Mutually Exclusive Rules in Logic Programming
Kjell Post
Computer Science Center
University of California
Santa Cruz, CA 95064 USA
e-mail: kjell@cse.ucsc.edu

Abstract
A technique to detect that pairs of rules are "mutually exclusive" in a
logic program is described. In contrast to previous work our algorithm de-
rivs mutual exclusion by looking not only on built-in, but also user-defined
predicates. This technique has applications to optimization of the execution of
programs containing these rules. Additionally, the programmer is less depen-
dent on non-logical language features, such as Prolog's "cut", thus creating
more opportunities for parallel execution strategies.

Key Words: Mutual Exclusion, Dataflow Analysis, Logic Programming, PROLOG

1 Introduction
When a relation is defined by several rules in a logic program, there is sometimes
the opportunity to realize that if tuples of a certain pattern are derived by one of
these rules, no tuples of that pattern can be derived by other rules. This presents
an optimization opportunity.

The problem of "mutual exclusion", or disjointness, of rules has certain sim-
ilarities to constraint inference, but is actually quite a different problem. Both
may study order constraints, such as $X > Y$, but constraint inference attempts to
use properties of "$>$", such as transitivity, to infer additional constraints. How-
ever, in mutual exclusion detection, the basic fact available is that, for a given pair
$(X,Y)$ it is not possible that both $(X > Y)$ and $(Y > X)$ hold: they are mutu-
ally exclusive. In addition, patterns of mutual exclusion can occur without regard
to any ordering concept. For example, we might have tokentype($X$, number)
and tokentype($X$, identifier) that are known or given as disjoint on their first argument.

Let us mention a few applications of mutual exclusion detection.

1. In a top-down execution strategy with sideways information passing, if a tuple
for some subgoal\(^1\) $q^{c\ldots d}(X, \ldots, Z)$ is derived using one rule for $q$ and it is known
that all rules for $q$ are mutually exclusive on their first arguments, then none
of the remaining rules need to be tried with this value of $X$.

Detection of mutual exclusion is particularly useful when a goal matches two or
more rules with recursive definitions — if a recursive invocation happens when
other rules have yet to be tried, the execution engine must save backtracking
information in case the selected rule fails. But if the rules are mutually
exclusive to each other, such information can be discarded if the success of
all subgoals appearing before the recursive call contradicts the other rules.
This creates more opportunities for tail recursion optimization [War86] and
will also prevent the execution engine from wasting time on the other rules,
in case all solutions are asked for.

\(^1\) We use the notation that superscripts $c$ and $d$ denote "constant" (ground) and "don't know"-
arguments at the time the subgoal $q$ is "called."
2. In a bottom-up execution strategy the union of relations from mutually exclusive rules is a disjoint union; duplicates cannot occur.

3. In a logic language that contains the nonlogical “cut” operation, which is explicit in Prolog, and implicit in some others, there is a distinction drawn between “green” cuts and “red” cuts. Recall that “cut” in Prolog is a directive to the execution engine to cancel certain backtracking activities that would normally have occurred in the future. A cut is said to be “green” if the cancelled backtracking could not have produced any additional solution tuples (for reasons known to or believed by the programmer), otherwise it is “red”. Red cuts are often considered bad style for much the same reason as “go to”s in a procedural language. Mutual exclusion analysis can provide a sufficient condition for cuts to be “green”. In a reasonably expressive language (say SQL) the question is undecidable (by essentially the same arguments that show undecidability of “domain independence”).

4. The detection of functionality [DW89] represents an important space and time saving optimization for Prolog-like languages. Various forms of functionality have been considered; the algorithm by Debray and Warren defines functionality as when “all alternatives produce the same result, which therefore need not be computed repeatedly” [DW89]. With this information at hand, the execution engine does not need to waste time on, or save state information for finding alternative solutions. Debray and Warren demonstrates that a predicate is functional by showing that each individual clause is functional, and that the clauses are pairwise mutually exclusive. The latter requirement is not a necessary, but sufficient condition. Information produced by the methods outlined in this paper can therefore improve the precision of their algorithm.

5. In a parallel execution strategy with or-parallelism, a set of processors working on mutually exclusive rules can be relieved of their duties as soon as one of them succeeds. Also, in the Andorra model [HB88], goals with only one matching clause are executed before other goals. Static analysis has been used to detect such properties of goals [CWY91, PN91].

After discussing related work and establishing some required terminology, we illustrate the ideas of our technique by means of a small example before describing the full algorithm. A larger example concludes the paper.

1.1 Related Work

The idea of recognizing mutual exclusion in Prolog programs has been considered by Hickey and Mudambi [HM89], Debray and Warren [DW89], and Van Roy [VR90]. Their methods works on the level of primitives, i.e., only built-in predicates such as arithmetic comparisons and unifications are examined. In contrast to earlier work, our algorithm also examines user-defined predicates, even those with recursive definitions.

Several methods for static or dynamic inference of determinacy in Andorra programs have been proposed [PN91, CWY91, KT91]. In the Andorra model, a goal is determinate if it has at most one matching rule; thus the effort has been directed into checking whether two or more rules can satisfy a goal invocation. Whether our method, which is based not only on the information in the head of the rule, but also the presence of mutually exclusive subgoals in the bodies, can be useful for this purpose, remains to be explored.
A related topic, that of detecting functionality\(^2\), has been investigated by Debray and Warren [DW89], Sawamura and Takeshima [ST85], and Mellish [Mel85]. A functional predicate is one where all alternatives produce the same result. The notion is related to mutual exclusion, but is not the same, as the following two rules demonstrate:

\[
\text{son}(X, Y) \leftarrow \text{male}(X), \text{father}(X, Y) \\
\text{son}(X, Y) \leftarrow \text{female}(X), \text{mother}(X, Y)
\]

A goal \(\leftarrow \text{son}(\text{carl}, Y)\) could potentially have several solutions for \(Y\) — one for each of Carl’s sons — although only the first clause is applicable. Therefore \text{son} is determinate, but not functional.

Various methods for constraint inference in logical rules have been proposed [UVG88, BS89a, BS89b, KPR90, Las90, SVG91, VG91, BS91], but to our knowledge none have been implemented. The performance of our algorithm could be enhanced in practice if such an inference system were available, by simple rule transformations: whenever a constraint \(c(X, Y)\) is inferred to hold for the head of a rule, append it explicitly as an additional subgoal. Then run our algorithm as described on the resulting set of rules.

### 1.2 Summary of Contributions

A new technique for recognizing mutual exclusion among rules in logic programs is presented. While in general the problem is undecidable, a conservative analysis works on many programs, such as databases, natural language parsers, or any type of program where predicates operating on a certain domain divide it into equivalence classes.

Previous researchers have considered what we call primitive mutual exclusion, that is, the analyzer only looks at built-in predicates such as “\(=\)” or “\(>\)” . In contrast, our work facilitates a notation for propagating mutual exclusion information to user-defined predicates, even when they have recursive definitions. (Without recursion, a simple unfolding strategy would reduce the problem back to recognizing primitive mutual exclusion.)

Output from the analysis can be used to optimize various execution strategies. For instance, if all rules of a goal are mutually exclusive to each other, a top-down interpreter may at a certain point be able to commit to the rule it is working on. In languages where recursion is the only looping construct, such information could mean the difference of executing in constant space as opposed to space being linear with time.

### 2 Definitions

Following the standard terminology of [Llo87] some additional terms are defined for this paper.

The position of a term \(t\) in an atom \(A\) is given by the relation \(A \xrightarrow{\psi} t\) where \(\psi\) is a sequence of numbers that “spells” the argument path to \(t\) in \(A\) (“Dewey notation”). The head of a sequence \(\psi\) is denoted \(hd(\psi)\). As an example, if \(A\) is the

\(^2\)Sometimes called determinacy by some authors. We reserve this term for goals that have mutually exclusive rules.
atom \( p(X, g(X, Y)) \) we have both \( A \xrightarrow{1} X \) and \( A \xrightarrow{2} X \). The relation “\( \xrightarrow{\cdot} \)” is transitive: if \( A \xrightarrow{\psi_1} t_1 \) and \( t_1 \xrightarrow{\psi_2} t_2 \), it follows that \( A \xrightarrow{\psi_1 \psi_2} t_2 \).

The projection operation “\( \pi \)” from relational algebra is extended in the following manner. If \( \Psi = \langle \psi_1, \ldots, \psi_k \rangle \) is a tuple of paths, and \( A \) is an atom with a relation \( R \) then \( \pi_\Psi(R) = \langle t_1, \ldots, t_k \rangle \) where \( A \xrightarrow{\psi_i} t_i \). Sometimes it is convenient to have a relation or a tuple conform to the scheme of another relation; we use the customary notation \( \pi_R(S) \) to mean the projection of \( S \) onto \( R \)'s columns, and \( \mu[S] \) for the components of \( \mu \) in the attributes of \( S \).

The notation “\( R \| S \)” means that the relations \( R \) and \( S \) are disjoint.

Variables appearing in \( c \)-moded argument positions of rule heads are of particular interest to us. We restrict the “\( \xrightarrow{\cdot} \)”-relation to such ground variables as follows: if \( \langle \delta_1, \ldots, \delta_n \rangle \) is the mode tuple for a predicate \( p \) and \( p \xrightarrow{\psi} X \), then \( p \xrightarrow{\psi_{\beta}} c \) holds iff \( \delta_{\text{hd}(\psi)} = c \).

2.1 Rule/Goal Graphs

The input program is stored in a data structure called rule/goal graph. A rule/goal graph is basically a call graph for the program where rules and goals have been made explicit. Starting with the program’s goal rule, which we can assume to be of the form “\( \leftarrow B \)” for simplicity, we create a start goal node for \( B \). If there is a rule \( A \leftarrow B_1, \ldots, B_n \) such that \( A = B\theta \) then \( \theta(A \leftarrow B_1, \ldots, B_n) \) constitutes a rule node and becomes a child of \( B \). The rule node itself becomes the parent of \( n \) goal nodes, one for each subgoal in the body of the rule.

Goals are considered equal up to renaming of variables; if a goal already has a node in the graph we don’t expand it further but rather create a cyclic edge to the variant subgoal. It is helpful to think of the rule/goal graph as a directed acyclic graph with some occasional backarcs. The “leafs” in the graph then correspond to goals for built-in procedures such as “\( = \)” , “\( > \)” , true, or extensional database (EDB) predicates.

3 Deriving Mutual Exclusion

Two nodes in the rule/goal graph are said to be mutually exclusive if the relations they represent are disjoint. It is convenient to think of mutual exclusion not only between two rules, but also between a rule and a goal, or between two goals. Thus, mutual exclusion is a symmetric binary relation that can be represented by an edge between two nodes in the rule/goal graph.

In fig. 1 we see the rule/goal graph for a program and its goal, along with some mode information. Ultimately we are interested in deriving a mutual exclusion between all rules of a max (fig. 1c) to show that it is determinate.

\footnote{Some researchers use the notation \( \{b, f\} \) instead of \( \{c, d\} \) and speak of mode information as “adornments”.}
Underlying the mutual exclusion between two intermediate nodes is always some initial mutual exclusion between two built-in goals. When variables appear in the built-in goals it is essential to show that the variables in the second goal would have been bound in the same way as in the first goal, before we exclude the second goal.

In general we do not have to ensure that all variables in the built-in goals become bound; only certain variable positions are critical for establishing mutual exclusion. For instance, only $X$ needs to be bound to make the subgoals $(X = \{0|T\})$ and $(X = \{1|T\})$ mutually exclusive. If we think of the nodes as representing relations, we can state what the critical positions are by projecting on the interesting columns. For instance, the requirement for the initial edge between node 4 and 5 in fig. 1a is $\pi_{1,2}(\leq 4) \parallel \pi_{1,2}(> 5)$. (Relations are subscripted according to the nodes they appear in.) This states that the first and second column in the respective relations are necessary to demonstrate the mutual exclusion. Another edge, not shown in fig. 1 to simplify the exposition, is from node 5 to itself, with the requirement $\pi_{1,2}(> 5) \parallel \pi_{2,1}(> 5)$. This simply states that “$>$” is anti-symmetric.

We will now illustrate how the mutual exclusion between $\leq 4$ and $\max 3$ can be derived, based on the existing mutual exclusion of $\leq 4$ and $> 5$. It is convenient to think of this step as trying to lift the edge in fig. 1a along its right side. The intuition behind this move is that a node $n$ may be mutually exclusive with a rule node if $n$ is mutually exclusive with at least one of the subgoals of the rule. (The other case, showing mutual exclusion between a node $n$ and a goal node requires that $n$ is mutually exclusive with all rules for the goal.)

We must now keep track of what happens to the variables $X$ and $Y$ as the goal $> 5$ is “sucked up” into its parent rule. Notice first that only the first and second argument positions of $\max 3$ are specified as having ground mode. If the variables in the goal of $\max 3$ are to be ground, they must pass through $c$-modeled argument positions in the head. In this case, both variables do appear in $c$-modeled positions of the head: $\max 3(X, Y, X) \xrightarrow{1} c X$ and $\max 3(X, Y, X) \xrightarrow{2} c Y$ both hold. The condition for mutual exclusion between relations $\leq 4$ and $\max 3$ can be now be stated as follows

$$\pi_{1,2}(\leq 4) \parallel \pi_{1,2}(\max 3) \leftarrow \pi_{1,2}(\leq 4) \parallel \pi_{1,2}(> 5).$$

Since the right hand side is already given, the conclusion follows immediately.

In the next step the algorithm would try to lift the right side of the edge in fig. 1b and show the mutual exclusion between $\leq 4$ and $\max 1$. This fails however, since $\leq 4$ is not mutually exclusive with all the rules of $\max 1$ (in particular $\max 2$). Not being able to make any more progress on the right side of the edge, it is time to start working on the left side. The rules $\max 2$ and $\max 3$ are candidates for mutual exclusion because node $\max 3$ is already mutually exclusive with $\leq 4$, one of $\max 2$’s subgoals. The variables in the goal appear in $c$-modeled positions in the head of the rule, thus

$$\pi_{1,2}(\max 2) \parallel \pi_{1,2}(\max 3) \leftarrow \pi_{1,2}(\leq 4) \parallel \pi_{1,2}(\max 3).$$

Again, the conclusion follows immediately, and, as we shall see in the next section, that is always the case. To find out if $\max$ is determinate we could now ask the question: is there a fixed set of argument positions that differentiates all rules of $\max$, i.e., is there a $\Psi$ such that $\pi_{\Psi}(\max 2) \parallel \pi_{\Psi}(\max 3)$? The answer is yes, since $\Psi = (1, 2)$ has been shown to do so.

While we hope that this small example has illustrated the principles of our method, the scenario is generally more complicated because of the ways variables in the built-in goals are aliased as we work our way up the graph. In the next section we state the full algorithm and discuss its complexity.
3.1 Algorithm for Propagating Mutual Exclusion

A set $\textsc{needs\_processing}$ is used to hold triples of the form $\langle x, y, u \rangle$, representing mutual exclusion between $x$ and $y$. Since rules may have several goals serving as mutual exclusion witnesses, we maintain a third element $u$ which is the node used in the process of adding the edge from $x$ to $y$. Without it, some opportunities may be missed because an incorrect assumption was made about what the distinguishing goal was.

After the rule/goal graph has been built, “leaf” nodes are scanned and used to access a table with requirements for mutual exclusion between built-in goals. When applicable, these requirements are added as initial axioms to a database $\textsc{mutex\_facts}$. For instance, one entry in the table might read “for two goals $(V < m)$ and $(W > n)$ (where $m$ and $n$ are integers) the axiom $\pi_1(V < m) \parallel \pi_1(W > n)$ can be added if $m \leq n$."

Finally, for each pair $(x, y)$ of leaf nodes that lead to an initial axiom, the triple $\langle x, y, \_ \rangle$ is added to the set $\textsc{needs\_processing}$. The main algorithm consists of picking an edge and trying to lift it up along one of its sides.

while $\textsc{needs\_processing} \neq \emptyset$ do
  let $\langle x, y, v \rangle$ be an element of $\textsc{needs\_processing}$;
  if not lift_right$(x, y)$ then lift_right$(y, x)$;
  $\textsc{needs\_processing} := \textsc{needs\_processing} - \{\langle x, y, v \rangle\}$;
end.

In the function lift_right$(x, y)$, we will use the function $\text{parents}(y)$ to mean the set of immediate predecessors of a node. (As the name implies, $\text{parents}(y)$ does not return $v$ if there is a backarc from $v$ to $y$.) Since goal nodes can have more than one parent, lifting the right side of an edge can in general be done in more than one way, the effect being that the current edge of $\textsc{needs\_processing}$ is replaced with zero or more new edges. If no mutually exclusive edges are derived between $x$ and any parent of $y$, lift_right returns false to indicate that no more progress can be made on this side and that it is time to start lifting the other side.

function lift_right$(x, y)$
  progress := false;
  for all $p \in \text{parents}(y)$
    if $(x, p, y) \notin \textsc{needs\_processing}$ then
      progress := (progress or try$(x, p, y)$);
    end:
  return progress;
end:

The function try$(x, p, y)$ attempts to derive mutual exclusion between nodes $x$ and $p$, based on the existing mutual exclusion of $x$ and $y$, where $p$ is a parent of $y$.

When $p$ is a rule node, only one of its goals needs to be mutually exclusive with $x$. That goal is $y$. If the critical argument positions of $y$ can be propagated to the head of $p$ we can safely establish a mutual exclusion between $x$ and $p$. We use the relation $\text{prop}(\Psi_y, \Psi_p)$ for propagating the critical argument positions of $y$ (denoted $\Psi_y$) to the corresponding positions in $p$ (denoted $\Psi_p$). The definition of the prop relation, being rather technical, is given later (c.f. definition 3.2).

The other case, when $p$ is a goal node, requires that $x$ is mutually exclusive with all rules $p_1, \ldots, p_k$ of $p$. In the relational view, since $p$ is the union of all the $p_i$ relations, the same column numbers used to differentiate $x$ and $p$ must be used to
differentiate $x$ from all $p_i$'s. Hence, a fixed set of argument positions $\Psi_p$ is used for this purpose.

**function** try($x, p, y$)
  
  if $p$ is a rule node (with $y$ as one of its goals) **then**
  
  FACTS := \{ $\pi_{\Psi_x}(x) \| \pi_{\Psi_p}(p)$ such that $\pi_{\Psi_x}(x) \| \pi_{\Psi_y}(y) \& prop(\Psi_y, \Psi_p)$ \};
  
  else ($p$ is a goal node with rules $p_1, \ldots, p_k$)
  
  FACTS := \{ $\pi_{\Psi_x}(x) \| \pi_{\Psi_p}(p)$ such that $\pi_{\Psi_x}(x) \| \pi_{\Psi_x}(p_1) \& \cdots \& \pi_{\Psi_x}(x) \| \pi_{\Psi_x}(p_k)$ \};
  
  if |FACTS| > 0 **then**
  
  NEEDS_PROCESSING := NEEDS_PROCESSING + \{ $(x, p, y)$ \};
  
  add FACTS to MUTEX_FACTS;
  
  return true;
  
  else return false;
  
  end;

When the algorithm has traversed the entire rule/goal graph, queries about mutual exclusion between nodes can be answered using MUTEX_FACTS. As an example, to find out if a goal $g$ is determinate, i.e., whether all its rules are pairwise mutually exclusive, we can use the following function.

**function** determinate($g$)
  
  ($g$ is a goal with rules $r_1, \ldots, r_k$)
  
  for all pairs $(r_i, r_j)$
  
  if there is no $\Psi_g$ such that $(\pi_{\Psi_x}(r_i) \| \pi_{\Psi_x}(r_j)) \in$ MUTEX_FACTS **then**
  
  return false;
  
  return true;
  
  end;

**3.2 Termination and Complexity**

Two properties of the algorithm guarantee termination.

1. The current mutual-exclusion edge, represented by the triple $(x, y, v)$, is always replaced by zero or more new edges.

2. New mutual-exclusion edges have at least one node closer to the root of the rule/goal graph, and such edges are not propagated across backarcs.

To derive an upper bound on the time complexity of the algorithm we observe that no triple $(x, y, v)$ is processed more than once. With $n$ nodes in the rule/goal graph there can be at most $n^3$ such triples. However, the time to process one triple depends on how many arguments appear in the relevant nodes, as mutual exclusion is checked for various permutations of those arguments. Assuming the number of arguments in any relation never exceeds some predefined constant, the time per triple can be considered $O(1)$. In conclusion, our algorithm is guaranteed to terminate in so-called semi-polynomial time ($O(n^3)$), i.e., polynomial in the size of the rule/goal graph and exponential in the maximum arity of any relation in the program.

**3.3 Correctness**

As already pointed out, the mutual exclusion detection problem is, in general, unsolvable (c.f. [HM89] for an example) and hence no complete algorithm exists.
Soundness can be verified by making sure that the function \textit{try} does not derive a mutual exclusion between two nodes when in fact they are not mutually exclusive. In the proof, we study \textit{variable binding} relations. These are relations whose attributes correspond to the variables appearing in a node, and whose values represent possible bindings for the variables. For a precise definition of variable bindings, see [Ull89, page 748].

**Example 3.1:** If node \(n\) contains the goal \(p(f(X), g(h(Y), X))\), and is described by the relation

\[
\{(f(a), g(h(c), a)), (f(f(a)), g(h(d), f(a)))\},
\]

then the variable bindings are \(V_n = \{(a, c), (f(a), d)\} \) with the scheme \(\{X, Y\}\).

**Definition 3.1:** In a rule \(p' \leftarrow p_1, \ldots, p_m\) the variable bindings \(V'\) for \(p'\) can be described in terms of a natural join between the variable bindings \(V_i\) for the subgoals in the body: \(V' = V_1 \Join \cdots \Join V_m\). (c.f. [Ull89, page 751].)

**Lemma 3.1:** If \(R = R_1 \Join \cdots \Join R_m\) then \(\pi_{R_i}(R) \subseteq R_i\).

**Proof:** Assume by contradiction that there is a \(\mu\) in \(R\) such that \(\mu[R_i]\) is not in \(R_i\). This implies that \(\mu\)'s attributes didn't agree with \(R_i\). However, this can't be true for otherwise \(\mu\) would not have been in \(R\) in the first place.

**Theorem 3.2: Mutex_Facts** is sound.

**Proof:** By induction on the height of the rule/goal-graph, modulo backarcs.

**Basis.** The base case is two “leaf” nodes, \(x\) and \(y\), representing built-in goals. In this case, it is assumed that the initial axioms describing the mutual exclusion between built-in goals are correct. These axioms are statements of the form \(\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_y)\).

**Induction.** Consider facts added by the function \textit{try}(\(x, p, y\)) where \(p\) is a parent of \(y\). The rest of this proof proceeds in two separate cases:

1. \(p\) is a rule node, with node \(y\) as one of its subgoals. The inductive hypothesis is that \(\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_y) = \emptyset\). By definition 3.1 we have \(V_p = \cdots \Join V_y \Join \cdots\). From lemma 3.1 it follows that \(\pi_{V_p}(V_p) \subseteq V_y\). Thus, with the inductive hypothesis, \(\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_p) = \emptyset\).

   Since the variables in \(y\) must appear in the head of the rule we can simplify to \(\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_p) = \emptyset\), i.e., \(\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_p)\), which is what we wanted to show.

2. \(p\) is a goal node with rule nodes \(y_1, \ldots, y_k\). Again, the inductive hypothesis is that \(\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_y)\) is correct for \(i = 1 \ldots k\). (Note that \(\Psi_y\) is applied to all rules.) As before, the inductive hypotheses may also be stated as \(\pi_{\Psi_x}(V_x) \cap \pi_{\Psi_y}(V_{y_i}) = \emptyset\), \(i = 1 \ldots k\). Hence, \(\pi_{\Psi_x}(V_x) \cap (\bigcup_i \pi_{\Psi_y}(V_{y_i})) = \emptyset\).
Since projection distributes over union, \( \pi_{\Psi_y}(V_x) \cap \pi_{\Psi_y}(\bigcup_i V_y^i) = \emptyset \). It remains to show that \( \pi_{\Psi_y}(\bigcup_i V_y^i) = \pi_{\Psi_y}(V_p) \). Intuitively, the left hand side of this conjecture represents a set of values found by reaching inside the \( V_y^i \)’s and the question is, can these values also be found inside \( V_p^i \)? Observe that, by construction, the rule heads in \( y_1 \ldots y_k \) are various instances of the goal in node \( p \). Therefore, the same value held by a variable \( Y \) in some rule head may also be found in some variable \( Z \) in the goal node (although the value might be more “embedded”), provided, of course, that \( Y \) is found in the same position in each and every rule head (hence \( \Psi_{\Psi_y} \)). More formally, let \( \theta_i \) be the unifier between the goal node and rule head \( i \). If \( Z/t \in \theta_i \) and \( t \mapsto^\rho Y \), \( i = 1 \ldots k \), then \( \pi_{\Psi_y}(V_y^i) = \pi_{\rho+\Psi_y}(Z) \).

### 3.4 Description of prop

We now turn to the specification of \( \text{prop} \), used by the function \( \text{try} \) in section 3.1. The problem we face is to keep track of certain variable positions for two disjoint built-in goals, and make sure that when they both can be invoked from two different rules, only one of them can succeed since the variables will be bound to the same value (whatever that value is). The relation \( \text{prop} \) is the sole mechanism used for describing how the path to these values change as we move from one node to another in the rule/goal-graph.

In the proof we have already outlined a way to keep track of values within variables. Variable bindings were introduced in the proof simply because it was easier to describe how variable values, rather than full relations, are propagated from subgoals to the head of the rule (definition 3.1).

To determine the position of a value inside a literal we only need to append the variable’s position in the literal to the value’s position in the variable. The reader may verify that the following definition of \( \text{prop} \) captures the above ideas.

**Definition 3.2:** Let \( H \leftarrow G_1, \ldots, G_k \) be a rule where a the position \( \psi \) of some \( G_i \) needs to be propagated to a corresponding position \( \psi' \) in \( H \).

\[
\text{prop}(\psi, \psi') \leftarrow \psi = \rho \tau, G_i \mapsto^\rho X, \var(X), H \mapsto^\rho X, \psi' = \rho' \tau.
\]

To simplify the notation we will “lift” \( \text{prop} \) to work on tuples of paths, \( \text{prop}(\Psi, \Psi') \), with the obvious component-wise meaning.

**Example 3.2:** Consider the rules

\[
q(f(X, g(Y))) \leftarrow Y > 0.
\]

\[
p(U, V, Z) \leftarrow q(f(X, Z)).
\]

The original critical position for “\( Y > 0 \)” is “1” which corresponds to \( Y \). When propagated to the head of \( q \), \( Y \)’s position becomes “1.2.1”, hence \( \text{prop}(1, 1.2.1) \) holds for \( q \)’s rule. As we try to follow the path “1.2.1” in the subgoal of \( p \) (a more general atom than \( q \)’s head) we run into \( Z \) after “1.2”. Mapping “1.2” to the head of \( p \) gives us “3” (\( Z \)’s position in the head) so the critical argument position is now “3” appended with the trailing “1”. Hence \( \text{prop}(1.2.1, 3.1) \) holds for \( p \)’s rule.
4 A Larger Example

The program in fig. 2 implements a parser for a small natural language. All queries are assumed to be ground. In the program, the rules for np2 specify two ways to form a noun phrase. Being able to follow only one branch at a time, a top-down interpreter would pick the first rule and save backtracking information on the stack so that it can go back and try the second rule, in case the first one fails. Once the interpreter has managed to solve adj it is possible to discard the backtracking information just created, since adj and noun are mutually exclusive in their first argument. No system today, that we are aware of, recognizes this opportunity.

Since np2 is recursive, a top-down interpreter that doesn’t recognize the mutual exclusion would generate an amount of backtracking information that is proportional to the number of adjectives in the input string. This information stays around until the user is satisfied with the answer, and quits. If all solutions are asked for, the interpreter would start a series of meaningless attempts on the second rule for np2, which we know will fail.

We will now trace the execution of our algorithm for the parsing program, and see how the mutual exclusion between the two rules of np2 is derived. The rule/goal graph for the interesting part of the program can be seen in fig. 2. Initially, all leaf nodes (16, 22, and 20) are mutually exclusive with each other, provided that the variable L is bound. Stated in terms of our axioms, we start with the initial mutex facts:

\{π1(16)∥π1(22), π1(16)∥π1(20), π1(22)∥π1(20)\}

and needs-processing set to \{(16, 22, _), (16, 20, _), (22, 20, _).\}

The table in fig. 3 shows the edges picked from the needs-processing set, along with the new mutual exclusion facts added, and how they were derived. After the execution, we can query the function determinate with the goals noun18 and np212. In both cases, the answer is true. For the entire program, the algorithm would also discover that np is determinate.

5 Limitations

At the moment, we are not even trying to derive mutual exclusion if a local variable of a test goal, which does not occur in a bound argument of the head of the rule, would be involved. The following program illustrates why this is unsound, in general.

\[
\begin{align*}
q(b). & \quad q(c). & \quad r(a,b). & \quad s(a,c). \\
p(X) & \leftarrow q(Y), r(X,Y). \\
p(X) & \leftarrow q(Y), s(X,Y).
\end{align*}
\]

Assume that \(p\) is called with its argument ground. Even though the relations for \(r\) and \(s\) are disjoint, and are only called with ground arguments, it is not correct to say that the two rules for \(p\) are mutually exclusive. For instance, in a top-down execution, the local variable \(Y\) may get rebound upon backtracking into \(q\).

However, there are other situations where, by detection of “functionality” (Section 1.1), one can safely conclude that a local variable cannot be bound to more than one value.

The rather coarse description of arguments into modes \{c,d\} are for practical purposes too rigid to be useful. As an example, for the two goals \(X = [0|T]\) and \(X = [1|T]\) it is too restrictive to demand that \(X\) should be c-moded considering the number of programs that deal with partially instantiated data structures.
6 Conclusion and Future Work

A new conservative approximation technique for the undecidable problem of recognizing mutual exclusion among rules in logic programs has been presented. The information is derived statically (at compile-time), and may aid in both time and space optimizations during execution. Additionally, the programmer is less dependent on non-logical language features, such as Prolog’s “cut”, thus creating more opportunities for parallel execution strategies.

An extension to this research would be to analyse propagation of bindings from head variables to local variables. As mentioned in Section 1.1 and 5, constraint inference methods and detection of “functionality” can be applied.

As with many other analysis programs, some sort of type informations should replace the mode system that is currently in use.

Further analysis and improvement of this algorithm may lead to a description in terms of abstract interpretation [CC92, BJCD87]. Since our method mimics a bottom-up execution, the integration with [MS88] seems most probable.

Acknowledgements

This research was partially supported by NSF grants CCR-89-58590 and IRI-9102513, and by equipment donations from Sun Microsystems, Inc., and software donations from Quintus Computer Systems, Inc.

The author gratefully acknowledges valuable insights from his advisor Allen Van Gelder and would also like to thank Saumya Debray, Peter Van Roy, and Ola Petersson for background information and interesting discussions. The IILPS94 committee contributed with important comments on the paper.

References


Figure 1: Propagation of mutual exclusion (denoted by thick edges) in the rule/goal graph for the program \{max(X, Y, Y) ← X ≤ Y, max(X, Y, X) ← X > Y\} and the goal "← max_{c.e.d}(X, Y, Z)."
\[ s(L) \leftarrow np(L, R), \text{vp}(R, _), \]
\[ np(L, R) \leftarrow \text{det}(L, T), \text{np}(T, R). \]
\[ np(L, R) \leftarrow \text{np}(L, R). \]
\[ np2(L, R) \leftarrow \text{adj}(L, T), \text{np}(T, R). \]
\[ np2(L, R) \leftarrow \text{noun}(L, R). \]
\[ \text{vp}(L, R) \leftarrow \text{verb}(L, R). \]
\[ \text{vp}(L, R) \leftarrow \text{verb}(L, T), \text{np}(T, R). \]
\[ \text{det}([\text{the}[R], R]). \]
\[ \text{det}([\text{a}[R], R]). \]
\[ \text{adj}([\text{new}[R], R]). \]
\[ \text{noun}([\text{code}[R], R]). \]
\[ \text{noun}([\text{bugs}[R], R]). \]
\[ \text{verb}([\text{has}[R], R]). \]
\[ \leftarrow s([\text{the}, \text{new}, \text{code}, \text{has}, \text{bugs}]). \]

---

**Figure 2:** A parsing program and part of its rule/goal graph.

<table>
<thead>
<tr>
<th>((x, y, v))</th>
<th>New Facts</th>
<th>Derived From</th>
</tr>
</thead>
<tbody>
<tr>
<td>((16, 22, _))</td>
<td>(\pi_1(16)|\pi_1(21)) (\pi_1(16)|\pi_1(22))</td>
<td></td>
</tr>
<tr>
<td>((16, 20, _))</td>
<td>(\pi_1(16)|\pi_1(19)) (\pi_1(16)|\pi_1(20))</td>
<td></td>
</tr>
<tr>
<td>((22, 20, _))</td>
<td>(\pi_1(16)|\pi_1(19)) (\pi_1(16)|\pi_1(20))</td>
<td></td>
</tr>
<tr>
<td>((16, 21, 22))</td>
<td>(\pi_1(16)|\pi_1(18)) (\pi_1(16)|\pi_1(21))</td>
<td></td>
</tr>
<tr>
<td>((16, 19, 20))</td>
<td>(\pi_1(19)|\pi_1(15)) (\pi_1(19)|\pi_1(16))</td>
<td></td>
</tr>
<tr>
<td>((22, 19, 20))</td>
<td>(\pi_1(19)|\pi_1(21)) (\pi_1(19)|\pi_1(22))</td>
<td></td>
</tr>
<tr>
<td>((16, 18, 21))</td>
<td>(\pi_1(16)|\pi_1(27)) (\pi_1(16)|\pi_1(18))</td>
<td></td>
</tr>
<tr>
<td>((19, 15, 16))</td>
<td>(\pi_1(19)|\pi_1(14)) (\pi_1(19)|\pi_1(15))</td>
<td></td>
</tr>
<tr>
<td>((16, 27, 18))</td>
<td>(\pi_1(27)|\pi_1(15)) (\pi_1(27)|\pi_1(16))</td>
<td></td>
</tr>
<tr>
<td>((19, 14, 15))</td>
<td>(\pi_1(19)|\pi_1(13)) (\pi_1(19)|\pi_1(14))</td>
<td></td>
</tr>
<tr>
<td>((27, 15, 16))</td>
<td>(\pi_1(27)|\pi_1(14)) (\pi_1(27)|\pi_1(15))</td>
<td></td>
</tr>
<tr>
<td>((27, 14, 15))</td>
<td>(\pi_1(27)|\pi_1(13)) (\pi_1(27)|\pi_1(14))</td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 3:** Execution trace for the parsing program.