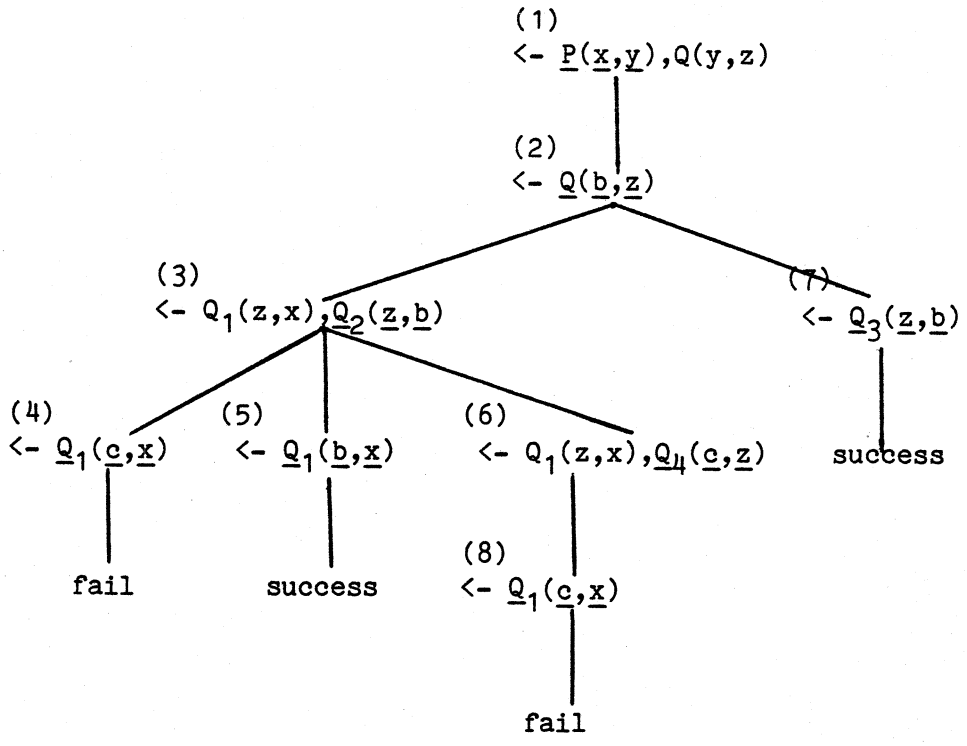
Logic Program:

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

Query:

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

Search Tree:




From the proof tree, the following information can be exteracted. From 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 8. This constraint is local to node 3 and its children since that is the node that bound z to c. Thus, node 6 has inherited this local constraint and thereby prevents z from being bound to c. As can be seen from the example an alternative expansion of node 2 giving node 7 succeeds with z bound to c, which illustrates that z = c at node 3 is a local constraint.

## 3.4. Fully Expanded Nodes

Implicit integrity constraints can be derived not only from failures but also from fully expanded nodes.

If the same local integrity constraint is generated by every expansion of a given node, then this constraint can be propagated globally, irrespective of whether or not some expansion of this node succeeded. Also any node which fails for every possible expansion, can be propagated as a global integrity constraint.

## 3.5. Generation and Propagation of Conditions

Implicit integrity constraints are generated at the leaf nodes of the search space and are then propagated either globally or as locally applicable integrity constraints to some parent node of the leaf node. The rules for generating and propagating these dynamic 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 $\alpha_i$, $i = 1,\ldots,n$ are constants or variables, fails for every expansion of P, then $\leftarrow P(\alpha_1, \alpha_2, \ldots, \alpha_n)$ is a global integrity constraint. This is because $P(\alpha_1, \alpha_2, \ldots, \alpha_n)$ can never succeed, given the current state of the 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 the failure causing bindings receive as local integrity constraints, the information that these bindings must not be repeated along alternative expansions of the nodes which created the bindings. That is, if $P(\alpha_1, \alpha_2, \ldots, \alpha_n)$ fails and there is some ancestor $P'$ of P such that some $\alpha_i$ of P is bound by 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 $\alpha_i$ which have been bound in different clauses then the conjunction of these bindings must be propagated to the binding clauses. That is, if $\beta_1, \beta_2 \ldots, \beta_m$ are constants, where $\beta_i = \alpha_j$ $i = 1,\ldots,m$; $j = 1,\ldots,n$, such that $\beta_i$ was bound by some ancestor $P_i$ of P, and if $P_1$ is the most recent ancestor of P and $P_m$ is the least recent ancestor of P, then $\leftarrow x_1 = \beta_1$ is propagated to the clause containing $P_1$ and $\leftarrow x_1 = \beta_1, x_2 = \beta_2, \ldots, x_i = \beta_i$ is propagated to the clause containing $P_i$. This is because undoing the $\beta_1$ binding at $P_1$ may suffice to remove the cause of failure whereas at $P_2$, undoing either of the $\beta_1$ or $\beta_2$ bindings may suffice. We do not propagate $\leftarrow x_1 = \beta_1, x_2 = \beta_2, \ldots, x_i = \beta_i$ to every $P_k$ since $x_i$ has been replaced by $\beta_i$ in $P_i$ and thus does not occur in any $P_k$, $k < i$.

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 of $\beta_i$ to $x_i$ in the node containing $P_i$ was due to the selection of some atom other than $P_i$ in that node. Thus, $P_i$ will be present in every expansion of that node and the binding of $\beta_i$ to $x_i$ will cause $P_i$ to eventually fail.

**Theorem:**

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.
Let $IC_i$ be the set of local integrity constraints associated with each $P_i$. Then $\bigcap_i IC_i$ is propagated to P.

**Proof :**

Let $IC_i$ be the set of local integrity constraints applicable to $P_i$, i.e.,

$IC_i$ is the set of constraints generated by the descendents of $P_i$ using the rules explained above.

Let $IC_P$ be the set of local integrity constraints applicable to P, then $\bigcap_i IC_i \subseteq IC_P$. Let $\gamma \lessdot \bigcap_i IC_i$, then $\gamma \lessdot IC_i$ $\forall$ i=1,...,n and thus every $P_i$ will fail for any binding that satisfies $\bar{\gamma}$. Thus since every $P_i$ fails, P must fail with any binding that satisfies $\bar{\gamma}$. Thus $\gamma \lessdot IC_P$ i.e. $\gamma \lessdot \bigcap_i IC_i \Rightarrow \gamma \lessdot IC_P$ and $\bigcap_i IC_i \subseteq IC_P$.

## 4. Summary

A theory has been developed for function-free logic programs to permit control of the 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 failures result in the creation of integrity constraints. Failure analysis is also used to determine backtrack points which are more likely to succeed. The concepts of local and global constraints have been introduced and are to be used to inhibit exploring fruitless alternatives. Subsumption is employed to take advantage of the constraints. A logic program is provided 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.

## 5. Acknowledgements

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

## 6. References

[Bruynooghe 1978]
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.

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

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

[Kasif, Kohli, and Minker 1983]
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.

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

[McSkimin and Minker 1977]
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.

[Pereira 1982]
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.

[Pietrzykowski 1982]
Pietrzykowski, T., and Matwin, A., Exponential Improvement of Efficient Backtracking: A Strategy for Plan Based Desduction, Proceedings of the 6th Conference on Automated Deduction, Springer Verlag, New York, June 1982, pp 223-239.

[Reiter 1978]
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.

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

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

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

## Appendix: The Interpreter

In this appendix we describe the interpreter, for function-free logic programs, which implements the theory described in the previous sections.

Defn: An open node is a node which has not been selected for expansion.

Defn: An active node is a node which has been selected for expansion but has not yet been fully solved.

Defn: A closed node is a node which has been completely expanded.

### 1. Interpreter Specification

1. Initialise the search space with the initial goals.
2. Select an open node from the search space. Mark it as active.
3. If the selected node is subsumed by a global integrity constraint then go to step 15.
4. Select an atom from the selected node.
5. Unify the selected atom with all procedure heads which have the same predicate letter.
6. Delete all procedures which require bindings which violate the local integrity constraints of the selected node, from the set of procedures which were selected in step 5.
7. If the set of procedures obtained after step 6 is empty, go to step 14.
8. Add the set of selected procedures to the set of all selected but unconsumed procedures generated so far.
9. Select one or more procedures from the set of unconsumed procedures.
10. For each of the procedures selected in step 8: generate descendant nodes of the node which was expanded to obtain these procedures. The descendants are generated by replacing the selected literal by the body of a matching procedure, and applying the substitution obtained from the unification process to the newly generated node. Mark the newly generated node as open.
11. If any of the newly generated nodes is empty (null clause), STOP.
12. Initialise the local constraint sets for the newly generated nodes, with the local constraint sets of their parent node.
13. go to step 2.
14. Add the selected node with all its current bindings to the list of global integrity constraints.
15. Mark the current node as closed. For each ancestor node which made a binding in the current node, add a local constraint which is the negation of the bindings.
16. Propagate the local constrint set to all children of each affected node.
17. Go to step 2.

## 2. Logic Program for the Interpreter

In this section we define the logic program for the interpreter described in the previous section.

A node is a 3-tuple

n(state,clause,localic)

where, state is either open, active or closed; clause is the set of goals represented by this node; localic is the set of local integrity constraints.

A body is a 2-tuple,

b(clause,sublist)

where, clause is the procedure body; and, sublist is the substitution set generated by unification of the literal and the procedure head.

```
solve(goals,prog,ic) <-
          init(goals,tree),
          interpret(tree,prog,ic).

init(goals,tree) <-
          inittree(n(open,goals,nil),tree).

interpret(tree,prog,ic) <-
          empty(tree).
interpret(tree,prog,ic) <-
          selectnode(tree,node,newtree1),
          expandnode(node,newtree1,prog,ic,newtree,newic),
          interpret(newtree,prog,newic).

expandnode(node,tree,prog,ic,newtree,newic) <-
          icok(node,ic),
          selectatom(node,atom),
          unify(atom,prog,bodies),
          checklocalic(node,bodies,newbodies),
          insertbodies(newbodies,node,atom,tree,ic,newtree,newic).

expandnode(node,tree,prog,ic,newtree,ic) <-
          not(icok(node,ic)),
          failurenode(node,tree,newtree).

icok(node,ic) <-
          empty(ic).
icok(node,ic) <-
          selectic(ic,oneic,restic),
          not(subsume(oneic,node)),
          icok(node,restic).

checklocalic(node,nil,nil,ic,ic) <- .
checklocalic(n(x,y,lic),b(z,subs).restbodies,b(z,subs).newbodies,ic,newic) <-
          checklicok(lic,subs),
          checklocalic(n(x,y,lic),restbodies,newbodies,ic,newic).
```

failurebindings(node,bindings) creates a list of bindings in node, undoing any one or more of which may undo the cause of failure.

addlocalic(parents,bindings,newparents) updates the local integrity constraint sets for parents with those elements of bindings which are applicable, and creates the new node list newparents.

propagatelocalic(parents,tree,newtree) propagates the local integrity constraint sets of each element of parents to every child of that node. It generates a newtree with all the newly modified nodes.

addic(node,ic,newic) adds node to the set of global integrity constraints ic to generate the new global integrity constraint set newic.

selectic(ic,singleic,restic) selects an integrity constraint singleic from the set of integrity constraints ic and creates the set restic which is ic - singleic.

```
checklocalic(n(x,y,lic),b(z,subs).restbodies,newbodies,ic,newic) <-
          not(checklicok(lic,subs)),
          addic(z,ic,newic1),
          checklocalic(n(x,y,lic),restbodies,newbodies,newic1,newic).

checklicok(nil,subs) <-
checklicok(lic.restlic,subs) <-
          not(member(lic,subs)),
          checklicok(restlic,subs).

insertbodies(bodies,node,literal,tree,ic,newtree,ic) <-
          not(empty(bodies)),
          inserttree(node,literal,bodies,tree,newtree).
insertbodies(bodies,node,literal,tree,ic,newtree,newic) <-
          empty(bodies),
          addic(node,ic,newic),
          failurenode(node,tree,newtree).

failurenode(node,tree,newtree) <-
          bindingnodes(node,tree,parents),
          failurebindings(node,bindings),
          addlocalic(parents,bindings,newparents),
          updatetree(parents,newparents,tree,newtree1),
          propagatelocalic(newparents,newtree1,newtree).
```

where,

inittree(node,tree) creates tree with node as its root node.

inserttree(node,literal,bodies,tree,newtree) creates a newtree which is formed from tree by generating children nodes of node by replacing literal in node by the atoms in each of the bodies and applying the substitutions to the newly generated nodes.

updatetree(nodes,newnodes,tree,newtree) modifies tree to produce newtree by replacing each element of nodes in the tree by the corresponding element in newnodes.

selectnode(tree,node,newtree) selects an open node from tree and creates a newtree which has node replaced by a new node marked as active.

selectatom(node,atom) selects an atom from node.

unify(atom,prog,bodies) selects those procedures from prog which have the same name as atom. It then applies the unification algorithm to each of these procedure heads and forms a list of those bodies which match the given atom, along with the resulting substitution lists.

subsume(ic,node) determines whether the integrity constraint ic subsumes node.

member(element,set) determines if element is a member of set.

empty(list) determines if list is empty(nil).

bindingnodes(node,tree,parents) sets parents to be the list of all ancestors of node which created a binding in node.