Fully Distributed, AND/OR Parallel Execution of Logic Programs

Prabhakaran Raman Eugene W. Stark *
TR-88/02
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794 USA
E-mail : raman@sbcs.sunysb.edu

February 13, 1988

Abstract

We consider a distributed model for the execution of logic programs, in which a process is assigned to each node of the AND/OR tree for a program, and in which each process communicates directly only with its immediate neighbors in the tree. We derive an interpreter, or execution method, for this model, in which each node process maintains in its state a set of substitutions that represents a current local approximation to the set of answer substitutions. Execution consists of each process repeatedly sending its current state to its neighbors, and updating its state using information it receives. Eventually, "exact" answer substitutions accumulate at the root node, and are output. The interpreter supports both AND- and OR-parallelism in a completely unrestricted fashion. In particular, bidirectional communication can occur between two children of the same AND-node, a feature not present in previous work. We prove that our interpreter is sound and complete. The proof, which is nontrivial, hinges on an interesting property of information flow in the AND/OR-tree.

*This research was supported in part by NSF Grant CCR-8702247.

1 Introduction

A *logic program* consists of a finite, nonempty set of *definite Horn clauses*, which are closed formulas of first-order logic of the form $\forall x_1, \ldots x_m (g_1 \wedge \ldots \wedge g_n \supset h)$, where $g_1, \ldots, g_n$ and h are *literals*, or atomic formulas. It is customary, in the logic programming literature, to write such a formula as $h \leftarrow g_1, \ldots, g_n$, and to read it as "*h if* g_1 *and* ... *and* g_n." The literal h is called the *head* of the clause, and the sequence of literals $g_1, \ldots g_n$ is called its *body*.

Execution of a logic program begins when an input literal g, called the *goal*, has been specified. The objective of an *interpreter*, or execution method, is to *solve* the goal g. That is, the interpreter searches for an assignment of terms to the variables of g, such that if g' is the instance of g obtained by substituting each occurrence of a variable by the corresponding term, then g' is a logical consequence of the set of program clauses. Such an assignment is called an *answer substitution*. In general, there will be more than one, and possibly infinitely many, answer substitutions for a given program and goal, so the output of the interpreter will be a finite or infinite sequence of substitutions. An interpreter is *sound* if it only outputs answer substitutions, and *complete* if every ground answer substitution is an instance of some output substitution.

The operation of an interpreter presented with a logic program $\mathcal{P}$ and a goal g is conveniently described in terms of the *AND/OR-tree* [Kow79] for $\mathcal{P}$ and g. This (potentially infinite) tree contains two kinds of nodes, *OR-nodes*, which are labeled by literals, and *AND-nodes*, which are labeled by program clauses[1]. The root of the tree is an OR-node, which is labeled by g. Each OR-node in the tree has one AND-node child for each program clause, a variant (renamed version) of which is used to label the child. Each AND-node in the tree, labeled by a clause $h \leftarrow g_1, \ldots, g_k$, has k OR-node children, which are labeled by variants of $g_1, \ldots, g_k$, respectively. Variants are selected so that the labels of distinct nodes in the tree have no variables in common. With each edge in the AND/OR-tree between a parent node m and a child node n we associate an *edge equation*, defined as follows: If m is an AND-node labeled by a clause $h \leftarrow g_1, \ldots, g_k$ and n is the ith OR-node child of m, labeled by a variant g of g_i, then the edge equation is $g_i = g$. If m is an OR-node labeled by a literal g, and n is the AND-node child of m labeled by a clause $h \leftarrow g_1, \ldots, g_k$, then the edge equation is $g = h$.

It is a consequence of the completeness of the resolution inference rule for first-order logic in clausal form, that if an instance g' of a goal g is a logical consequence of the clauses of program $\mathcal{P}$, then there exists a proof of g' from $\mathcal{P}$ whose representation in tree form is isomorphic to a subgraph of the AND/OR-tree for $\mathcal{P}$ and g. The subgraphs that correspond to proof trees are those finite subgraphs of the AND/OR-tree that are connected, *conjunctive* in the sense that each OR-node in the subgraph has exactly one child in the subgraph, and *maximal* in the sense that they are not properly contained in any other connected, conjunctive subgraph. We use the term *neighborhood* to refer to a connected subgraph of an AND/OR-tree, and we call maximal conjunctive neighborhoods *prime*. A prime neighborhood determines a proof tree once we have specified a substitution σ that *solves* the neighborhood, which means that it solves (unifies) the equation associated with each of the edges in the neighborhood.

The task of an interpreter presented with a logic program $\mathcal{P}$ and goal g can therefore be thought of as a search for substitutions that solve finite prime neighborhoods of the AND/OR-tree for $\mathcal{P}$ and g. Sequential interpreters for logic programs use a deterministic strategy to enumerate and

[1] Authors differ on which nodes are called "AND-nodes," and which are called "OR-nodes." Our usage agrees with [CK85] and is the opposite of [LM86].

solve finite prime neighborhoods. For example, the Prolog interpreter traverses the AND/OR-tree in a depth-first fashion. At each OR-node, one child is chosen for search, and the choice is recorded on stack. No choice is made at AND-nodes; rather, all children are searched in sequence. The choice information on the stack at any time determines a finite, conjunctive neighborhood of the AND/OR-tree. Also computed by the interpreter is a substitution that solves this neighborhood. This substitution is updated incrementally as the neighborhood is extended by the search procedure.

Extension of the current conjunctive neighborhood can be interrupted in two ways: either the neighborhood is maximal, or else the addition of a new edge to the neighborhood results in a neighborhood having no solution. In the first case, the search is said to *succeed*, the neighborhood that has been searched is prime, and the substitution that has been computed is output as an answer. In the second case, the search is said to *fail*, and no answer is produced. In either case, the interpreter then backtracks to the most recent choice point (if any) on the stack, and restarts the search with the next untried choice. Such an interpreter, though sound, is obviously not complete since the presence of infinite branches in the AND/OR-tree can prevent some prime neighborhoods from being discovered.

The AND/OR-tree paradigm also suggests methods for exploiting parallelism in the execution of logic programs. For example, a Prolog-like interpreter may search all children of an OR-node in parallel, rather than choosing one and recording the choice for later backtracking. This kind of parallelism, called *OR-parallelism*, is particularly easy to exploit, since the search of each child of an OR-node can be carried out completely independently of the others, and the answers produced by all of them simply collected together. In *AND-parallelism*, the search of the children of an AND-node is carried out in parallel. This kind of parallelism is more difficult to exploit, since the presence of variables shared between the literals in the body of a clause implies that the answers produced by the searches of each of the children cannot merely be collected together, but must be merged in such a way that the semantics of shared variables is respected. The merging can be done either after the answers are produced, or it can be done during the search by having the child processes communicate with each other. The latter approach is attractive, since information provided by one child of an AND-node can be used to prune the search space of another, resulting in increased efficiency.

In this paper, we consider a model for parallel execution of logic programs in which a process is assigned to each node of the AND/OR-tree. Each process in our model executes asynchronously and independently of the others; thus the model supports unrestricted AND- and OR-parallelism. Processes obey a simple program, in which they repeatedly transmit their current state to their neighbors in the tree, and incorporate state information received from neighbors into their own state. Processes communicate directly only with their neighbors in the tree. From time to time, answer substitutions are output by the root process.

We consider the question of whether there exists a sound and complete interpreter for logic programs, based on this model, in which each node process maintains in its state a certain set of substitutions. Roughly, we would like this set of substitutions to represent an approximation to the set of answer substitutions for the subtree rooted at that node, based on information collected from other processes so far during execution. We would like processes to retain only substitutions that are in some sense relevant to the computation of answer substitutions at the root. Information arriving during execution could then be used to prune irrelevant substitutions. Also, we would like the substitutions maintained by a process to include only information about the bindings of variables that are "local" to the corresponding node, in the sense that they occur in the literal or clause labeling that node.

We show that such an interpreter does exist. Our interpreter has the following characteristics:

- The interpreter is fully distributed and asynchronous. No restriction is placed on the pattern of communication between nodes, except that a minimal "bottom-up fairness" assumption must be satisfied.

- The interpreter supports both AND- and OR-parallelism. In particular, information arriving from one child of an AND-node can propagate to another child, causing pruning of the search space for the latter. This bidirectional communication between AND-parallel processes is a feature not present in previous work.

- State information at a node is completely local to that node. In particular, no system-wide unique variable names are required.

- The interpreter is provably sound and complete.

The soundness and completeness proof, which is nontrivial, involves establishing an invariant relationship, between the state of a given node at any point during execution, and the "neighborhood" of all nodes from which the given node has received information so far. The proof hinges on an interesting property of information flow in the AND/OR-tree, which says that if m and n are adjacent nodes in the tree, then m always "knows more" than n about the state of the tree in the direction away from n.

In the form presented here, our interpreter is not suitable for direct implementation, since we ignore a number of issues of practical importance. For example, we assume that all processes (potentially infinitely many of them) exist from the start of execution, and do not consider how a practical interpreter would construct the AND/OR tree and allocate processes to its nodes. We also make no attempt to minimize communication costs between processes or to show how a process can efficiently update its state with information received in a message. Nevertheless, we feel the interpreter has great potential for the incorporation of various performance-improving optimizations and heuristics, which would ultimately lead to a practical algorithm. We are currently investigating possibilities here, some of which are sketched at the end of the paper.

Irrespective of the practical potential of our interpreter, we feel that its derivation and correctness proof represents an interesting exercise in distributing an algorithm over a number of processors. It should be pointed out that we did not originally discover the algorithm in the final form presented here, and subsequently prove its correctness. Rather, we began by looking for algorithms of the same general form, and then were guided to the final algorithm by solving problems that arose in the correctness proof. Although the idea underlying the the algorithm is not particularly difficult, to see why it works, and to deal correctly with the various special cases that arise, demands the construction of a careful correctness argument.

1.1 Informal Description of the Interpreter

To motivate the subsequent formal development, we give here a slightly more detailed description of our interpreter.

As mentioned above, our interpreter assigns a process to each node of the AND/OR tree. Each process maintains in its state a finite set of substitutions, which we call *alternatives*. At any point during execution, the set of alternatives for node n represents an approximation, based on the information received so far at node n, to a set of answer substitutions for the subtree rooted at n. Not all substitutions that are answers for the subtree rooted at n are necessarily covered by the set of alternatives. Rather, only those that are relevant to the computation of answers at the root need be retained. Processes refine their approximations by exchanging state information with their neighbors in the tree. Eventually, "exact" approximations accumulate at the root, where they are output.

Various complications arise when one attempts to elaborate the above ideas into an algorithm. First, we find it necessary to maintain slightly different state information at OR-nodes than at AND-nodes. The state of an AND-node contains a finite set of alternatives as described above. However, the state of an OR-node contains not just a single set of alternatives, but rather a vector of such sets, with one component for each of its children in the tree. Second, we must make sure that the state of each node contains information about which alternatives represent exact approximations. Ultimately, this information would be used by the root process to determine whether an alternative can be output as an answer.

The third difficulty we face is to discover the proper "update functions" which a node uses to incorporate information about its neighbors' states into its own. We did not find the solution particularly obvious. In the end, there are four different kinds of update functions, to cover the four cases defined by whether an AND- or an OR-node is receiving state information from a parent or child. To give an idea of how the update functions work, we consider the case in which an AND-node n receives state information from its OR-node parent m. The other cases use similar ideas. Let g be the literal labeling the OR-node, and let h be the head of the clause labeling the AND-node. To compute its new set of approximate answer substitutions, the AND-node takes the vector of sets maintained by its parent OR-node, and selects out the component corresponding to itself. Call the old set of approximations for the AND-node Σ and the selected component of the state of the OR-node Σ_n. The AND-node then proceeds as follows: For each substitution σ in Σ and τ in Σ_n, renamings are applied to obtain variants σ' and τ' having no range variables in common. Then σ' is applied to g to obtain $g\sigma'$, and τ' is applied to h to obtain $h\tau'$. A most general unifier μ (if it exists) of $g\sigma'$ and $h\tau'$ is computed, and is composed with τ' to obtain a substitution $\rho = \tau'\mu$. The set of all ρ obtained in this way from substitutions in Σ and Σ_n is collected, and subsumption is applied to reduce its size. The resulting set is the new set of alternatives for the AND-node.

1.2 Related Work

We classify previous work in parallel interpreters for logic programs on the basis of the AND-parallelism each supports. In [CK85, LM86, Kal86] AND-parallelism is restricted to *independent* goals (those that do not share variables). In [Wis86, KL87] AND-goals are evaluated bottom-up without any exchange of information between the AND-parallel goals. The "committed choice" languages of [CG86, Sha83] evaluate AND-goals in parallel with bidirectional exchange of information through shared variables, but eliminate OR-parallelism by having each OR-node "commit" to only one child. There are several proposals for purely OR-parallel interpreters that evaluate AND-goals sequentially. We note that no previous interpreters feature the full OR-parallelism coupled with bidirectional communication between AND-processes, which ours supports.

The AND/OR process model of Conery [CK83, CK85], was one of the first suggested models for parallel evaluation of logic programs. In this message-passing model, the AND/OR tree is developed dynamically, and processes are assigned to each node. Each OR-node develops all its children in parallel, thus the method supports OR-parallelism. However, AND-parallelism is restricted to independent goals. Independence is ensured by dynamically maintaining a dependency graph between literals in the body of a clause, and solving literals in parallel only when they can be determined to be independent. The method of [CK85] produces only one answer at a time, with alternative answers produced on demand by backtracking. Distributed backtracking is complex and expensive, and in [LM86] it is eliminated in favor of a multiple-answer data-driven model in which solutions to literals in the body of a clause are piped between AND-parallel processes in accordance with the dependency graph. Kale [Kal87] has pointed out that both methods [CK85] and [LM86] are incomplete, the authors' claims to the contrary notwithstanding. In [Kal86], Kale has given a modified version, which is apparently complete. In [CDD85, Her86] it is shown how the expense of maintaining the dependency graphs can be reduced by compile-time preprocessing.

In [Wis86] and [KL87], AND-parallelism is supported by computing answers in a bottom-up fashion, and merging answer sets for separate conjuncts to resolve conflicting bindings for shared variables. Whereas [KL87] merges the answers incrementally, [Wis86] does not merge the answers until one of the conjuncts has produced all answers, and his method fails to be complete for this reason [Kal87]. In both methods, the bottom-up approach means that it is impossible for the answers produced by one conjunct to affect the computation of answers by a parallel conjunct.

The "committed-choice" languages of [CG86] and [Sha83] evaluate AND-goals in parallel with bidirectional exchange of information through shared variables. However, OR-parallelism is eliminated by "committing" to only one child of each OR-node. At the other extreme are the fully OR-parallel interpreters [YN84, UT83, WADK84, CH83, GTM84, LP84] which evaluate AND-goals sequentially. Other approaches are taken by Pollard [Pol81], who proposes a rather complex shared-memory AND/OR parallel model, and in [CJ87], where we find a theorem-proving-style model that does not attempt to use AND- or OR-parallelism but rather distributes work between processors based on locality of data.

1.3 Outline of the Paper

The remainder of the paper is organized as follows: In Section 2, we introduce some preliminary material, leading up to a formal definition of an "interpreter." In Section 3 we present our interpreter, which we call the "basis sets interpreter." In Section 4 we prove that the basis sets interpreter is correct. The proof is accomplished by using "simulations" to compare the basis sets interpreter to an abstract interpreter whose correctness is obvious. Finally, in Section 5, we summarize what has been achieved and point out possibilities for future work.

2 Logic Programs and Interpreters

Let a countably infinite set $\mathcal{V}$ of *variables* be fixed. If S is a first-order signature, then let $\mathcal{T}_S$, $\mathcal{A}_S$ and $\mathcal{C}_S$ denote, respectively, the set of all terms, literals (atomic formulas), and clauses constructed from variables in $\mathcal{V}$ and symbols in S. We write $\mathrm{vars}(e)$ to denote the set of all variables occurring in an expression (term, literal, or clause) e.

A *logic program* $\mathcal{P}$ is a pair (S, C), where S is a first-order signature, and $C \subseteq \mathcal{C}_S$ is a nonempty, finite set of definite Horn clauses. We use the customary notation $h \leftarrow g_1, \ldots, g_k$ to denote a clause in C.

2.1 Substitutions

Since our results depend heavily on properties of substitutions, we require a careful development of the pertinent definitions. We shall always speak of substitutions under the assumption that a first-order signature S has been specified in advance.

A *substitution* is a function $\sigma : V \to \mathcal{T}_S$, where V is a subset of $\mathcal{V}$, called the *domain* of σ and denoted $\mathrm{dom}(\sigma)$. The set $\{v \in \mathrm{dom}(\sigma) : \sigma(v) \neq v\}$ is called the *support* of σ and is denoted $\mathrm{supp}(\sigma)$. The *range variables* $\mathrm{rvars}(\sigma)$ of σ is the set $\bigcup\{\mathrm{vars}(\sigma(v)) : v \in \mathrm{dom}(\sigma)\}$. We say that σ is *ground* if $\mathrm{rvars}(\sigma) = \emptyset$. Substitutions σ and σ' are called *range-disjoint* if $\mathrm{rvars}(\sigma) \cap \mathrm{rvars}(\sigma') = \emptyset$. If e is an expression, and σ is a substitution with $\mathrm{vars}(e) \subseteq \mathrm{dom}(\sigma)$, then define the *application* of σ to e to be the result of replacing each occurrence of a variable v in e by the term $\sigma(v)$, in the usual way. We write $e\sigma$ to denote the application of σ to e.

We distinguish two special kinds of substitutions, "global" and "local." A substitution σ is called *global* if $\mathrm{dom}(\sigma) = \mathcal{V}$ and *local* if $\mathrm{dom}(\sigma)$ is a finite subset of $\mathcal{V}$. If σ is a substitution, and τ is a global substitution, then the *composition* of σ and τ is the substitution $\sigma\tau$, with $\mathrm{dom}(\sigma\tau) = \mathrm{dom}(\sigma)$, such that $x(\sigma\tau) = (x\sigma)\tau$ for all $x \in \mathrm{dom}(\sigma)$. Note that $\sigma\tau$ is defined when, and only when, τ is global. In particular, the composition of arbitrary global substitutions is defined.

If σ is a substitution and $V \subseteq \mathcal{V}$, then the *restriction* of σ to V is the substitution $\sigma{\restriction}V$ such that $\mathrm{dom}(\sigma{\restriction}V) = \mathrm{dom}(\sigma) \cap V$, and $(\sigma{\restriction}V)(v) = \sigma(v)$ for all $v \in \mathrm{dom}(\sigma) \cap V$. Note that for all substitutions σ, all global substitutions ρ, and all sets of variables V, we have $(\sigma\rho){\restriction}V = (\sigma{\restriction}V)\rho$. If $\mathrm{supp}(\sigma) \cap \mathrm{supp}(\tau) = \emptyset$, then the *union* of σ and τ is the substitution $\sigma \cup \tau$ such that $\mathrm{dom}(\sigma \cup \tau) = \mathrm{dom}(\sigma) \cup \mathrm{dom}(\tau)$ and

$$v(\sigma \cup \tau) = \begin{cases} v\sigma, & \text{if } v \in \mathrm{supp}(\sigma), \\ v\tau, & \text{if } v \in \mathrm{supp}(\tau), \\ v, & \text{for all other } v \in \mathrm{dom}(\sigma) \cup \mathrm{dom}(\tau). \end{cases}$$

A *renaming* is a global substitution that is an injection from $\mathcal{V}$ to $\mathcal{V}$. A particularly important renaming is the identity global substitution id. An expression e' is a *variant* of e iff $e' = e\rho$ for some renaming ρ. Similarly, a substitution σ' is a *variant* of σ if $\sigma' = \sigma\rho$ for some renaming ρ. We shall need to obtain variants of expressions and substitutions in a standardized fashion. Therefore, let us fix a range-disjoint pair of renamings ω and ω' (such a pair exists because $\mathcal{V}$ is infinite).

A substitution σ *subsumes* a substitution τ, and we write $\sigma \precsim \tau$, if there exists a global substitution ρ such that $\tau = \sigma\rho$. The relation $\precsim$ is easily seen to be reflexive and transitive, moreover if $\sigma \precsim \tau$ and $\tau \precsim \sigma$, then τ is a variant of σ.

Expressions e and e' are are called *unifiable* if there exists a global substitution τ, called a *unifier*, such that $e\tau = e'\tau$. Similarly, substitutions σ and σ' are *unifiable* if there exists a global substitution τ such that $\sigma\tau = \sigma'\tau$. A unifier μ of a pair of expressions or substitutions is called *most general* if $\mu \precsim \tau$ for any other unifier τ of that pair of expressions or substitutions. We have the following standard result:

Proposition 2.1 (Unification Theorem) *If a pair of expressions or local substitutions has a unifier, then it has a most general unifier.*

Let mgu denote a function that maps each unifiable pair of expression or local substitutions to a most general unifier.

2.2 AND/OR Trees

Suppose $\mathcal{P} = (S, C)$ is a logic program, and g is a literal, which we call the *goal.* An *AND/OR tree* for $\mathcal{P}$ and g is a (potentially infinite) node-labeled, bipartite tree $T = (A, O, r, E, L)$, where

- $A \cup O$ is the set of *nodes*, with A the set of *AND* nodes and O the set of *OR* nodes,
- $r \in O$ is the *root* node,
- $E \subseteq (A \times O) \cup (O \times A)$ is the set of directed *edges*,
- L is the *labeling map*, which assigns a literal to each OR node, and a clause from C to each AND node,

such that the following conditions hold:

1. $L(r) = g$.
2. For each OR node o, there is exactly one child a of o for each clause $h \leftarrow g_1, \ldots, g_k$ in C, and $L(a)$ is a variant of that clause.
3. For each AND node a, if $L(a) = h \leftarrow g_1, \ldots, g_k$, then a has exactly k children, $o_1, \ldots, o_k$, and $L(o_i)$ is a variant of g_i, for $1 \leq i \leq k$.
4. If m and n are distinct nodes, then $L(m)$ and $L(n)$ have no variables in common.

It follows from these conditions that two AND/OR trees for $\mathcal{P}$ and g are variants of each other in the sense that the underlying trees are isomorphic, and corresponding node labels are variants. Since nothing we shall say depends in any way on a particular choice of variables in the node labels, we shall henceforth speak of *the* AND/OR tree for $\mathcal{P}$ and g as if it were unique.

When considering an AND/OR tree T, we write $m \prec n$ or $m = \text{parent}(n)$ if node m is the parent of node n, $m \prec^* n$ if m is an ancestor of n, and $m \wedge n$ for the least common ancestor of m and n. We write LV_n for the set of *local variables* of node n; that is, for the set $\text{vars}(L(n))$. We write LV for $\bigcup_{n \in A \cup O} \text{LV}_n$.

A set U of nodes of T is *conjunctive* if for each $m, n \in U$, either $m \wedge n$ is an AND node, or else one of m, n is an ancestor of the other. We say that nodes m, n are conjunctive if the set $\{m, n\}$ is conjunctive, and that node m is *conjunctive with a set of nodes U* if $\{m\} \cup U$ is conjunctive. A connected set U of nodes of T is called a *neighborhood* in T. If U is a neighborhood, and $n \in U$ then U is called a *neighborhood of n.* A neighborhood is *prime* if it is maximal conjunctive; that is, if it is conjunctive and is not a proper subset of any conjunctive neighborhood. Clearly, every conjunctive neighborhood (in particular, a neighborhood consisting of a single node) is contained in a prime neighborhood.

To each pair of nodes (m, n) in T, with $m \prec n$, we define the *associated edge equation* to be the pair of literals (l, l') defined as follows:

- If $m \in A$, $L(m) = h \leftarrow g_1, \ldots, g_k$, node $n \in O$ is the child of m corresponding to g_i, and $L(n) = g$, then $l = g_i$ and $l' = g$.
- If $m \in O$, $L(m) = g$, and $L(n) = h \leftarrow g_1, \ldots, g_k$, then $l = g$ and $l' = h$.

A substitution σ *solves neighborhood* U if $\mathrm{LV}_n \subseteq \mathrm{dom}(\sigma)$ for all $n \in U$ and if for all edges (m,n) in U, with associated edge equation (l, l'), we have $l\sigma = l'\sigma$. We say σ *solves edge* (m,n) if σ solves the neighborhood $\{m,n\}$. Clearly, if τ solves some finite prime neighborhood P (of r) in the AND/OR tree for $\mathcal{P}$ and g, then application of τ to P yields a proof tree for $g\tau$ from axioms C. Conversely, if an instance g' of g is provable from axioms C, then there exists a substitution τ and a finite prime neighborhood P in the AND/OR tree for $\mathcal{P}$ and g, such that τ solves P and such that $g' = g\tau$. Thus, we say that a substitution σ is an *answer* for program $\mathcal{P}$ and goal g if $\sigma = \tau \restriction \mathrm{LV}_r$, where τ solves some finite prime neighborhood P (of r) in the AND/OR tree for $\mathcal{P}$ and g.

2.3 Interpreters

In general, an "interpreter" for logic programs is an algorithm, which accepts as input a logic program $\mathcal{P}$ and a goal g, and produces as output a (possibly infinite) sequence of substitutions $\sigma_0, \sigma_1, \ldots$ with common domain LV_r. An interpreter is "sound" if each σ_i is an answer for $\mathcal{P}$ and g, and "complete" if whenever σ is a ground answer for $\mathcal{P}$ and g, then $\sigma_i \precsim \sigma$ for some $i \geq 0$.

In this paper, we are concerned with interpreters of a specific form: namely, those in which a program $\mathcal{P}$ with goal g is executed by a distributed algorithm in which one process is assigned to each node of the AND/OR tree for $\mathcal{P}$ and g. Each process executes a simple program, in which it repeatedly transmits its entire current state to its neighbors in the tree, and incorporates into its state the information it receives from its neighbors.

Formally, we define an *interpreter* to be a function $\mathcal{I}$ that takes an AND/OR tree, and annotates it with the following information:

- To each node n is assigned:
  - A set $Q_n^{\mathcal{I}}$ of *states*.
  - A distinguished element $\iota_n^{\mathcal{I}} \in Q_n^{\mathcal{I}}$, called the *initial state*.
- To the root r is assigned an *output function* $O^{\mathcal{I}}$ that maps $Q_r^{\mathcal{I}}$ to sets of substitutions with domain LV_r.
- To each directed edge (m,n), with $m \prec n$, is assigned a pair of *update functions*:

  $$\delta_{mn}^{\mathcal{I}} : Q_m^{\mathcal{I}} \times Q_n^{\mathcal{I}} \to Q_n^{\mathcal{I}}, \qquad \delta_{nm}^{\mathcal{I}} : Q_n^{\mathcal{I}} \times Q_m^{\mathcal{I}} \to Q_m^{\mathcal{I}}.$$

  The update function $\delta_{mn}^{\mathcal{I}}$ ($\delta_{nm}^{\mathcal{I}}$) is used by node n (node m) to compute its new state after receiving a message from node m (node n).

If T is an AND/OR tree and $\mathcal{I}$ is an interpreter, then $\mathcal{I}(T)$ denotes the *annotated tree* obtained by applying $\mathcal{I}$ to T.

A *global state* for an annotated tree is a vector $\bar{q} = (q_n : n \in A \cup O)$, where $q_n \in Q_n$ for all $n \in A \cup O$. We write $\bar{q}^n$ to denote the component q_n of $\bar{q}$. A *transition* is an expression of the form $\bar{q} \xrightarrow{mn} \bar{r}$, where $\bar{q}$, $\bar{r}$ are global states and m, n are adjacent nodes, such that for all $p \in A \cup O$,

$$\bar{r}^p = \begin{cases} \delta_{mn}(\bar{q}^m, \bar{q}^n), & \text{if } p = n, \\ \bar{q}^p, & \text{otherwise.} \end{cases}$$

A *schedule* for an annotated tree is an infinite sequence of ordered pairs

$$(m_0, n_0), (m_1, n_1), \ldots,$$

where for each $i \geq 0$, the nodes m_i and n_i are adjacent. A schedule is *bottom-up-fair* if it contains as a subsequence a bottom-up traversal of a neighborhood P, whenever P is finite and prime. For example, every schedule that contains infinitely often each edge (n, m) with $m \prec n$, is bottom-up-fair. Each schedule for an annotated tree determines a corresponding *computation*, which is a sequence of transitions of the form:

$$\bar{q}_0 \xrightarrow{m_0 n_0} \bar{q}_1 \xrightarrow{m_1 n_1} \ldots,$$

where $\bar{q}_0 = (\iota_n : n \in A \cup O)$ is the initial global state.

A global state $\bar{q}$ is *reachable* if it appears in some computation. A property of global states is called *invariant* if it holds for all reachable global states, and *inevitable* if it holds for some state in every computation corresponding to a bottom-up-fair schedule. A property P of global states is *inductive* if P holds of the initial global state, and whenever $\bar{q} \xrightarrow{mn} \bar{r}$, if P holds of $\bar{q}$, then P holds of $\bar{r}$.

Proposition 2.2 (Computational Induction) *If a property of global states is inductive, then it is invariant.*

An interpreter $\mathcal{I}$ is *sound* if for every AND/OR tree T corresponding to a logic program P and goal g, it is invariant for global states $\bar{q}$ of the annotated tree $\mathcal{I}$, that every element of $O(\bar{q}^r)$ is an answer for P and g. An interpreter $\mathcal{I}$ is *complete* if for every AND/OR tree T corresponding to P and g, and every ground answer σ for P and g, it is inevitable for global states $\bar{q}$ of $\mathcal{I}(T)$ that there exists $\tau \in O(\bar{q}^r)$ with $\tau \sqsubsetsim \sigma$. We say that an interpreter is *correct* if it is both sound and complete.

3 The Basis Sets Interpreter

In this section we define our interpreter, which we call the *basis sets interpreter*. Each process in the basis sets interpreter maintains in its state a finite set of substitutions, which we shall call *alternatives*. At any time during execution, the set of alternatives at node n represents an approximation, based on the information received so far at node n, to a set of answer substitutions for the subtree rooted at n. The state of each node must also contain information about which alternatives represent exact approximations. Ultimately, this information is used by the root process to determine which alternatives can be output as answers. Thus, nodes actually maintain two sets of alternatives, "upper bound" alternatives, which approximate all possible answers, and "lower bound" alternatives, which are those approximations that are known to be exact.

3.1 Preliminary Definitions

Suppose an AND/OR tree T has been given and n is a non-root node in T. We first define operations $\underset{n}{\Leftarrow}$ and $\underset{n}{\Rightarrow}$ on sets of substitutions. Suppose $m \prec n$ in T, and (l, l') is the equation associated with the edge (m, n). Suppose τ and τ' are substitutions, with $\mathrm{LV}_m \subseteq \mathrm{dom}(\tau)$ and $\mathrm{LV}_n \subseteq \mathrm{dom}(\tau')$. Recall that we had fixed a range-disjoint pair of renamings ω and ω'. If $l\tau\omega$ and $l'\tau'\omega'$ are unifiable, with $\mu = \mathrm{mgu}(l\tau\omega, l'\tau'\omega')$, then define

$$\tau \underset{n}{\Leftarrow} \tau' = (\tau\omega\mu)\restriction \mathrm{LV}_m, \qquad \tau \underset{n}{\Rightarrow} \tau' = (\tau'\omega'\mu)\restriction \mathrm{LV}_n.$$

Extend this definition to sets of substitutions Σ, Σ' as follows:

$$\begin{aligned}\Sigma \underset{n}{\Leftarrow} \Sigma' &= \{\tau \underset{n}{\Leftarrow} \tau' : \tau \in \Sigma, \tau' \in \Sigma', l\tau\omega \text{ and } l'\tau'\omega' \text{ are unifiable}\} \\ \Sigma \underset{n}{\Rightarrow} \Sigma' &= \{\tau \underset{n}{\Rightarrow} \tau' : \tau \in \Sigma, \tau' \in \Sigma', l\tau\omega \text{ and } l'\tau'\omega' \text{ are unifiable}\}.\end{aligned}$$

Next, we define operations $\bigvee$ and $\bigwedge$ on sets of substitutions. If Σ is a set of substitutions, then define the *span* $[\Sigma]$ of Σ to be the set of all τ such that $\sigma \precsim \tau$ for some $\sigma \in \Sigma$. We say that a set of substitutions Σ is *independent* if it is pairwise incomparable under $\precsim$. A *basis* for Σ is an independent subset Ξ of Σ such that $[\Xi] = [\Sigma]$. Since the strict part of $\precsim$ is well-founded on local substitutions, every set of such substitutions Σ contains a basis, and it is not difficult to see that two bases for the same set differ only up to renamings applied to their elements. Define a *join* of a collection $\{\Sigma_i : i \in I\}$ of sets of local substitutions to be a basis for the set $\bigcup_{i\in I}[\Sigma_i]$. Similarly, if I is nonempty, then define a *meet* of $\{\Sigma_i : i \in I\}$ to be a basis for the set $\bigcap_{i\in I}[\Sigma_i]$. Let $\bigvee$ be an arbitrary function that maps each collection of local substitutions $\{\Sigma_i : i \in I\}$ to a join $\bigvee_{i\in I}\Sigma_i$, and let $\bigwedge$ map each nonempty collection of local substitutions $\{\Sigma_i : i \in I\}$ to a meet $\bigwedge_{i\in I}\Sigma_i$.

Lemma 3.1 *Suppose $\{\Sigma_i : i \in I\}$ is a finite collection of sets of local substitutions, where each Σ_i has a finite basis. Then*

1. *$\bigvee_{i\in I}\Sigma_i$ is finite.*
2. *$\bigwedge_{i\in I}\Sigma_i$ is finite, if I is nonempty.*

Proof – For each $i \in I$, let Ξ_i be a finite basis for Σ_i.

(1) It suffices to show that there is a finite set Σ such that $[\Sigma] = \bigcup_{i\in I}[\Sigma_i]$. Clearly, $\Sigma = \bigcup_{i\in I}\Xi_i$ is such a set.

(2) It suffices to show that there is a finite set Σ such that $[\Sigma] = \bigcap_{i\in I}[\Sigma_i]$. We construct Σ by induction on the number of elements $|I|$ of I. If $|I| = 1$, let i be the single element of I, then clearly $\Sigma = \Xi_i$ has the required properties. If $|I| = n+1$, then $I = \{i\} \cup I'$, where I' has n elements. Let $\Sigma' = \bigwedge_{i\in I'}\Sigma_i$, which is finite by induction hypothesis. Define

$$\Sigma = \{\sigma\omega\mu : \sigma \in \Sigma', \xi \in \Xi_i, \mu = \mathrm{mgu}(\sigma\omega, \xi\omega')\}.$$

Clearly, Σ is finite. That $[\Sigma] = \bigcap_{i\in I}[\Sigma_i]$ follows easily by the properties of most general unifiers. ∎

We note in passing that the proof of Lemma 3.1 can be refined to give an algorithm for computing $\bigvee_{i\in I}\Sigma_i$ and $\bigwedge_{i\in I}\Sigma_i$, when I and each Σ_i are finite.

Finally, some vector notation will significantly shorten our definitions. If n is a node, then we write $(\bar{\Upsilon}^p : n \prec p)$, (or just $\bar{\Upsilon}$ when n is clear) to denote a vector of sets of substitutions, one for each child p of node n. The notation $\bigvee\bar{\Upsilon}$ abbreviates $\bigvee_{n\prec p}\bar{\Upsilon}^p$. The notation $\bigwedge\bar{\Upsilon}$ is used similarly, when n is not a leaf. The notation $\bar{\Upsilon}\{p/\Sigma\}$ stands for the vector $\bar{\Upsilon}$ with its pth component replaced by Σ. We write $\Sigma \underset{n}{\Rightarrow} \bar{\Upsilon}$ as an abbreviation for $(\Sigma \underset{n}{\Rightarrow} \bar{\Upsilon}^p : n \prec p)$, and similarly for $\underset{n}{\Leftarrow}$.

3.2 The Interpreter

For the *basis sets interpreter* $\mathcal{B}$,

- The state sets are defined as follows:

1. If n is an AND-node, then the set $Q^{\mathcal{B}}_n$ consists of all pairs $(\Upsilon, \bar{\Lambda})$, where Υ is a finite set of substitutions with domain LV_n, and $\bar{\Lambda} = (\bar{\Lambda}^p : n \prec p)$ is a vector of finite sets of substitutions with domain LV_n.
2. If n is an OR-node, then the set $Q^{\mathcal{B}}_n$ consists of all pairs $(\bar{\Upsilon}, \bar{\Lambda})$, where $\bar{\Upsilon} = (\bar{\Upsilon}^p : n \prec p)$ and $\bar{\Lambda} = (\bar{\Lambda}^p : n \prec p)$ are vectors of finite sets of substitutions with domain LV_n.

The first (*i.e.* Υ) component of the state of a node represents the "upper bound" information, and the second (*i.e.* Λ) component represents "lower bound" information.

- The initial states are defined as follows:

1. If n is an AND-node, then $\iota^{\mathcal{B}}_n = (\{\mathrm{id}\restriction \mathrm{LV}_n\}, (\{\} : n \prec p))$.
2. If n is an OR-node, then $\iota^{\mathcal{B}}_n = ((\{\mathrm{id}\restriction \mathrm{LV}_n\} : n \prec p), (\{\} : n \prec p))$.

The symbol "id" denotes the identity global substitution.

- The output function $O^{\mathcal{B}}$ maps a state $(\bar{\Upsilon}, \bar{\Lambda}) \in Q^{\mathcal{B}}_r$ to the set $\bigvee \bar{\Lambda}$.
- The update functions for edge (m, n), with $m \prec n$, are defined as follows:

1. If m is an AND-node and n is an OR-node,

$$\begin{aligned}
\delta^{\mathcal{B}}_{mn}((\Upsilon_m, \bar{\Lambda}_m), (\bar{\Upsilon}_n, \bar{\Lambda}_n)) &= (\Upsilon_m \underset{n}{\Rrightarrow} \bar{\Upsilon}_n,\ \Upsilon_m \underset{n}{\Rrightarrow} \bar{\Lambda}_n) \\
\delta^{\mathcal{B}}_{nm}((\bar{\Upsilon}_n, \bar{\Lambda}_n), (\Upsilon_m, \bar{\Lambda}_m)) &= (\Upsilon_m \underset{n}{\Lleftarrow} \bigvee \bar{\Upsilon}_n,\ (\bar{\Lambda}_m \underset{n}{\Lleftarrow} \bigvee \bar{\Upsilon}_n)\{n/\Upsilon_m \underset{n}{\Lleftarrow} \bigvee \bar{\Lambda}_n\}).
\end{aligned}$$

2. If m is an OR-node and n is an AND-node,

$$\delta^{\mathcal{B}}_{mn}((\bar{\Upsilon}_m, \bar{\Lambda}_m), (\Upsilon_n, \bar{\Lambda}_n)) = (\bar{\Upsilon}^n_m \underset{n}{\Rrightarrow} \Upsilon_n, \bar{\Upsilon}^n_m \underset{n}{\Rrightarrow} \bar{\Lambda}_n)$$

$$\begin{aligned}
&\delta^{\mathcal{B}}_{nm}((\Upsilon_n, \bar{\Lambda}_n), (\bar{\Upsilon}_m, \bar{\Lambda}_m)) \\
&\qquad = \begin{cases} (\bar{\Upsilon}_m\{n/\bar{\Upsilon}^n_m \underset{n}{\Lleftarrow} \Upsilon_n\},\ \bar{\Lambda}_m\{n/\bar{\Upsilon}^n_m \underset{n}{\Lleftarrow} \bigwedge \bar{\Lambda}_n\}), & \text{if } n \text{ is not a leaf,} \\ (\bar{\Upsilon}_m\{n/\bar{\Upsilon}^n_m \underset{n}{\Lleftarrow} \Upsilon_n\},\ \bar{\Lambda}_m\{n/\bar{\Upsilon}^n_m \underset{n}{\Lleftarrow} \Upsilon_n\}), & \text{if } n \text{ is a leaf.} \end{cases}
\end{aligned}$$

4 Correctness Proof

In this section, we prove the correctness of the basis sets interpreter. The technique we use is to relate this interpreter to a known correct interpreter through the use of "simulations," which are mappings between interpreters that preserve and reflect correctness. To get started with this proof technique, we introduce the "neighborhoods interpreter," in which the state of a process at any time is simply the neighborhood in the tree from which information has been received so far. Although the neighborhoods interpreter would not be useful in practice, it has the advantage of being obviously correct. Next, we refine this interpreter to obtain the "solution sets interpreter," in which each process maintains sets of substitutions called "solution sets." At any time during execution, the solution set maintained by node n is the set of all substitutions that are restrictions, to the local variables for node n, of solutions to intersections of prime neighborhoods with the neighborhood from which information has been received so far. We prove the correctness of the solution sets interpreter by exhibiting a simulation to it from the neighborhoods interpreter.

The solution sets interpreter has the same form as the basis sets interpreter, except that solution sets are infinite, whereas basis sets are finite. In fact, at any time during execution, the infinite solution set maintained by a process in the solution sets interpreter is the set of all instances of substitutions in the set maintained by the corresponding process in the basis sets interpreter. We use this fact to exhibit a simulation from the basis sets interpreter to the solution sets interpreter. Since simulations preserve and reflect correctness, the correctness of the basis sets interpreter follows.

4.1 Simulations

Suppose $\mathcal{I}$ and $\mathcal{J}$ are interpreters. If $T = (A, O, r, E, L)$ is an AND/OR tree, then a *simulation* from $\mathcal{I}(T)$ to $\mathcal{J}(T)$ is a pair $(R, \mathcal{F})$, where R is a property of global states of $\mathcal{I}(T)$ and $\mathcal{F} = \{\mathcal{F}_n : n \in A \cup O\}$ is a collection of functions with $\mathcal{F}_n : Q_n^{\mathcal{I}} \to Q_n^{\mathcal{J}}$, such that

1. R is inductive for $\mathcal{I}(T)$.
2. $\mathcal{F}(\iota_n^{\mathcal{I}}) = \iota_n^{\mathcal{J}}$ for all $n \in A \cup O$.
3. $\mathcal{F}_n(\delta_{mn}^{\mathcal{I}}(\bar{q}^m, \bar{q}^n)) = \delta_{mn}^{\mathcal{J}}(\mathcal{F}_m(\bar{q}^m), \mathcal{F}_n(\bar{q}^n))$ for all adjacent nodes $m, n \in A \cup O$ and all global states $\bar{q}$ for $\mathcal{I}(T)$ that satisfy R.
4. $O^{\mathcal{J}}(\mathcal{F}_r(\bar{q}^r)) = O^{\mathcal{I}}(\bar{q}^r)$ for all global states $\bar{q}$ for $\mathcal{I}(T)$ that satisfy R.

Lemma 4.1 *Suppose $\mathcal{I}$ and $\mathcal{J}$ are interpreters, such that for all AND/OR trees T, there exists a simulation from $\mathcal{I}(T)$ to $\mathcal{J}(T)$. Then $\mathcal{I}$ is correct iff $\mathcal{J}$ is correct.*

Proof – The conditions in the definition of a simulation imply that, if a computation of $\mathcal{I}(T)$ and a computation of $\mathcal{J}(T)$ are determined by the same schedule, then each state in the computation of $\mathcal{J}(T)$ is the image under the simulation mapping of the corresponding state in the computation of $\mathcal{I}(T)$. Moreover, the output maps of $\mathcal{I}(T)$ and $\mathcal{J}(T)$ yield identical values on corresponding states. Since correctness is a property only of the set of output sequences of bottom-up-fair computations, it follows that $\mathcal{I}$ is correct iff $\mathcal{J}$ is correct. ∎

4.2 The Neighborhoods Interpreter

The *neighborhoods interpreter* $\mathcal{N}$ is defined as follows: Given an AND/OR tree T,

- For each node n,
  - The state set $Q_n^{\mathcal{N}}$ is the set of all finite neighborhoods of n.
  - The initial state $\iota_n^{\mathcal{N}}$ is the singleton neighborhood $\{n\}$.
- The output map $O^{\mathcal{N}}$ takes a finite neighborhood U of r to the set of all $\tau \restriction \mathrm{LV}_r$, where τ solves some prime neighborhood of r contained in U.
- For all edges (m, n), with $m \prec n$, and all states $q_m \in Q_m^{\mathcal{N}}$, $q_n \in Q_n^{\mathcal{N}}$,

$$\delta_{mn}(q_m, q_n) = \delta_{nm}(q_n, q_m) = q_m \cup q_n.$$

It is easily checked that if m and n are adjacent nodes, q_m is a finite neighborhood of m, and q_n is a finite neighborhood of n, then $q_m \cup q_n$ is a finite neighborhood of both m and n; thus the update functions are well-defined.

Theorem 1 *The neighborhoods interpreter $\mathcal{N}$ is correct.*

Proof – Soundness is a direct consequence of the definition of the output function. To show completeness, suppose σ is an answer for T. Then $\sigma = \tau \restriction \mathrm{LV}_r$, where τ solves some finite prime neighborhood P of the root. By definition, every bottom-up-fair schedule for T embeds a bottom-up traversal of P, and in any state subsequent to the completion of such a traversal, the set P is a subset of the state of the root. Thus, it is inevitably the case that σ is an output. ▮

We now present an important property of information flow between nodes, upon which all our subsequent results depend. Intuitively, this property says that if m and n are adjacent nodes in the tree, then m always "knows more" than n about the state of the tree in the direction towards m. To state this formally, we need some additional notation.

If U is a neighborhood, and n is a node, then define

$$U \downarrow n = \{m \in U : m \wedge n = n\}, \qquad U \uparrow n = \{m \in U : m \wedge n \neq n\}$$

The expressions $U \downarrow n$ and $U \uparrow n$ are read as "U below n," and "U above n," respectively.

If n is a node, and U, U' are neighborhoods, then define $\mathrm{INC}_n(U, U')$ to be true exactly when n is not the root node, U is a finite neighborhood of parent(n), U' is a finite neighborhood of n, and the following inclusions hold:

$$U \uparrow n \supseteq U' \uparrow n, \qquad U \downarrow n \subseteq U' \downarrow n.$$

Lemma 4.2 *Suppose T is an AND/OR tree. Let* INC *be the property of global states $\bar{q}$ of $\mathcal{N}(T)$ which is true iff* $\mathrm{INC}_n(\bar{q}^m, \bar{q}^n)$ *holds for all nodes m, n with $m \prec n$. Then* INC *is inductive.*

Proof – Straightforward. ▮

4.3 The Solution Sets Interpreter

In this section, we refine the neighborhoods interpreter to obtain the solution sets interpreter. This is done in several steps: First, we define functions primes$_n$ and cprimes$_n$ that map a neighborhood U of node n to certain "relatively prime" sub-neighborhoods of U. We then establish some homomorphic properties of these functions with respect to the operation $\cup$ on sets, used in the definition of the update functions for the neighborhoods interpreter. In particular, we show that under conditions that are invariant for the neighborhoods interpreter, the maps primes$_n$ and cprimes$_n$ translate $\cup$ to an operation $\underset{n}{\bowtie}$ on neighborhoods.

Second, we define a map solns, which takes a set of neighborhoods to the set of all substitutions that solve one of its elements. We show that, under certain conditions, the map solns is homomorphic with respect to $\underset{n}{\bowtie}$ and that it translates this operation to a related operation, which we also denote by $\underset{n}{\bowtie}$, on sets of substitutions.

Third, we show that restriction of substitutions to variables local to a node is homomorphic with respect to $\underset{n}{\bowtie}$ on sets of substitutions, and that restriction translates $\underset{n}{\bowtie}$ into one of two operations, which we denote $\underset{n}{\leftarrow}$ and $\underset{n}{\rightarrow}$, on sets of substitutions. Finally, we combine all our results and obtain both the definition of the solution sets interpreter and a proof of its correctness.

4.3.1 Prime Neighborhoods

Suppose n is a node. For neighborhoods V and V', and sets of neighborhoods Γ and Γ', define

$$V \underset{n}{\bowtie} V' = (V \uparrow n) \cup (V' \downarrow n), \qquad \Gamma \underset{n}{\bowtie} \Gamma' = \{V \underset{n}{\bowtie} V' : V \in \Gamma, V' \in \Gamma'\}.$$

The importance of $\underset{n}{\bowtie}$ is established by the following result.

Lemma 4.3 *Suppose n is a node, and P, P' are prime neighborhoods of n. Then $P \underset{n}{\bowtie} P'$ is also a prime neighborhood of n.*

Proof – To show that $P \underset{n}{\bowtie} P'$ is conjunctive, suppose $m, m' \in P \underset{n}{\bowtie} P'$. Then either both m and m' are in $P \uparrow n$, or both m and m' are in $P' \downarrow n$, or one of m, m' is in $P \uparrow n$ and the other is in $P' \downarrow n$. In the first two cases, m and m' are conjunctive because P and P' are conjunctive. For the third case, suppose without loss of generality that $m \in P \uparrow n$ and $m' \in P' \downarrow n$. Then $m \wedge m' = m \wedge n$, which is either m or an AND node, by the fact that P is conjunctive and $n \in P$. Finally, suppose $P \underset{n}{\bowtie} P'$ were not prime. There there would exist m, conjunctive with $P \underset{n}{\bowtie} P'$, but not in $P \underset{n}{\bowtie} P'$. There are two cases, either $m \wedge n = n$ or $m \wedge n \neq n$. If $m \wedge n = n$, then m would be conjunctive with P' but not in P', a contradiction with the fact that P' is prime. Similarly, if $m \wedge n \neq n$, then m would be conjunctive with P but not in P, contradicting the fact that P is prime. ∎

Lemma 4.4 *Suppose* $\mathrm{INC}_n(U, U')$. *Then for all neighborhoods V, V'*

$$(V \cap U) \underset{n}{\bowtie} (V' \cap U') = (V \underset{n}{\bowtie} V') \cap (U \cup U').$$

Proof –

$$\begin{aligned}
(V \underset{n}{\bowtie} V') \cap (U \cup U') &= (V \uparrow n \cup V' \downarrow n) \cap (U \cup U') \\
&= (V \cap U) \uparrow n \cup (V \cap U') \uparrow n \cup (V' \cap U) \downarrow n \cup (V' \cap U') \downarrow n \\
&= (V \cap U) \uparrow n \cup (V' \cap U') \downarrow n \\
&= (V \cap U) \underset{n}{\bowtie} (V' \cap U').
\end{aligned}$$

where the third equality uses the hypothesized inclusions. ∎

Suppose n is a node and U is a finite neighborhood. Define $\mathrm{primes}_n(U)$, the set of *prime neighborhoods of n, relative to U*, by

$$\mathrm{primes}_n(U) = \{P \cap U : P \text{ prime}, n \in P\}.$$

Define $\mathrm{cprimes}_n(U)$, the set of *complete* prime neighborhoods of n, relative to U, by

$$\mathrm{cprimes}_n(U) = \{P \cap U : P \text{ prime}, n \in P, P \downarrow n \subseteq U\}.$$

Intuitively, if the process for node n has heard from nodes in a neighborhood U, then $\mathrm{primes}_n(U)$ represents what node n knows about the prime neighborhoods in the AND/OR tree, and $\mathrm{cprimes}_n(U)$ consists of those elements of $\mathrm{primes}_n(U)$, which are "complete" in the sense that they correspond to proof trees rooted at n.

Lemma 4.5 *Suppose n is a node and U is a finite neighborhood of n.*

1. *If n is an AND-node, then*

   (a) $\mathrm{primes}_n(U) = \mathrm{primes}_p(U)$, *if $n \prec p$.*

   (b) $\mathrm{cprimes}_n(U) = \begin{cases} \bigcap_{n \prec p} \mathrm{cprimes}_p(U), & \text{if } n \text{ is not a leaf,} \\ \mathrm{primes}_n(U), & \text{if } n \text{ is a leaf.} \end{cases}$

2. *If n is an OR-node, then*

   (a) $\mathrm{primes}_n(U) = \bigcup_{n \prec p} \mathrm{primes}_p(U)$.

   (b) $\mathrm{cprimes}_n(U) = \bigcup_{n \prec p} \mathrm{cprimes}_p(U)$.

Proof – Obvious. ∎

Lemma 4.6 *Suppose* $\mathrm{INC}_n(U, U')$ *and $n \prec^* p$. Then for all neighborhoods $V \subseteq U \cup U'$, the following are equivalent:*

1. $V \in \mathrm{primes}_p(U \cup U')$.

2. $V \cap U \in \mathrm{primes}_n(U)$ *and* $V \cap U' \in \mathrm{primes}_p(U')$.

3. $V = W \underset{n}{\bowtie} W'$ *for some* $W \in \mathrm{primes}_n(U)$ *and* $W' \in \mathrm{primes}_p(U')$.

Proof – (1) implies (2): Suppose $V \in \mathrm{primes}_p(U \cup U')$. Then there exists a prime neighborhood P of p, such that $V = P \cap (U \cup U')$. Since $n \prec^* p$, we know that P is also a prime neighborhood of n. Then $V \cap U = P \cap U$, and $V \cap U' = P \cap U'$, hence $V \cap U \in \mathrm{primes}_n(U)$ and $V \cap U' \in \mathrm{primes}_p(U')$.

(2) implies (3): Suppose $V \cap U \in \mathrm{primes}_n(U)$ and $V \cap U' \in \mathrm{primes}_p(U')$. Let $W = V \cap U$ and $W' = V \cap U'$. Then $W \underset{n}{\bowtie} W' = ((V \cap U) \uparrow n) \cup ((V \cap U') \downarrow n)$, which by the hypothesized inclusions is just $(V \cap U) \cup (V \cap U') = V$.

(3) implies (1): Suppose $W \in \mathrm{primes}_n(U)$ and $W' \in \mathrm{primes}_p(U')$ are such that $V = W \underset{n}{\bowtie} W'$. Then there exist a prime neighborhood P of n and a prime neighborhood P' of p such that $W = P \cap U$ and $W' = P' \cap U'$. Let $R = P \underset{n}{\bowtie} P'$, then clearly we have $p \in R$. By Lemma 4.4, $R \cap (U \cup U') = V$. By Lemma 4.3, R is prime, thus showing that $V \in \mathrm{primes}_p(U \cup U')$. ∎

Lemma 4.7 *Suppose* $\mathrm{INC}_n(U, U')$ *and $n \prec^* p$. Then for all neighborhoods $V \subseteq U \cup U'$, the following are equivalent:*

1. $V \in \mathrm{cprimes}_p(U \cup U')$.

2. $V \cap U \in \mathrm{primes}_n(U)$ *and* $V \cap U' \in \mathrm{cprimes}_p(U')$.

3. $V = W \underset{n}{\bowtie} W'$ *for some* $W \in \mathrm{primes}_n(U)$ *and* $W' \in \mathrm{cprimes}_p(U')$.

Proof – Similar to Lemma 4.6. ∎

Lemma 4.8 *Suppose n is an OR-node and* $\mathrm{INC}_n(U, U')$ *holds. If $m \prec n$ and $m \prec l \neq n$, then the following are equivalent :*

1. $V \in \text{cprimes}_l(U \cup U')$.

2. $V \cap U \in \text{cprimes}_l(U)$ *and* $V \cap U' \in \text{primes}_n(U')$.

3. $V = W \underset{n}{\bowtie} W'$ *for some* $W \in \text{cprimes}_l(U)$ *and* $W' \in \text{primes}_n(U')$.

Proof – Similar to Lemma 4.6. ∎

Lemma 4.9 *Suppose* $\text{INC}_n(U, U')$, $m \prec n \prec^* p$ *and* $m \prec l \neq n$. *Then*

1. $\text{primes}_p(U \cup U') = \text{primes}_n(U) \underset{n}{\bowtie} \text{primes}_p(U')$.

2. $\text{cprimes}_p(U \cup U') = \text{primes}_n(U) \underset{n}{\bowtie} \text{cprimes}_p(U')$.

3. $\text{cprimes}_l(U \cup U') = \text{cprimes}_l(U) \underset{n}{\bowtie} \text{primes}_n(U')$, *if* n *is an OR-node.*

Proof – Immediate from Lemma 4.6, 4.7, 4.8. ∎

The following is the main technical lemma upon which our correctness proof is based. It shows how information about prime sets relative to $U \cup U'$ can be characterized in terms of information about prime sets relative to U and U' separately.

Lemma 4.10 *Suppose* $\text{INC}_n(U, U')$ *and* $m \prec n$.

1. *If* m *is an AND-node and* n *is an OR-node, then*

   (a) *For all* p *with* $n \prec p$,

      i. $\text{primes}_p(U \cup U') = \text{primes}_m(U) \underset{n}{\bowtie} \text{primes}_p(U')$.

      ii. $\text{cprimes}_p(U \cup U') = \text{primes}_m(U) \underset{n}{\bowtie} \text{cprimes}_p(U')$.

   (b) *For all* l *with* $m \prec l$,

      i. $\text{primes}_m(U \cup U') = \text{primes}_m(U) \underset{n}{\bowtie} \bigcup_{n \prec p} \text{primes}_p(U')$.

      ii. $$\text{cprimes}_l(U \cup U') = \begin{cases} \text{primes}_m(U) \underset{n}{\bowtie} \bigcup_{n \prec p} \text{cprimes}_p(U'), & \text{if } l = n, \\ \text{cprimes}_l(U) \underset{n}{\bowtie} \bigcup_{n \prec p} \text{primes}_p(U'), & \text{if } l \neq n. \end{cases}$$

2. *If* m *is an OR-node and* n *is an AND-node, then*

   (a) *For all* p *with* $n \prec p$,

      i. $\text{primes}_n(U \cup U') = \text{primes}_n(U) \underset{n}{\bowtie} \text{primes}_n(U')$.

      ii. $\text{cprimes}_p(U \cup U') = \text{primes}_n(U) \underset{n}{\bowtie} \text{cprimes}_p(U')$.

   (b) *For all* l *with* $m \prec l$,

      i. $$\text{primes}_l(U \cup U') = \begin{cases} \text{primes}_n(U) \underset{n}{\bowtie} \text{primes}_n(U'), & \text{if } l = n, \\ \text{primes}_l(U), & \text{if } l \neq n. \end{cases}$$

      ii. $$\text{cprimes}_l(U \cup U') = \begin{cases} \text{primes}_n(U) \underset{n}{\bowtie} \bigcap_{n \prec p} \text{cprimes}_p(U'), & \text{if } l = n, n \text{ not a leaf}, \\ \text{primes}_n(U) \underset{n}{\bowtie} \text{primes}_n(U'), & \text{if } l = n, n \text{ a leaf}, \\ \text{cprimes}_l(U), & \text{if } l \neq n. \end{cases}$$

Proof – All cases except (2bi) and (2bii) for $l \neq n$ are proved by straightforward applications of Lemma 4.9, and applying Lemma 4.5 to make use of the assumptions about m and n. We now argue the remaining cases.

(2bi) If $l \neq n$, then $\mathrm{primes}_l(U \cup U') = \mathrm{primes}_l(U)$ holds because the fact that m is an OR node and $l \neq n$ means that if P is a prime neighborhood of l, then $P \downarrow n = \emptyset$, hence $P \cap (U \cup U') = P \cap U$ by $\mathrm{INC}_n(U, U')$.

(2bii) Same argument as (2bi). ∎

Suppose Γ and Γ' are sets of neighborhoods. We say that $\Gamma \underset{n}{\bowtie} \Gamma'$ is *union-representable* if each $V \in \Gamma \underset{n}{\bowtie} \Gamma'$ is $W \cup W'$ for some $W \in \Gamma$ and $W' \in \Gamma'$. The next technical property is required as a hypothesis by Lemma 4.12.

Lemma 4.11 *Suppose* $\mathrm{INC}_n(U, U')$. *Then each set defined by* $\underset{n}{\bowtie}$ *in Lemma 4.10 is union-representable.*

Proof – It suffices to show that the following three sets are union-representable. Then each of the sets defined by $\underset{n}{\bowtie}$ in Lemma 4.10 can be shown to be union-representable by making use of Lemma 4.5.

If $m \prec n \prec^* p$ and $m \prec l \neq n$. Then the following sets are union-representable :

1. $\mathrm{primes}_n(U) \underset{n}{\bowtie} \mathrm{primes}_p(U')$.
2. $\mathrm{primes}_n(U) \underset{n}{\bowtie} \mathrm{cprimes}_p(U')$.
3. $\mathrm{cprimes}_l(U) \underset{n}{\bowtie} \mathrm{primes}_n(U')$, if n is an OR-node.

The proof of (1), (2), and (3) are immediate from Lemma 4.6, 4.7, and 4.8 respectively. ∎

4.3.2 Solution Sets

If U is a neighborhood and n is a node, then the *solution set* $\mathrm{solns}(U)$ of U is the set of all substitutions σ, with $\mathrm{dom}(\sigma) = \mathrm{LV}$, such that σ solves U. We extend this definition to sets Γ of neighborhoods by: $\mathrm{solns}(\Gamma) = \bigcup\{\mathrm{solns}(U) : U \in \Gamma\}$.

If σ is a substitution and n is a node, then define

$$\sigma \uparrow n = \sigma \restriction (\mathrm{LV} \uparrow n), \qquad \sigma \downarrow n = \sigma \restriction (\mathrm{LV} \downarrow n)$$

where

$$\mathrm{LV} \uparrow n = \bigcup\{\mathrm{LV}_m : m \in (A \cup O) \uparrow n\}, \qquad \mathrm{LV} \downarrow n = \bigcup\{\mathrm{LV}_m : m \in (A \cup O) \downarrow n\}.$$

Suppose $m \prec n$, and (l, l') is the equation associated with the edge (m, n). Suppose τ and τ' are substitutions, with $\mathrm{LV}_m \subseteq \mathrm{dom}(\tau)$ and $\mathrm{LV}_n \subseteq \mathrm{dom}(\tau')$. If ρ is a unifier of $l\tau$ and $l'\tau'$, then define

$$\tau \overset{\rho}{\underset{n}{\bowtie}} \tau' = (\tau\rho \uparrow n) \cup (\tau'\rho \downarrow n).$$

If n is a node and Σ, Σ' are sets of substitutions, such that $\mathrm{LV}_m \subseteq \mathrm{dom}(\tau)$ for each $\tau \in \Sigma$ and $\mathrm{LV}_n \subseteq \mathrm{dom}(\tau')$ for each $\tau' \in \Sigma'$, then define

$$\Sigma \underset{n}{\bowtie} \Sigma' = \{\tau \overset{\rho}{\underset{n}{\bowtie}} \tau' : \tau \in \Sigma, \tau' \in \Sigma', \rho \text{ is a unifier of } l\tau, l'\tau'\}.$$

Lemma 4.12 *Suppose $m \prec n$, Γ is a set of neighborhoods of m, and Γ' is a set of neighborhoods of n, such that $\Gamma \underset{n}{\bowtie} \Gamma'$ is union-representable. Then*

$$\mathrm{solns}(\Gamma \underset{n}{\bowtie} \Gamma') = \mathrm{solns}(\Gamma) \underset{n}{\bowtie} \mathrm{solns}(\Gamma').$$

Proof – Suppose $\sigma \in \mathrm{solns}(\Gamma \underset{n}{\bowtie} \Gamma')$. Then σ solves some $V \in \Gamma \underset{n}{\bowtie} \Gamma'$, so by hypothesis there exist $W \in \Gamma$ and $W' \in \Gamma'$ with $V = W \cup W'$. Now, if σ solves V, then it solves W, W' and edge (m, n). Thus, $\sigma = \sigma \overset{\rho}{\underset{n}{\bowtie}} \sigma \in \mathrm{solns}(\Gamma) \underset{n}{\bowtie} \mathrm{solns}(\Gamma')$, where $\rho = \mathrm{id}$.

Conversely, suppose $\sigma \in \mathrm{solns}(\Gamma) \underset{n}{\bowtie} \mathrm{solns}(\Gamma')$. Then $\sigma = \tau \overset{\rho}{\underset{n}{\bowtie}} \tau'$, where $\tau \in \mathrm{solns}(\Gamma)$ and $\tau' \in \mathrm{solns}(\Gamma')$. Thus, there exist $W \in \Gamma$ and $W' \in \Gamma'$ such that τ solves W and τ' solves W'. Now, σ solves $W \underset{n}{\bowtie} W'$, because $\sigma \uparrow n = \tau\rho \uparrow n$ and τ solves W, $\sigma \downarrow n = \tau'\rho \downarrow n$ and τ' solves W', and $l\sigma = l\tau\rho = l'\tau'\rho = l'\sigma$. Hence $\sigma \in \mathrm{solns}(\Gamma \underset{n}{\bowtie} \Gamma')$. ∎

4.3.3 Local Variables

Suppose $m \prec n$, and (l, l') is the equation associated with the edge (m, n). Suppose τ and τ' are substitutions, with $\mathrm{LV}_m \subseteq \mathrm{dom}(\tau)$ and $\mathrm{LV}_n \subseteq \mathrm{dom}(\tau')$. If ρ is a unifier of $l\tau$ and $l'\tau'$, then define

$$\tau \overset{\rho}{\underset{n}{\leftarrow}} \tau' = \tau\rho{\restriction}\mathrm{LV}_m, \qquad \tau \overset{\rho}{\underset{n}{\rightarrow}} \tau' = \tau'\rho{\restriction}\mathrm{LV}_n.$$

If n is a node and Σ, Σ' are sets of substitutions, such that $\mathrm{LV}_m \subseteq \mathrm{dom}(\tau)$ for all $\tau \in \Sigma$ and $\mathrm{LV}_n \subseteq \mathrm{dom}(\tau')$ for all $\tau' \in \Sigma'$, then define

$$\begin{aligned}
\Sigma \underset{n}{\leftarrow} \Sigma' &= \{\tau \overset{\rho}{\underset{n}{\leftarrow}} \tau' : \tau \in \Sigma, \tau' \in \Sigma', \rho \text{ is a unifier of } l\tau \text{ and } l'\tau'\} \\
\Sigma \underset{n}{\rightarrow} \Sigma' &= \{\tau \overset{\rho}{\underset{n}{\rightarrow}} \tau' : \tau \in \Sigma, \tau' \in \Sigma', \rho \text{ is a unifier of } l\tau \text{ and } l'\tau'\}.
\end{aligned}$$

Lemma 4.13 *Suppose $m \prec n$, and Σ, Σ' are sets of substitutions, such that $\mathrm{LV}_m \subseteq \mathrm{dom}(\tau)$ for all $\tau \in \Sigma$ and $\mathrm{LV}_n \subseteq \mathrm{dom}(\tau')$ for all $\tau' \in \Sigma'$. Then*

$$(\Sigma \underset{n}{\bowtie} \Sigma'){\restriction}\mathrm{LV}_m = (\Sigma{\restriction}\mathrm{LV}_m) \underset{n}{\leftarrow} (\Sigma'{\restriction}\mathrm{LV}_n), \qquad (\Sigma \underset{n}{\bowtie} \Sigma'){\restriction}\mathrm{LV}_n = (\Sigma{\restriction}\mathrm{LV}_m) \underset{n}{\rightarrow} (\Sigma'{\restriction}\mathrm{LV}_n)$$

Proof – We consider only the first equality, the other is similar. Let (l, l') be the equation associated with the edge (m, n). First note that if τ and τ' are substitutions, then a substitution ρ is a unifier of $l\tau$ and $l'\tau'$ iff it is a unifier of $l(\tau{\restriction}\mathrm{LV}_m)$ and $l'(\tau'{\restriction}\mathrm{LV}_n)$. Moreover, if ρ is a unifier of $l\tau$ and $l'\tau'$, then

$$\begin{aligned}
(\tau \overset{\rho}{\underset{n}{\bowtie}} \tau'){\restriction}\mathrm{LV}_m &= ((\tau\rho \uparrow n) \cup (\tau'\rho \downarrow n)){\restriction}\mathrm{LV}_m, && \text{by definition} \\
&= \tau\rho{\restriction}\mathrm{LV}_m, && \mathrm{LV}_m \subseteq \mathrm{LV} \uparrow n \\
&= (\tau{\restriction}\mathrm{LV}_m)\rho{\restriction}\mathrm{LV}_m, && \text{property of restriction} \\
&= (\tau{\restriction}\mathrm{LV}_m) \overset{\rho}{\underset{n}{\leftarrow}} (\tau'{\restriction}\mathrm{LV}_n), && \text{by definition.}
\end{aligned}$$

The asserted equality now follows easily from this fact and the pointwise definitions of $\underset{n}{\bowtie}$ and $\underset{n}{\leftarrow}$ on sets of substitutions. ∎

4.3.4 The Solution Sets Interpreter

We now define the *solution sets interpreter* S. The interpreter has the same form as the basis sets interpreter, except that solutions sets are infinite whereas basis sets are finite. It will be clear in Section 4.4 that the solution set maintained by a process in this interpreter is the set of all instances of substitutions in the set maintained by the corresponding process in the basis sets interpreter.

- The state sets are defined as follows:
  1. If n is an AND-node, then the set $Q_n^{\mathcal{S}}$ consists of all pairs $(\Sigma, \bar{\Upsilon})$, where Σ is a set of substitutions with domain LV_n, and $\bar{\Upsilon} = (\bar{\Upsilon}^p : n \prec p)$ is a vector of sets of substitutions with domain LV_n.
  2. If n is an OR-node, then the set $Q_n^{\mathcal{S}}$ consists of all pairs $(\bar{\Sigma}, \bar{\Upsilon})$, where $\bar{\Sigma} = (\bar{\Sigma}^p : n \prec p)$ and $\bar{\Upsilon} = (\bar{\Upsilon}^p : n \prec p)$ are vectors of sets of substitutions with domain LV_n.
- The initial states are defined as follows:
  1. If n is an AND-node, then $\iota_n^{\mathcal{S}} = (\mathrm{Sub}(\mathrm{LV}_n), (\{\} : n \prec p))$.
  2. If n is an OR-node, then $\iota_n^{\mathcal{S}} = ((\mathrm{Sub}(\mathrm{LV}_n) : n \prec p), (\{\} : n \prec p))$.

  Here $\mathrm{Sub}(\mathrm{LV}_n)$ denotes the set of all substitutions with domain LV_n.
- The update functions for edge (m, n), with $m \prec n$, are defined as follows:
  1. If m is an AND-node and n is an OR-node,

$$\begin{aligned} \delta_{mn}^{\mathcal{S}}((\Sigma_m, \bar{\Upsilon}_m), (\bar{\Sigma}_n, \bar{\Upsilon}_n)) &= (\Sigma_m \underset{n}{\rightarrow} \bar{\Sigma}_n,\ \Sigma_m \underset{n}{\rightarrow} \bar{\Upsilon}_n) \\ \delta_{nm}^{\mathcal{S}}((\bar{\Sigma}_n, \bar{\Upsilon}_n), (\Sigma_m, \bar{\Upsilon}_m)) &= (\Sigma_m \underset{n}{\leftarrow} \bigcup \bar{\Sigma}_n,\ (\bar{\Upsilon}_m \underset{n}{\leftarrow} \bigcup \bar{\Sigma}_n)\{n/\Sigma_m \underset{n}{\leftarrow} \bigcup \bar{\Upsilon}_n\}) \end{aligned}$$

  2. If m is an OR-node and n is an AND-node,

$$\begin{aligned} &\delta_{mn}^{\mathcal{S}}((\bar{\Sigma}_m, \bar{\Upsilon}_m), (\Sigma_n, \bar{\Upsilon}_m)) = (\bar{\Sigma}_m^n \underset{n}{\rightarrow} \Sigma_n,\ \bar{\Sigma}_m^n \underset{n}{\rightarrow} \bar{\Upsilon}_n) \\ &\delta_{nm}^{\mathcal{S}}((\Sigma_n, \bar{\Upsilon}_n), (\bar{\Sigma}_m, \bar{\Upsilon}_m)) \\ &\qquad = \begin{cases} (\bar{\Sigma}_m\{n/\bar{\Sigma}_m^n \underset{n}{\leftarrow} \Sigma_n\},\ \bar{\Upsilon}_m\{n/\bar{\Sigma}_m^n \underset{n}{\leftarrow} \bigcap \bar{\Upsilon}_n\}), & n \text{ not a leaf}, \\ (\bar{\Sigma}_m\{n/\bar{\Sigma}_m^n \underset{n}{\leftarrow} \Sigma_n\},\ \bar{\Upsilon}_m\{n/\bar{\Sigma}_m^n \underset{n}{\leftarrow} \Sigma_n\}), & n \text{ a leaf}. \end{cases} \end{aligned}$$

  In the above definitions we have used several abbreviations similar to those in the definition of the basis sets interpreter (see Section 3.1).
- The output function $O^{\mathcal{S}}$ maps a state $(\bar{\Sigma}, \bar{\Upsilon}) \in Q_r^{\mathcal{S}}$ to the set $\bigcup \bar{\Upsilon}$.

4.3.5 Correctness of the Interpreter

If n is a node and U is a finite neighborhood, then define

$$\mathrm{solns}_n(U) = \mathrm{solns}(\mathrm{primes}_n(U)), \qquad \mathrm{csolns}_n(U) = \mathrm{solns}(\mathrm{cprimes}_n(U)).$$

Lemma 4.14 *Suppose* $m \prec n$ *and* $\mathrm{INC}_n(U, U')$.

1. *If m is an AND-node and n is an OR-node, then*

   (a) *For all p with $n \prec p$,*

      i. $\mathrm{solns}_p(U \cup U')\restriction \mathrm{LV}_n = \mathrm{solns}_m(U)\restriction \mathrm{LV}_m \underset{n}{\rightarrow} \mathrm{solns}_p(U')\restriction \mathrm{LV}_n$.

      ii. $\mathrm{csolns}_p(U \cup U')\restriction \mathrm{LV}_n = \mathrm{solns}_m(U)\restriction \mathrm{LV}_m \underset{n}{\rightarrow} \mathrm{csolns}_p(U')\restriction \mathrm{LV}_n$.

   (b) *For all l with $m \prec l$,*

      i. $\mathrm{solns}_m(U \cup U')\restriction \mathrm{LV}_m = \mathrm{solns}_m(U)\restriction \mathrm{LV}_m \underset{n}{\leftarrow} \bigcup_{n\prec p} \mathrm{solns}_p(U')\restriction \mathrm{LV}_n$.

      ii. $\mathrm{csolns}_l(U \cup U')\restriction \mathrm{LV}_m = \begin{cases} \mathrm{solns}_m(U)\restriction \mathrm{LV}_m \underset{n}{\leftarrow} \bigcup_{n\prec p} \mathrm{csolns}_p(U')\restriction \mathrm{LV}_n, & \text{if } l = n, \\ \mathrm{csolns}_l(U)\restriction \mathrm{LV}_m \underset{n}{\leftarrow} \bigcup_{n\prec p} \mathrm{solns}_p(U')\restriction \mathrm{LV}_n, & \text{if } l \neq n. \end{cases}$

2. *If m is an OR-node and n is an AND-node, then*

   (a) *For all p with $n \prec p$,*

      i. $\mathrm{solns}_n(U \cup U')\restriction \mathrm{LV}_n = \mathrm{solns}_n(U)\restriction \mathrm{LV}_m \underset{n}{\rightarrow} \mathrm{solns}_n(U')\restriction \mathrm{LV}_n$.

      ii. $\mathrm{csolns}_p(U \cup U')\restriction \mathrm{LV}_n = \mathrm{solns}_n(U)\restriction \mathrm{LV}_m \underset{n}{\rightarrow} \mathrm{csolns}_p(U')\restriction \mathrm{LV}_n$.

   (b) *For all l with $m \prec l$,*

      i. $\mathrm{solns}_l(U \cup U')\restriction \mathrm{LV}_m = \begin{cases} \mathrm{solns}_n(U)\restriction \mathrm{LV}_m \underset{n}{\leftarrow} \mathrm{solns}_n(U')\restriction \mathrm{LV}_n, & \text{if } l = n, \\ \mathrm{solns}_l(U)\restriction \mathrm{LV}_m, & \text{if } l \neq n. \end{cases}$

      ii. $\mathrm{csolns}_l(U \cup U')\restriction \mathrm{LV}_m$

$$= \begin{cases} \mathrm{solns}_n(U)\restriction \mathrm{LV}_m \underset{n}{\leftarrow} \bigcap_{n\prec p} \mathrm{csolns}_p(U')\restriction \mathrm{LV}_n, & \text{if } l = n, n \text{ not a leaf}, \\ \mathrm{solns}_n(U)\restriction \mathrm{LV}_m \underset{n}{\leftarrow} \mathrm{solns}_n(U')\restriction \mathrm{LV}_n, & \text{if } l = n, n \text{ a leaf}, \\ \mathrm{csolns}_l(U)\restriction \mathrm{LV}_m, & \text{if } l \neq n. \end{cases}$$

Proof – The result follows immediately from Lemmas 4.10, 4.11, 4.12, and 4.13. ∎

Theorem 2 *Suppose T is an AND/OR tree. For each node n in T, define a map $\mathcal{NS}_n : Q_n^{\mathcal{N}} \to Q_n^{\mathcal{S}}$ as follows:*

1. *If n is an AND-node, then define $\mathcal{NS}_n(U) = (\mathrm{solns}_n(U)\restriction \mathrm{LV}_n, (\mathrm{csolns}_p(U)\restriction \mathrm{LV}_n : n \prec p))$.*

2. *If n is an OR-node, then define*

$$\mathcal{NS}_n(U) = ((\mathrm{solns}_p(U)\restriction \mathrm{LV}_n : n \prec p), (\mathrm{csolns}_p(U)\restriction \mathrm{LV}_n : n \prec p)).$$

Then the pair $(\mathrm{INC}, \mathcal{NS})$ is a simulation from $\mathcal{N}(T)$ to $\mathcal{S}(T)$.

Proof – We must verify the conditions in the definition of a simulation. The output function condition is obvious. The initial state conditions follow directly from the fact that $\mathrm{primes}_n(\{n\}) = \{\{n\}\}$, and if $n \prec p$, then $\mathrm{primes}_p(\{n\}) = \{\{n\}\}$ and $\mathrm{cprimes}_p(\{n\}) = \{\}$.

To verify the update function conditions, we use Lemma 4.14. There are two cases, each with two subcases:

1. If m is an AND-node, n is an OR-node, and $\mathrm{INC}_n(U_m, U_n)$ holds, then

   (a) $\mathcal{NS}(\delta_{mn}^{\mathcal{N}}(U_m, U_n)) = \delta_{mn}^{\mathcal{S}}(\mathcal{NS}(U_m), \mathcal{NS}(U_n))$.

(b) $\mathcal{NS}(\delta^{\mathcal{N}}_{nm}(U_n, U_m)) = \delta^{\mathcal{S}}_{nm}(\mathcal{NS}(U_n), \mathcal{NS}(U_m))$.

2. If m is an OR-node and n is an AND-node, and $\mathrm{INC}_n(U_m, U_n)$ holds, then

   (a) $\mathcal{NS}(\delta^{\mathcal{N}}_{mn}(U_m, U_n)) = \delta^{\mathcal{S}}_{mn}(\mathcal{NS}(U_m), \mathcal{NS}(U_n))$.

   (b) $\mathcal{NS}(\delta^{\mathcal{N}}_{nm}(U_n, U_m)) = \delta^{\mathcal{S}}_{nm}(\mathcal{NS}(U_n), \mathcal{NS}(U_m))$.

Each subcase is established by using the definitions of $\mathcal{NS}$, $\delta^{\mathcal{P}}_{mn}$, and $\delta^{\mathcal{S}}_{mn}$, and then applying the corresponding part of Lemma 4.14. The details are straightforward, and are omitted. ∎

Corollary 4.15 *The solution sets interpreter is correct.*

4.4 Correctness of Basis Sets Interpreter

We now establish the correctness of the basis sets interpreter by exhibiting a simulation from it to the solution sets interpreter. The simulation mapping is in fact the *span* mapping that was defined in Section 3.1. We use the following "lifting lemma" to show that span has the required properties.

Lemma 4.16 (Lifting Lemma) *Suppose η and η' are range-disjoint substitutions. Suppose substitutions τ and τ' are such that $\eta \precsim \tau$ and $\eta' \precsim \tau'$. Let l, l' be literals with $\mathrm{vars}(l) \subseteq \mathrm{dom}(\eta)$ and $\mathrm{vars}(l') \subseteq \mathrm{dom}(\eta')$. If $l\tau$ and $l'\tau'$ have a unifier ρ, then $l\eta$ and $l'\eta'$ are unifiable. Moreover, if $\mu = \mathrm{mgu}(l\eta, l'\eta')$, then there exists λ such that $\eta\mu\lambda = \tau\rho$ and $\eta'\mu\lambda = \tau'\rho$.*

Proof – Since η, η' are range-disjoint, we may choose κ, κ', with disjoint supports, such that $\eta\kappa = \tau$ and $\eta'\kappa' = \tau'$. Let the substitution θ be defined as follows:

$$\theta(v) = \begin{cases} v\kappa\rho, & \text{if } v \in \mathrm{supp}(\kappa), \\ v\kappa'\rho, & \text{if } v \in \mathrm{supp}(\kappa'), \\ v\rho, & \text{otherwise.} \end{cases}$$

Now, ρ is a unifier of $l\tau$ and $l\tau'$, so θ is a unifier of $l\eta$ and $l'\eta'$. Let $\mu = \mathrm{mgu}(l\eta, l'\eta')$, then $\mu \precsim \theta$, so there exists λ such that $\mu\lambda = \theta$. Then $\eta\mu\lambda = \eta\theta = \tau\rho$ and $\eta'\mu\lambda = \eta'\theta = \tau'\rho$. ∎

Lemma 4.17 *Suppose $m \prec n$, and Σ, Σ' are sets of substitutions, such that $\mathrm{LV}_m \subseteq \mathrm{dom}(\tau)$ for all $\tau \in \Sigma$ and $\mathrm{LV}_n \subseteq \mathrm{dom}(\tau')$ for all $\tau' \in \Sigma'$. Then*

$$[\Sigma \underset{n}{\Rightarrow} \Sigma'] = [\Sigma] \underset{n}{\rightarrow} [\Sigma'], \qquad [\Sigma \underset{n}{\Leftarrow} \Sigma'] = [\Sigma] \underset{n}{\leftarrow} [\Sigma'].$$

Proof – (1) and (2) are similar; we prove only (1).

Suppose $\sigma \in [\Sigma \underset{n}{\Rightarrow} \Sigma']$. Then there exists $\eta \in \Sigma$ and $\eta' \in \Sigma'$ such that $l\eta\omega$ and $l'\eta'\omega'$ are unifiable, and if $\mu = \mathrm{mgu}(l\eta\omega, l'\eta'\omega')$, then there exists κ such that $\sigma = (\eta'\omega'\mu\kappa)\restriction\mathrm{LV}_n$. Let $\tau = \eta\omega\mu\kappa$ and $\tau' = \eta'\omega'\mu\kappa$, then $\sigma = \tau \overset{\rho}{\underset{n}{\rightarrow}} \tau'$, where $\rho = \mathrm{id}$. Since $\tau \in [\Sigma]$ and $\tau' \in [\Sigma']$, we have shown that $\sigma \in [\Sigma] \underset{n}{\rightarrow} [\Sigma']$.

Conversely, suppose $\sigma \in [\Sigma] \underset{n}{\rightarrow} [\Sigma']$. Then $\sigma = \tau \overset{\rho}{\underset{n}{\rightarrow}} \tau' = (\tau'\rho)\restriction\mathrm{LV}_n$ for some $\tau \in [\Sigma]$, $\tau' \in [\Sigma']$, and unifier ρ of $l\tau$ and $l'\tau'$. Let $\eta \in \Sigma$ and $\eta' \in \Sigma'$ be such that $\eta \precsim \tau$ and $\eta' \precsim \tau'$. It is easy to see that we also have $\eta\omega \precsim \tau$ and $\eta'\omega' \precsim \tau'$. Since ρ is a unifier of $l\tau$ and $l'\tau$, by Lemma 4.16 $l\eta\omega$ and $l'\eta'\omega'$ are unifiable. Moreover, if $\mu = \mathrm{mgu}(l\eta\omega, l'\eta'\omega')$, then there exists λ such that $\eta\omega\mu\lambda = \tau\rho$ and $\eta'\omega'\mu\lambda = \tau'\rho$. Note that $\eta \underset{n}{\Rightarrow} \eta' = (\eta'\omega'\mu)\restriction\mathrm{LV}_n \in \Sigma \underset{n}{\Rightarrow} \Sigma'$. But then $(\eta \underset{n}{\Rightarrow} \eta')\lambda = ((\eta'\omega'\mu)\restriction\mathrm{LV}_n)\lambda = (\eta'\omega'\mu\lambda)\restriction\mathrm{LV}_n = \sigma$, thus showing that $\eta \underset{n}{\Rightarrow} \eta' \precsim \sigma$ and hence that $\sigma \in [\Sigma \underset{n}{\Rightarrow} \Sigma']$. ∎

Theorem 3 *Suppose T is an AND/OR tree. For each node n in T, define a map $\mathcal{BS}_n : Q_n^{\mathcal{B}} \to Q_n^{\mathcal{B}}$ as follows:*

1. *If n is an AND-node, then define $\mathcal{BS}_n(\Xi, \bar{\Lambda}) = ([\Xi], ([\bar{\Lambda}^p] : n \prec p))$.*
2. *If n is an OR-node, then define $\mathcal{BS}_n(\bar{\Xi}, \bar{\Lambda}) = (([\bar{\Xi}^p] : n \prec p), ([\bar{\Lambda}^p] : n \prec p))$.*

Let **true** *denote the property that holds of all global states of $\mathcal{B}(T)$. Then the pair $(\mathbf{true}, \mathcal{BS})$ is a simulation from $\mathcal{B}(T)$ to $\mathcal{S}(T)$.*

Proof – The proof is immediate from Theorem 2, Lemma 4.17, and the definition of $\bigvee$ and $\bigwedge$. ∎

Corollary 4.18 *The basis sets interpreter $\mathcal{B}$ is correct.*

5 Discussion

We have described a simple distributed interpreter for logic programs that exploits both AND- and OR-parallelism, and we have proved its soundness and completeness. A feature of our interpreter is that it supports bidirectional communication between AND-parallel goals. This feature is important because such communication can result in increased efficiency through pruning of irrelevant alternatives. A good example of an application in which such pruning would seem to be useful is alpha-beta minimax search.

There are several issues that must be addressed before a practical interpreter can be based on our results. Clearly, data and communication requirements of our interpreter must be optimized. For example, it is not necessary for processes to maintain the set of lower-bound alternatives separately from the upper-bound set. Rather, tags can be used to identify those elements of the upper-bound set that are also members of the lower-bound set. It is also unnecessary for processes to repeatedly send their entire state to their neighbors, since this state will mostly consist of information already known to the neighbor. We believe it is possible, though, to design a highly efficient protocol for the incremental communication of state information between nodes. Such a protocol would require the design of data structures with which sets of substitutions can be recorded, and the incremental updates efficiently computed. We can optimize the data structures by observing that the substitutions in a state are some instances of substitutions in the previous state and so on up to the initial state, and hence two substitutions in the current state can share the structure of their common ancestor substitutions.

The design of an interpreter to run on a particular parallel or distributed architecture must address the problems of how to dynamically construct the AND/OR tree, assign processes to processors and schedule processes assigned to the same processor. Solution to these problems will obviously depend on the particular architecture, and probably also on the properties of application programs to be run. It seems clear that a fully distributed implementation of the interpreter described here will probably not be desirable in most circumstances. Rather, it will probably be necessary for efficiency reasons to combine the tasks of a number of node processes into one. This can be done statically, for example by assigning one processor to handle the tasks of all AND-nodes labeled by the same program clause, or dynamically, for example by assigning one processor to handle the tasks of all nodes in some neighborhood of the AND/OR tree. The results of this paper will be useful in showing what must be accomplished by the processors to ensure soundness and completeness. It also seems clear that not all application programs will benefit from the completely unrestricted communication patterns supported by our model. However, as our model does not impose any restrictions other than a minimal bottom-up fairness on the pattern of communication between processes, we feel that it is easy to introduce performance-improving heuristics for scheduling communication between processes.

An important issue in the design of distributed interpreters is the problem of detection of duplicate goals and the elimination of such redundant computations. Ullman [Ull84] has proposed a method in which the states of all processes are sorted at fixed intervals, so that two or more processes with identical states (these correspond to duplicate goals in his method) can be detected. In such cases, only one of the processes is allowed to proceed while others wait for answers from that process. However, his method is not directly applicable to our interpreter because in our interpreter a goal can be affected by other goals conjunctive with it, and hence identical goals may not generate the same set of answers. The extension of Ullmans' method to our interpreter merits further investigation.

References

[CDD85] Jung-Herng Chang, Alvin M. Despain, and Doug DeGroot. AND-parallelism of logic programs based on a static data dependency analysis. In *Digest of Papers, Compcon85*, pages 218–225, February 1985.

[CG86] Keith Clark and Steve Gregory. PARLOG : parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[CH83] Andrzej Ciepielewski and Seif Haridi. A formal model for OR-parallel execution of logic programs. In R.E.A Mason, editor, *Information Processing 83*, pages 299–305, North-Holland, September 1983.

[CJ87] P. Daniel Cheng and J.Y. Juang. A parallel resolution procedure based on connection graph. In *Sixth National Conference on Artificial Intelligence*, pages 13–17, AAAI87, July 1987.

[CK83] J.S Conery and D.F. Kibler. AND parallelism in logic programs. *Proceedings of 8th IJCAI*, 539–543, August 1983.

[CK85] J.S Conery and D.F. Kibler. AND parallelism and nondeterminism in logic programs. *New Generation Computing*, 3:43–70, 1985.

[GTM84] A. Goto, H. Tanaka, and T. Moto-Oka. Highly parallel inference engine PIE—goal rewriting model and machine architecture. *New Generation Computing*, 2:37–58, 1984.

[Her86] M.V. Hermenegildo. An abstract machine for restricted AND-parallel execution of logic programs. In *Lecture Notes in Computer Science, Vol 225*, pages 25–39, Third International Conference on Logic Programming, Springer-Verlag, July 1986.

[Kal86] L.V. Kale. *Parallel Architectures for Problem Solving.* PhD thesis, SUNY Stony Brook, 1986.

[Kal87] L.V. Kale. 'completeness' and 'full parallelism' of parallel logic programming schemes. In *Proceedings 1987 Symposium on Logic Programming*, pages 125–133, August 1987.

[KL87] Michael Kifer and E. Lozinskii. Implementing logic programs as a database system. In *Third International Conference on Data Engineering*, February 1987.

[Kow79] R.A. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland, 1979.

[LM86] Peyyun Peggy Li and Alain J. Martin. The SYNC model: a parallel execution method for logic programming. In *Proceedings 1986 Symposium on Logic Programming*, pages 223–234, IEEE, September 1986.

[LP84] Gary Lindstrom and Prakash Pananagaden. Stream-based execution of logic programs. In *International Symposium on Logic Programming*, pages 168–176, February 1984.

[Pol81] George H. Pollard. *Parallel Execution of Horn Clause Programs.* PhD thesis, Dept. of Computing, Imperial College, London, 1981.

[Sha83] Ehud Y. Shapiro. *A subset of Concurrent Prolog and its Interpreter.* Technical Report, ICOT Tokyo, 1983.

[Ull84] Jeffrey D. Ullman. Flux, sorting, and supercomputer organization for ai applications. *Journal of Parallel and Distributed Computing*, 133–151, 1984.

[UT83] Shinji Umeyama and Koichiro Tamura. A parallel execution model of logic programs. In *Tenth International Symposium on Computer Architecture*, pages 349–355, June 1983.

[WADK84] D.S. Warren, M. Ahamad, S.K. Debray, and L.V. Kale. Executing distributed prolog programs on a broadcast network. In *Proceedings of the 1984 Logic Programming Symposium*, February 1984.

[Wis86] Michael J. Wise. *Prolog Multiprocessors.* Prentice-Hall, 1986.

[YN84] Hiroshi Yasuhara and Kazuhiko Nitadori. ORBIT: a parallel computing model of prolog. *New Generation Computing*, 277–288, 1984.